Deliverable #2 Template

SE 3A04: Software Design II – Large System Design

Tutorial Number: T01
Group Number: G6
Group Members:

- Jane Klavir
- Nathan Luong
- Areez Visram
- Jennifer Ye

IMPORTANT NOTES

- Please document any non-standard notations that you may have used
  - *Rule of Thumb*: if you feel there is any doubt surrounding the meaning of your notations, document them
- Some diagrams may be difficult to fit into one page
  - Ensure that the text is readable when printed, or when viewed at 100% on a regular laptop-sized screen.
  - If you need to break a diagram onto multiple pages, please adopt a system of doing so and thoroughly explain how it can be reconnected from one page to the next; if you are unsure about this, please ask about it
- Please submit the latest version of Deliverable 1 with Deliverable 2
  - Indicate any changes you made.
- If you do NOT have a Division of Labour sheet, your deliverable will NOT be marked

1 Introduction

This section should provide an brief overview of the entire document.

1.1 Purpose

State the purpose and intended audience for the document.

1.2 System Description

Give a brief description of the system. This could be a paragraph or two to give some context to this document.

1.3 Overview

Describe what the rest of the document contains and explain how the document is organised (e.g. ”In Section 2 we discuss...in Section 3...”).

2 Analysis Class Diagram

This section should provide an analysis class diagram for your application.

3 Architectural Design

This section should provide an overview of the overall architectural design of your application. Your overall architecture should show the division of the system into subsystems with high cohesion and low coupling.

3.1 System Architecture

- Identify and explain the overall architecture of your system
- Be sure to clearly state the name of the architecture you used (this is the name of the architectural pattern, not the name of your system)
- Provide the reasoning and justification of the choice of architecture
- Provide a structural architecture diagram showing the relationship among the subsystems (if appropriate)
- List any design alternatives you considered, but eliminated (and explain why you eliminated them)

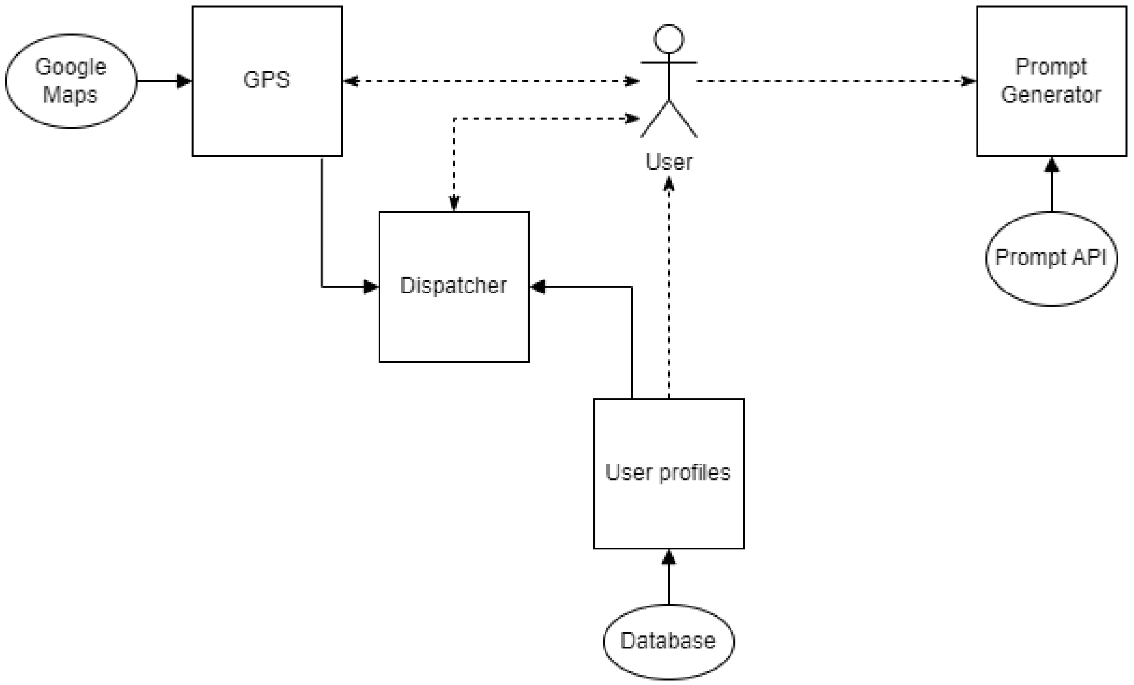
3.1.1 Overall Architecture and Justification

The overall architecture of the system is the Model-View-Controller (MVC) architecture, which is an interaction oriented architecture. The MVC architecture is the architecture that best fits the system that we are designing. The MVC style is best suited for interactive applications which contain multiple views for various data models. The application being implemented is heavily focused on the UI on the mobile app, and so an architecture style that allows for easy display of data into a UI view is suitable. Furthermore, our system contains multiple data models, for example, models for Users, Rides, Prompts and multiple pages/views across the application will interact with these data models. Furthermore, the MVC architecture is suitable for applications which are prone to frequent data changes. Our system is most definitely prone to frequent data changes, as rides are constantly being offered, arriving, and starting, which means the Rides model will be constantly updated and written to. Finally, the MVC architecture will allow us to separate the various logic functions into separate controllers. The architecture will allow for clear division of logic between controllers, which will make it easier to extend and modify the system. For example, we would have a UserController to handle user operations, a RidesController to handle ride operations and various other controllers to control the logic of other parts of the application. Given these benefits and the functionality of the system, we have chosen to implement the MVC architecture style for the main system.

3.1.2 Subsystem Architectures and Justification

Another architecture style will also be used for the various subsystems of our application. One of the subsystems within the main system is the Dispatcher subsystem, which is responsible for storing all information on taxis and carpool offerings, as well as deciding how to match carpool offers with requests that come in. The repository architecture style is best suited for systems (in this case a subsystem) in which a central data store stores information and various agents communicate with it. This architecture style fits the Dispatcher subsystem because the Dispatcher will contain a central data store which will store all the information about carpools. The data store will be passive, it will be like a database which just stores information and can be read from. The various agents for the repository will be the users who are requesting and offering carpools. These agents are active, as they drive the flow of the subsystem by sending carpool requests and offering carpools. The agents do not communicate directly with each other, they communicate via the data store by sending requests to the data store, and the data store facilitates the requests and decides appropriate matches. Given this, the repository architecture style is the most suitable for the Dispatcher subsystem.

3.1.3 Structural Architecture Diagram



3.1.4 Design Alternatives Considered

One design alternative for the main architecture that was considered was the Repository architectural style, which is a data centered architecture. We envisioned the system having a central passive database repository, with multiple active agents communicating with the data store. The agents were envisioned to be the different parts of the system, such as a user agent, a rides agent, a prompts agent, and then a dispatcher, all which communicated with the central data store. However, this design was not chosen because the system that is being built is a single mobile application connected to a database. The system does not involve multiple agents which must communicate with a central data store. If there were multiple applications that needed to communicate, this architecture style would be more applicable. However, is is a single application and the different parts of it can be represented using controllers within the architecture rather than external agents, so this design was not chosen.

The second design alternative for the main architecture that was considered was the Presentation-Abstraction-Control (PAC) architecture, which is also an interaction oriented architecture. Both MVC (the architecture that was chosen) and PAC could work for this system, but we ultimately chose MVC over PAC for various reasons. PAC was not chosen because PAC is more useful in complex applications, which require a hierarchical layering of agents. Our system is not complex, it is a single application that communicates with a database. In PAC, the only communication that can occur is between agents, and in our system, we would not have enough agents to justify having communication only between them. This would only add complexity. Furthermore, PAC is more useful for concurrent systems, which our app does not need to be to fulfill its requirements. Finally, PAC is less publicized and less widely used, and so for a somewhat inexperienced development team, MVC is a better choice as there are more resources, information and examples available to help in the design.

3.2 Subsystems

The first subsystem is the dispatcher. The dispatcher is a data store for all communications within our system. It acts as the data store in the repository architecture which is the architecture style that this subsystem follows. The dispatcher has several purposes. The first is to store information about active taxis in the fleet. It also stores information about user offers and requests. Subsequently, the dispatcher suggests "offerers" to "requesters", and once a "requester" requests to join the "offerer's" taxi, the dispatcher must also display an updated fare to the "offerer" so that they can make an informed decision. This match-making process is another fundamental purpose of the dispatcher.

The next subsystem is the user profile subsystem, that handles tasks having to do with the processing and storage of user profiles. One main purpose of this subsystem is to facilitate the registration of new users. It records their information and remembers it in a database. Furthermore, another purpose is to allow registered users to make updates to their profile. When users edit their profiles, the user profile subsystem must make the necessary updates to the database so that all the information is up to date. Additionally, this subsystem allows registered users to delete their profiles, thenceforth these profiles are removed from the database.

Another subsystem is the GPS. The purpose of this subsystem is to allow users to track the route of their taxi. Also, it uses the carpoolers' locations to determine distances between carpoolers. The GPS subsystem requires interfacing with Google Maps.

The final subsystem is the prompt generator. This subsystem executes our additional innovative feature, which is allowing carpoolers to socialize through a series interesting prompts. The prompt generator must also implement an API to fulfill its purpose of coming up with prompts.

The subsystems interact with one another when it comes to the match-making procedure described in the dispatcher subsystem. The dispatcher is at the centre of it all, and for it to perform its role, it uses the user profile and GPS subsystems to obtain critical information. First, the user profiles subsystem provides data on users' social preferences, which helps the dispatcher predict user compatibility. Then, the GPS subsystem returns distances between the taxi and potential matches, which the dispatcher uses in showing trip conditions (estimated fare, time, and distance). Ultimately, this relationship is one in which the dispatcher is the central unit, and it uses the other subsystems to get the necessary parameters for its algorithm.

4 Class Responsibility Collaboration (CRC) Cards

Class Name: RegistrationPage	
Responsibility:	**Collaborators:**
Knows username	
Knows password	
Knows email	
Knows RegistrationController	
Handles ”Done” click event button	RegistrationController

Class Name: RegistrationSuccessPage	
Responsibility:	**Collaborators:**
Handles ”Finish” click event button	TaxiSessionController
Knows RegistationController	

Class Name: RegistrationErrorPage	
Responsibility:	**Collaborators:**
Handles ”Go Back” click event button	RegistrationController
Knows RegistationController	
Knows RegistationPage	

Class Name: RegistrationController	
Responsibility:	**Collaborators:**
Know RegistrationPage	
Handle user registration request	DataControllerClass, RegistrationSuccessPage, RegistrationFailurePage
Know RegistrationSuccessPage	
Know RegistrationFailurePage	

Class Name: CustomerInfo	
Responsibility:	**Collaborators:**
Knows username	
Knows password	
Knows email	
Knows birth date	
Knows prompt preferences	
Knows SettingController	

Class Name: CustomerEditPage	
Responsibility:	**Collaborators:**
Know password	
Know username	
Know prompt preferences	
Handles ”Save” click event button (to check is user changes are possible)	SettingController
Handles ”Delete Account” click event button	SettingController

Class Name: EditSuccessPage	
Responsibility:	**Collaborators:**
Handles ”Finish” click event button	SettingController
Knows SettingController	

Class Name: EditErrorPage	
Responsibility:	**Collaborators:**
Handles ”Go Back” click event button	SettingController
Knows SettingController	
Knows CustomerEditPage	

Class Name: SettingController	
Responsibility:	**Collaborators:**
Handle new user profile information edit request	DataControllerClassr, EditSuccessPage, EditErrorPage
Handle user account deletion	DataControllerClass
Know CustomerInfo	
Know EditSuccessPage	
Know EditErrorPage	
Know CustomerEditPage	
Knows DataControllerClass	

Class Name: PromptDisplayPage	
Responsibility:	**Collaborators:**
Know list of prompts	PromptController
Handle ”Refresh” click event button	
Handle ”Exit” click event button	PromptController

Class Name: PromptController	
Responsibility:	**Collaborators:**
Handle exit prompt request click event button	DataControllerClass
Handle prompt list request based on user preference	DataControllerClass

Class Name: CarpoolOfferPage	
Responsibility:	**Collaborators:**
Handle scanning QR code	
Knows number of people to offer the carpool to	
Knows the destination of the carpool	
Knows taxiID	
Knows if the carpool would like to participate in the prompt feature	
Handle ”Submit Offer” click event button	Dispatcher

Class Name: TaxiSelectionPage	
Responsibility:	**Collaborators:**
Handle ”Destination” click event button	Dispatcher
Sorts taxi listing based on criteria selected though ”Sort by” click event button	
Handle ”Details” click event button on each taxi listing	Dispatcher
Filters taxi carpool listing based on selected criteria	
Requests list of taxi carpool in the users current area	Dispatcher
Knows basic details (destination, number of people in the offer, time) of each listing	

Class Name: TaxiDetailPage	
Responsibility:	**Collaborators:**
Handle ”Confirm” click event button	Dispatcher
Handle ”←” click event button	Dispatcher
Handle list information request on page load	Dispatcher
Know all the listing’s information (prompt preferences, car type, number of people already in the carpool, number of people wanted for the carpool, estimated price, etc.)	

Class Name: LocationSelectionPage	
Responsibility:	**Collaborators:**
Handle ”←” click event button	Dispatcher
Handle ”Save” click event button	Dispatcher
Knows desired destination of a user	
Knows difference in distance between current and desired location	
Knows if destination entered is possible	

Class Name: Dispatcher	
Responsibility:	**Collaborators:**
Handle submitting carpool offer	DataControllerClass
Handle display enter destination screen request	LocationSelectionPage
Handle getting more details for a listing request	DataControllerClass, TaxiDetailPage
Handle carpool offer list requests	DataControllerClass
Handle user selecting the carpool offer through confirm request	DataControllerClass
Handle going back to taxi carpool listing page	TaxiSelectionPage
Handle saving destination location request	DataControllerClass
Know if user is registered	
Know CarpoolOfferPage	
Know TaxiSelectionPage	
Know TaxiDetailPage	
Know LocationSelectionPage	
Know DataControllerClass	

Class Name: FareDisplayPage	
Responsibility:	**Collaborators:**
Handle ”←” click event button	TaxiSessionController
Calculates and displays the fare for the user when carpool ends	

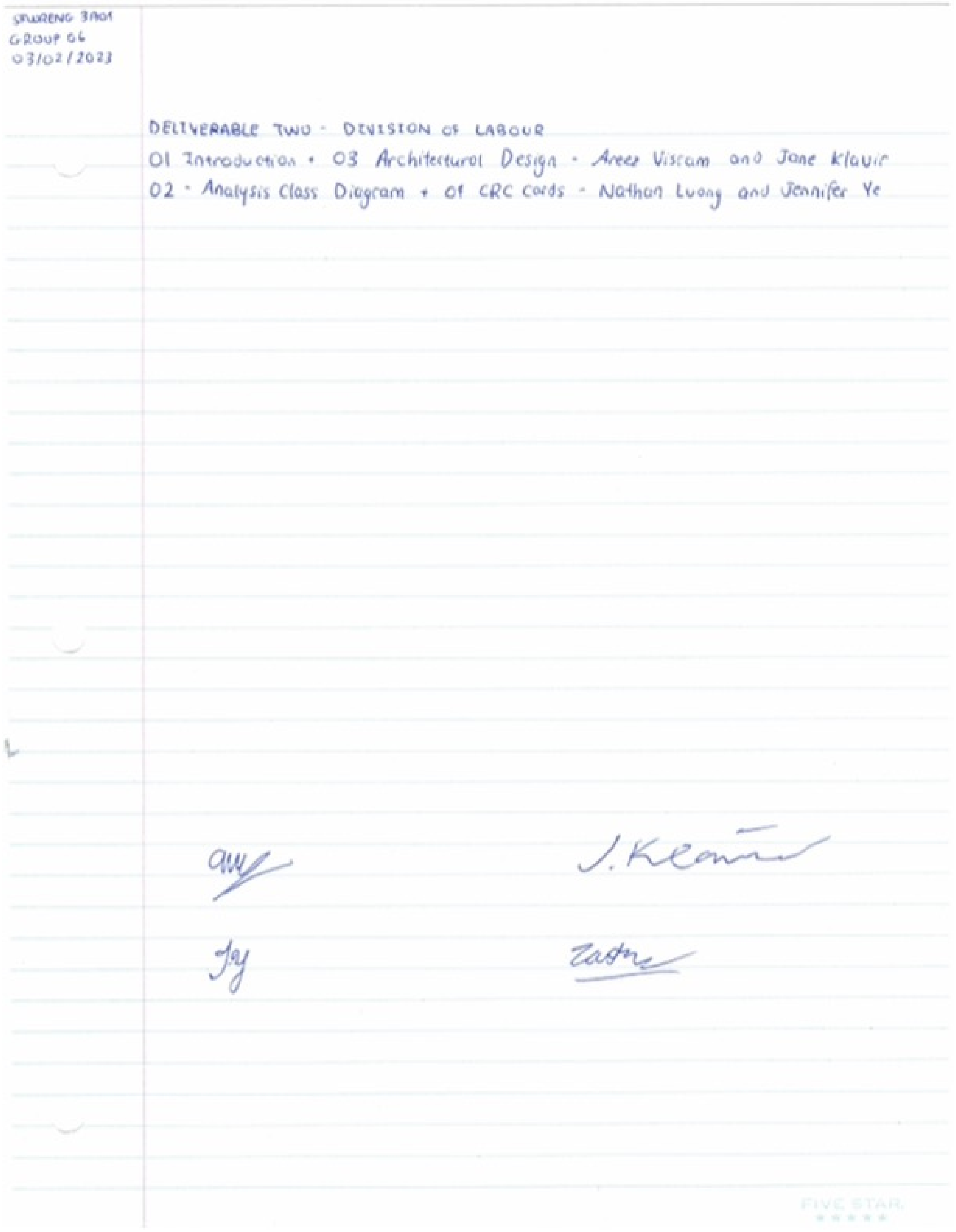
Class Name: RatingPage	
Responsibility:	**Collaborators:**
Handle ”Submit” click event button	TaxiSessionController
Knows number of stars user gives the trip	
Knows number of stars user gives the app experience	

Class Name: LoginPage	
Responsibility:	**Collaborators:**
Know username	
Know password	
Handles ”Login” click event button	AuthenticationController
Know AuthenticationController	

Class Name: LogoutPage	
Responsibility:	**Collaborators:**
Handles ”Logout” click event button	AuthenticationController
Knows AuthenticationController	

Class Name: AppDB	
Responsibility:	**Collaborators:**
Accomplish the following two tasks in one atomic step: (1) Verifies the nonexistence of a username, password and email pair, (2) if successful, insert the pair into the database	DataEncryptionController
Accomplish the following two tasks in one atomic step: (1) Verifies the nonexistence of a username, password and email pair, (2) if successful, allow user to access the app	DataEncryptionController
Accomplish the following two tasks in one atomic step: (1) Verifies the nonexistence of a username, password and email pair, (2) if successful, remove the pair into the database	DataEncryptionController
Accomplish the following two tasks in one atomic step: (1) Assuming user is verified, use username to find prompt preferences (2) Return list of prompts based on the preference	DataEncryptionController
Assuming user is verified, insert all information of the carpool that is being offered	DataEncryptionController
Return all information of a carpool when details are requested	DataEncryptionController
Return list of carpools that has not met its number of people needed	DataEncryptionController
Accomplish the following two tasks in one atomic step: (1) Get the taxiID from the carpool listing currently on screen (2) Insert the taxiID to the user	DataEncryptionController
Assuming user is verified, save the entered destination to the user	DataEncryptionController

A Division of Labour



SFWRENG 3A04
GROUP 06
03/02/2023

DELIVERABLE TWO - DIVISION OF LABOUR
01 Introduction + 03 Architectural Design - Areez Visram and Jane Klavir
02 - Analysis Class Diagram + 04 CRC cards - Nathan Luong and Jennifer Ye

[illegible] [illegible]

[illegible] [illegible]