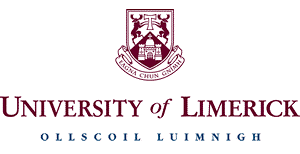
****

**Department of Computer Science & Information Systems**

Restaurant Booking System

Team 10

System Analysis and Design

Yiran Gao 17083214

Sean Wright 15145271  
Gavin Randles 12130583

Patryk Leszczynski 15178536  
Michael Keegan 15170756

Table of Contents

[**Knowledge**](#_6ier5g67pir5) **3**

[Description of Business Scenario](#_5739tubtp7ko) 3

[Business Logic](#_719wugy50fgs) 4

[Software Development Lifecycle](#_8xw1n9g01lyf) 6

[V- Model](#_l4gstk993v6j) 6

[Agile](#_nskpfxwhpy0j) 7

[Project Plan](#_4z3bb2g7lcm1) 8

[Project Roles](#_dxeszz9e1ldg) 9

[**Project Requirements**](#_ngv9owm3edpx) **10**

[Requirement](#_3tgidjtjuvmq) 10

[· Use case](#_4x9q1o5oj2y1) 10

[· Customer](#_sonb2imcoq6n) 10

[· Staff (Front of house/Receptionist)](#_a7md3di53k2w) 10

[· Manager](#_hw3aubkhw2xi) 11

[Optional Additions](#_k5la931a07zr) 11

[· Customer](#_argxfx6payba) 11

[· Staff](#_cvwajmsodcmk) 11

[Use Case Diagrams](#_7op75j6q9wl4) 12

[Use case 1:](#_43e7d7pk32oo) 12

[Use case 2:](#_ahrlpk9hvlg4) 13

[Use Case Descriptions](#_a81efooh43sx) 14

[Pre-developement](#_2qy20qdeoqdk) 14

[Post-developement](#_kr45gt7onwm1) 18

[Successfully Implemented Use Cases](#_16248gtdm4tl) 18

[Use Case Diagram](#_7iqzc8v68pjx) 21

[GUI Prototypes](#_t3k194oiouub) 22

[GUI After Implementation](#_rf16wip1147k) 26

[E-R Diagram](#_f0m93xbsgaft) 30

[Pre-developement](#_tcx1ldlw1y4v) 30

[Post-developement](#_ytprwp7l0pqk) 31

[**System Architecture**](#_bs6l97pwmvpp) **32**

[Candidate Classes](#_lgcb28155jme) 32

[Package diagram](#_cwrz329847qu) 33

[Make Booking sequence diagram](#_8iudriksnda) 34

[Modify Booking sequence diagram](#_9yvjyvdaoyr5) 35

[Architectural discussions Taken](#_94vligxdqz4l) 36

[Class Analysis Pre-developement](#_58mhyjp3p5lq) 37

[**Code**](#_sv07dauo5lxw) **39**

[Packages](#_qpwm7sjnteb0) 39

[MVC](#_85eir0uj5w22) 44

[Collaboration Diagram](#_gkwhfg1wd1rj) 47

[Final Package Diagram](#_q0ubznvr3a8l) 47

[Register Customer class diagram](#_kxbn8f7ivv00) 48

[DAO](#_sz6acfj82mjs) 49

[Model/Bean](#_rbdxgzj9dok2) 50

[Controller](#_4vzzqrxvd458) 51

[View](#_8hzsrl18dngv) 51

[Full system uml](#_2zjcurcqdmyz) 52

[Final login squence diagram](#_i0kbm2jgv3js) 53

[**Design Pattern**](#_rxmg5jbj3hq) **54**

[Refactoring](#_s2o7ym4n2kbl) 57

[**Added Value**](#_qy895fqd5wd3) **62**

[Git & Github](#_ol54iwnlo0jk) 62

[DATABASE - MySQL](#_j80i1n1r5vyv) 64

[Tables](#_2vgmu7w1pnhi) 64

[JUnit](#_mya0gc8avyt8) 65

[**Critique**](#_sefs95vc433k) **66**

[**Reference**](#_11kzrhe4z13v) **69**

[**Marking Scheme**](#_mvjtbrbaux5t) **70**

# Knowledge

## Description of Business Scenario

The purpose of our project is to make a Restaurant Booking System which let customers make a reservation online. The system would involve a booking system and record system. The booking system allows user to make, change and cancel a booking, and the record system would record every feedback mark from customer, also it will contain the overbooking rate (customer who made booking but not show up) and a reputation for each customer.

There are four types of user, first user type is customer, a customer can only make a booking online when the customer is register and login to the system, otherwise the only thing he can do is to view existing booking. Customer could select a date and restaurant location, then the system will send it to server and the server will respond a page with valid tables on it. Customer will use his E-mail address, phone number and name as references on the booking. And after making a reservation on the booking page, system will give customer a unique number as an evidence of order and customer could use this number to modify the booking. A customer would change the booking and cancel booking and after eating at restaurant the customer would give a feedback form to mark waiter and front-house as a reward. Second user type is administrator, an admin could modify the booking include changing and cancelling. Also administrator could look through the feedback and doing statistics on staff reward marks. Third and last user types are waiter and front-house, both of them could modify the existing booking. Once customer shows up, front-house will record the covers and give the table number at restaurant, this table number also will send to waiter. If customer did not show up, the front-house will record it and put it into overbooking rate, also the system would reduce the reputation of that customer. For waiter, he will lead customers to the table and set cost of money on booking order.

## Business Logic

Our business logic based on restaurant reservation and loyalty of each customer. The restaurant booking system has different restaurants in the list and customer could choose their favourite one to make a booking on it. Different restaurant provides different type of discount when customer making reservation on it. There are two types of discount we have implemented on La Cucina and Milano (restaurant name that exist in database) one is based on the day if it is Tuesday then customer will get 5% and for Milano you may get 10% on Friday. Also for each chain of restaurants we will give extra discount for customer who has great loyalty number. As For loyalty number, each end user (customer) will have a loyalty number(100) when they register in the system. And every time they check in the reservation they will get 100 added on the loyalty number and if they do not show up on time then will take out 50 points of loyalty number.

So based on the specific business logic, the system will encourage people more on select chain restaurants, that would stabilise the market of customers. Also with the daily off discount in the reservation, people will prefer to go out eating in restaurant in holiday rather than cooking in home.

## Software Development Lifecycle

### V- Model

Ref <https://diagramchartwiki.com/systems-engineering-v-diagram/systems-engineering-v-diagram-verification-and-validation-the-mitre-corporation/>

Diagramchartwiki.com (2014)

The V-model is a graphical representation of the systems development lifecycle. It summarizes the main steps to be taken in conjunction with the corresponding deliverables within computerized system validation framework.  
  
The V represents the sequence of steps in a project life cycle development. It describes the activities to be performed and the results that have to be produced during product development. The left side of the "V" represents the decomposition of requirements, and creation of system specifications. The right side of the V represents integration of parts and their validation.

### Agile



Agile software development is an approach to software development under which requirements and solutions evolve through the collaborative effort of self-organizing and cross-functional teams and their customer. It advocates adaptive planning, evolutionary development, early delivery, and continual improvement, and it encourages rapid and flexible response to change.

In total, Agile is more suitable for the projects where requirements change rapidly. V-model is suitable where requirement changes are almost none.

For our team project, the requirements are kind static, so that means we do not need to mandates customer interaction on a regular basis. And will not change direction concept. But we are keeping watching requirements and scenarios, and decided whether look back or ahead changing is necessary for the software development lifecycle.

## Project Plan

|  |  |  |  |
| --- | --- | --- | --- |
| **Section** | **Description** | **Responsibility** | **Week** |
| **Purpose and Concept** | Describe the project and scenarios | Yiran | 4 |
| **Software requirements** | Describe the project requirements and software development lifecycle | Yiran | 5 |
| **Roles** | Complete the specific roles for each team member | Group | 4 |
| **System Architecture** | Analyse system and give basic architecture of it | Sean,Yiran,Patryk | 6 |
| **Analysis Sketches** | Create diagrams on each analysed function | Sean,Yiran | 6 |
| **Coding** | Implement the system | Group | 7-10 |
| **Testing** | Testing system | Sean, MJ, Patryk, Yiran | 11 |
| **Acceptance** | Make sure all functions are highly satisfied to requirements | Gavin | 11 |
| **Project Report** | Create a project report to describe the system | Group | 12 |
| **Reference** | Inventory of Resources | Group | 12 |

## Project Roles

|  |  |  |
| --- | --- | --- |
| **Role** | **Description** | **Designated Team Member** |
| **Project Manager** | Sets up group meetings, get agreement on the project plan and tracks progress. | Yiran |
| **Documentation Manager** | Responsible for sourcing relevant supporting documentation from each team member and composing it in the report. | MJ |
| **Business Analyst/Requirements Engineer** | Responsible for section 6- Requirements. | Patryk |
| **Architect** | Defines system architecture | Sean |
| **Systems Analysts** | Creates conceptual class model | Sean, Yiran |
| **Designer** | Responsible for recovering design time blueprints from implementation | Patryk |
| **Technical Lead** | Leads the implementation effort | Gavin |
| **Programmers** | Each team member to develop at least 1 package in the architecture | All |
| **Tester** | Coding of automated test cases | Sean, MJ, Patryk, Yiran |
| **Dev Ops** | Must ensure that each team member is competent with development infrastructure. | MJ |

# Project Requirements

## Requirement

### · Use case

o A booking System should:

§ Record a new booking information (also called booking sheet)

§ Cancel a booking information

§ Record arrival of a customer

§ Table transfer

o

o Also, system should collect those info:

§ Name, Phone number and covers. (on booking sheet)

§ For each booking, occupies an available table.

§ Record walk-ins and booking but not taking table customers. (only for counting, no phone number and name record)

§ Update the information on booking sheet

§ Whether has changed the table

§ Whether arrived or not

§ Cancellation

o So, the basic functions are:

§ Record booking

§ Updating booking sheet (on time)

### · Customer

o Make bookings (receives booking confirmation if reference number)

o Cancel bookings

o Change bookings (Reschedule or change size of party)

### · Staff (Front of house/Receptionist)

o Log in to system

o Can do everything a customer can do

o Checking customers in or out

### · Manager

o Can do everything the staff can do

o Print up to date list of bookings

o Creating seating plan

o Modify seating plan

### Optional Additions

#### · Customer

o Choose a table

o Pre-order food

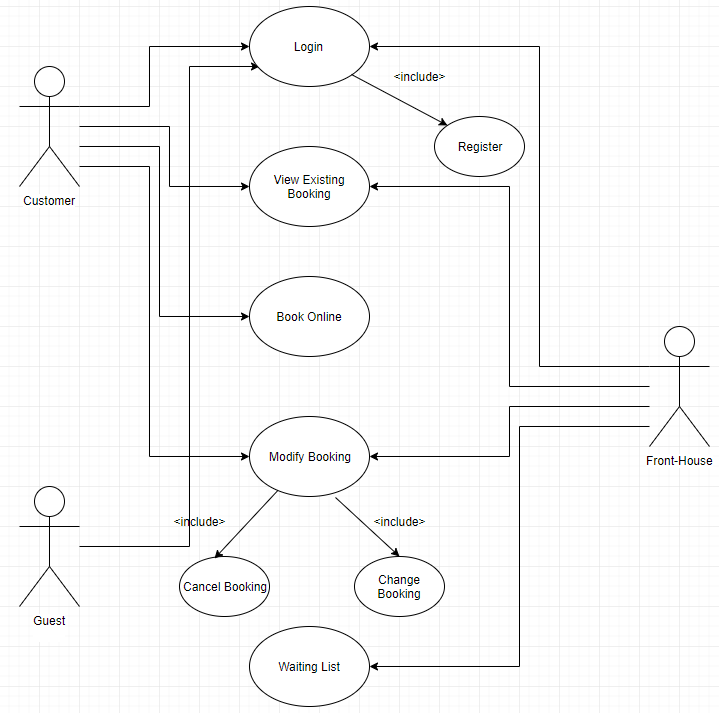
#### · Staff

o Submit food orders

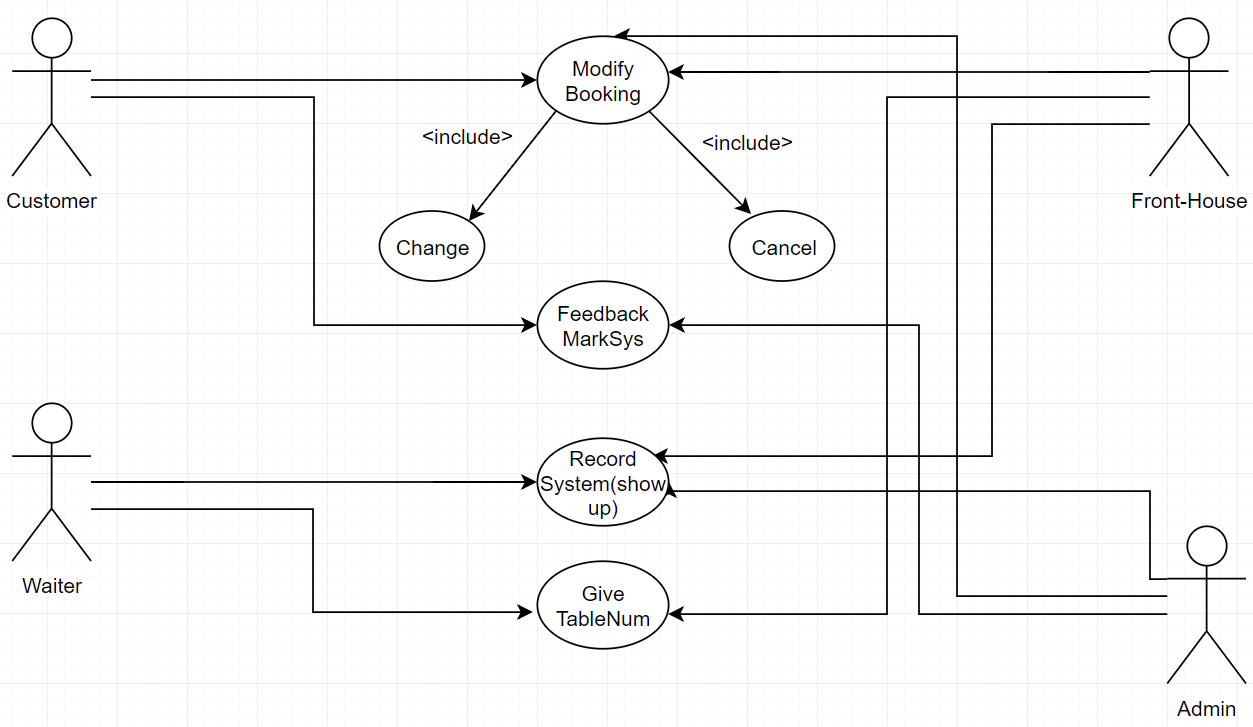
• If not enough table, the system needs to give an option to customer:  
 If other restaurant has available table   
 Else if Using 2 or more tables to make a big table.  
 Else if Setting in waiting list.  
 Cannot order.

## Use Case Diagrams

### Use case 1:



### Use case 2:



## Use Case Descriptions

### Pre-developement

|  |  |
| --- | --- |
| Use Case 1 | Staff Login |
| Actor Action | System Response |
| 1 – Staff enters username and password. | 2 – Login details are checked. Displays main menu. |
| Alternative route | |
| 1 (a) – Staff enters incorrect username or password. | 2 (b) – Login check fails. Error message is displayed. |

**Non-functional requirement: Security**User password must be encrypted. Store hashed passwords in the database.

|  |  |
| --- | --- |
| Use Case 2 | Staff Log Out |
| Actor Action | System Response |
| 1 – Staff selects log out option. | 2 – System displays log out screen. |

|  |  |
| --- | --- |
| Use Case 3 | Customer Registration |
| Actor Action | System Response |
| 1 – Customer chooses registration. | 2 – System displays registration screen. |
| 3 – Customer enters his/her details. | 4 – System displays customer main menu. |
| Alternative route | |
| 3 (a) – Customer enters invalid details. | 4 (a) – Registration fails. Appropriate error message displayed. |
| 3 (b) – Customer already has an existing account. | 4 (a) - Registration fails. Appropriate error message displayed. |

**Non-functional requirement: Security**User passwords must be encrypted. Store hashed passwords in the database. System should store the actor details safely.

|  |  |
| --- | --- |
| Use Case 4 | Customer Login |
| Actor Action | System Response |
| 1 – Customer chooses login. | 2 – System displays login screen. |
| 3 – Customer enters username and password | 4 - Login details are checked. Displays main menu. |
| Alternative route | |
| 3 (a) – Customer enters incorrect username or password. | 4 (b) – Login check fails. Error message is displayed. |

**Non-functional requirement: Security**User password must be encrypted. Store hashed passwords in the database.

|  |  |
| --- | --- |
| Use Case 5 | Customer Log Out |
| Actor Action | System Response |
| 1 – Customer selects log out option. | 2 – System displays log out screen. |

|  |  |
| --- | --- |
| Use Case 6 | Administrator Login |
| Actor Action | System Response |
| 1 – Administrator chooses login. | 2 – System displays login screen. |
| 3 – Administrator enters username and password | 4 - Login details are checked. Displays main menu. |
| Alternative route | |
| 3 (a) – Administrator enters incorrect username or password. | 4 (b) – Login check fails. Error message is displayed. |

**Non-functional requirement: Security**User password must be encrypted. Store hashed passwords in the database.

|  |  |
| --- | --- |
| Use Case 7 | Administrator Log Out |
| Actor Action | System Response |
| 1 – Administrator selects log out option. | 2 – System displays log out screen. |

|  |  |
| --- | --- |
| Use Case 8 | Table Reservation |
| Actor Action | System Response |
| 1 - User chooses date and restaurant location | 2 - System returns current table states |
| 3 - User chooses a table and fills booking form | 4 - System records form and gives verify number |
| Alternative route | |
| 3 (a) - User puts incorrect info to booking form | 4 (b) - Booking fails, error message is displayed. |

**Non-functional requirement: Usability**It should provide a friendly GUI and let user complete the form easily and quickly.

|  |  |
| --- | --- |
| Use Case 9 | Change Reservation |
| Actor Action | System Response |
| 1 - User gives verify number and personal info | 2 - System returns reservation |
| 3 - User chooses a new date | 4 - System modifies update the reservation |
| Alternative route | |
| 1 (a) - User gives wrong number and info | 2 (b) - Changing fails, system returns error. |

**Non-functional requirement: Reliability**Because this use case is based on the user operation, it should be recorded without any crash in it, and make sure the changing info exactly upload to the reservation.

|  |  |
| --- | --- |
| Use Case 10 | Cancel Reservation |
| Actor Action | System Response |
| 1 - User gives verify number and personal info | 2 - System returns reservation |
| 3 - User chooses a new date | 4 - System cancels the reservation |
| Alternative route | |
| 1 (a) - User gives wrong number and info | 2 (b) - Cancel fails, system returns error. |

**Non-functional requirement: Reliability**Because this use case is based on the user operation, it should be recorded without any crash in it, and make sure no miss on any cancel operation.

|  |  |
| --- | --- |
| Use Case 11 | Feedback |
| Actor Action | System Response |
| 1 - User gives marks on waiter and front-house | 2 - System records the feedback mark |
| Alternative route | |
| 1 (a) - User gives wrong marks | 2 (b) - Submission fails, system returns error. |

**Non-functional requirement: Performance**The feedback system needs to take quick response of operation and it is important that survey comes quickly so that it won’t take much time on customer.

|  |  |
| --- | --- |
| Use Case 12 | Customer Checkin |
| Actor Action | System Response |
| 1 - User puts verify number and customer info | 2 - System changes the table state and returns that table number. |
| 3 - User sends the table number and selects a waiter | 4 - System records the table number and waiter ID. |
| Alternative route | |
| 1 (a) - User gives wrong number and info | 2 (b) - Checkin fails, system returns error. |

**Non-functional requirement: Reliability**Because this use case is one of the most important parts in the booking system, it should have high reliability on it, no missing operation is allowed during operating.

|  |  |
| --- | --- |
| Use Case 13 | Waiter Checkout |
| Actor Action | System Response |
| 1 - User gives feedback mark and money the customer cost. | 2 - System records the money and feedback mark, then return a finish message. |
| Alternative route | |
| 1 (a) - User gives wrong mark and info | 2 (b) - Submission fails, system returns error. |

**Non-functional requirement: Reliability**Because this use case is based on the user operation, it should be recorded without any crash in it, and make sure the feedback form is completed and recorded in the system database.

|  |  |
| --- | --- |
| Use Case 14 | Reputation System |
| Actor Action | System Response |
| 1 - User sets reservation state to overbooking | 2 - System records it and reduces the reputation of that customer |
| 3 - User inputs reservation booking and customer info | 4 - System cancels the reservation |
| Alternative route | |
| 1 (a) - User gives wrong info and states | 2 (b) - Changing states fails, system returns error. |

**Non-functional requirement: Reliability**Because this use case is based on the user operation, it should not miss any cancel and the reputation value needs to be in a correct range after changing it.

|  |  |
| --- | --- |
| Use Case 15 | View Feedback System |
| Actor Action | System Response |
| 1 - User gives waiterID and front-houseID | 2 - System returns feedback marks |
| Alternative route | |
| 1 (a) - User gives wrong ID | 2 (a) - Selection fails, system returns error. |

**Non-functional requirement: Usability**This feedback view system is for administrator to see the feedback marks on each staff, so it should be easily to use and administrator could quickly find information on it.

### Post-developement

#### Successfully Implemented Use Cases

* Use Case 1: Staff Login
  + Author: Gavin
  + Packages:
    - View: LoginView
    - Controller: LoginController
    - Bean: StaffBean
    - DAO: StaffDAO
  + Description:

The initial screen that is displayed when the application is run, it allows a member of staff to log in to the system.

* Use Case 2: Staff Log Out
  + Author: Gavin
  + Packages:
    - View: MainmenuView
    - Controller: MainmenuController
  + Description:

A button on the main menu view that brings the user back to the log in.

* Use Case 3: Customer Registration
  + Author: Sean
  + Packages:
    - View: CustomerRegisterView
    - Controller: CustomerRegisterController
    - Bean: CustomerBean
    - DAO: CustomerDAO
  + Description:

Once entered from the main menu the customer registration displays a form to create a new customer.

* Use Case 6: Administrator Login
  + See use case 1
* Use Case 7: Administrator Log Out
  + See use case 2
* Use Case 8: Table Reservation
  + Author: Patryk
  + Packages:
    - View: ReservationBookingView
    - Controller: BookingController
    - Bean: BookingBean
    - DAO: BookingDAO
    - Discount: All
  + Description:

Table Reservation is accessed through the main menu screen, the view displays a form to be filed out with the relevant data.

* Use Case 9: Modify Reservation
  + Author: Patryk
  + Packages:
    - View: SearchView, ModifyReservationView
    - Controller: SearchController, ModifyBookingController
    - Bean: BookingBean
    - DAO: BookingDAO
  + Description:

After searching using the booking ID the user is brought to the modify booking view.

* Use Case 10: Cancel Reservation
  + Author: Michael & Gavin
  + Packages:
    - View: SearchView
    - Controller: SearchController, CancelReservation
    - Bean: BookingBean
    - DAO: BookingDAO
  + Description:

After successfully searching the booking ID the booking is removed from the system.

* Use Case 11: Feedback
  + Author: Michael
  + Packages:
    - View: FeedBackView
    - Controller: FeedbackController
    - Bean: FeedbackBean
    - DAO: FeedbackDAO
  + Description:

A staff member can enter the feedback from a customer on a scale of 1-5. That score is added to the staff members info.

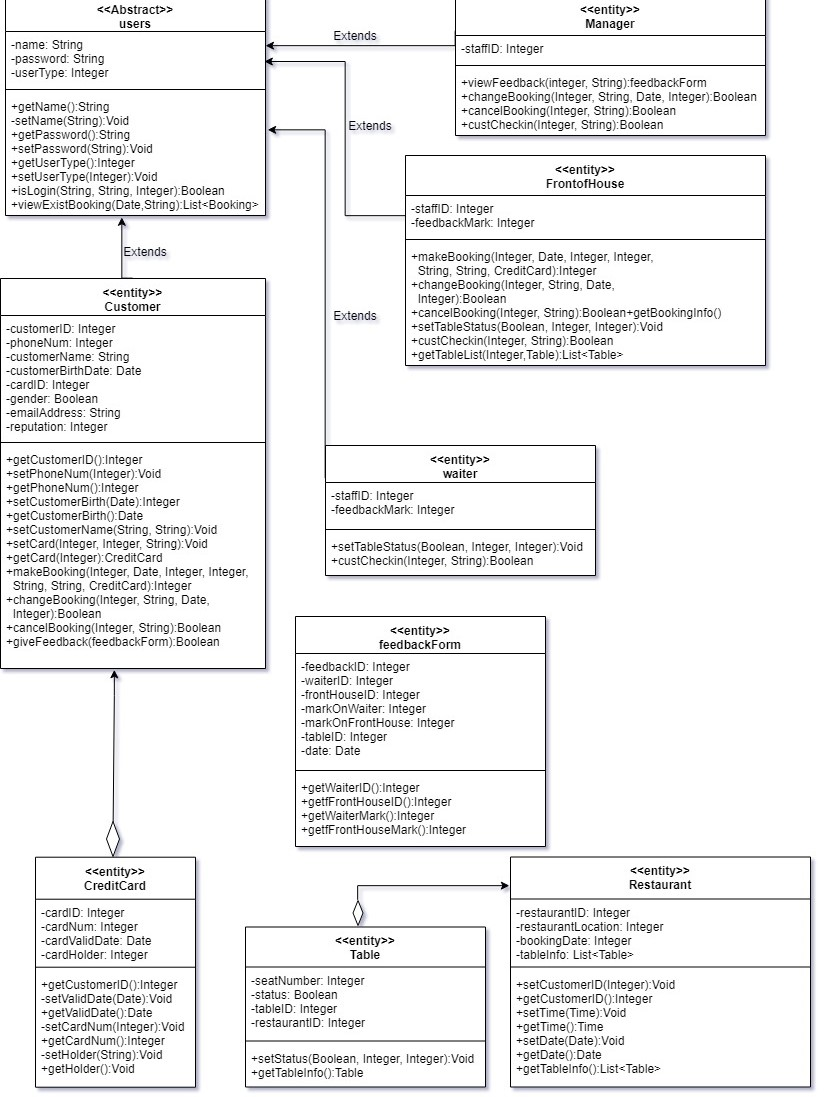
* Use Case 12: Customer Checkin
  + Author: Gavin
  + Packages:
    - View: SearchController
    - Controller: SearchController, CheckInController
  + Description:

After the booking ID is searched the system will display the discount and allergies associated with the booking.

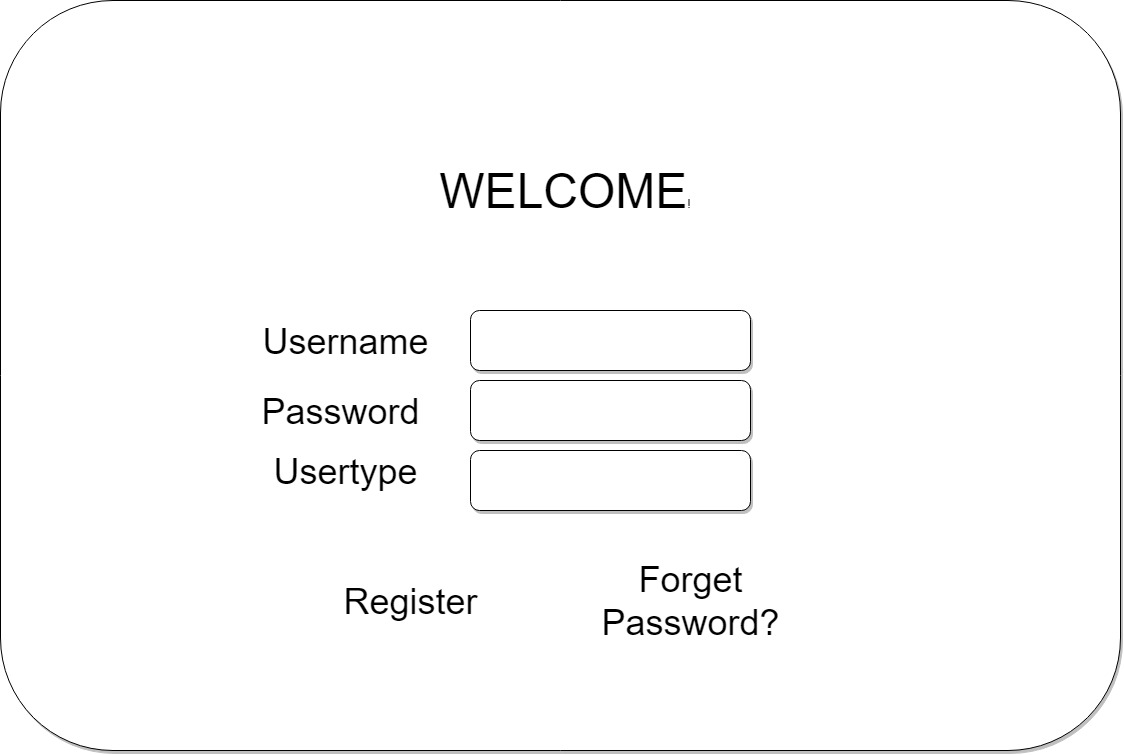
* Use Case 14: Reputation System
  + Description:

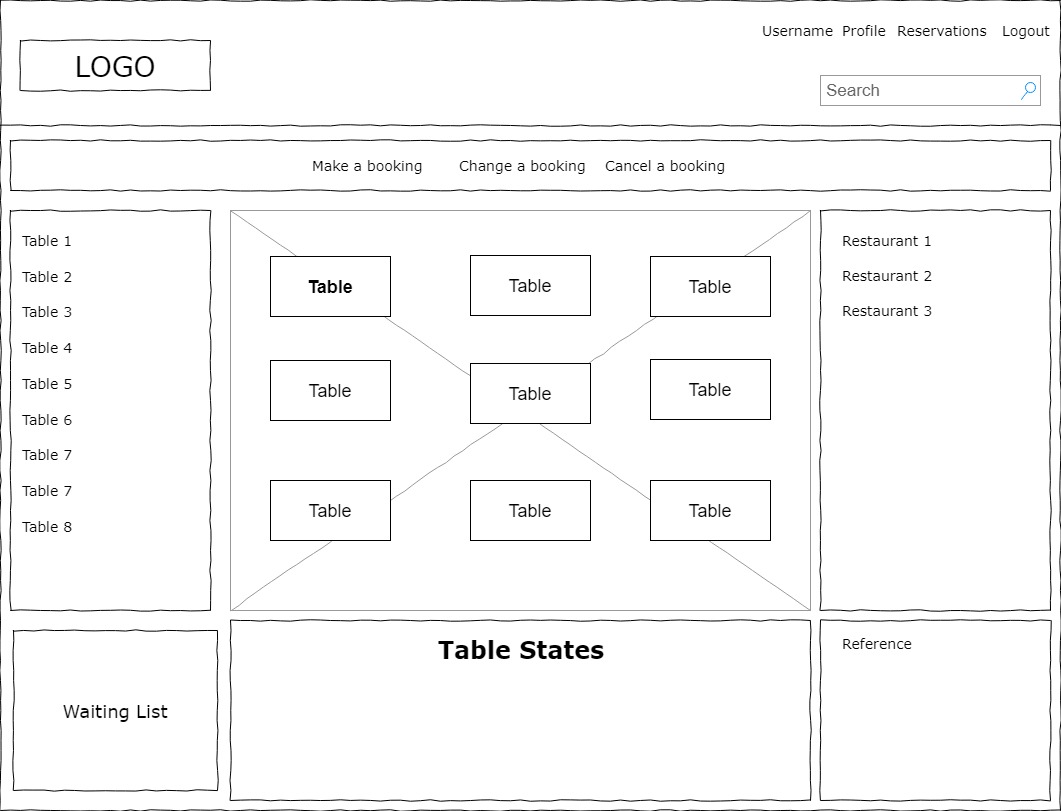
Renamed to Loyalty Points, this use case is involved in Use Cases: 8 and 10. When creating a booking the customer receives 100 points, upon Cancelling a booking the customer will lose 50 points.

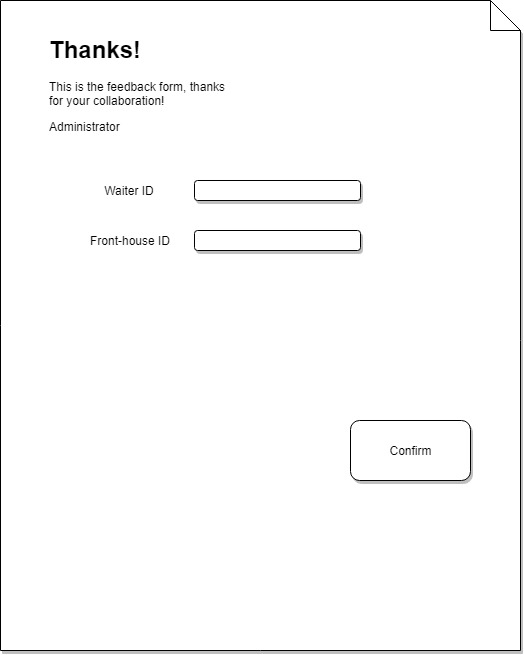
## Use Case Diagram

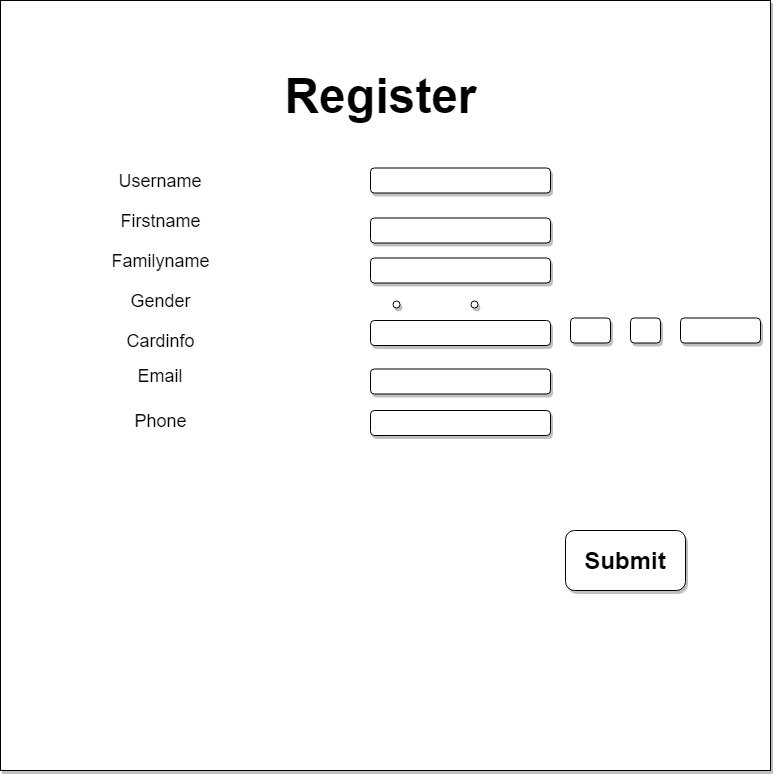


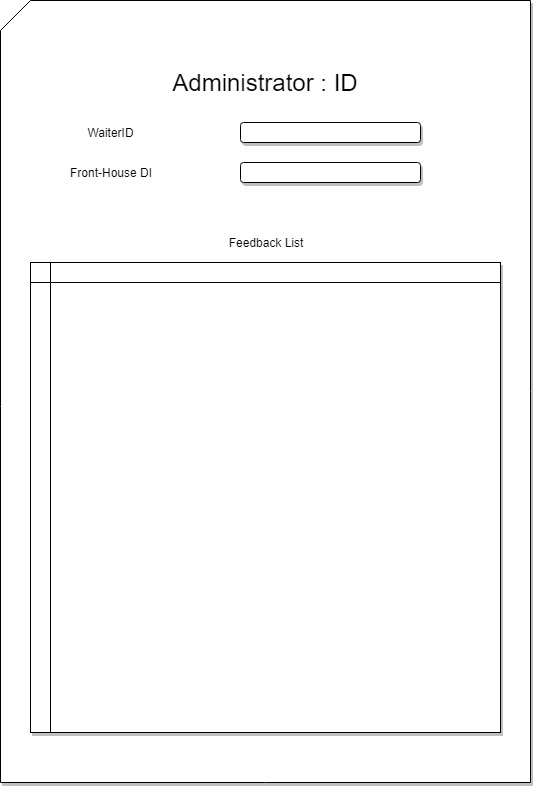
## GUI Prototypes

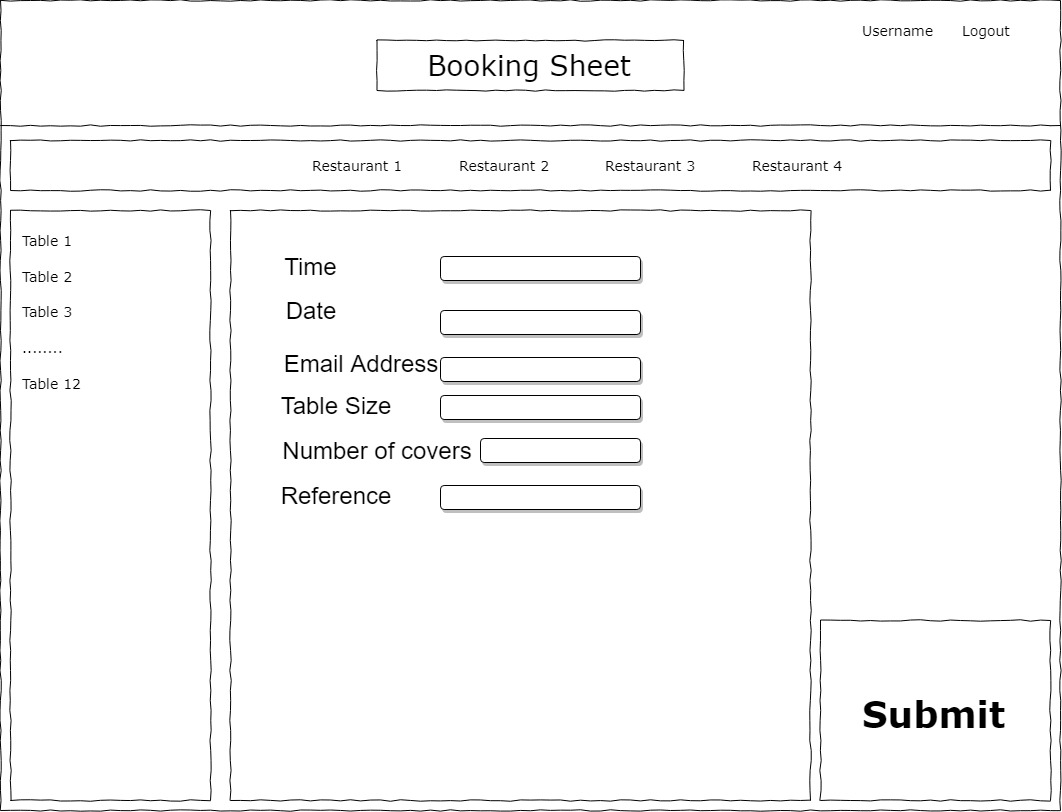






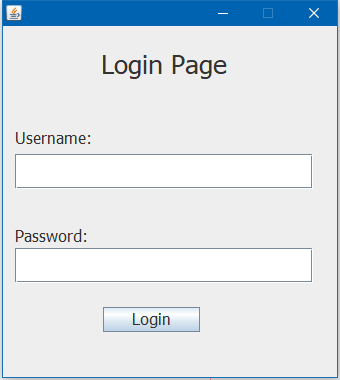




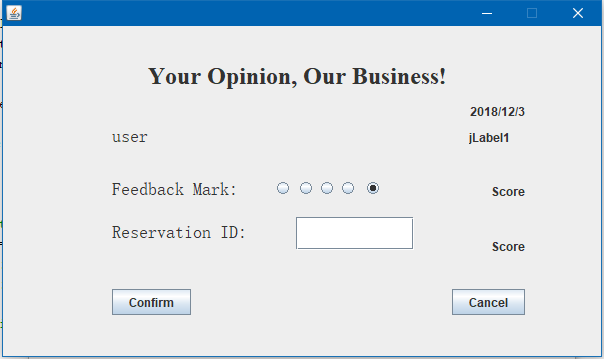


## GUI After Implementation

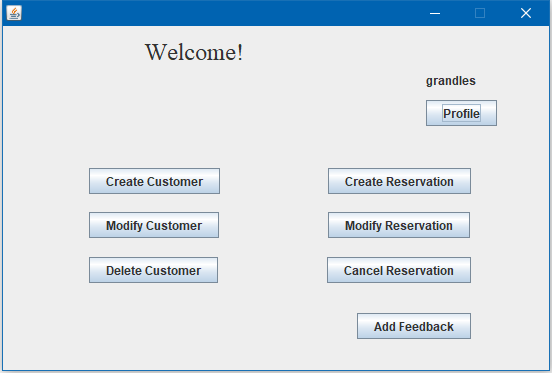
Login



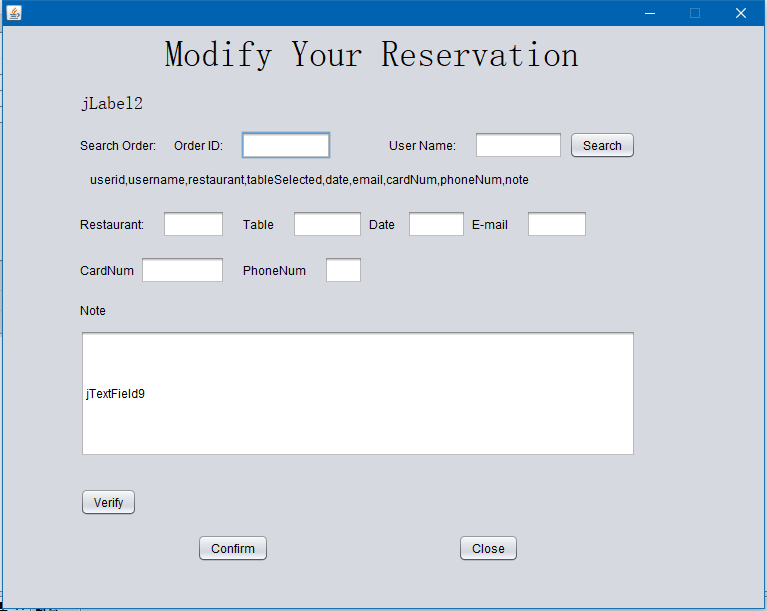
FeedBack



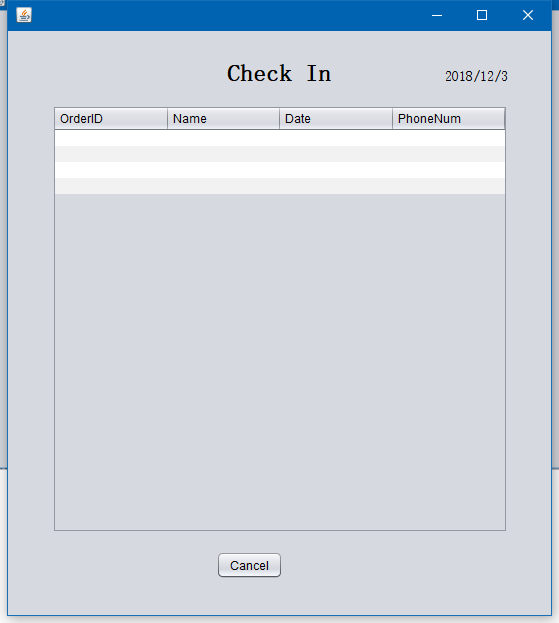
Mainmenu



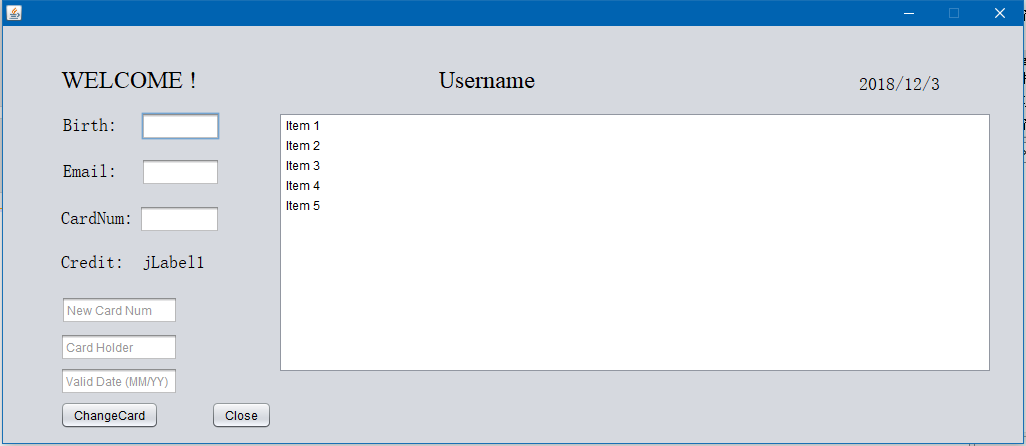
Modify Reservation



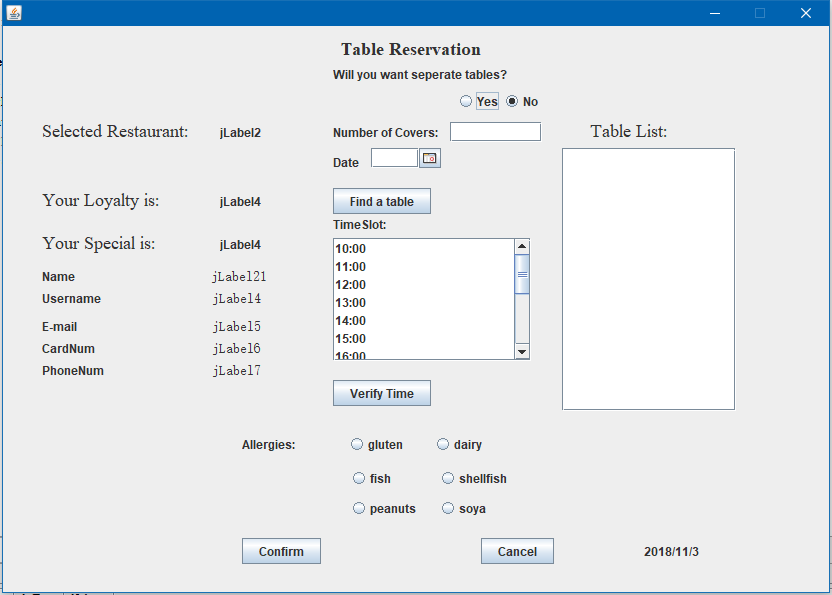
CheckIn



UserProfile

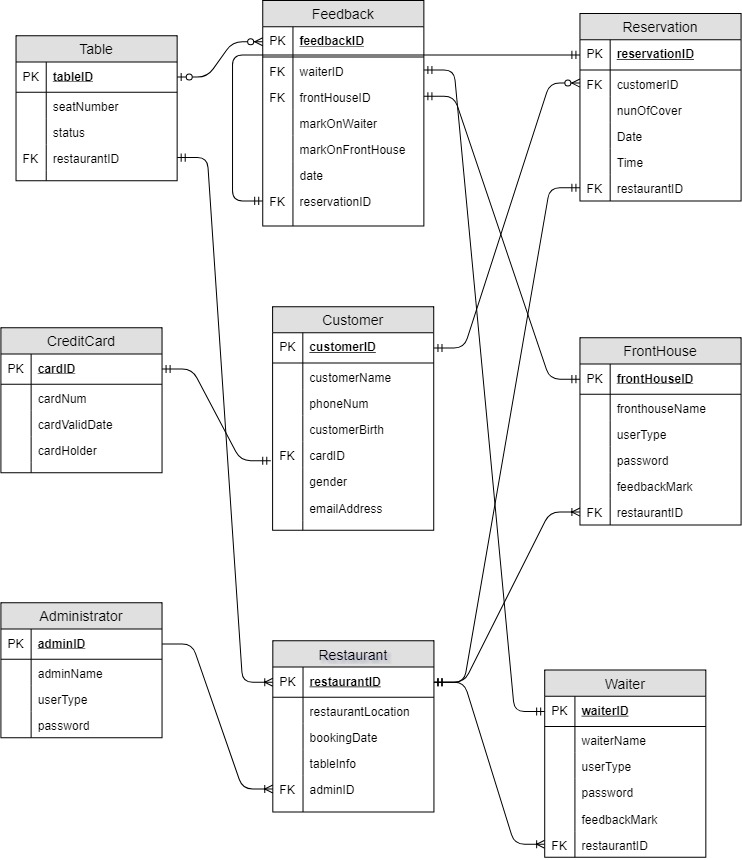


ReservationBooking



## E-R Diagram

### Pre-developement



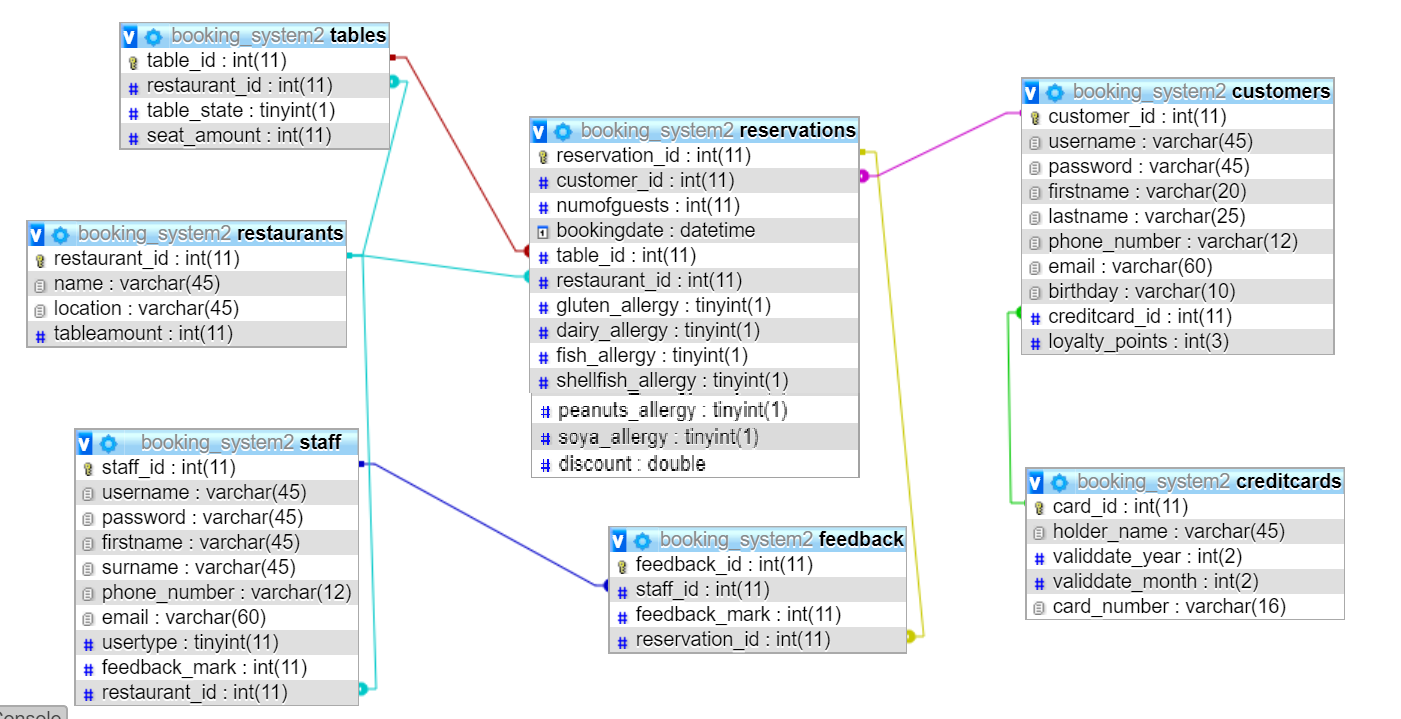
This image displays the first version of the database entity relations

The first version of the database was developed by Yiran upon initial conception of the project idea. It was later modified by Gavin and Patryk to refactor tables and add data to tables to accommodate business logic.

### Post-developement

Gavin combined the Administration, Waiter and FrontHouse tables as these all had userType and a member of either table could essentially have the same ID, this could get confusing later on. Gavin also add columns to Reservation to allow for allergies to be recorded and used to notify staff upon Check In. Loyalty Points was also add to allow for further business logic involved in earning or losing points due to cancelation or failure to check in on time.

Patryk added Discount to Reservations Table to implement the discount business logic.



This image displays the latest version of the database entity relations.

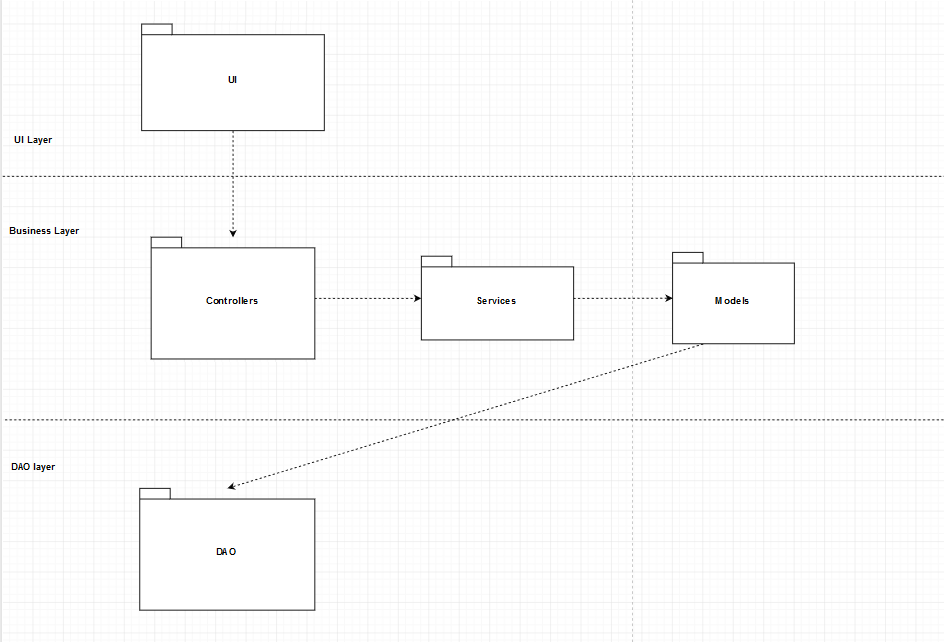
# System Architecture

## Candidate Classes

It is very important to identify nouns present in our project requirements in order to identify which of these will make suitable classes for our system.

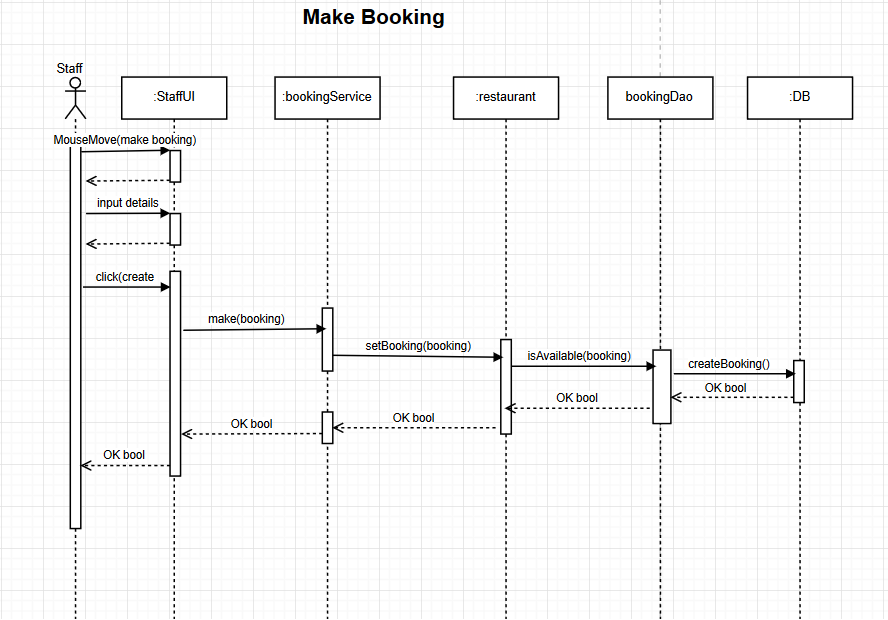
|  |  |
| --- | --- |
| Before | After |
| customer  staff  booking  ~~allergy~~ - to specific  cerditcard  feedback  resturant  table  discount  ~~Reservation~~ - and booking are the same  ~~Reputation~~ - to specific  Register  modify  cancel | customer  staff  booking  cerditcard  feedback  resturant  table  Register  modify  cancel  discount |

## Package diagram



This diagram shows a generic what a generic MVC might look like for our system.

## Make Booking sequence diagram



## Modify Booking sequence diagram

###### 

## Architectural discussions Taken

The system will be implemented using a Java SOAP server this will allow customer to remotely access the system through a web interface, this will allow all the functionality outline in the requirements. The use of the SOAP server will also allow for the desktop application to be deployed in many different restaurants while still allowing access to the same databases. The system will use a number of design patterns to achieve this, these include the Abstract Factory pattern which will be used to make different type of user e.g. a user super type which both staff and customer can inherit from. We will also implement the observer design pattern when calculating customer loyalty points and then using these points to calculate the discounts or fines for cancellations or “no shows”. The final design pattern we will rely heavily on is MVC as this will allow the system to seamlessly pass data around the system as it’s needed to either create users, create booking or modify any of these.

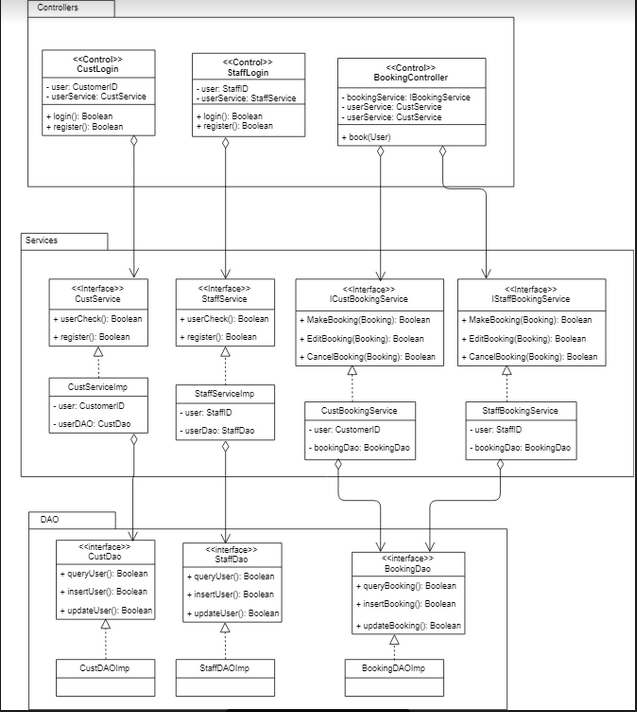
Draw.io

Will be our go to for creating use case diagrams, sequence diagrams, package diagrams, E-R diagrams or any other diagrams we will need during implementation.

easyUML tool

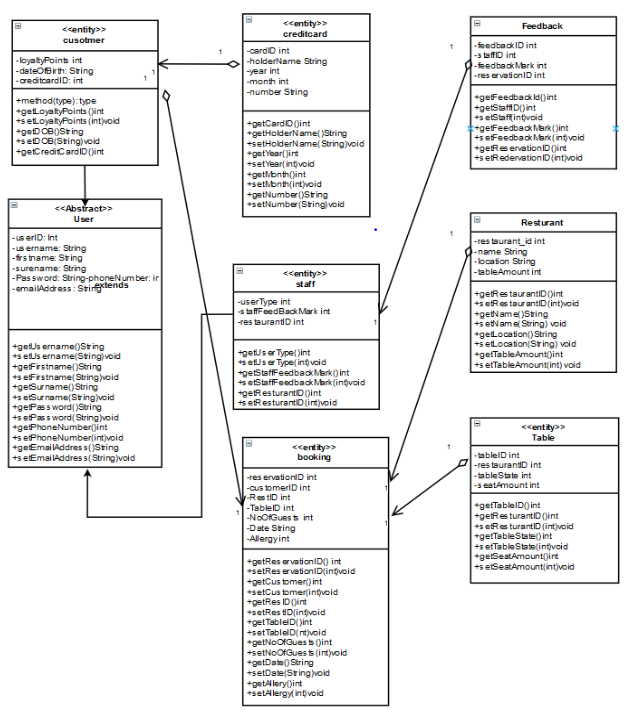
Will be used after implementation to gather package diagrams, architecture diagrams, design-time class diagram from NetBean.

## Class Analysis Pre-developement



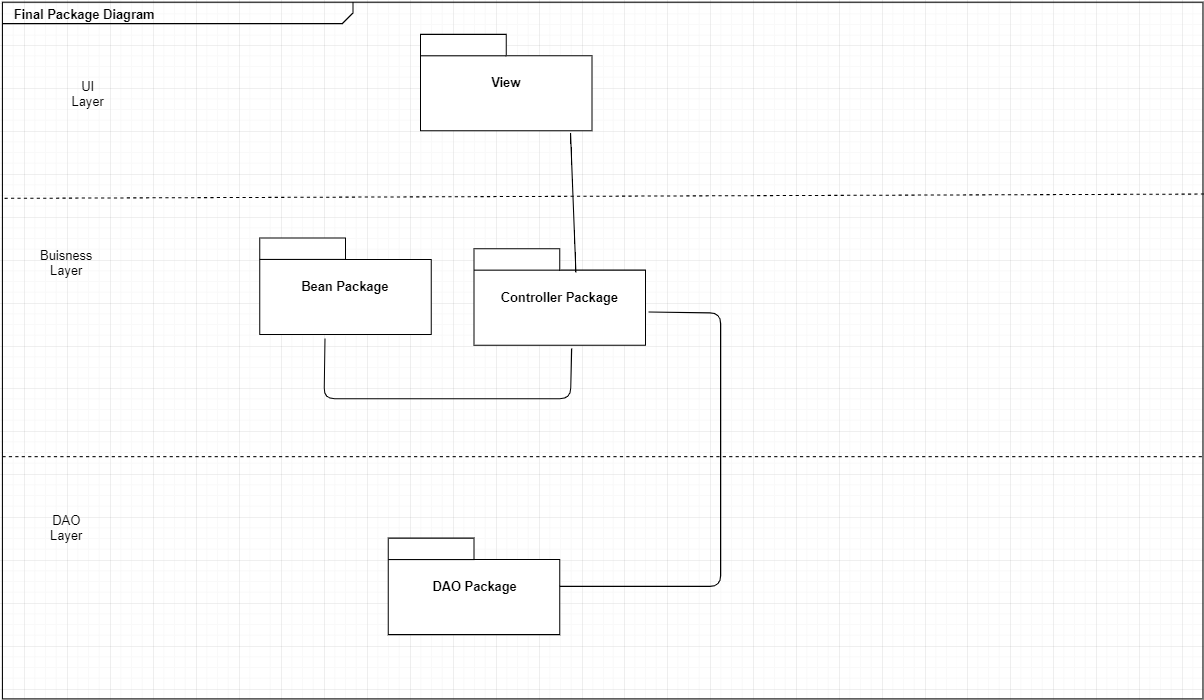
See Bean package below

Bean



# Code

## Packages



Package: DAO

|  |  |  |
| --- | --- | --- |
| **Class** | **Author** | **LinesOfCode** |
| BookingDAO | MJ | 186 |
| CreditCardDAO | Gavin | 103 |
| CustomerDAO | Sean | 193 |
| FeedbackDAO | Sean | 74 |
| RestaurantDAO | MJ | 75 |
| StaffDAO | Sean | 158 |
| TableDAO | Gavin & Patryk | 109 |

Package: DAO

|  |  |  |
| --- | --- | --- |
| Deals | Patryk | 45 |
| Discount | Patryk | 25 |
| SimpleDealsFactory | Patryk | 17 |
| LaCucinaDeals | Patryk | 28 |
| MilanoDeals | Patryk | 28 |
| LaCucinaLoyalDiscount | Patryk | 14 |
| LaCucinaPremiumDiscount | Patryk | 14 |
| MilanoLoyalDiscount | Patryk | 14 |
| MilanoPremiumDiscount | Patryk | 14 |

Package: Bean

|  |  |  |
| --- | --- | --- |
| BookingBean | MJ | 131 |
| CreditCardBean | Gavin | 63 |
| CustomerBean | Sean | 46 |
| FeedbackBean | Gavin | 52 |
| RestaurantBean | Gavin | 54 |
| StaffBean | Sean | 45 |
| TableBean | Gavin | 54 |
| UserBean | Sean | 84 |

Package: Controller

|  |  |  |
| --- | --- | --- |
| BookingController | Patryk & MJ |  |
| CheckInController | Gavin | 51 |
| CreateBookingController | Patryk | 52 |
| CreditCardController | Sean | 51 |
| CustomerRegisterController | Sean | 111 |
| deleteBookingController | MJ | 43 |
| feedbackController | MJ | 39 |
| LoginController | Gavin | 40 |
| MainMenuController | Patryk & Gavin | 96 |
| ModifyBookingController | Patryk | 89 |
| SearchController | Gavin | 118 |
| StaffProfileController | MJ | 20 |
| StaffRegisterController | Sean | 99 |
| DeleteCustomerController | Sean | 19 |

Package: Util

|  |  |  |
| --- | --- | --- |
| DBConnection | Gavin | 25 |

Package: View

|  |  |  |
| --- | --- | --- |
| CancelReservation | Yiran | 238 |
| CheckInView | Yiran | 154 |
| creditCardView | Sean & Yiran | 106 |
| CustomerRegisterView | Sean | 144 |
| FeedBackView | Yiran | 294 |
| LoginView | Gavin | 85 |
| MainMenuView | Yiran & Gavin | 380 |
| ModifyReservaitonView | Yiran | 382 |
| ReservationBookingView | Yiran | 672 |
| SearchView | Gavin | 99 |
| StaffProfileView | MJ | 139 |
| UserProfileView | Yiran | 330 |
| CreateRegisterStaffView | MJ | 152 |

|  |  |  |
| --- | --- | --- |
| LoginControllerTest | Yiran | 52 |
| FeedBackTest | Yiran | 56 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Lines of Code | Gavin | MJ | Sean | Patryk | Yiran |
| Approx Total | 1200 | 950 | 980 | 760 | 2600 |

Total lines of code in project: 6000+

BookingDAO: Able to write, receive, update and delete bookings from the database.

CreditCardDAO: Adds credit card to database, can receive credit card information from the database.

CustomerDAO: Adds customer to database, gets customer information from database, update customer information in database, authenticate user details, verification, update loyalty points in database, delete customer from database.

FeedBackDAO: sends users feedback to database and can get user feedback information to display in staff profile.

RestaurantDAO: adds restaurant id, name, location, table-amount to the database and can receive this information to display also.

StaffDAO: Authenticates staff user details, can add a new staff member to the database, update their details, and retrieve their details to display in a view.

TableDAO: adds a restaurant id, table state and amount of seats to the TableDAO table in database, get all free tables from database.

The discount is applied using an abstract factory method patern.

SimpleDealsFactory: a simple factory class which decides what type to assign to the deals abstraction.

Deals: an abstract class factory class. Provides an abstraction to apply deals for various restaurant chains.

Discount: an abstract class which is an abstraction of the so called “product” in factory method pattern.

LaCucinaDeals: a sub-class of Deals. Initilises the dealDay and dayRate variables. Overrides the getDiscount() method to factory method which decides which sub-class of Discount for the restaurant La Cucina to return.

MilanoDeals: a sub-class of Deals. Initilises the dealDay and dayRate variables. Overrides the getDiscount() method to factory method which decides which sub-class of Discount for the restaurant Milano to return.

Subclasses of Discount (LaCucinaLoyalDiscount, LaCucinaPremiumDiscount, MilanoLoyalDiscount and MilanoPremiumDiscount). They override the getRate() method by assigning their unique discount rates.

The Java Beans are used to encapsulate many objects into a single object(ie. the bean) so that they can be passed around as a single bean object instead of as multiple individual objects making for neater, cleaner code. They are used as a temporary state prevention having getters and setters providing additional functionality. The beans that we have created and implemented are: BookingBean, CreditCardBean, CustomerBean, Discount, FeedbackBean, loginBean, LoyalDiscount, PremiumDiscount, RestaurantBean, StaffBean, TableBean, UserBean.

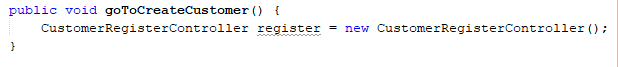
The Controllers act on both the beans and the views. Each controller controls the data flow into bean objects and updates the view whenever data changes. Each view collects and passes information to the controller. The controller then creates a bean and passes it to the DAO. The controllers we have created are: booking controller and createBookingController that extends bookingController. CreditCardController, CustomerRegisterController, deleteBooking, feedbackController, loginController, MainMenuController, ModifyBookingController, SearchController, StaffBookingController, StaffProfileController, StaffRegisterController, deleteCustomerController.

Each view takes information from the end user and passes the information to the controller. The views we have created for our Restaurant Booking System were cancelReservation, checkInView, creditCardView, CustomerRegisterView, FeedbackView, LoginView, MainMenuView, ModifyReservationView, ReservationBookingView, SearchView, StaffProfileView, UserProfileView and a createRegisterStaffView.

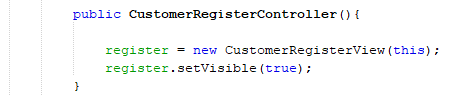
## MVC

An example of how we used MVC is shown below.

The MainMenuController talks to every other controller. In this example we will show how a customer is created. From the MainMenuController a CustomerRegisterController instance is created.



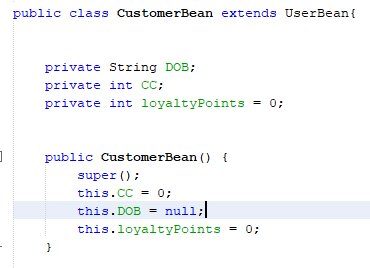
From the CustomerRegisterController an instance of the CustomerRegisterView is created. The view collects information from the user and passes it back to the controller where validation takes place and then the CustomerRegisterBean is created.



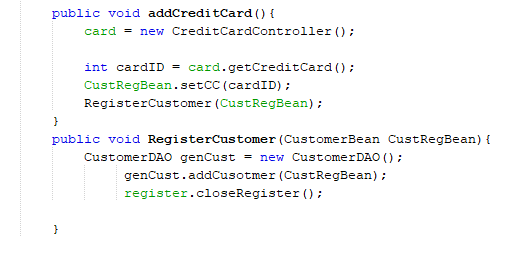
Here the CustomerBean is created.



The CustomerBean encapsulates many objects into a single object so that they can be passed around as a single bean object instead of as multiple individual objects making for neater, cleaner code. Here we can see CustomerBean extends UserBean this is an example of inheritance, more specifically polymorphism as Customer takes on the form of a user adding 3 more data fields.



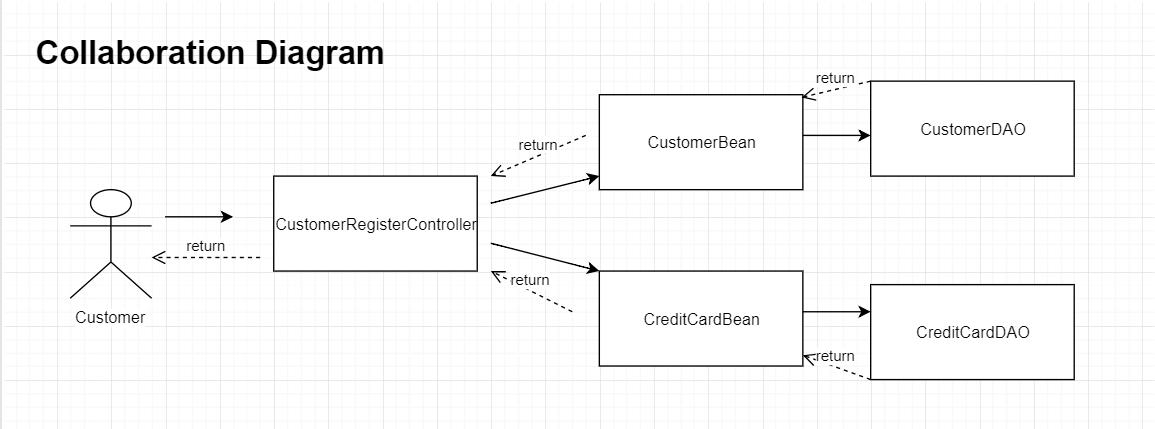
CreateCustomerController has methods where you can add a credit card and register the customer, when the register customer method is called the customer information gets passed to the DAO.



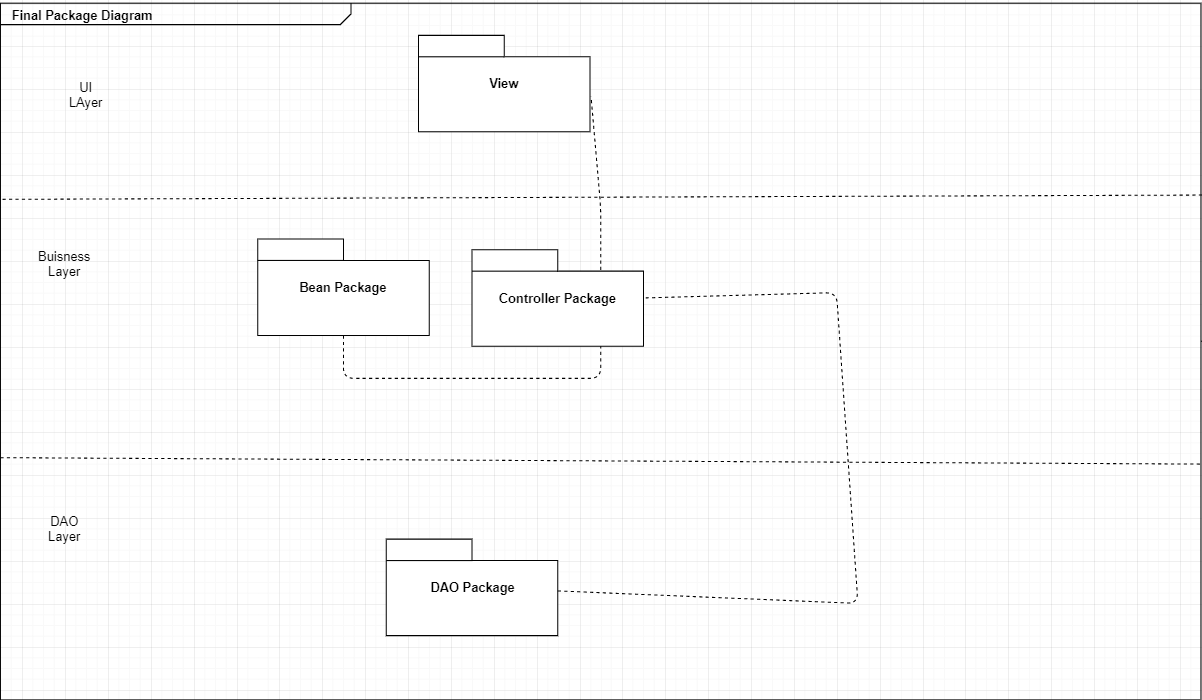
In the customerDAO the customers information is sent to the database.



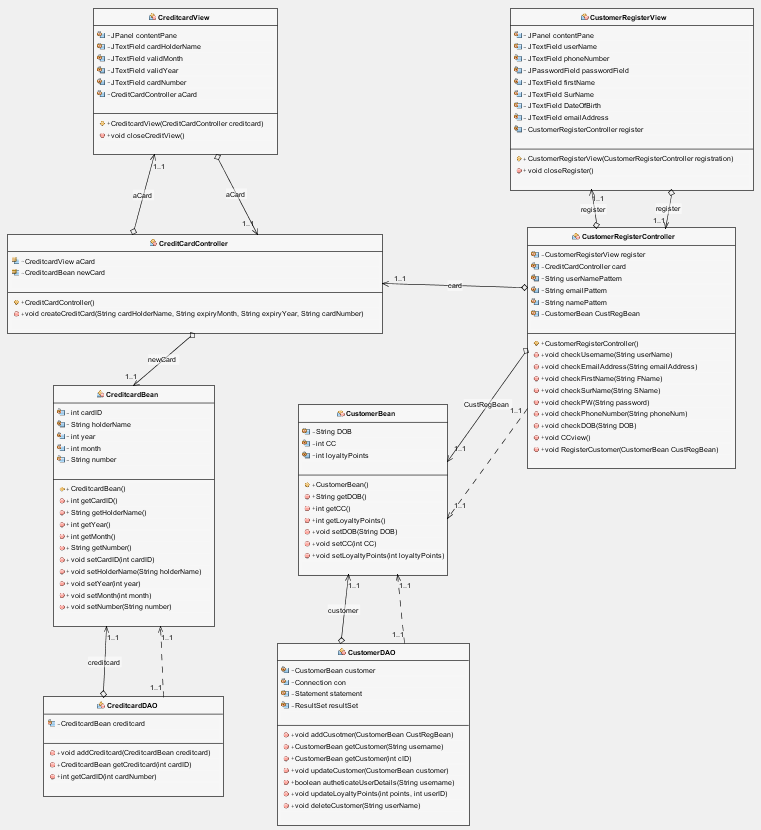
## Collaboration Diagram



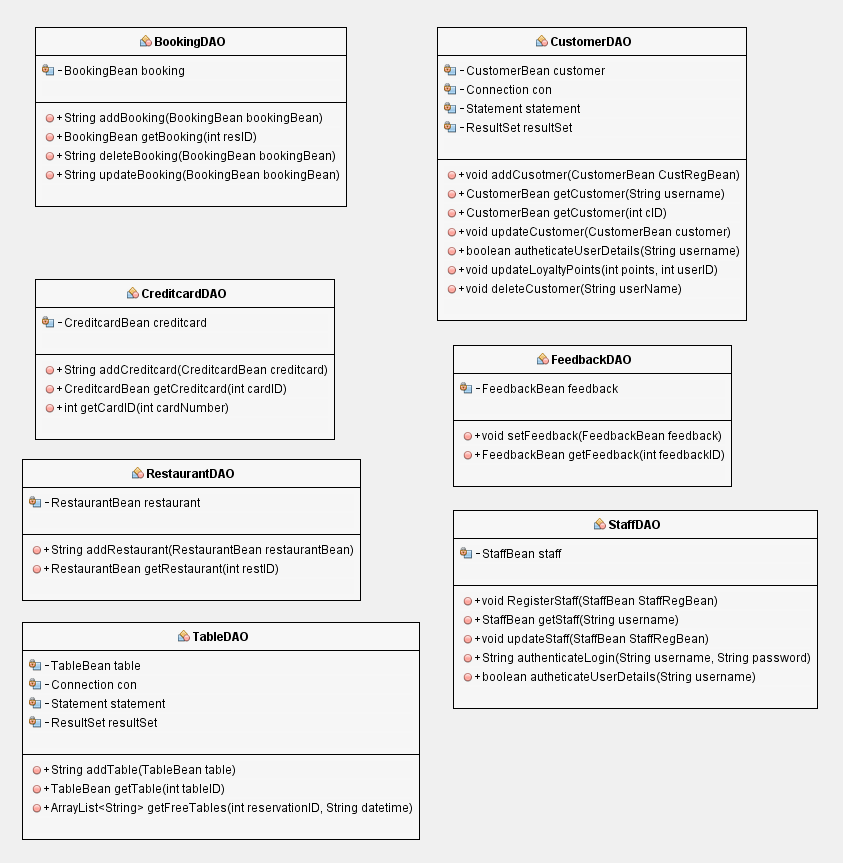
### Final Package Diagram



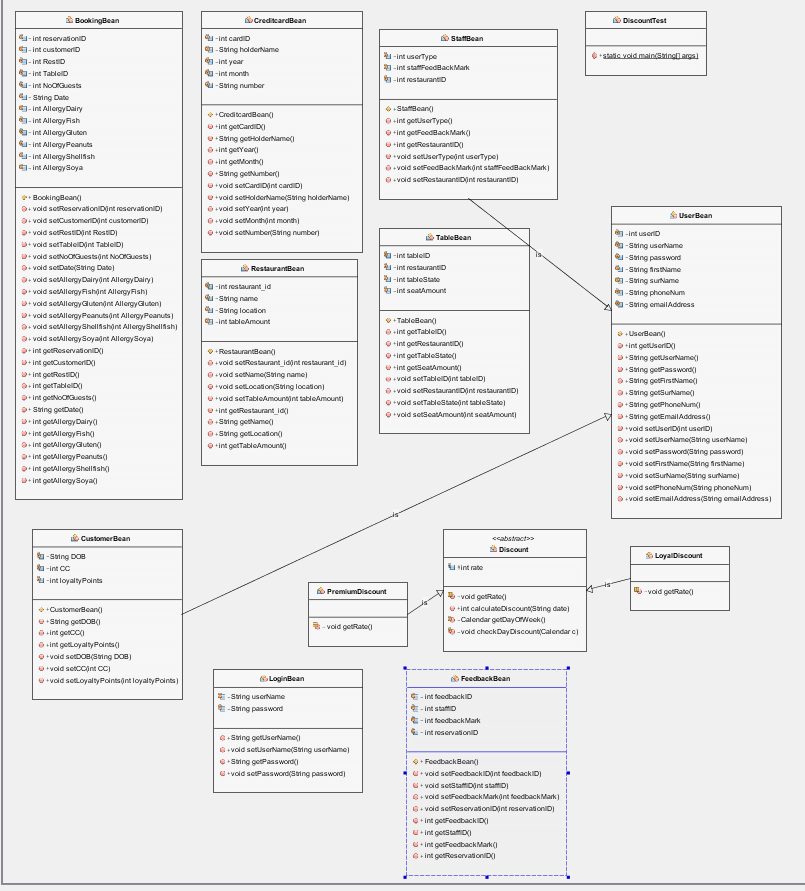
### Register Customer class diagram



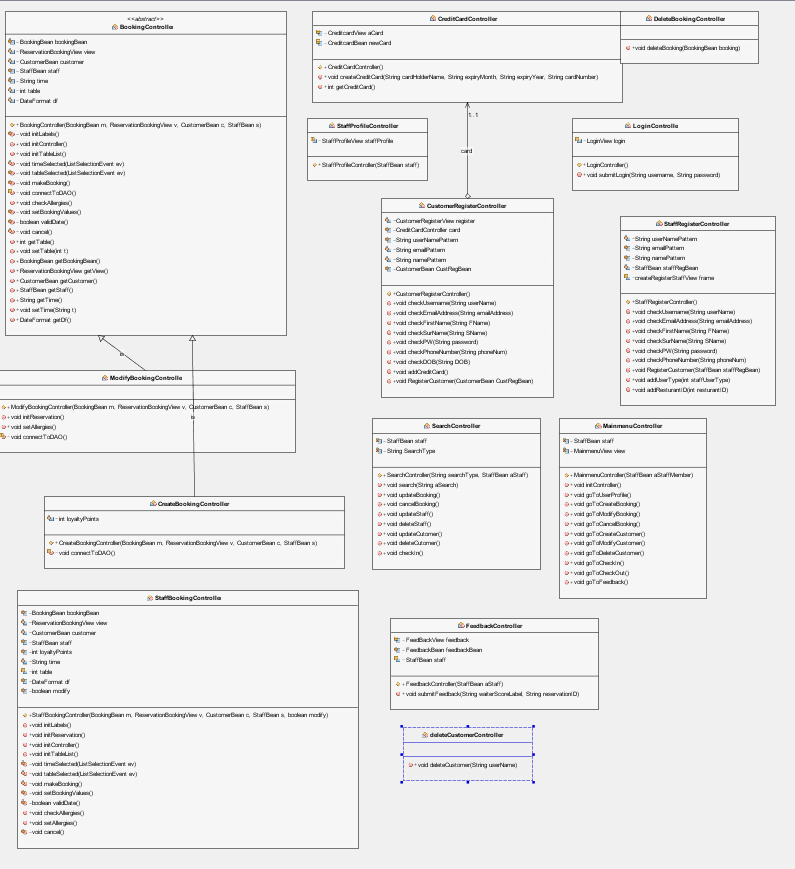
### DAO



### Model/Bean

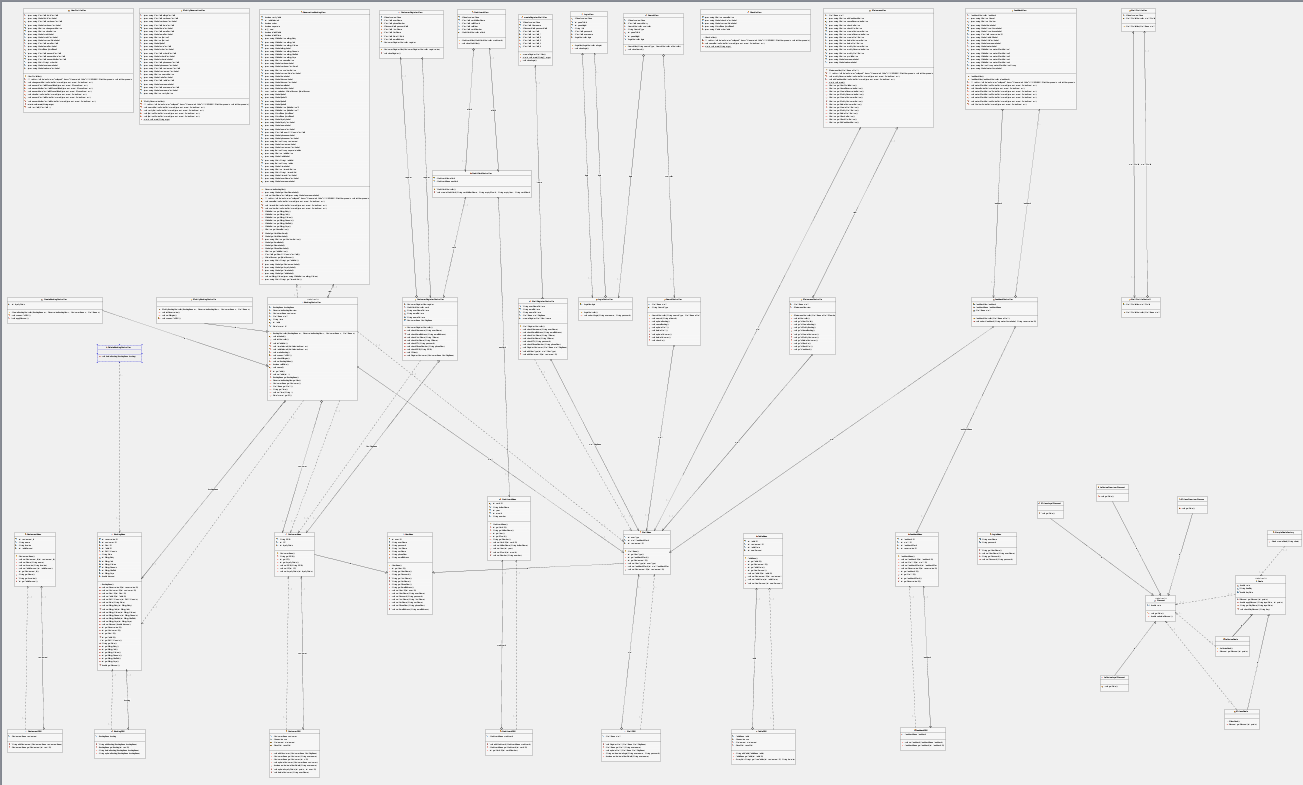


### Controller

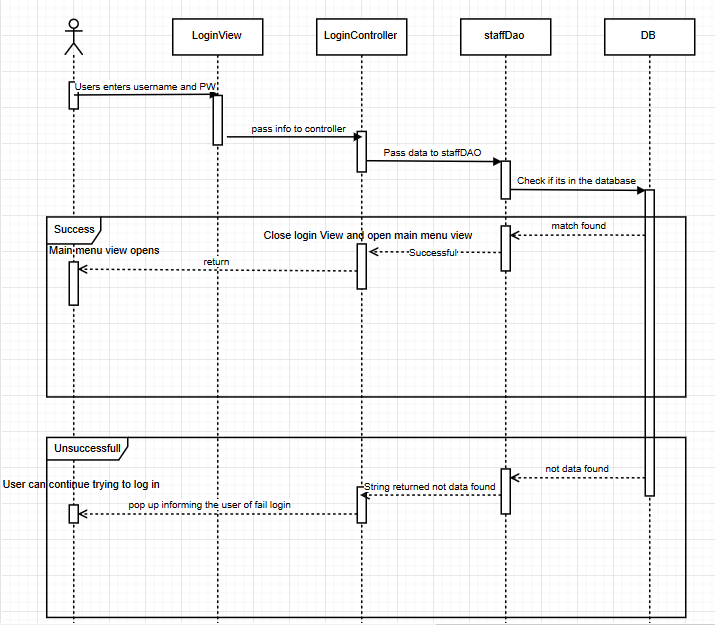


### View

#### Full system uml



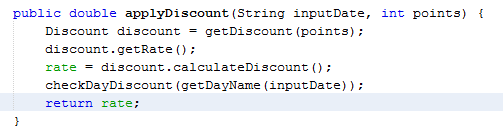
#### Final login squence diagram



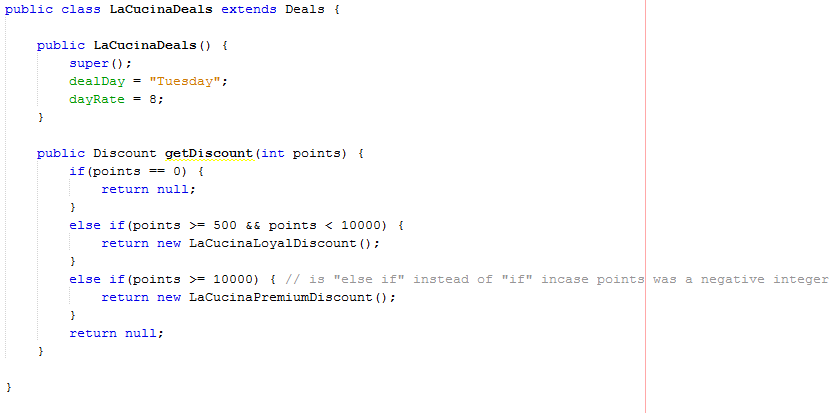
# Design Pattern

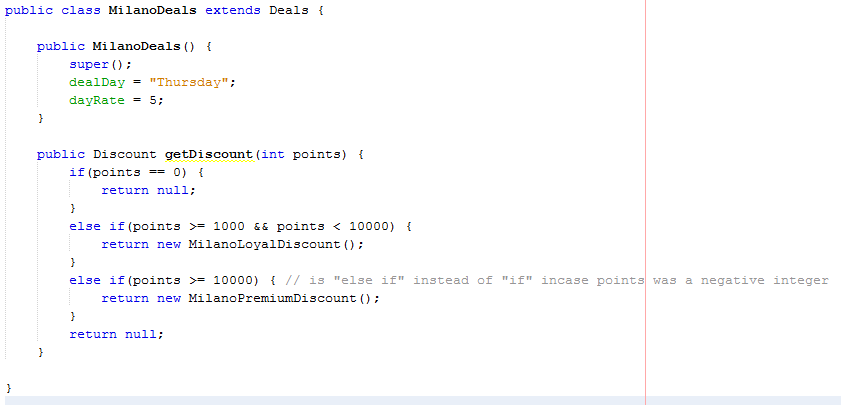
I decided to use the abstract factory pattern for the implementation of discount rates for reservations. The discount rates vary for different Restaurant chains and depend on factors such as day of the week and the number of loyalty points a customer has. The use of an abstract factory pattern provides easier modification and extensibility of the code.

Deals is an abstract class that provides the checkDayDiscount(), which checks whether the day of the reservation matches the discount day for the particular restaurant chain. The applyDiscount() method declares a Discount object and uses the abstract method getDiscount() to return a sub-class of class Discount depending on the restaurant chain. It calculates the discount rate and returns it.

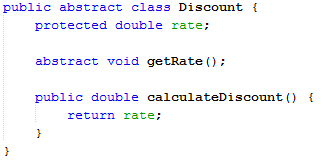


MilanoDeals and LaCucina deals are sub-classes of Deals. They provide different values for dealDay and dayRate. The getDiscount() method is overridden and it is used as a factory method for returning a different sub-class of Discount depending on the restaurant chain and amount of loyalty points.

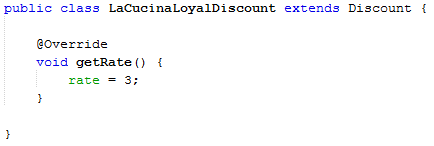


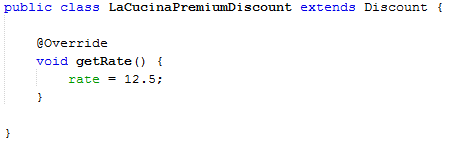


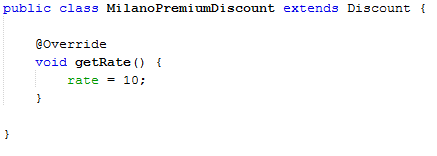
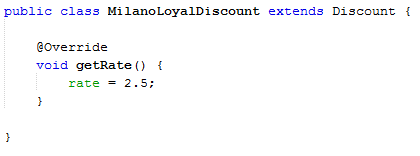
Discount is an abstract class. CalculateDiscount returns the rate of the discount. The getRate() method is abstract.



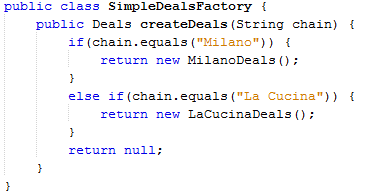
Sub-classes of Discount (LaCucinaLoyalDiscount, LaCucinaPremiumDiscount, MilanoLoyalDiscount and MilanoPremiumDiscount) override the getRate() method by assigning rate with a hard coded value.



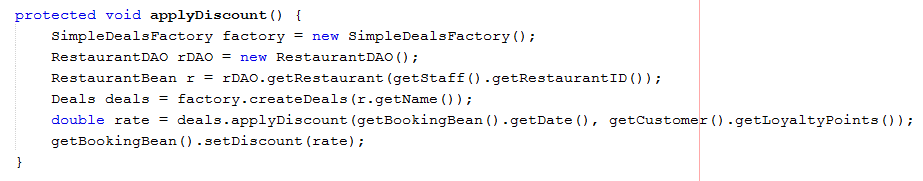




SimpleDealsFactory uses the simple factory method. A simple factory method is not a design pattern but provides cleaner and more organised code in my opinion. It returns different sub-classes of Deals depending on the String argument being passed which is the restaurant chain name.



Here is the code snippet showing the applyDiscount() method inside the CreateBookingController. The SimpleDealsFactory is called, it returns a sub-class of Deals which varies for different restaurant chains.



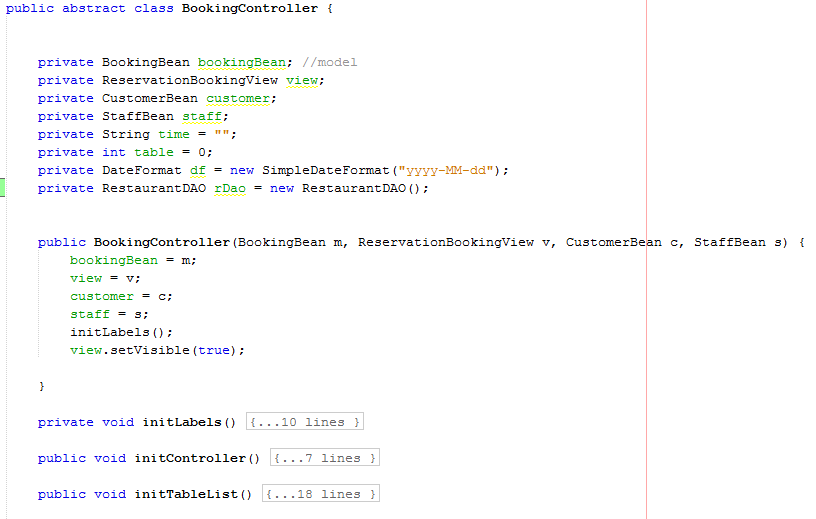
This design pattern provides easy modification of discount rates for different restaurant chains but also provides extensibility. A new restaurant chain or a new type of discount can be easily added without changing much of the code.

## Refactoring

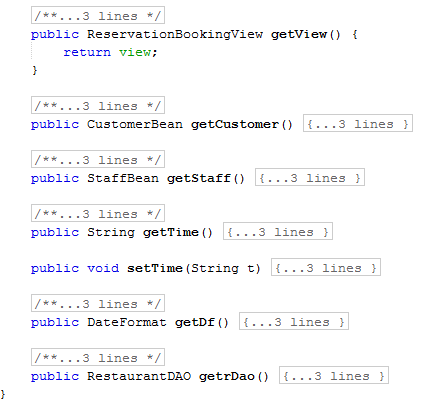
While implementing the CreateBookingController and knowing that ModifyBookingController will be my next step, I realised that implementing the ModifyBookingController will be highly time consuming considering the size and time it took to implement CreateBookingController and ReservationBookingView. Yiran was working on the ReservationBookingView.

I thought that it would be a good idea to reuse the controller and the view for modifying a booking. The view should be the same for both, creating and modifying a booking, with the only difference that for creating a booking the view should have empty input fields and for modifying a booking these input fields should be set. The user is free to edit these fields. When confirm is pressed the data is sent to the database using a BookingDAO. No matter if the fields will be edited or not, the data can be sent to the database. At this stage I knew that the ReservationBookingView can be reused for both create and modify. The controller could be mostly reused excluding the initialization of the input fields in view when modifying a booking and a different method being called from the BookingDAO to push the data to the database. For a booking creation we required a “CREATE Table...” SQL query and for booking modification we needed a “UPDATE Table…” query. So we needed separate methods for creating and modifying a view in the BookingDAO. However the code in the controller would be different for create and modify because a different BookingDAO method would need to be called.

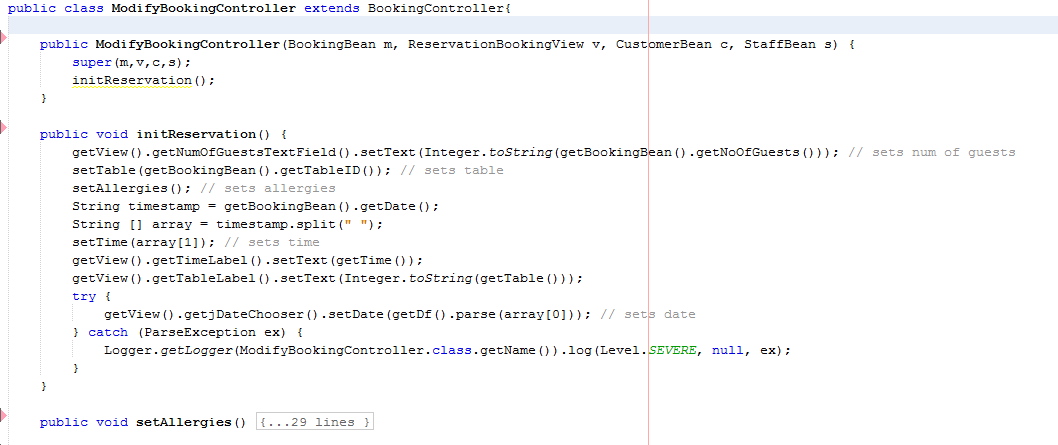
I considered using polymorphism but I realised that the connectToDAO() method should be called in the base class but implemented in the sub-class. So I decided to make an abstract class BookingController from which CreateBookingController and ModifyBookingController inherit. They both need to override the connectToDAO() method and call the appropriate method from BookingDAO().

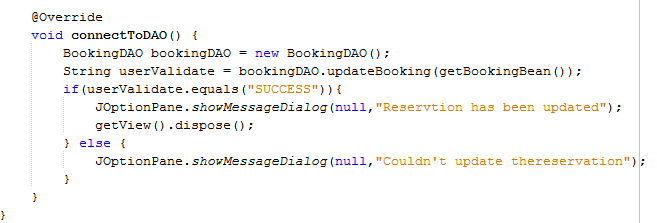










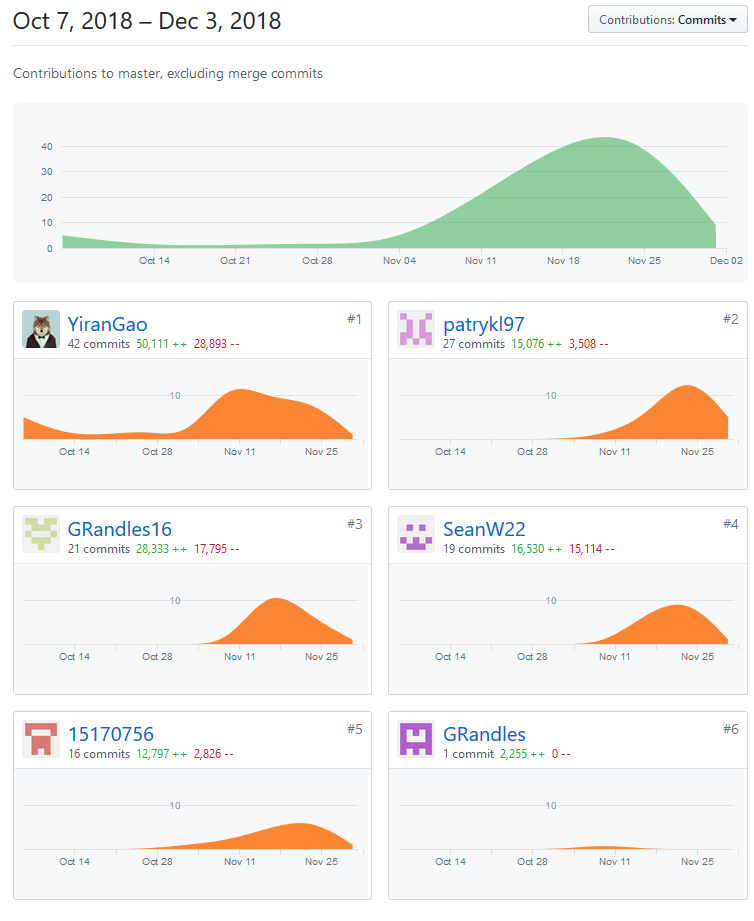


# Added Value

## Git & Github

Git is a distributed revision control system which allows the user to track changes to any file that is text, revert to any previous version easily, visualize changes between versions, and a variety of other useful things (creating branches of a project, asynchronous collaboration, etc.). GitHub is an enormously popular web-service that allows the user to host Git repositories publicly. There are a number of excellent resources online with Git and GitHub. GitHub also has graphical applications available for Mac and Windows machines, as well as integration with Eclipse, Netbeans, Emacs, and most other text editors with an active community. In addition to the aforementioned official Git GUIs, there are a number of 3rd party applications that make using Git quite easy. (Jones 2013)

A complete research project hosted on GitHub is reproducible and transparent by default in a more comprehensive manner than a typical journal mandated replication archive. (Jones 2013) With a typical journal replication archive, the final data and code to run the final set of models is provided. This leaves to the imagination most of the details of the data collection and/or data manipulation that produced the final data set, what model specifications preceded the ones present in the final script, and how the manuscript changed during its journey from idea to publication.



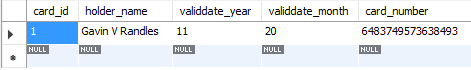
## DATABASE - MySQL

### Tables:

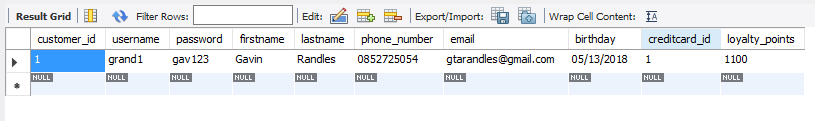


Details:

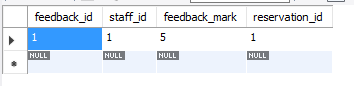
creditcards\_table



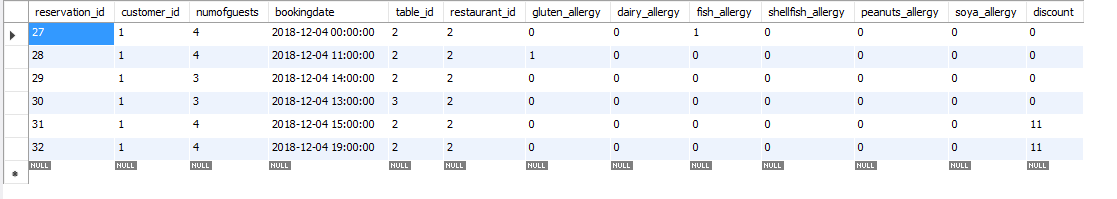
customers\_table



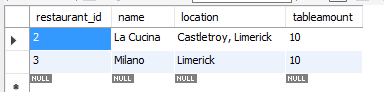
feedback\_table



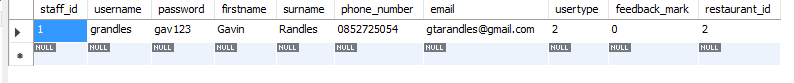
reservations\_table



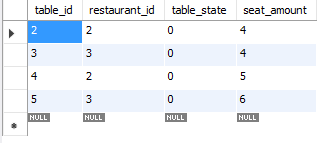
restaurants\_table



staff\_table



tables\_table



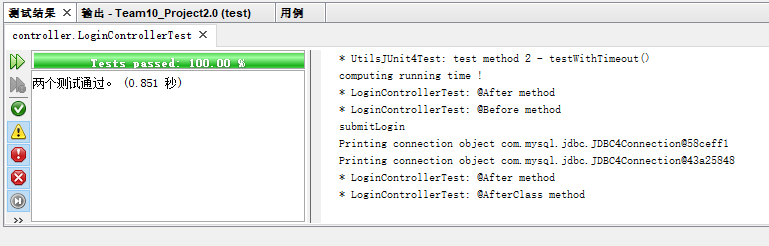
## JUnit

JUnit is a testing framework that developers use for writing test cases while developing the software. They would write and run test cases for every function they write. So using this, it will make sure that every single tits and bits in your software is tested even before module or System level testing is performed.(Gorav 2012)

Every time programmer make a small or a big modification in the code (in any function), we can make sure that the function is performing well and has not broken any older functionality by executing all JUnit test cases in one go written for that function. (Gorav 2012) So, we write a test case once, and go on using the test case again and again to make sure that- everytime software is modified it is working as expected. Using JUnit, our team can easily create and most importantly manage a rich unit test case suite for the entire software.

JUnit Test Example:

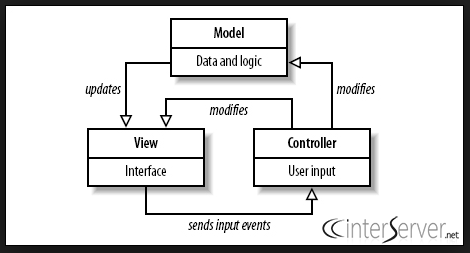
LoginControllerTest.java



# Critique

Taking a look at the analysis sketches in part 8 of the report and comparing these sketches and diagrams to the final blueprints of our application, the final version is much more in-depth with much more functions than we originally planned out. For starters the controllers, there was originally three controllers. CustLogin, StaffLogin and BookingController. However with more research and knowledge on MVC we quickly realised the importance of controllers and how the views and models talk to controllers. In the final version we implemented much more controllers for every function we wished the booking system would have.

Controllers processes all the business logic and incoming requests, manipulate data using the Model component, and interact with the Views to render the final output. It receives input and initiates a response by making calls on model objects.



Comparing our original services to the beans we created, the final version has extra classes where we tried to implement more business logic adding loyal and premium discounts to customers who visit the restaurant on a regular basis. Each bean also has much more parameters. We updated our database along the way to more appropriate tables collecting just the right information off our customers to allow for the best experience of the application.

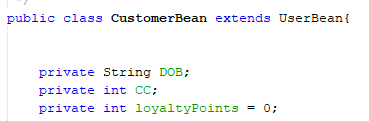
Our DAO packages also changed from their basic three DAO design of custDAO, staffDAO and BookingDAO to a more complex yet improved DAO package containing seven DAO files to write and receive information to and from the database.

**Language Selection**

We decided to implement our reservation system project in JAVA, for multiple reasons. Being as this is a restaurant booking system we wanted this system to be flexible and allow it to be installed on any machine with ease, JVM allows for this very easily. Java is also based around object oriented programming which was ideal for this project. As a team we agreed we would code our project in NetBeans due to its compatibility with glassFish server, as we were originally hoping to make a java based application for restaurants and an online web application written in JSP and servlet using a SOAP server however we under anticipated how much time the project would take to implement and ran into multiple problems along the way that set us back each time. Other reasons we chose netbeans was because of its powerful GUI builder, its excellent debugging tools and helpful suggestions and it was free and open source.

**Design Patterns**

A design pattern is a general repeatable solution to a commonly occurring problem in software design. At the beginning we struggled to understand design patterns and how to implement them. However with more research we were able to implement multiple design patterns to a satisfactory standard. Implementing design patterns in our project allowed for reusable code in multiple projects, provides solutions to help define system architecture. It makes the system more robust and highly maintainable. We implemented MVC design pattern making our code much more readable and easier to manipulate and update in future versions. We also implemented Factory design patterns. Factory design pattern is used when we have a super class with multiple sub-classes and based on input, we need to return one of the sub-class. This pattern take out the responsibility of instantiation of a class from client program to the factory class. The picture below is an example of where we implemented factory design patterns.



**Problems Encountered**

While implementing this project we encountered a number of problems, these included our struggles to work as a team at times this problem was caused by a lack of communication at and during meetings. We quickly addressed this as soon as it came to light and insured we met more regularly and stuck to the meeting agenda. This first problem had a knock on effect that we had some issue balancing the workload of some group members and spread out work evenly while ensuring that members were happy with the amount of work they received and could comfortably make progress with their work.

Another effect of this lack of meetings and communication was that we struggled with time and the amount of work left to-do, this was also partially caused by not fully understanding the task we had undertaken in terms of the best way to implement important design patterns to insure learning and proper enterprise development.

Even with the problems we encountered we believe that this project has taught us a lot in terms of understanding how to develop enterprise level systems, implementing design patterns and how to communicate better in future projects to insure these mistake are lessons and not habits.

# Reference

Diagramchartwiki.com (2014) *Systems Engineering V Diagram Verification And Validation* [image], available:

<https://diagramchartwiki.com/systems-engineering-v-diagram/systems-engineering-v-diagram-verification-and-validation-the-mitre-corporation/> [accessed 27 Oct 2018].

Jones, M.Z. (2013) ‘Git/GitHub, transparency, and legitimacy in quantitative research’, *The Political Methodologist,* available:

<http://zmjones.com/static/papers/git.pdf> [accessed 20 Nov 2018].

Gorav, A. (2012) ‘Benefits of using JUnit framework’, *gontu.org,* 24 Jul, available: <https://www.gontu.org/benefits-of-using-junit-framework/> [accessed 18 Nov 2018].

<https://www.interserver.net/tips/kb/mvc-advantages-disadvantages-mvc/>

# Marking Scheme

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **PENALTIES** | | | | | | |
|  | **Description** | **Yiran** | **Sean** | **Gavin** | **MJ** | **Patryk** |
| 1 | Late Submission |  |  |  |  |  |
| 2 | Failure to contribute to coding effort |  |  |  |  |  |
| 3 | Failure to contribute to writing of report |  |  |  |  |  |
| 4 | Failure to report problems with team dynamics |  |  |  |  |  |
| 5 | Failure to contribute to demo week 13 |  |  |  |  |  |
|  | **Sub-Total (B)** |  |  |  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **FINAL MARKS AWARDED** | | | | | | |
|  | **(A-B)** | **Yiran** | **Sean** | **Gavin** | **MJ** | **Patryk** |
|  |  |  |  |  |  |  |