Inf2-SEPP 2022-23

Coursework 2

Creating a software design for a customisable events app

SAMPLE SOLUTION

1 Task 1: Ambiguities and addressing them

Note: The solutions to ambiguities from this task apply to the rest of these sample answers. You may have identified different ambiguities, or proposed different assumptions in response to them. These are all fine. What is important for a high mark is for all assumptions from this coursework to have been mentioned here, and for them to be consistent with your later solutions in this coursework.

Some ambiguities (there may be others) and possible solutions to them are as follows:

- Ambiguity: If a confirmation of the booking is only displayed, won't this risk consumers not being able to show anything in the form of tickets, and even worse forget their booking number and be unable to later cancel their booking?
  Solution: We assume that the kiosk could be used to print a screen out (separate from the system), and the consumer could use this to print their confirmation message. They could also be advised as part of their confirmation to make a note of their booking number.
- Ambiguity: How will tags be represented in the system?
  Solution: We assume that as name-value pairs, with a default value provided. When creating a new tag, staff can set its name, possible values it could take and the default value. They can select the value for each tag when creating an event. Then, on setting up their profile, consumers can select the values for all tags that had been used by staff throughout their events.

- Ambiguity: Will there be any default tags?
  Solution: We assume so, for social distancing, air filtration and venue capacity.
- Ambiguity: Are there any special checks when creating a new event that correct tags are provided?
  Solution: We assume so: the system will need to check that the provided tags will only include known names and values.
- Ambiguity: Can staff leave event reviews?
  Solution: We assume that not, like in the original requirements.
- Ambiguity: How are addresses (for consumers, venues) represented in the system?
  Solution: We assume that using latitude-longitude coordinates.
- Ambiguity: Is there any point where the consumer's address will be checked for correct format?
  Solution: We assume that the system will check that an address is provided correctly as latitude-longitude coordinates which additionally fall within map system boundaries, when the consumer registers, as well as when they update their profile (which may involve updating their address too).
- Ambiguity: Is there any point where a venue's address will be checked for correct format?
  Solution: We assume that yes, the system will check that an address is provided correctly as latitude-longitude coordinates which additionally fall within map system boundaries, when an event is created by a staff member, and thus the venue address is provided.
- Ambiguity: How will loading (importing) data work exactly in terms of the system making sure that importing (i.e. adding) the data from the file does not result in duplicated data or invalid objects (only some examples are provided in the changed/new requirements)?
  Solution: We assume that the following will cause the whole loading operation to be cancelled: for event tags, clashing tags (with the same name but different values), as this would affect events and consumers using these tags; any user email clashes for users that are not the same; for events, clashing events (with the same title, start date and tiime, and end date and time) that are not identical because otherwise this could result in either duplicated events or issues with connected bookings; for bookings, clashing bookings (by the same consumer for the same event with the same booking date and time) because otherwise this could result in duplicated bookings. Moreover, the current user is not replaced. If there are no clashes, identifiers (like event numbers or booking numbers) are renumbered for the newly loaded items to prevent conflicts.

2 Task 2: Old System's UML class model

1. The following are good answers for each of the taught design principles. Different opinions about how well a design principle is addressed overall will be accepted, as long as justifications demonstrate a good understanding of that principle and are convincing.

   - **Cohesion** is reasonable throughout the class diagram, but could be improved. Classes for important real-life entities (like User, Consumer, Entertainment-Provider, Booking, Event) with clearly individual and focused responsibilities have been chosen, and the names of all of the other classes do not lead to any doubts that their responsibilities would be focused and individual. This is confirmed by looking at the old code and the operations (i.e. responsibilities) available in the different classes, which are usually closely related to one another and to the name of the classes to which they belong. An exception is the classes in the command package, which could be argued do too much. Some of the responsibilities which concern accessed objects from the model classes could have been moved to these classes.

   - **Coupling** is reasonable, but could be improved. It has been considered by using the Command pattern (see below). The Controller and model classes have been decoupled: the controller does not need to know which model classes the commands act on or how, it only works with an object of generic type Command and at runtime the concrete command implementation will know all the details and the model classes to act on. However, the classes in the command package are usually coupled with numerous classes in the model package, while it could be argued that a better solution which delegates some responsibilities to these classes and takes benefit of the relationships between these classes could have been found.

   - **Abstraction** is high throughout the class diagram: classes usually only expose what is needed to external clients in terms of public methods; interfaces are used to present a protocol of how communication can be done with external systems, and to control how commands are being used.

   - **Encapsulation** is high throughout the class diagram: all attributes are private and only accessible if ever needed and through public getters and setters; methods that are only needed by the class itself are private; details of how external systems work are hidden behind interfaces.

   - **Separation of interface and implementation** is again high, for the reasons exposed in abstraction and encapsulation.

   - **Decomposition, modularisation** is high: the system has been split into classes with distinct responsibilities and well-defined interfaces, using a combination of real-world entities and entities brought about by the use of the Controller and MVC patterns (see below) for choosing classes.

2. The **Command** and **MVC** patterns are used. For the Command pattern, the Controller class acts as the invoquer, we have the ICommand interface and subclasses corresponding to the different commands as in the pattern (and with the execute operation also as in the pattern), and the receivers would be the Context, View and classes in the model package. It was beneficial because it decouples the controller and receivers of the commands, the controller does not need to know how the commands are to be performed or on who they act on but can just be parametrised with a generic command that handles these. Moreover, commands can be added and removed easily, without needing to change code anywhere else (e.g. in Controller). For the MVC pattern, the model, state, external, command package classes, and the Context class act as the model, the Main and Controller classes act as the controller, and the view package classes act as the view. It separates concerns about receiving and interpreting inputs; managing data, logic and rules of the application; and defining and managing how data is presented to the users. It helps with understanding, maintaining, and testing the system. Moreover, it will allow multiple developers to work on different components of the system at the same time.

3.
   - **Changes for having only one entertainment provider**:
     - EntertainmentProviderSystem is deleted (events now contain the number of tickets left and manage this directly)
     - EntertainmentProvider class is replaced by Staff class
     - Staff contains less data – only email and password for logging in
     - Organisation information (name, address, email address for payments) is now stored in Context, as it is shared by all Staff. Moreover, an organisational secret String is added in Context, as a solution to the security issue. These details are hardcoded in Main. Checks for registering duplicate organisations are removed.
     - Argument of type EntertainmentProvider is removed from the method createEvent in EventState and IEventState
     - Events are now shared across all Staff (organiser property is removed, checks for organiser are removed). This is why userEventsOnly no longer does anything for Staff.
   - **Changes for the use cases**:
     - **Register (Staff)**: RegisterEntertainmentProviderCommand class is renamed to RegisterStaffCommand class. In this class: orgName and orgAddress attributes are removed (as registration now only requires email address and password), and the orgSecret string is added. These are also reflected in the class's constructor. Other modifications will be in the execute method, to be made in the code

- **Log in**: leads to no changes in the class diagram as the functionality of the new system is the same as for the old system (i.e. here there was only a need to reverse-engineer from the old code).
- **Add Event Tags**: add new EventTag class associated to EventState with multiplicity *-1. EventTag has attributes holding a set of possible String values, and a default String value. EventState holds a map possibleTags associating Strings (names of tags) with EventTag objects (their values). Its constructors are updated to include this additional attribute and the default constructor initialises possibleTags with tags for social distancing, air filtration and venue capacity. It also includes a createEventTag method to be able to add to the map and return a new EventTag object, and a getPossibleTags method to return possibleTags. We add a new concrete command: the AddEventTags class which realises ICommand interface. It has attributes for the tag name, values, default values, the returned object of EventTag, and the usual methods. The rest of the modifications will be in the execute method, to be made in the code.
- **Create Event**: The EventTagCollection class is added to the Model package, as associated with Event with multiplicity 1-1. The Event holds an attribute of the type of this collection. The EventTagCollection class holds a map of names and values, corresponding each to an event's tag name and the value selected for it. The constructor can receive a String of form name1=value2,name2=value2 etc which it will parse to obtain this map. The createEvent methods from the EventState and IEventState have replaced the last three arguments (standing for previous 3 types of tags) with an EventTagCollection type attribute (add association EventState to EventTagCollection). The CreateEventCommand class is modified by adding an attribute of the type of the EventTagCollection class, such that events can be created with the collection of tags associated to them. Moreover, we remove the CreateEventCommand class's attributes on the default tags (because the default tags are now known by EventState which can be queried for them). The constructor is updated to reflect the above changes.The rest of the modifications will be in the execute method, to be made in the code. We will also need the map system to check the correctness of the venue address format and whether it fits within map boundaries. Therefore, we also set up a MapSystem interface with a convertToCoordinates method (taking the address as Strong and converting it to latitude and longitude) that would issue an exception if the conversion cannot be done, and an isPointWitinMapBounds method returning a boolean. We add a use dependency between the CreateEvent-Command and the MapSystem interface. We also add an attribute of type MapSystem to the Context class (creating an association between these classes).

- **Register (Consumer)**: The RegisterConsumeCommand class will only have modifications to the execute command method, to be made in the code. The map system interface was already set up with needed methods above. We add a use dependency between the RegisterConsumeCommand and the MapSystem interface.
- **Edit Profile (Consumer)**: Remove ConsumerPreferences class, as we can now benefit from the EventTagCollection class by associating the Consumer class to it. The Consumer class will hold an attribute of the EventTagCollection type. Therefore, in UpdateConsumerProfileCommand we can replace the attribute of type ConsumerPreferences with one of type EventTagCollection, and add an association to EventTagCollection. The constructor is updated to reflect the above change. The rest of the modifications will be in the execute method, to be made in the code. The map system interface was already set up with needed methods above. We add a use dependency between the UpdateConsumerProfileCommand and the MapSystem interface.
- **List Events (Consumer)**: This will involve modifications to the ListEventsCommand class, which also concerns staff listing events, but we only modify the parts about the consumer listing events. In the ListEventsCommand class, we modify the arguments of the eventSatisfiesPreferences as the ConsumerPreferences class no longer exists (see above): this method will now take the possible tags from this system as a map of String names and EventTag objects (add association with EventTag), the consumers preferences as an object of EventTagCollection (add association with EventTagCollection), and the event. The rest of the modifications will be in this method, and in the execute method, to be made in the code.
- **Book event**: The BookEventCommand class will only have modifications to the execute command method, to be made in the code. We add an integer attribute numTicketsLeft to the Event class, as this is no longer handled by an external entertainment provider system.
- **Request directions to venue**: We create a TransportMode enum which includes the different modes of transport, and we add it to the Model package. We add the GetEventDirectionsCommand class, realising the ICommand interface. It has as attributes an event number, the transport mode (object of type TransportMode and association to TransportMode), and the directions returned as a result as an array of Strings. It includes the usual methods for commands. The MapSystem interface (which we started setting up above) also needs some new methods for computing the route. We add a use dependency between the GetEventDirectionsCommand and the MapSystem interface.

  - **List Events by Distance**: Add a ListEventsMaxDistance class which extends ListEventsCommand class, because this use case shares data with the List Events use case (it takes the same inputs from the consumer) as well as functionality (filtering events by the consumer's profile preferences, whether active and/or for a certain date). The new class is given as attributes a TransportMode object (association with TransportMode), and a double maxDistance, and the inherited execute method is overriden.
  - **Review Event**: Add a Review class that is associated to both the Consumer class (1-1 multiplicity) and the Event class (1-1 multiplicity). It takes the consumer, event, creation date and time, and content as attributes. Add the ReviewEventCommand class which realises the ICommand interface. It has as attributes an event number, the String content, and the returned Review object. As methods, it has the usual methods for commands.
  - **List Reviews**: Add the ListEventReviewsCommand class which realises the ICommand interface. It has as attributes the event title for which reviews are sought and the resulting list of Review objects. It has the usual methods for commands.
  - **Export Data**: Add the SaveAppStateCommand class which realises the ICommand interface. It has as attributes the String file name and the resulting boolean of whether the operation succeeded. It has the usual methods for commands.

- **Changes for the security fix**: An organisational secret String is added in Context. RegisterEntertainmentProviderCommand class includes orgSecret as an attribute, and its execute method will now also check that the orgSecret provided in the constructor matches the one that is set up for this organisation from the context.

For changes to use cases, associations for the new command classes are presented in the solutions to the next task. It is also good to have extracted methods for the different checks in the execute methods into separate private methods used by execute. This would make the use cases more clearly covered by the solutions.

3 Task 3: New UML class model and description

3.1 Task 3.1: The class model

Figure 1 shows a high level class model, with changed classes (even if just an association to them added) indicated in yellow and new classes indicated in green. Figure 2 shows the detail of each class in terms of attributes and operations. As these will also be used for CW3, they considered all use cases and not only the required ones.

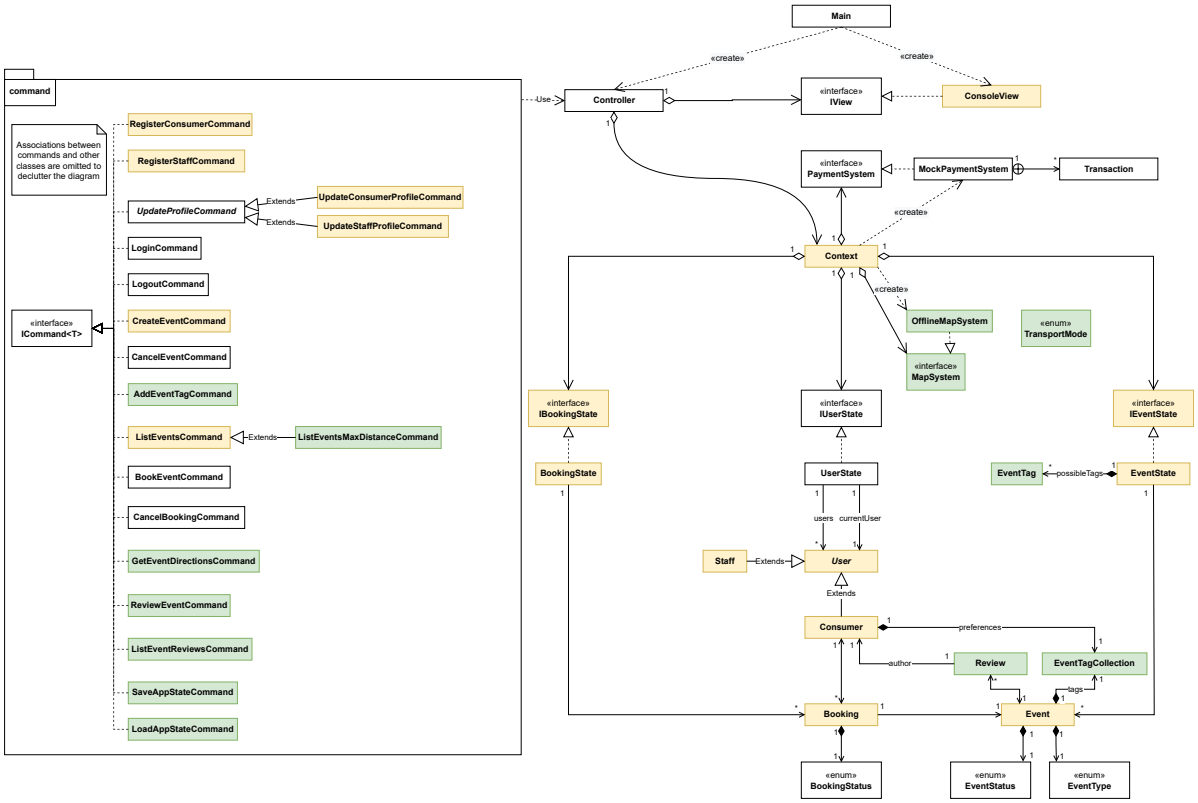

Figure 1: Class diagram of the new system: high level view

The following are the omitted associations from the command classes (where we don't include the association wih Context and the use dependency with IView which are default, and the multiplicity on the side of the command is always 1):
- RegisterConsumerCommand: UserState (1), User (*), Consumer (1), use dependency with MapSystem;
- RegisterStaffCommand: UserState (1), User (*), Staff (1)
- UpdateProfileCommand: UserState (1), User (*)
- UpdateConsumerProfileCommand: UserState (1), User (1), Consumer (1), EventState (1), EventTag (*), EventTagCollection (1), use dependency with MapSystem.
- UpdateStaffProfileCommand: UserState (1), User (1)
- LoginCommand: UserState (1), User (*)
- LogoutCommand: UserState (1), User (1)
- CreateEventCommand: UserState(1), User (1), EventState(1), EventTag (*), Event (*), EventTagCollection (1), use dependency with MapSystem
- CancelEventCommand: UserState(1), User (1), EventState(1), Event (1), BookingState (1), Booking (*), use dependency with PaymentSystem
- AddEventTagCommand: UserState (1), User (1), EventState (1), EventTag (*)
- ListEventsCommand: UserState(1), User (1), EventState(1), Event (*), EventTag (*)
- ListEventsMaxDistanceCommand: UserState(1), User (1), Consumer (1), EventState

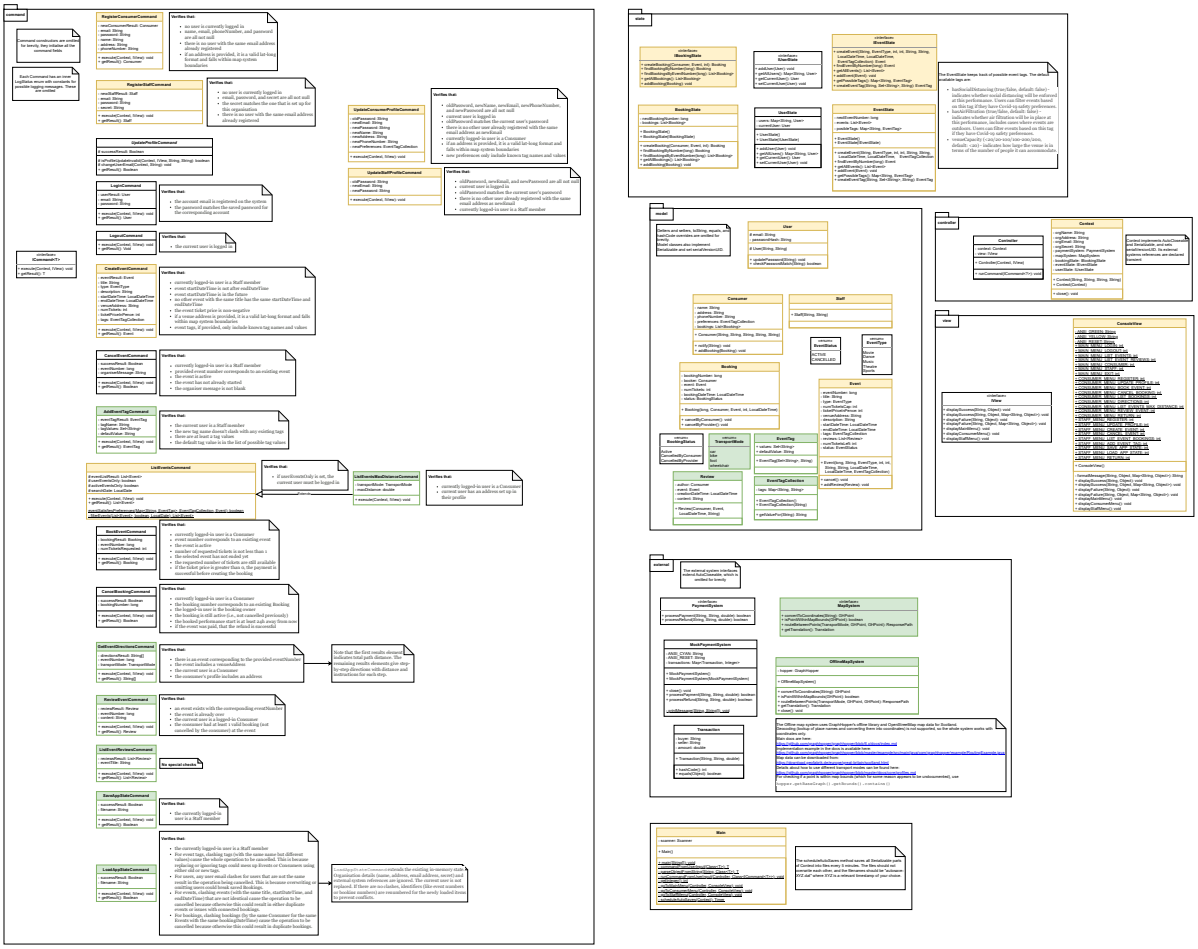

Figure 2: Class diagram of the old system: class detail

(1), Event (*), EventTag (*), TransportMode (1), use dependency with MapSystem
- BookEventCommand: UserState (1), User (1), Consumer (1). EventState(1), Event (1), BookingState(1), Booking (1), use dependency with PaymentSystem
- CancelBookingCommand: UserState(1), User (1), Consumer (1), BookingState (1), Booking (1), Event (1), use dependency with PaymentSystem
- GetEventDirectionsCommand: TransportMode(1). UserState(1), User (1), Consumer (1), EventState (1), Event (1), use dependency with MapSystem
- ReviewEventCommand: EventState(1), Event(1), UserState(1), User (1), Consumer (1), Booking (*), Review (1)
- ListEventReviewsCommand: EventState(1), Event (*), Review (*)
- SaveAppStateCommand: UserState (1), User (1)
- LoadAppStateCommand: UserState (1), User (1), EventState (1), EventTag (*)

3.2 Task 3.2: The description

- Here, expecting clarification of anything which was not already clear in the diagrams.
- Here, expecting similar explanations and examples as in Task 2.1, but referring to the new classes.

4 Task 4: UML Sequence Diagrams

Solutions for the sequence diagrams of the 4 required use cases are available to download as a .zip file from the following link, and will then open in the browser: sequence diagrams. Instead of showing a found message, it was good to instead show the sender being an object of class Controller.

5 Task 5: The Software Development

1. The answer is plan-driven software development process. Reasons: Design is a separate stage; it is perfected (as were requirements) before we proceed with implementation; outputs from design are used to plan implementation; the documentation that we produce is heavyweight; we use UML formally. Reasons why it is not agile: in agile, design is interleaved with implementation and requirements; documentation is lightweight; informal documents or design documentation is automatically generated by programming environment; Outputs of design may not be specification documents, but reflected later in the code; UML may be used but informally and to facilitate communication within the team.
2. The answer is Big Design Up Front (BDUF). This is because we complete and perfect the design before we start the implementation in CW3; a lot of time is spent in doing design properly and thoroughly documenting it. This will however result in doing extra things in the design than just in the original requirements, just for the fear that customer may keep changing or updating requirements (just like at the start of CW2) or things may change in the environment or systems that we use, which can be wasteful as we cannot predict all changes. This is against 'You Ain't Gonna Need it' (YAGNI), which advocates for not overengineering a design just because you think you may need some things later. Because we engineer the whole design, we also don't think of keeping solutions 'minimal' so that they could work quickly, which is what "Do the simplest thing that could possibly work (DTSTTCPW):" advocates.
3. Here, a discussion is expected. There are arguments for using YAGNI and/or DTSTTCPW for being able to deliver the solution quickly, be early on the market, attract customers and have a chance to beat competition. However, this depends on how great the demand for such a system is in the market, and the company's knowledge of its competition. There are also arguments for using them so that

we could prototype a solution and get feedback from sponsors and potential users, which would ultimately help it respect their needs the best way possible, before we deploy it. The requirements seem to still be in transition, as the updated requirements document still includes ambiguities, and these approaches are much better suited to being flexible to changing requirements. However, this will depend on the availability of the sponsors for more meetings, and for the company's success in finding potential future users to try out prototypes. There are also disadvantages: YAGNI and DTSTTCPW may lose sight of the architecture and essence of the design, and not build flexible components and frameworks right from the start, which are questionable decisions and may break the design. If urgency to launch is not of the essence, the possibilities to get a good amount of feedback from stakeholders and sponsors are slim, and moreover the requirements are now considered complete, BDUF can work better because it allows focusing all efforts on building a good design.

Borislav Ikonomov, Vidminas Vizgirda, Cristina Alexandru, 2023.