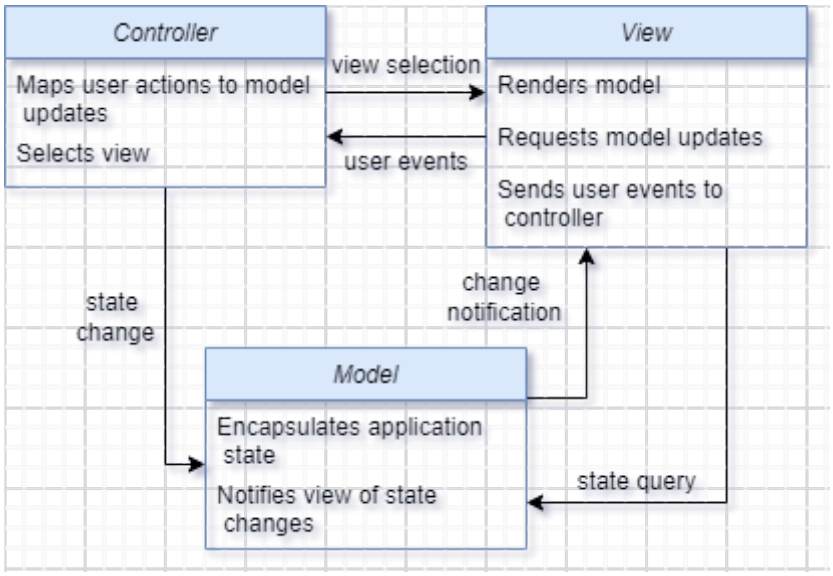
軟體工程 Homework #3

Part 1. Software architecture

(a) Model-View-Controller (MVC)

The Model-View-Controller (MVC) is an architecture pattern that separates an application into three main logic components. Each of these components are built to handle specific development aspects of an application. MVC is one of the most frequently used industry-standard web development framework to creates scalable and extensible projects.



Model

The Model component corresponding to all the data-related logic that user works with. This can represent either the data that is being transferred between the View and Controller components or any other business logic-related data. For example, a Customer object will retrieve the customer information from the database, manipulate it and updates it data back to the database or use it to render data.

View

The View component is used for all the UI logic of application. For example, the Customer view will include all the UI components such as text boxes, dropdowns, that the final user interacts with.

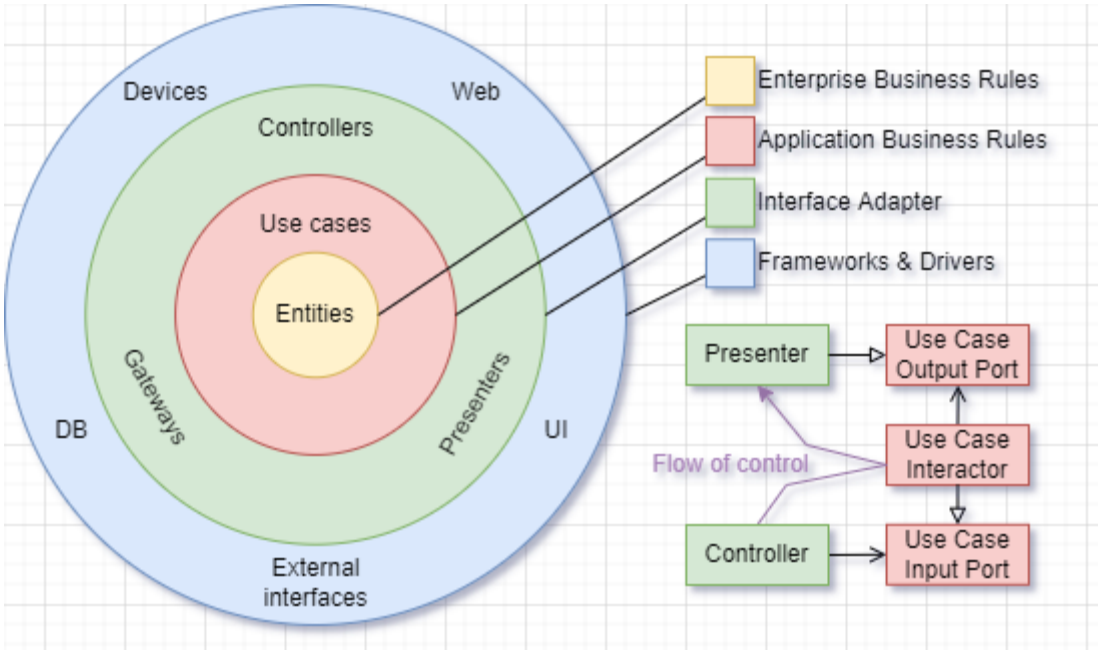
Controller

Controllers act as an interface between Model and View components to process all the business logic and incoming requests, manipulate data using the Model component and interact with the Views to render the final output. For example, the Customer controller will handle all the interactions and inputs from the Customer View and update the database using the Customer Model. The same controller will be used to view the Customer data.

(b) clean architecture

Clean Architecture 是一種設計理念，目的為「最小化建構與維護所需的人力」，可將一個系統設計的各項要素區分成多個層次，並使其邏輯能進行獨立分離，而不彼此依賴且可被測試，架構圖越靠近內部層，為越高的層次，意即越接近策略，越外層則代表機制，依賴關係由外層像內層推進，外層的改動不應該影響到內層，並且有三項原則：

1. 跨層原則：資料的傳遞、存取，不能跨層。
2. 分層原則：至少分成基礎的四層，可以更多。
3. 相依性原則：依賴關係必須由外向內，如果出現了由內向外的情況就用 Dependency Inversion Principle 來將其反轉。



Clean Architecture 層級分為四種：

Entities

封裝企業的業務規則，可以是一個物件或一種資料結構。這一層的物件都不應該被外層規則的變化所影響而更動。

Use Cases

Use case 定義了軟體要完成的任務，並指揮 Entity 層的物件來實現業務邏

輯計算

Interface Adapter

這一層級的職責是由 Adapter 將內外層級的資料格式進行轉換後，再輸入或輸出給上層級或下層級。

Frameworks and Drivers

這一層級由一些框架、工具、和外部服務組成，目的是為上面各層級提供工具。例如：資料庫、網頁框架

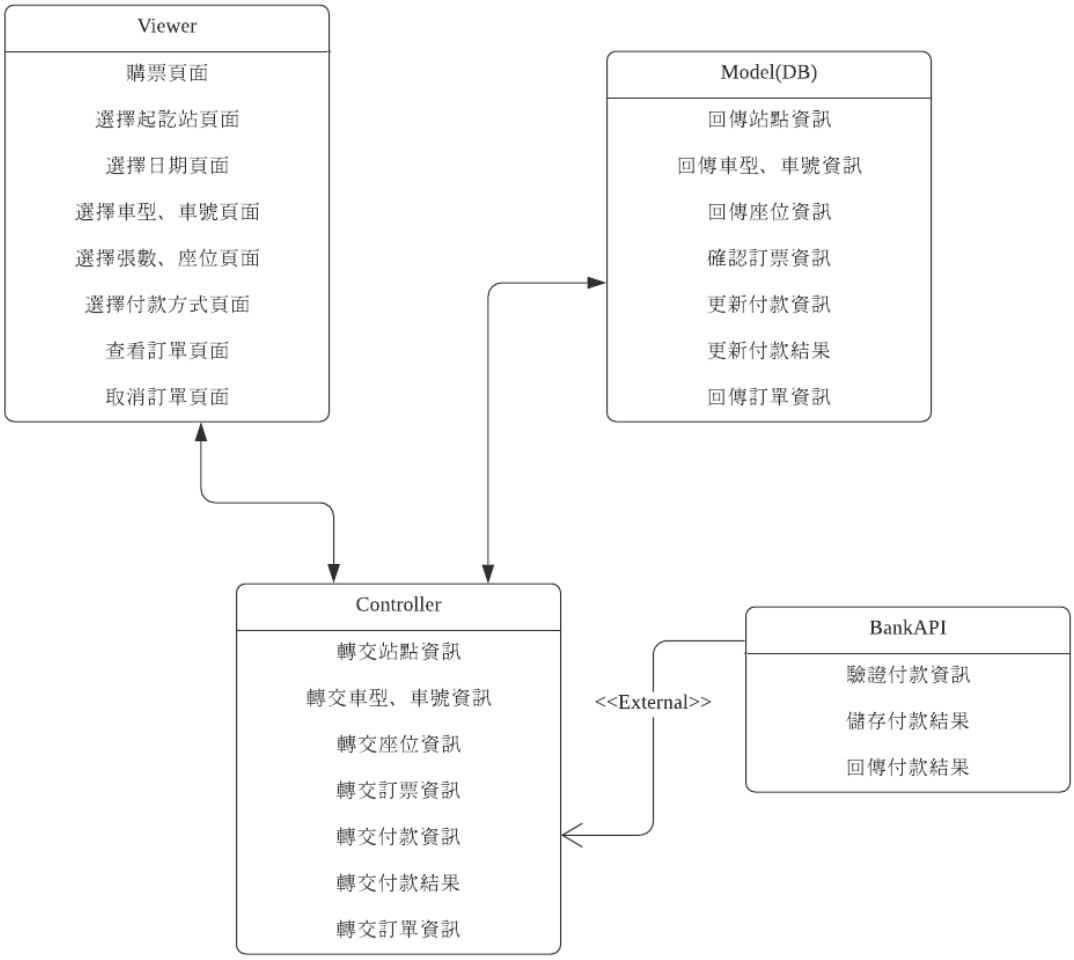
(c) microservices

Microservices Architecture 是一種將伺服器應用程式建置為一組小型服務集合的架構風格，透過將大型的應用程式架構成模組化的元件或服務，切割出單一責任與功能的元件，並使用定義明確的介面，如 API，來做為與其他服務溝通的橋樑，再透過建構模組來取得所需要的服務，每項服務專為一組功能設計，並著重於解決特定問題。

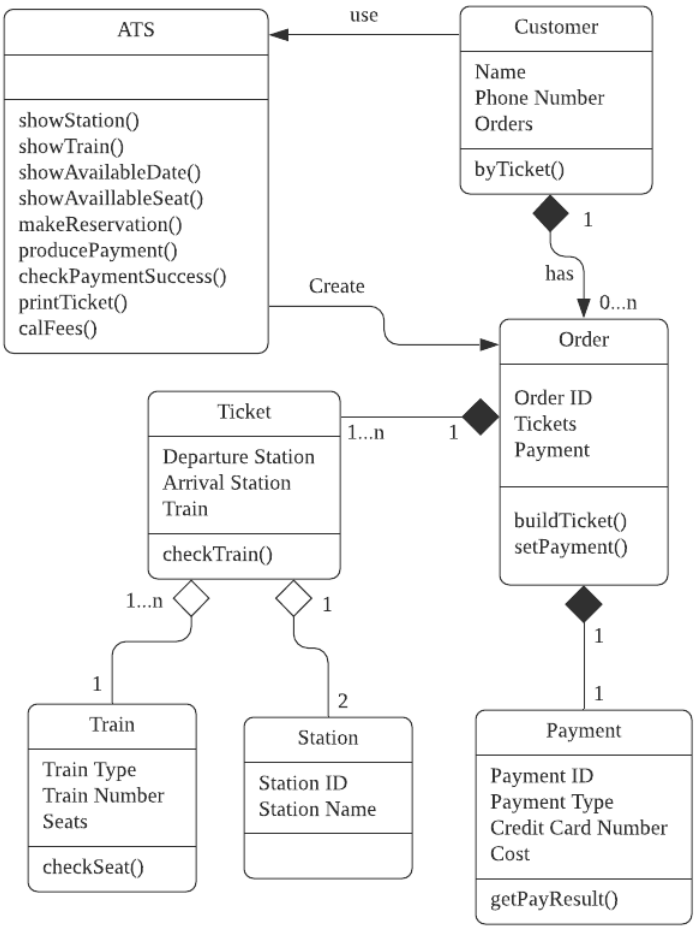
Part 2.

(a) architecture diagram

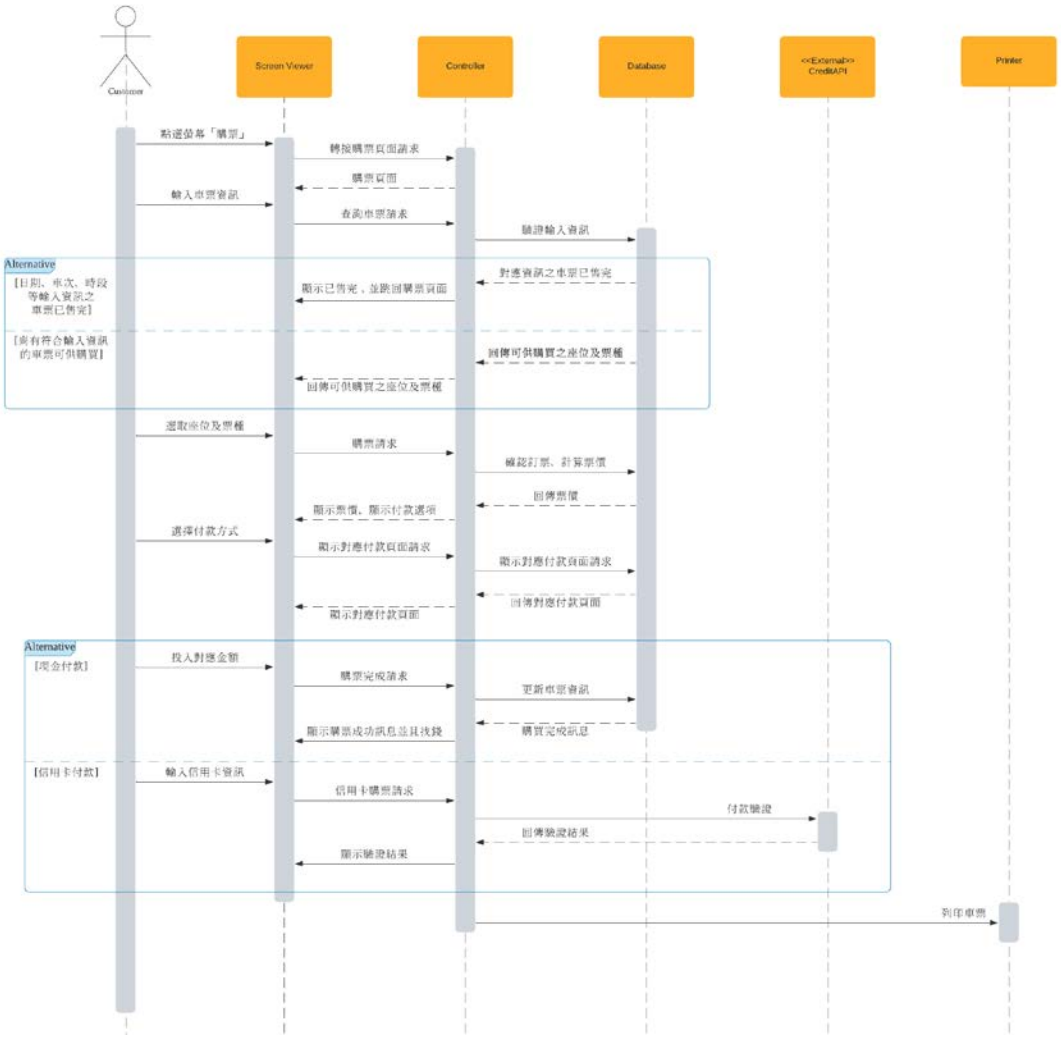
我選擇使用 MVC Architecture，考慮到之後可能可以擴充線上訂票系統，又或者可能加入新的付款方式，以及有可能新增新的功能，再來也有可能要新增新建的車站，或是停用舊車站，如新台中火車站以及舊台中火車站等等的因素，我認為採用 MVC Architecture 會相對來說比較好動工。比如新增新的付款方式的話可以依靠串接新的 External API，新增或刪除車站據點只需要對 Model(DB)內的站點資訊變更即可讓新增的站點資訊透過 Controller 顯示到 Viewer。


Viewer
購票頁面
選擇起訖站頁面
選擇日期頁面
選擇車型、車號頁面
選擇張數、座位頁面
選擇付款方式頁面
查看訂單頁面
取消訂單頁面
Model(DB)
回傳站點資訊
回傳車型、車號資訊
回傳座位資訊
確認訂票資訊
更新付款資訊
更新付款結果
回傳訂單資訊
Controller
轉交站點資訊
轉交車型、車號資訊
轉交座位資訊
轉交訂票資訊
轉交付款資訊
轉交付款結果
轉交訂單資訊
<<External>>
BankAPI
驗證付款資訊
儲存付款結果
回傳付款結果

(b) domain model (or the class diagram)



(c) sequence diagram for the “purchase tickets” scenario



Part 3. SOLID principles

Initial	Stands for	Name	Concept
S	SRP	Single Responsibility Principle 單一職責原則	物件應具有單一職責，若同一物件有兩個或以上職責時，當其中一個職責需要有所更動，極有可能會影響到另一職責。
O	OCP	Open Closed Principle 放封閉原則	軟體應該要對擴充保持開放，意即寫新的程式來新增功能；且對修改保持封閉，意即不該更動現有的程式碼。
L	LSP	Liskov Substitution Principle 里氏替換原則	物件應可被子類別替換，並且保持程式正確性，而子類別的修改內容應符合父類別的行為方向，由此提高程式的可擴張性。
I	ISP	Interface Segregation Principle 介面隔離原則	模組與模組之間的依賴，不應有用不到的功能可以被對方呼叫。在撰寫類別時應要最小化類別之間的介面，降低耦合。
D	DIP	Dependency Inversion Principle 依賴反轉原則	高層模組不應該依賴低層模組，兩者皆應該依賴抽象。若高層模組依賴低層模組，則滴成模組的變動會使得使用者高層模組受影響，此為不合理現象，應由高層行為去操作低層實作。

Part 4. UML activity diagram