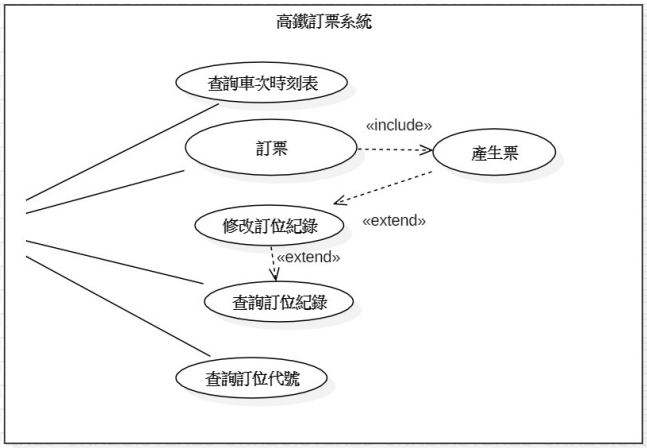
物件導向設計與模擬

Team 3: b04705003 林子雋, b04902007 楊舒瑄, b04902021 陳弘梵, b04902116 蘇景耀

Requirement

1. Use Case

我們依照使用者的操作行為，參考了高鐵售票系統，分為六種 Use Case，以下為其描述：



(1) 查詢車次時刻表:

輸入起訖站資訊

→ 顯示對應列車時刻表

(2) 產生票:

讀入列車資訊

→ 向資料庫、entity 索取對應列車座位 → 回傳成功訂單

(3) 訂票:

輸入身分證字號、列車資訊(或訂單編號)

→ 交付「產生票」→ 顯示欲訂票資訊 → 確認訂票成功

(4) 查詢訂位代號:

輸入身分證字號、列車資訊

→ 顯示已訂訂單編號

(5) 查詢訂位紀錄:

輸入身分證字號、訂單標號

→ 顯示已訂訂單、車票資訊

(6) 修改訂位紀錄:

選擇退票或修改「查詢訂位紀錄」已顯示訂單

→ 交付「產生票」→ 顯示欲改票資訊 → 確認改票成功

其中除了「產生票」、「修改訂位紀錄」除外，都是使用者可以直接啟動的功能。

在 Use Case 的規劃分類上，我們認為主要功能分為：查票、訂票兩種，其中查票初步判斷會是簡單的資料庫 query，而不更動資料庫、票 entity 等等的值；然而訂票，會包含用 filter 產生客製化選單，返還給使用者確認之後再更動資料庫等等，此部分共通的情況我們統一整理成為「產生票」。

「訂票」過程必須包含「產生票」，因此歸類於 include 標籤。

而「修改訂位紀錄」則是在「查詢訂位紀錄」之後才能選擇性觸發，其中「修改訂位紀錄」又分為刪除票、修改票兩類，只有在「修改票」時才會觸發「產生票」(新的票)，因此這二者歸類於 extend 標籤。

2. Analysis & Design

Analysis:

我們將專案的 package 分為三大類，Boundary, Controller, Entity，另外有連接的資料庫。

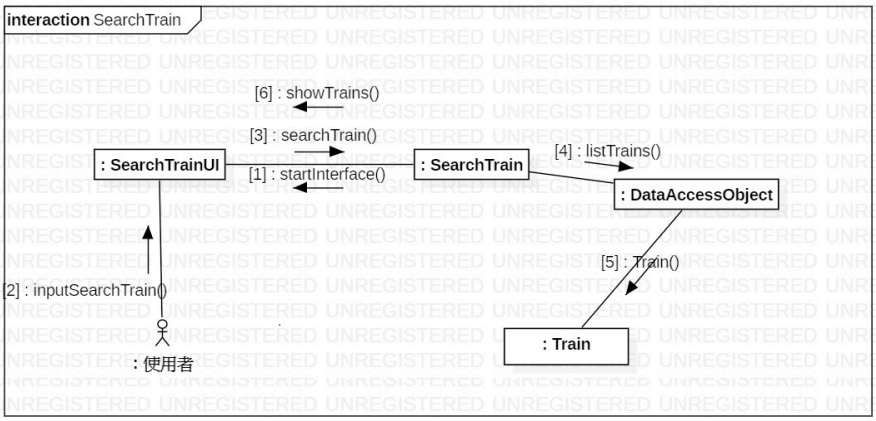
每一個 Use case 都有一個頁面的 Boundary (User Interface)，而負責讓 boundary 事件和其他物件互動的，都會是透過一個 controller 進行。

Controller 主要接觸的 entity，包含資料庫當中出現的一些 table，以及方便取用的參數匯整而成。而對於資料庫的變動，我們使用了一個 DataAccessObject 當作媒介，方便未來若是改動資料庫類型時，只需要更動此 DAO 當中對應函數即可。

Design: Interaction diagram

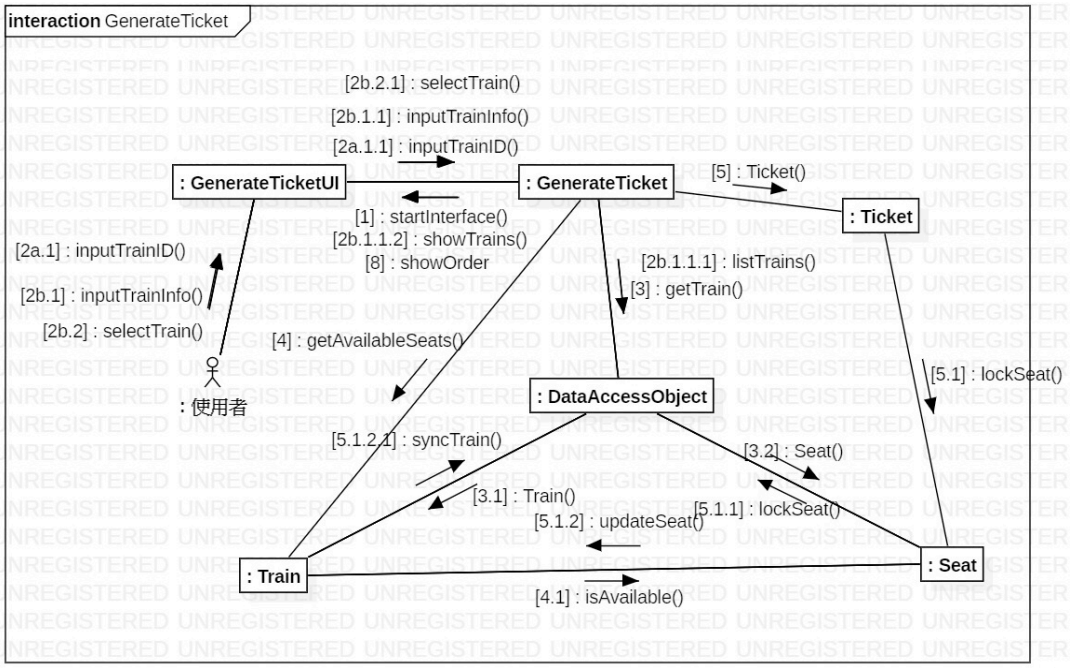
在 use case 已經確定的情況下，我們畫了以下的 communication diagram，目的是為了確立並統合 use case(是否能 reuse 或有過多 overlap)、物件之間互動的關係(method 大致上順序)、entity 實際上需要哪些等等。

(1) 查詢車次時刻表 (SearchTrain):



查詢時 controller 向 DAO 請求一個 TrainTime 的 list，取得之後回傳給 UI 顯示。

(2) 產生票 (GenerateTicket):

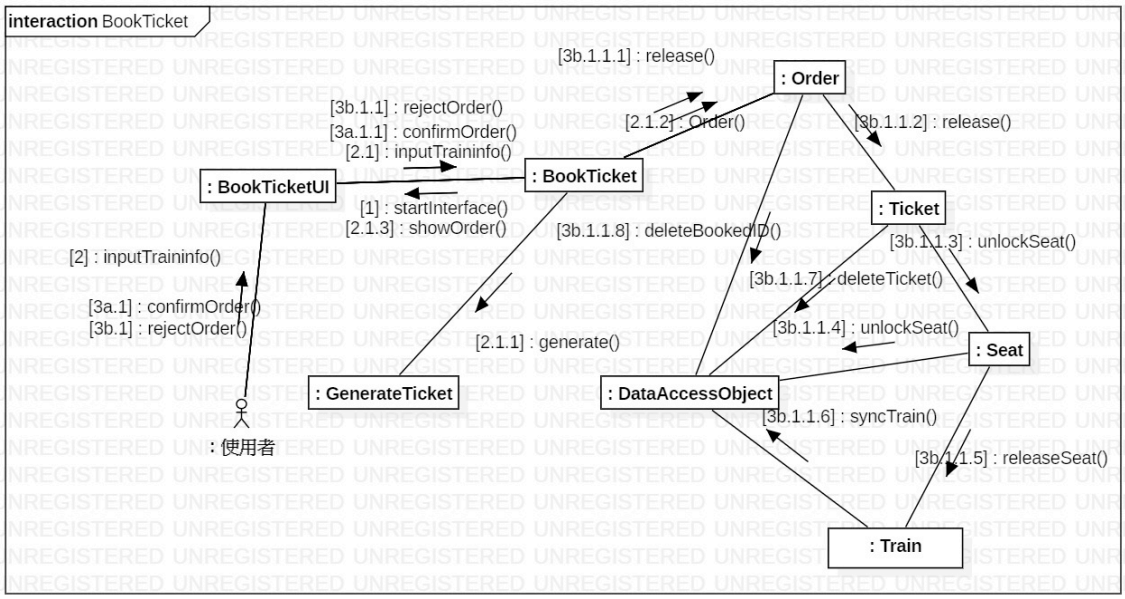


<step 1> Controller 從呼叫端收到一個傳入的「info」entity 後，向 DAO 取得符合 info 條件的 TrainTime list，再建立 UI 讓使用者選擇。

<step 2> 以選好的 TrainTime 向 DAO 請求空座位，並將此座位、車班資訊建立成一個新的 order 物件，返還給呼叫此 use case method 的函數。

< DESIGN > 我們設計的 entity 包含呼叫 DAO 中 method 的部分，而不是全部將工作由 controller 獨自完成。並且設立了 Seat, Ticket 等等 entity 讓資料管理更加清晰方便。

(3) 訂票 (BookTicket):

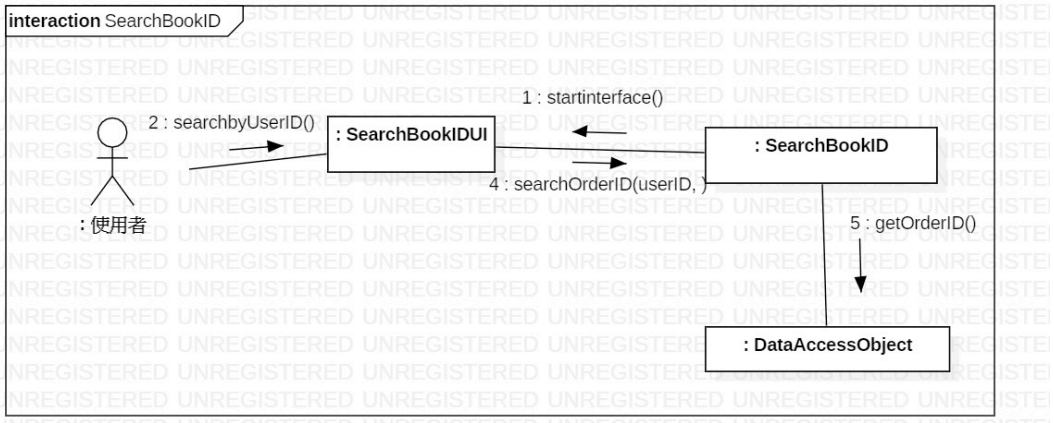


<step 1> Controller 將 info 傳入 GenerateTicket 並獲得新的 order 物件後，請 DAO 負責寫入資料庫。

<step 2> 回到 UI 詢問使用者是否確認訂單，若取消，controller 呼叫 order 物件讓他去把資料庫的對應 order record 刪除。

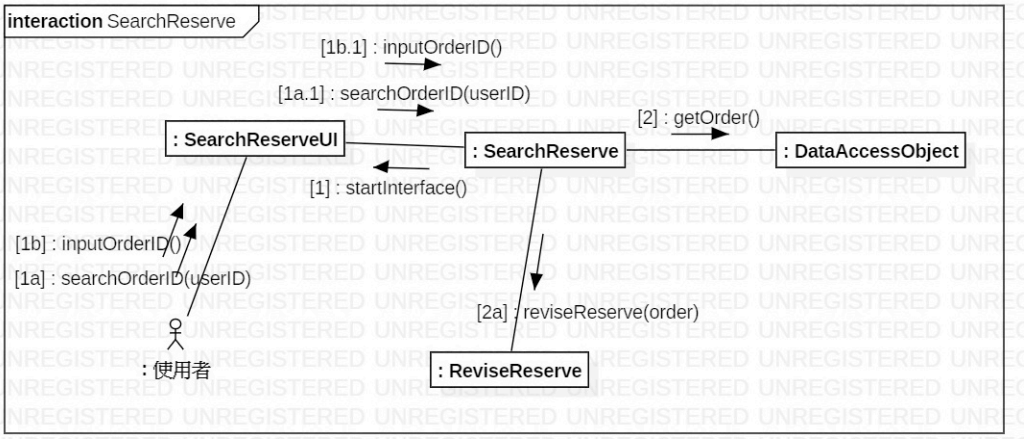
<DESIGN> 我們設計 order 物件在 initial 階段，由 GenerateTicket 負責創造，再由 BookTicket 寫入資料庫；release 階段則會由 order 親自去刪除，BookTicket 再釋放。

(4) 查詢訂位代號 (SearchBookID):



查詢時 controller 向 DAO 請求對應身分證字號、車班的所有 order id，回傳給 UI。

(5) 查詢訂位紀錄 (SearchReserve):



<step 1> 查詢時 controller 向 DAO 拿回指定的一項對應 order 物件。

<step 2> 回到 UI 詢問使用者是否取消或修改訂單。若取消，controller 呼叫 order 物件讓他去把資料庫的對應 order record 刪除；若修改，呼叫 ReviseReserve。

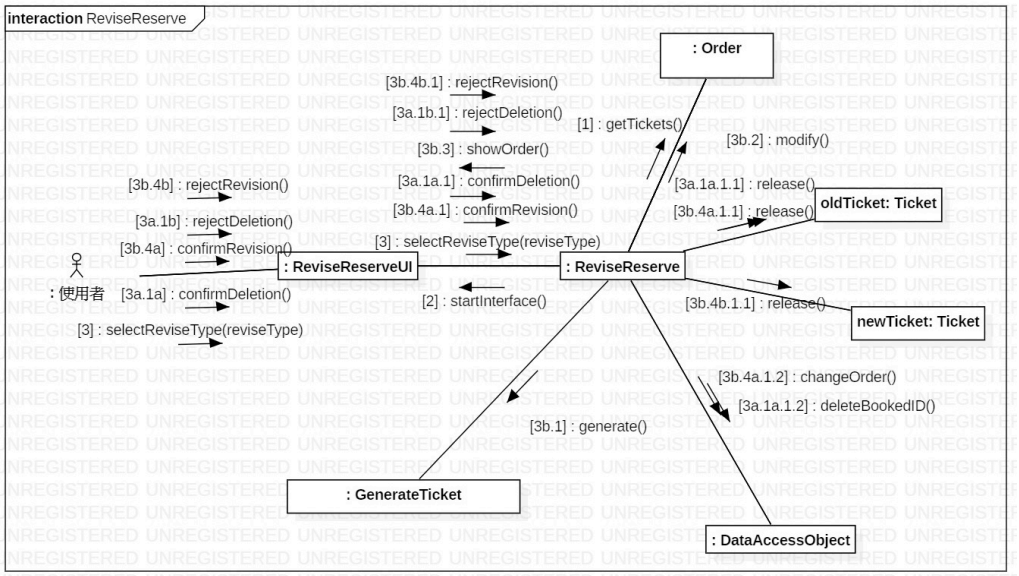
(6) 修改訂位紀錄 (ReviseReserve):

<step 1> 由 controller 呼叫 UI 顯示新的訂票頁面。

<step 2> controller 取得 info 傳入 GenerateTicket 並獲得新的 order 物件後，請 DAO 負責寫入資料庫。

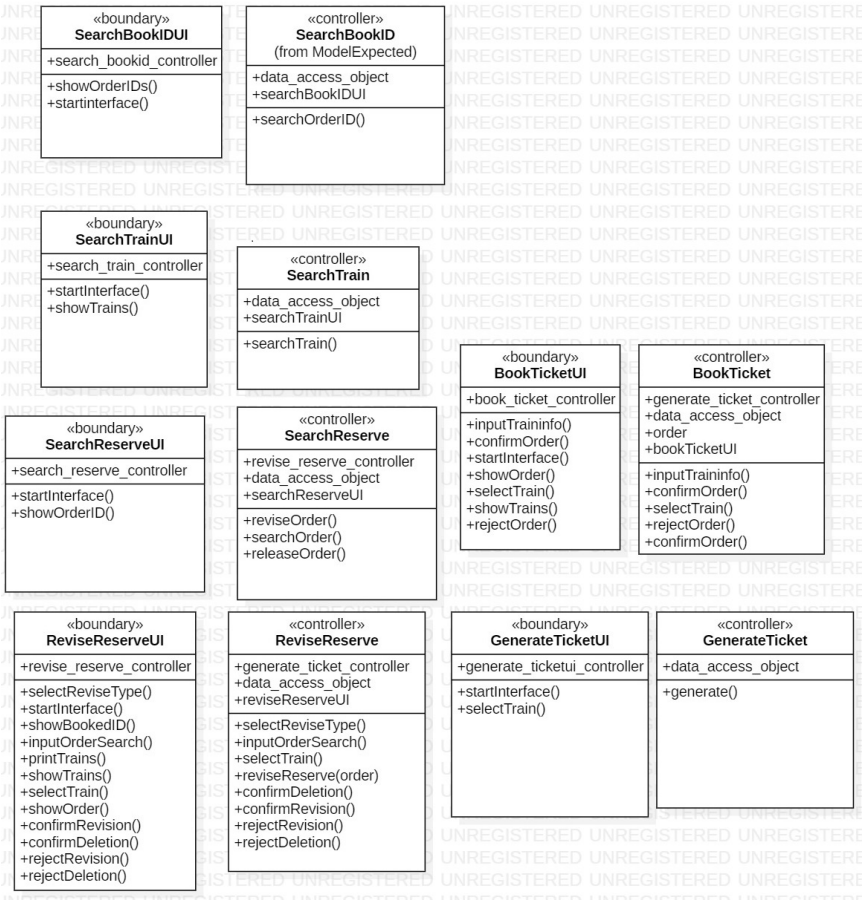
<step 3> 回到 UI 詢問使用者是否確認訂單，若取消，controller 呼叫 order 物件讓他去把資料庫的對應 order record 刪除。

<step 4> 已確認新的 order 寫入後，release 舊的 order。



3. Class Diagram

Boundary ,controller class 結構如右圖：

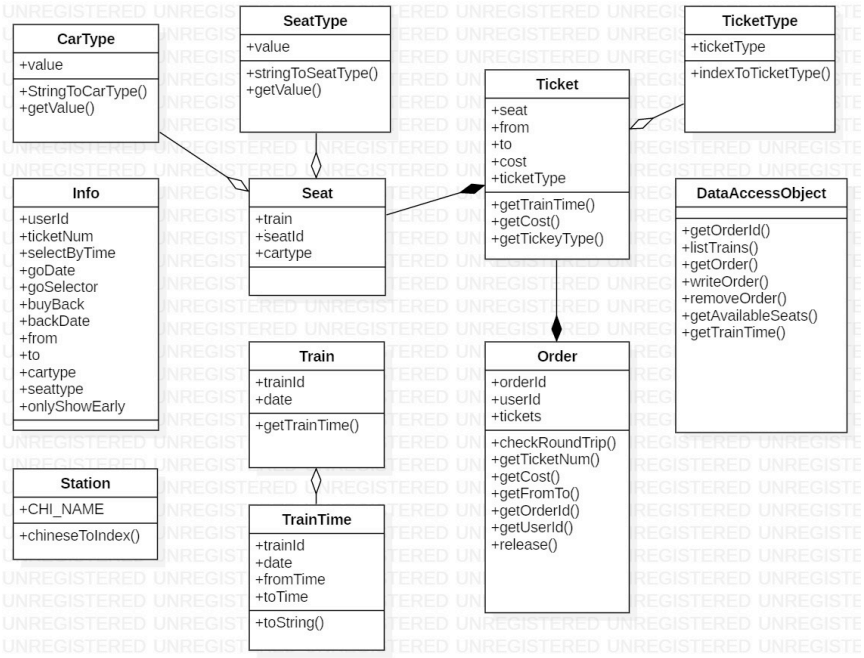


可以看到我們在任何一方都會存有對應的 boundary/controller 的物件，<1> 由 controller 喚起 UI 的 startInterface 佈置畫面

<2> 緊接著 UI 設定好各項按鍵 listener 等待使用者觸發 (我們使用 lambda function 所以並沒有列在 class entity 當中)

<3> UI 再呼喚 controller 接指令

<4> controller 開始對 entity 操作，將要呈現於畫面的結果回傳給 UI，use case 結束。



Entity class 結構如左圖：

<1> 如「info」方便傳遞的 class

<2> 如「carType、TicketType」等方便表示其他重要物件屬性 class(有增加其他屬性時只需更改此 class)

<3>「Seat、Ticket、Order」等模擬實體物件的 class

我們建立了這些 entity 來方便操作資料時可以完全與資料庫進行切割。

<4> DAO 的部分，還包含了一些 Private 區域變數，用來設定連上資料庫的方法和狀態。

其中，無論是 controller 或 entity object 呼喚 DAO 對資料庫進行改動時，我們在 method 當中也加入了 lock 資料庫的指令，使得系統若是面臨多使用者連線的情形，也會確保取到的 ticket 物件與資料庫內容的一致性，算是將資料庫的 concurrency control 運用在物件層級的溝通當中。

4. UI-bounding

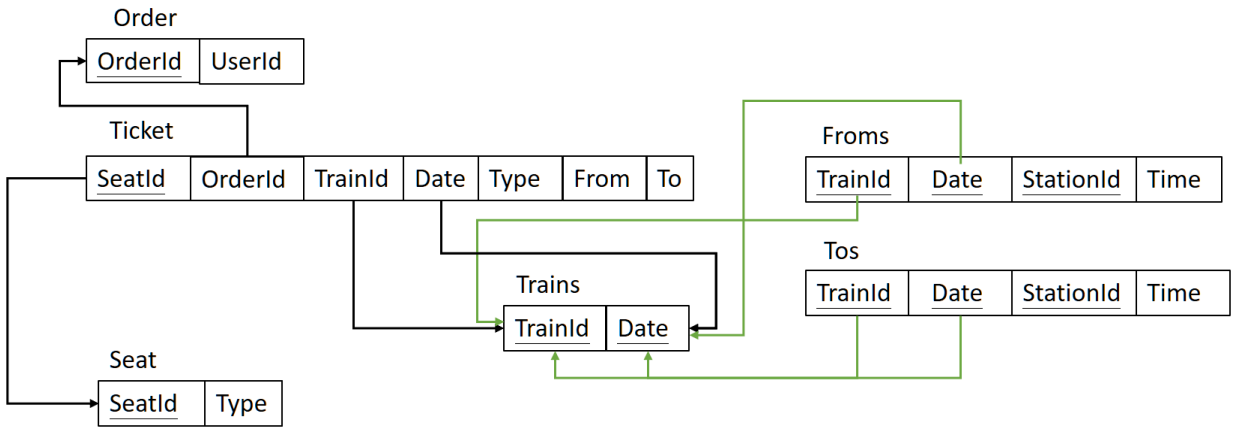
如同 use case 中所述，我們切分的 boundary 分為：

(1) 訂票
(2) 查詢車次時刻表
(3) 查詢訂位紀錄
(4) 查詢訂位代號

共通點為都從「查詢」的 user-friendly UI 著手，且每個 use case 可以任意跳轉到其他 use case 以及主畫面。

主畫面(main_page)是我們設立的初始化 UI class，也是我們劃分 use case 的 interface。因為在 main_page 先設立好了每個 controller 啟動的 listener，我們分工時能同時著手不同的 use case 前後端，也不會有內容衝突的情況發生。

5. Database



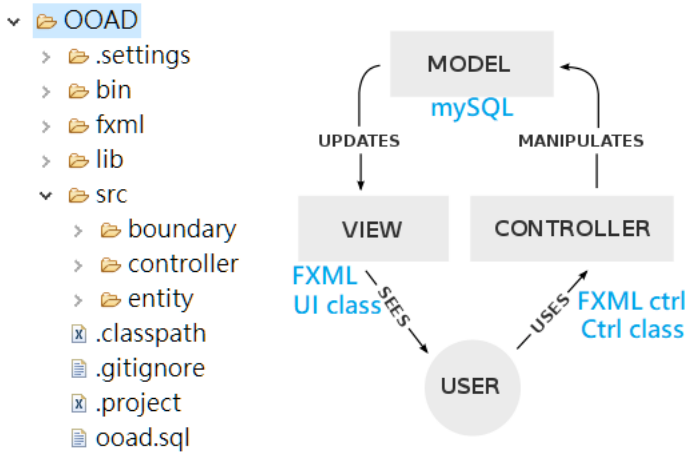
經過 ER diagram 以及 3NF 之後，我們的資料庫設計如上圖所示。

(1) 核心的資料圍繞在靜態的「Trains、Froms、Tos、Seat」四個龐大 Table 當中，我們會依序在 mysql 當中導入四個 table 的值，來保持他們的 foreign key constraint。

(2) 變動資料主要出現在 order 以及 ticket 的 insert 中，我們也在寫入的過程中幫資料庫上 lock，以保持不同使用者讀取資料前後的一致性。
(3) 這邊的設計與 entity 有相似之處，foreign key 所指向的 primary key 相當於 uml diagram 中表示的 composition；而資料庫當中，table 生出的每個 row 也類似，物件導向中一個 class new 出來的 object，帶有原本 prototype 設定好的對應關係(與其他 table 或 object 之間)。這裡我們在 analysis / design 階段有考慮，從而並且影響了我們的設計。

Implementation

1. multiple clients



實作 MVC 示意圖與 working repository

Database：

採用的是 RDBMS 的 MySQL，使用 sql 語法進行 model 部分資料存取的操作。同時在 Java 虛擬機的執行中，也有 entity 存取著資料庫中對應的內容，以供 controller 做訊息處理。

```
public static boolean writeOrder(Order order) throws Exception {
        PreparedStatement pstmt = connection.prepareStatement(
                "LOCK TABLES Orders WRITE, Tickets WRITE " +
                ";");
        pstmt.execute();
        pstmt = connection.prepareStatement("SELECT SeatId FROM Tic
        ResultSet rs;
        for( Ticket it:order.tickets ) {
            pstmt.setInt(1,it.seat.train.trainId);
            pstmt.setString(2,it.seat.train.date.toString());
            pstmt.setString(3,it.seat.seatId);
            pstmt.setInt(4,it.to);
            pstmt.setInt(5,it.from);
            pstmt.setInt(6,it.to);
            pstmt.setInt(7,it.from);
            rs = pstmt.executeQuery();
            if(rs.next()) {
                pstmt = connection.prepareStatement(
                    "UNLOCK TABLES " +
                    ";");
                pstmt.execute();
                return false;
            }
```

在 DAO 執行 sql query 時，將 table 上了 lock，使得其他使用者無法在此期間更動

UI：

使用 JavaFX 的 library 擺設前端頁面的物件，而此 library 可以使用 FXML 格式構圖，因此 view 負責顯示的部分包含以下 class：

(1) Use Case 所對應到的 UI class

(2) Use Case 跳轉到的新頁面的 UI class
(3) 每個 UI class 要顯示的頁面 FXML class

Controller：

使用 Java 進行撰寫，負責處理訊息的去向，同時連接著後方的 model 與前方的 view，在實作上 controller 包含了以下 class：

(1) Use Case 所對應到的 Control class
(2) 每個 FXML class 中負責接收物件名稱，且可編寫 method 的 FXML Control class

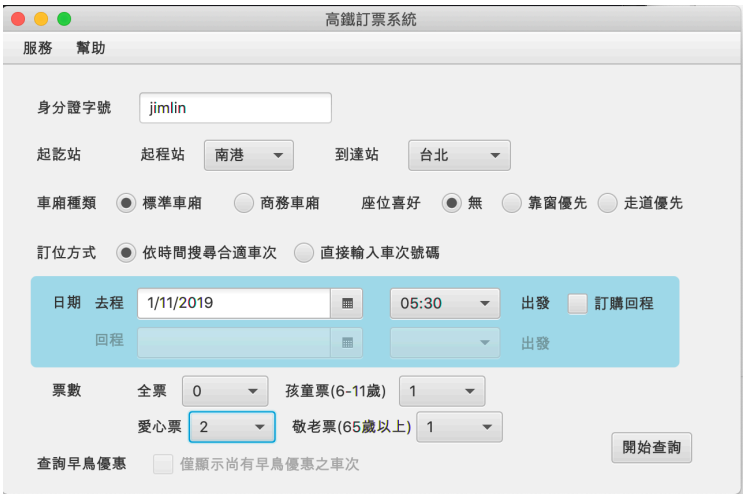
2. Demo

主畫面：

功能選單：

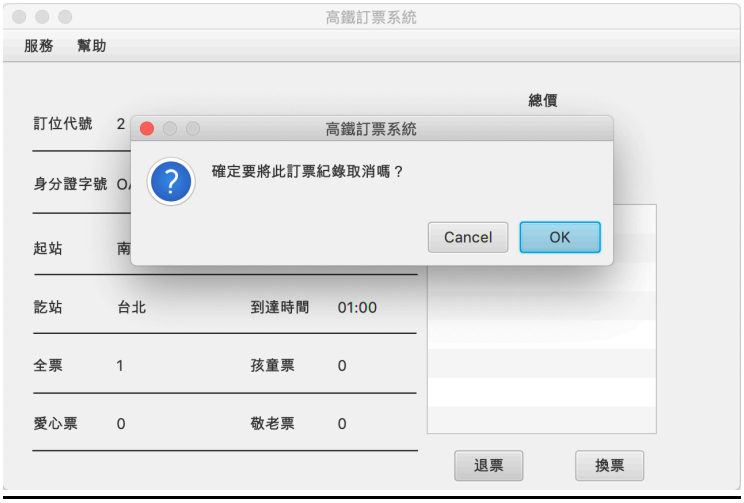
查詢車次：

輸入資訊訂票：



查詢訂票紀錄：

退票：



修改車票時間：

查詢訂票代號：

Comparison

1. UI

Sample code 使用 Swing 實作 UI 介面，每一個 component 的位置和大小都直接寫在 MainPage.java 裡，看起來版面有些雜亂。

我們的 UI 實作是使用 JavaFX 裡的 FXML，每個頁面會有自己的 .fxml 檔案和 FXML controller，.fxml 檔案是設定頁面格式以及上面的元件們，FXML controller 則是負責將頁面上的元件與 code 中的變數名稱做串連，以便指定 event listener。為了操作方便，我們沒有手刻 FXML，而是在 Scene Builder 拖拉我們想要的 component 並且設定簡單的 css，scene builder 會幫我們生成相對應的 .fxml 檔案。由於頁面元件的設定是分開來寫在 .fxml 檔案當中，且 .fxml 檔案格式與 HTML 的格式相仿，都是用一個一個標籤表示元件，我們的實作方式可以讓 code 看起來比較整齊。

2. Data Access Object

Sample code 中對於 database 的操作是直接將 SQL 指令寫在 controller 當中，這並不符合 MVC 架構中 database 獨立於 controller 的概念。於是我們新增了一個 DAO class 來做所有與 database 相關的操作，以 DAO 作為程式與 database 中間的介面，如此一來在 controller 當中不需要再管 SQL 的語法，只需要呼叫 DAO 的函式即可，如果要重構或者使用其他 database，也只需要改動 DAO 中的實作方式，不需要動其他 package。

3. 將管理介面從使用者介面上移除

Sample code 中，「更新車次時刻表」這個按鈕可以讓程式重新去網站抓車次資料，這不太符合實際的使用情境，實際上程式應該自己會在背景更新資料，而不是讓使用者去手動更新，因此我們移除了這個按鈕。

4. 明確的 MVC 架構

Sample code 中幾乎把所有頁面都寫在同一個 UI 中，以 UI 為主軸，觸發了什麼功能就去呼叫哪個 controller，但有一些操作寫在 UI 裡，也有一些錯誤訊息是在 controller 中被輸出；而在我們的架構中是以 controller 為主軸，一個 controller 會對應自己的 UI，每個 controller 分別處理不同的 use case，並且在被創建的時候去開啟自己的 UI，UI 只處理顯示的部份，剩餘則交給 controller 處理。

Reuse

1. coupling / coherent

我們讓每個 use case 分開來擁有自己的 controller 及 UI class，這些 class 只處理自己的 use case 而不做其他事情，module 內部有很高的 cohesion，因此很容易 reuse。同時為了降低 class 間的 coupling，我們將搜尋車票的資料集合成一個 Info class，避免在改變資料數目、種類的時候造成傳遞資料的兩邊 class 之間參數不同步。

2. Modularization & Componentization

在一開始切分 use case 時，我們發現了「訂票」以及「修改訂位紀錄」中，其實都會用到查詢、對 user 顯示車次、產生出票、對 database 寫入 order 等等一系列的程序，所以我們決定把「產生票」也獨立出來作為一個 use case，讓我們不必造兩次輪子，可以在兩個情況中都直接使用「產生票」的 controller 及 UI class。

而在不同 use case 中，有時候會需要用到相同的頁面，如「查詢車次時刻表」與「產生票」中都會用到顯示 order 明細的頁面，因此也可以共用此頁面的 .fxml 檔案及 FXML Control class。