Task 1 | Ambiguities

Ambiguity: Should MockPaymentSystem be kept even if the operation is done by an external system.

Solution: The payment system is assumed to use a tested and robust external system. We decided to remove the in-house testing system.

Ambiguity: Should the EntertainmentProviderSystem interface be kept when it will only be implemented once and does not adhere to the YAGNI.

Solution: Since each kiosk has their own version of the code stored locally, it makes sense to assume that there will be only one entertainment provider for each kiosk. Therefore, to make the code adhere to the YAGNI design principles, the interface for EntertainmentProviderSystem was removed. Instead, we merged EntertainmentProviderSystem and EntertainmentProvider into one class called Staff.

Ambiguity: How would save and import data work in the system.

Solution: To achieve this, our Data class has a direct association with the Context class. It uses this direct association to retreive all information about consumers, bookings and events via the classes UserState, BookingState and EventState which is then serialized and exported. When importing the data, the data will be interpreted and then added back to the system through the 'AddUser', 'createBookings', and 'addEvents' functions which assumes an older 'state' of the program prior to importing the data.

Ambiguity: How would a consumer have a record of the set of instructions to get to the venue.

Solution: We assume that external map system is what most of the industry uses (e.g., google maps). The current solution is to display it in the kiosk and consumers would have to take a picture with their phone or other ways to capture the display.

Ambiguity: How to tell if an event has taken place.

Solution: We add a function called 'hasEventTakenPlace' which takes an event number as input and returns a Boolean value on whether the event has taken place yet. This will automatically take the current date from the kiosks OS within the 'hasEventTakenPlace' function (so no LocalDateTime variable is required in the input) which then compares against startDateTime to return an answer.

Ambiguity: Whether to include a PaymentState and DirectionsState of type enum in our classs diagram.

Solution: Since we are using external systems here, keeping track of whether payments have been completed and directions successfully retrieved are going to be assumed completed externally through their respective systems. Therefore, it isn't necessary to show it on our diagram.

Ambiguity: How to prevent non-staff members from creating a staff account.

Solution: In our UserState class, we are going to add a new attribute called 'validStaffEmails' which is a list of Strings which contains a list of all emails which are validated to be Staff. Additionally, there will be an associated function 'addValidStaffEmail()', which will add any email address on input such that when

logging in, they will assume Staff level access. However, this will come with the pre-condition that the first time the system is booted up, the first account created will be a staff.

Ambiguity: What data type should tags be.

Solution: Since tags can technically be of any type, to simplify our implementation, we are going to assume tags will be Strings to accommodate all data types.

Ambiguity: Whether only Consumers can leave reviews or can Staff leave reviews as well.

Solution: We are assuming only consumers can leave reviews and designed our system only to allow consumers to leave reviews.

Ambiguity: How importing and exporting data works at a code level.

Solution: The staff class will have private methods 'importData()' and 'exportData()' which both return null. These functions will create an instance of the Data class. Depending on which function is called, if exportData() is called, it will prompt the staff to enter a file name and destination and then call the 'export()' method in the Data class. Similarly, the same will occur with the 'import()' method.

Task 2 | Old System's UML Class Model

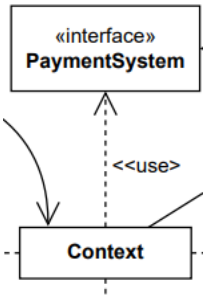
1)

Design principle #1: High cohesion

The old class diagram adheres to design principle # 1. For most of the classes, they have distinct and clear responsibilities. An example of this is the PaymentSystem. It demonstrates clear dependency and implementation of MockPaymentSystem and Transaction. It also shows a clear encapsulation of the payment system. In general, the whole diagram displays a clear association between functionalities and has good readability.

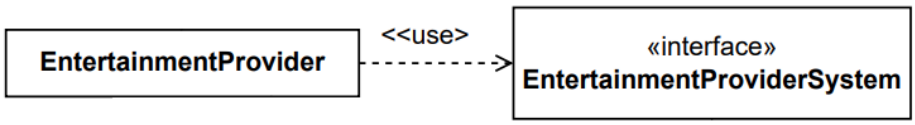
Design principle #2: Low coupling

The old class diagram adheres to design principle #2. This is done through the unidirectional associations between the classes. We can see this used in the relationship between Context and the PaymentSystem interface classes. We can also see low coupling through the use of the MVP design pattern as it decouples presentation from data and logic.



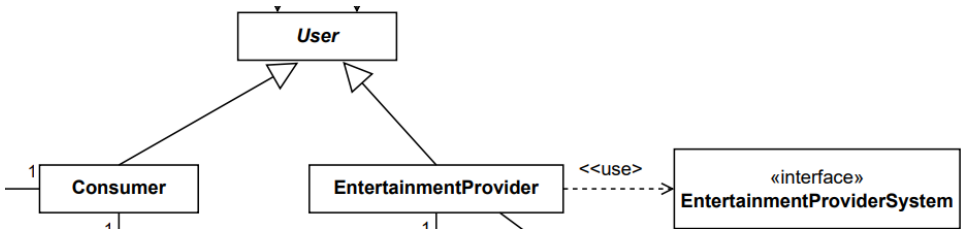
Design principle #3: Abstraction

The old class diagram adheres to design principle #3. The old class diagram achieves this by using interfaces which creates a barrier between classes which use the interface, and the inner workings of how it is implemented. An example of this is the EntertainmentProviderSystem interface which is implemented by EntertainmentProvider. We know this is true since a dotted arrow between them shows an implementation of the interface.



Design principle #4: Decomposition

The old class diagram adheres to design principle #4. This is because the class diagram divides a large system into well-defined interfaces. For example, there are distinct and well-defined interfaces such as EntertainmentProviderSystem which is shown using <<interface>>. The class diagram adheres to design principle #4 also because the large system divides into smaller components with distinct responsibilities such as User which divides into distinct responsibilities such as Consumer and EntertainmentProvider.

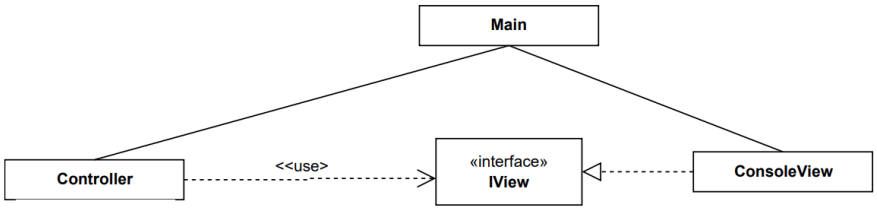


2)

MVC

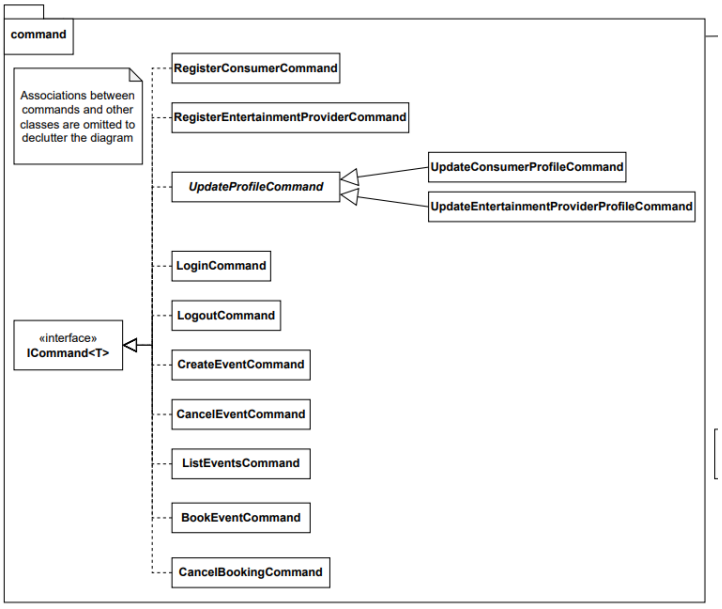
Model View Controller is a design pattern used in this class diagram. The program has the traditional three classes, Model, named as command, Controller and ConsoleView where ConsoleView implements the

interface IView. It is beneficial to use MVC as a design pattern since it adheres to Design principle #2 since we decouple presentation from data and logic while also adhering to Design principle #4 decomposition since we are delegating certain classes for certain responsibilities.



Command pattern

Command pattern is a design pattern used in this class diagram. The system has several classes such as LoginCommand, LogoutCommand that implement an ICommand interface. Using a command pattern is beneficial as it can help to easily log and track user requests as well as undo and redo as needed, and commands can be queued. This makes the system extensible as adding new commands is easy.



3)

Check valid staff email

The login and register requirements mean that we need to add the command CheckValidStaffEmail to the command section. This was added as an extra security feature to make sure logging in/registering as staff members is not open to everyone.

List Event By Distance

The Listing events requirement, specifically the Listing events by Distance requirement means that a ListEventByDistance is added to the command section. It is added in the command section as an extension

of the existing list event command as it is just an extension of this as it is just an ordered version of listEvents. It is added here as it does not require its own state and is just another way to list events.

Reviewing events

The reviewing events requirement means that the AddEventReviewCommand and ListReviewsCommand are added to the command system and a new review class was added. Reviews were not a function in the previous code or diagram and so we added a new class to hold all the review information for when a review is created, the AddEventReviewCommand and ListReviewsCommand were added to interact with this class.

Add event tag

The event tag requirement means that an AddTagCommand is added to the command section and a tag class is created. The tag class is needed to store the tag information and so the information can be accessed easily, the AddTagCommand is then added to interact with the class.

Getting directions to an event

The Getting directions requirement means to get directions to an event can be shown in the class diagram in a similar way to the external payment system is implemented. This means an interface association drawn out from the Context class to some interface which is implemented by some class which is associated with some class which holds directions to the event. Get directions command will also be added to the command system. This needs to be added as it was not a requirement for the previous app and so was not implemented or displayed.

Saving and loading System state

The saving and loading system state requirement means we added the saving and loading system state into the class diagram, by adding a 'Data' class which has a normal association with class 'Context'. The reason it is a normal association is because a 'Data' object calls an operation from a 'Context' object. In this case it will be some function which retrieves the state of the system. This is why the association is with the class 'Context' since many interfaces are associated with this class which contain information about the system. This includes information about user states, booking states and event states. There will also be a export data command added to the command system.

Handling entertainment providers

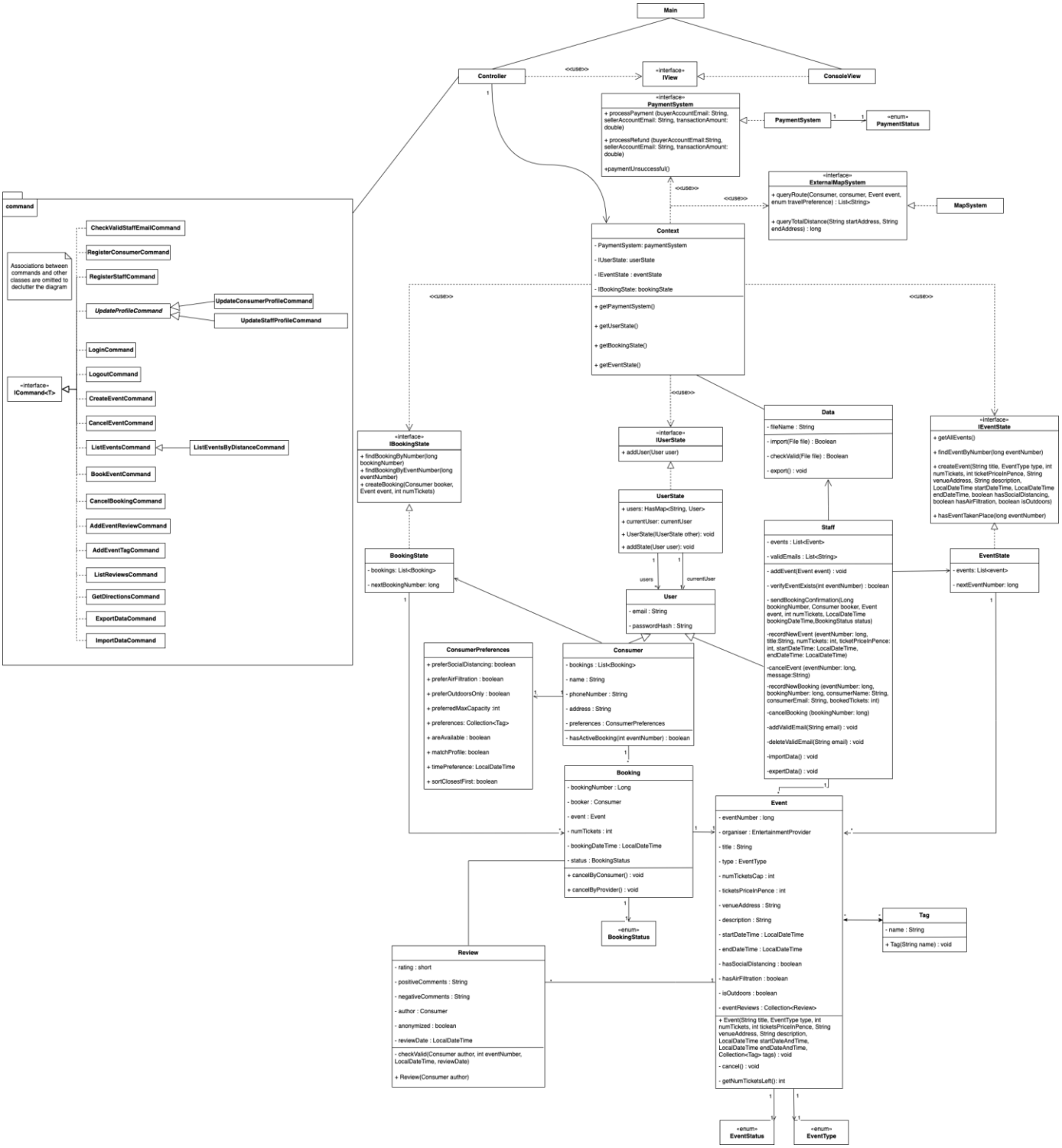
The Handling of entertainment providers requirement is addressed by our merging of the entertainment provider and entertainment provider system classes into one class called staff. We have done this to address the change from the original system which had to consider multiple entertainment providers to our new system in which each entertainment provider runs their own versions of the code. Meaning we no longer need a separate interface for the entertainment provider system and can have all the same functions in a staff class.

Checking event has taken place

As part of adding reviews to events, one of the conditions is that the event hasn't taken place. In order to satisfy this condition, we need to add a function into EventState which takes an event number as input and returns whether that event has taken place yet.

Task 3 | New UML Class Model and Description

Task 3.1



Associations with commands

We decided to omit the associations from commands to declutter the diagram. We will discuss the associations and multiplicity associated below. With the case a command has multiple associations, they will be listed via bullet points.

Command	Association	Multiplicity
CheckValidStaffEmailCommand	• Staff • Context • UserState • View	1-1 1-1 1-1 1-1
RegisterConsumerCommand	• UserState • Consumer • Context • View	1-1 1-1 1-1 1-1
RegisterStaffCommand	• UserState • Staff • Context • View	1-1 1-1 1-1 1-1
UpdateProfileCommand	• UserState • User • Context • View	1-1 1-1 1-1 1-1
UpdateConsumerProfileCommand	• UserState • Consumer • Context • View • ConsumerPreferences	1-1 1-1 1-1 1-1 1-1
UpdateStaffCommand	• UserState • Staff • Context • View	1-1 1-1 1-1 1-1
LoginCommand	• UserState • Consumer • Staff • Context • View	1-1 1-1 1-1 1-1 1-1
LogoutCommand	• UserState • Consumer • Staff • Context • View	1-1 1-1 1-1 1-1 1-1
CreateEventCommand	• UserState • Context • View • Staff	1-1 1-1 1-1 1-1

	• Event	1-1
CancelEventCommand	• UserState • Context • View • Staff • Consumer • Event	1-1 1-1 1-1 1-1 1-1 1-1
ListEventsCommand	• Context • View • Event • UserState • Consumer • Staff	1-1 1-1 1-..* 1-1 1-1 1-1
ListEventsByDistanceCommand	• Context • View • Event • UserState • Consumer • Staff • MapSystem	1-1 1-1 1-..* 1-1 1-1 1-1 1-1
BookEventCommand	• Context • View • Event • Booking • UserState • Consumer • Staff	1-1 1-1 1-1 1-1 1-1 1-1 1-1 1-1
CancelBookingCommand	• Context • View • Event • Booking • UserState • Consumer • Staff	1-1 1-1 1-1 1-1 1-1 1-1 1-1
AddEventReviewCommand	• Context • View • UserState • Consumer • Review • Event	1-1 1-1 1-1 1-1 1-1 1-1
AddEventTagCommand	• Context • View • UserState • Staff • Tag	1-1 1-1 1-1 1-1 1-1

ListReviewsCommand	• Context • View • UserState • Consumer • Event • Review	1-1 1-1 1-1 1-1 1-1 1-..*
GetDirectionsCommand	• Context • View • UserState • Consumer • Event • MapSystem	1-1 1-1 1-1 1-1 1-1 1-1
ExportDataCommand	• Context • View • UserState • BookingState • EventState • Staff • Data	1-1 1-1 1-1 1-1 1-1 1-1 1-1 1-1
ImportDataCommand	• Context • View • UserState • BookingState • EventState • Staff • Data	1-1 1-1 1-1 1-1 1-1 1-1 1-1

Task 3.1

1)

Security

We added a list of valid email addresses under the EntertainmentProviderSystem interface. Then we implemented a command called checkvalidEmailCommand, which is called when an entertainment provider staff account is being registered, to return a boolean value as to whether the email address provided is valid or not.

Tag System

In order to implement tags in our system, we created a Tag class which only has one attribute which is the tag name (the tag itself). We decided that tags will be of type String to reduce complexities within the program. This is because if we let numerical value tags be integers, it creates complexities because this means the system must deal with tags of potentially multiple types (such as boolean, string, long etc). Therefore, we believe a String is sufficient and numerical values can be represented as a string such as "<500" or Booleans with "wheelchair accessible".

Getting directions

When QueryRoute() is called, it prompts the external map system to find and return the shortest route automatically to the destination. It returns this as a list of directions (of type String) and gives information about each leg. The system uses an enumerated data type from a finite list of methods of transportation.

Export/Import data

In order to implement import/export data we created a Data class which handles this functionality. The class uses the Java Serialization API when exporting which all occurs under the export function. It retrieves all data such as bookings, events and consumers through the inputs of the function.

Payment System

Our payment system, despite implementing an external system, still has its own interface with functions. This is to aid the overall functionality of the application and achieve cohesion. While the actual payment process is managed externally, we collate information about transactions made and the state of these transactions (which are also visible to context). We use information about the consumer such as their email address to aid the payment process (even if done externally).

Review system

To successfully leave a review, the Review class has a private method which when called will verify that the event number is correct, the Consumer has an active booking for that event and that the review has taken place.

2)

In our UML Class diagram, we took inspiration from the old class diagram and added it. Therefore, we followed the same design principles such as abstraction, high cohesion, decomposition and low coupling.

Abstraction and cohesion are represented in our diagram through our interfaces, whereas cohesion is clearly represented by creating individual classes for the review system and tags.

Command Pattern

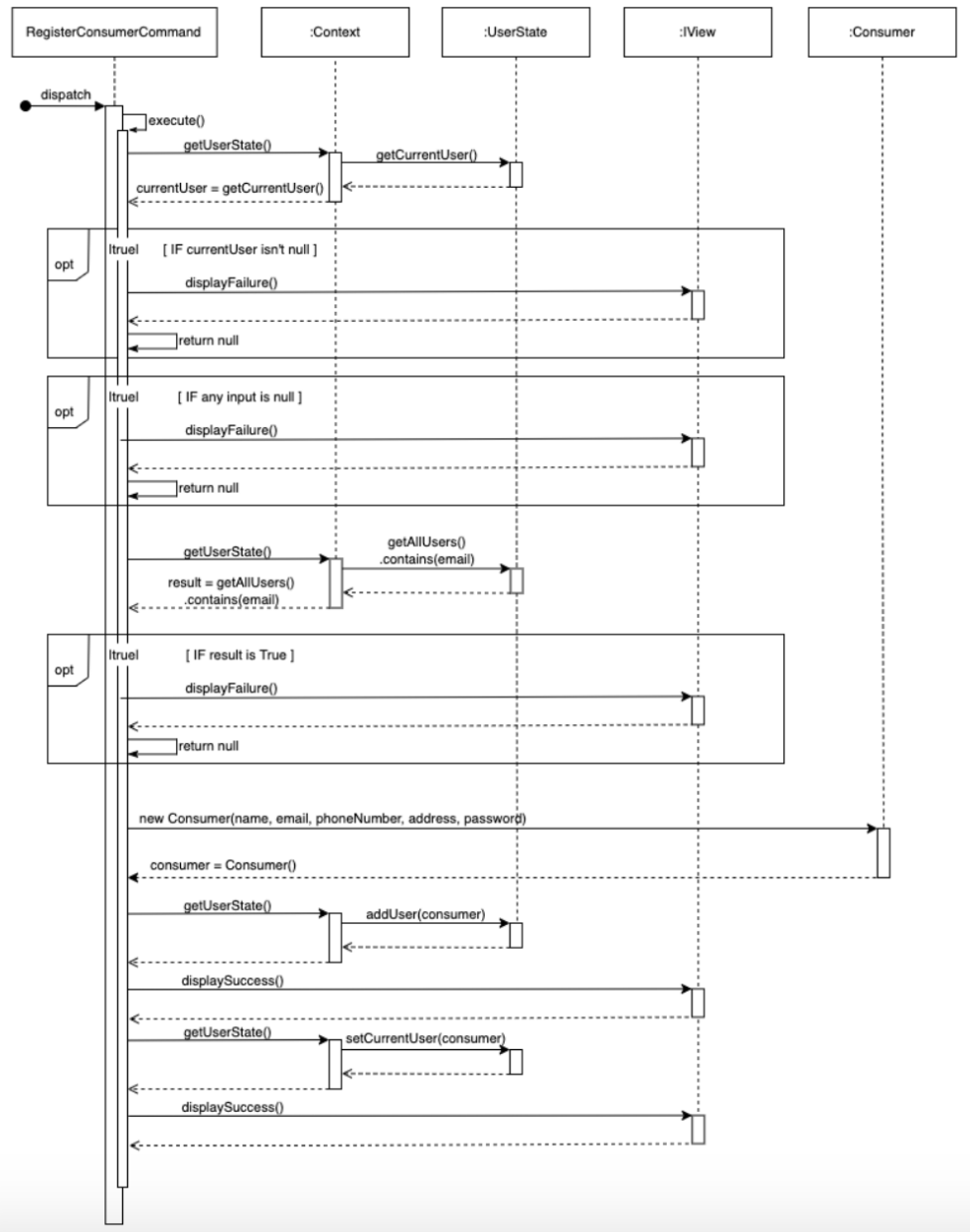
Our class diagram takes advantage of the command design pattern. The reason this is advantageous is because ‘commands’ can be modified and extended like any object. This means adding commands to the program isn’t difficult. Furthermore, using a command pattern decrease coupling between invoker and receiver. It also reduces code duplication.

MVC

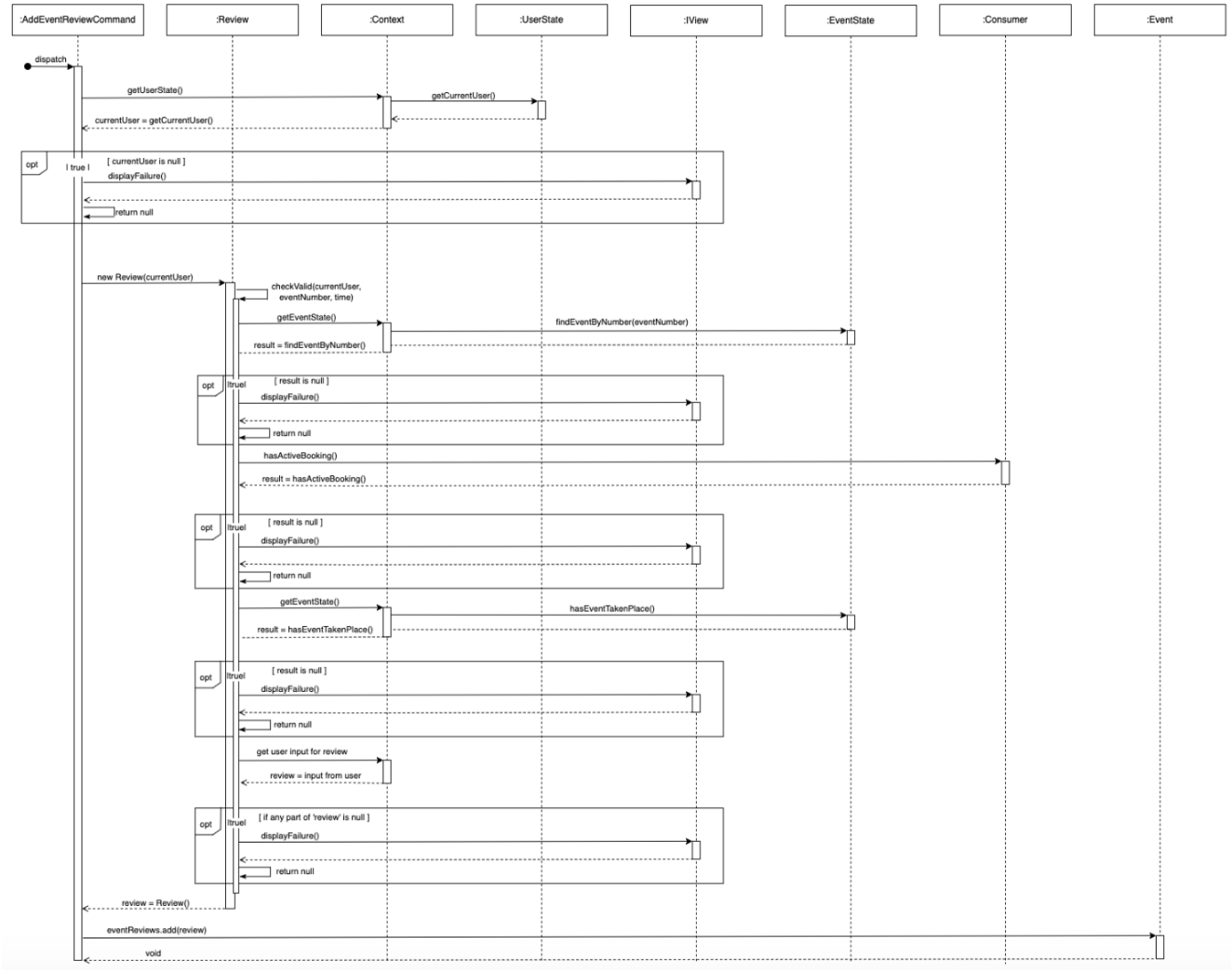
Our class diagram also highlights an MVC design pattern as well. This is notably presented through our decomposition separating the logic from presentation (Controller class vs ConsoleView class). This helps reduce complexity in our program as coupling decreases as we have split the program into separate parts. However, it can be argued that MVC pattern is only used sparingly since for our presentation, we are simply implementing an IView interface.

Task 4 | UML sequence diagrams

RegisterConsumerCommand Sequence Diagram



AddEventReviewCommand sequence diagram



Task 5 | The software development

1. a) Of the two types of software development processes that we have studied in this course, plan-driven software development is handling design more in this coursework.

   b) Reasons for plan driven:

   - Design is a separate stage in our software development process.
   - Architectural and detailed design is carried out thoroughly.
   - Formally using modelling and notation such as UML class diagram and sequence diagrams in the coursework has been carried out.
   - Output from design is going to be used in the implementation stage.

   c) Two reasons why it isn't agile development process is:

   - Design is not interleaved with requirements and implementation in each iteration. Instead of focusing on the most important unfinished features, we are focusing on all the features.
   - Formal, detailed, design documents are produced and not seen as a waste of time. Informal documents or design documentation is not automatically generated by programming environment in our case.

2)

Our team use the Big Design Up Front principle for their workflow by heavily utilising UML diagrams. This allowed us to carefully plan the project and eliminated any ambiguities. We do not use YAGNI due to strict requirements. We also did not use the DTSTTCW approach as it was not suited for a complex set of requirements and may result in incomplete features. Instead, we prioritise a stable system design with a clear understanding of each association between classes.

3)

a)

If we were to restart the whole design process, we believe it would be most advantageous to use the YAGNI and DTTSTCPW approach.

b)

Assuming the position of a company which has acquired legacy code and old UML class diagrams, a stronger understanding of requirements is attained as opposed to starting from scratch. This leads to requirements being better understood as it is possible to refer to the legacy software. Moreover, while BDUF does allow for a solid understanding of requirements before coding begins, it also means stakeholders are less involved in the development process. If requirements change, the non-iterative

nature of BDUF could hinder development which is costly to both Acme corp and the stakeholders of the application. As we were able to conduct interviews, acting on this regularly would benefit the development of the app as a further understanding of changing requirements and receiving early feedback is possible. Although a process conducted in BDUF as well, it is a tool that could be utilized more leading to a more complete final product.

Task 6: Reflection

Task 6.2.1

1. The Experience: This time, we tried to incorporate a detailed plan into our approach in completing the tasks. We created a timeline with deadlines for days of the week we would meet and what tasks we would aim to complete by. For example, we stated that by the end of week 1 (20th Feb – 24th Feb) we would have completed Tasks 2 and 3.

2. Reflection on Action: We followed our schedule which led to us being more time efficient. This method worked out well as compared to our previous coursework where we left tasks incomplete closer to the deadline, leading to a crunch.

3. Theory: We have learned that being organized with our schedule can lead to us having sufficient time to discuss tasks all together in a group. This proved to be better compared to splitting up tasks without any prior discussion, which often led to misunderstanding the given task and therefore delaying the coursework.

4. Preparation: Even though we followed our deadline, we still delayed some tasks due to inefficiencies in our work ethic resulting in more time allocated completing them than we initially decided. To improve, in the future, we would try to integrate an online planning tool that every member uses in the group such as Notion. By utilizing such tools, we can manage our time with team members actively updating the schedule based on our own availability, thus allowing for more efficient planning and coordination.

Task 6.2.2

1. The Experience: For the most part of the coursework, we think we handled all the tasks well. We managed to spend a lot of time discussing the given tasks and ways to best approach them by resolving any conflicts and making sure all team members were always on the same page. We ended up choosing the command design pattern for our class diagram as flexibility of “commands” as objects allows for easy modification and extension, making the task of adding new commands to the program a seamless process. It allows for the reduction in code duplication which will hopefully lead to a more consider and efficient codebase.

2. Reflection on Action:

Reflection on Action: We managed to extend the class diagram based on the new requirements quite substantially by adding well-chosen attributes, and interfaces for external systems used and respecting design principles. We mostly managed to cover use cases by reverse-engineering the old code provided to us. We referred to lecture slides as well as tutorials when appropriate, especially in making the sequence diagrams.

3. Theory: We have learnt a lot from this coursework regarding the design process. We have learnt how creating software design can have an impact on making such an application. Furthermore, we have learnt

how to use sequence diagrams and UML class diagrams based on the given requirements can please the stakeholders and meet their design criteria. Finally, we realized the importance of creating diagrams with precision as changes in the UML class diagram can result in changes to sequence diagrams and vice versa. This was particularly a challenge when creating the sequence diagram for the AddEventReviewCommand, as we had to ensure that functions that were called were consistent across both diagrams.

4. Preparation: Looking back, we could have improved our design by furthering our understanding of UML class diagrams and sequence diagrams. While we were able to successfully complete the tasks given to us, there were some areas where we spent lots of time debating what should and shouldn't be in the diagram. In the future, we will strive to allocate more time for research into design choices and hence make more informed decisions.