Inf2-SEPP 2021-22

Coursework 2

Creating a software design for an events app during the COVID-19 pandemic

SAMPLE SOLUTION

1 Task 1: System State

Components of the system state suggested by the coursework instructions include the following:

- **Consumer:**
  - ◇ full name
  - ◇ email address
  - ◇ phone number
  - ◇ password
  - ◇ social distancing preferred (true, false)
  - ◇ air filtration preferred (true, false)
  - ◇ outdoors only preferred (true, false)
  - ◇ preferred maximum capacity
  - ◇ preferred maximum venue size
  - ◇ bookings

- **Entertainment provider:**
  - ◇ organisation name
  - ◇ organisation address
  - ◇ phone number
  - ◇ name of main representative
  - ◇ email of main representative
  - ◇ payment account email
  - ◇ password (attached to the main representative's email)
  - ◇ list of names and emails of other representatives
  - ◇ events
- **Event:**
  - ◇ event number
  - ◇ entertainment provider
  - ◇ title
  - ◇ type (music, theatre, dance, movie, sports)
  - ◇ status (active, cancelled)
  - ◇ ticketed (true, false)
  - ◇ ticket price
  - ◇ number of tickets
  - ◇ event performances
  - ◇ sponsorship request
  - ◇ entertainment provider
- **Sponsorship request:**
  - ◇ request number
  - ◇ event
  - ◇ status (pending, accepted, rejected)
  - ◇ percentage of ticket price
- **Event performance:**
  - ◇ event

- ◇ date and time
- ◇ venue name
- ◇ venue address
- ◇ performer names
- ◇ social distancing (true, false)
- ◇ air filtration (true, false)
- ◇ outdoors (true, false)
- ◇ capacity
- ◇ venue size

- **Booking:**
  - ◇ booking number
  - ◇ performance
  - ◇ number of tickets
  - ◇ status (active, cancelled by consumer, cancelled by entertainment provider, payment failed)
  - ◇ amount paid
- **Government representative:**
- **Entertainment provider system:**
- **Payment system:**

Notes:

- The ways different entities are connected could also be always seen as bidirectional. Here, it is sometimes unidirectional because only relationships that are needed for the functionality of the system were kept.
- The sponsorship request could initially be seen as a piece of information (true or false) within the event, however the government should be able to see all requests and therefore it is clearer if it is a separate entity. Moreover, it benefits from a number of information attached to it.
- Ticket original price and percentage of ticket price which is discounted are best separated (ticket original price should be kept separate for reporting purposes), but the alternative will also be accepted.
- As not obvious from the requirements document, events, bookings and sponsorship requests need a status. One could originally think that cancelling an event or booking would delete it however this is not a good idea for reporting purposes.

Sponsorship requests (no matter if considered as a separate entity or information kept within event) would also benefit from distinguishing between the pending and addressed ones such that the government could filter to see only pending ones.

- Events and bookings have numbers according to use case descriptions. For consistency, sponsorship requests also have a number which identifies them uniquely and can be used to reference them. This is not needed if sponsorship requests are seen as information belonging to the event only.
- Splitting the Event entity into 'Ticketed' event and 'Non-ticketed event' is also acceptable.
- R2 said that created events have "date(s), time(s) and venue(s)", and "Additional advice document" clarified that a booking is for a certain date and time. Moreover, one could imagine Covid-19 measures to differ by the performance and its venue. Therefore, an event performance entity becomes necessary. However, because this was not very clear in the requirements document, we will accept the following alternative interpretations:
  - Each event being in fact a performance with a date, time and venue
  - The event entity having all dates, times and venues as information and being bookable as a whole (e.g. booking an entire festival); however, in this case the following class diagram should make it clear how the performances are distinguishable (e.g. use arrays or maps where position x in each shows date, time and venue for that performance, or use 'Collection' but explain how this would be achieved).
- Venues will not be accepted as separate entities because of "R16. Venues shall not be set as a separate entity in our app" from the Requirements document.
- The amount paid for a booking needs to be stored such that consumers who bought tickets before sponsorship is introduced can be refunded correctly.
- A government representative is apparent as a needed entity, because they need to be able to view and approve of sponsorship requests. No information was added for them here as it cannot be extracted from the requirements document, however having usual user information like name, email, password, is acceptable.
- No information is needed for the external systems as our system communicates with them only through messages.

2 Task 2: UML class model

A class model is shown in Fig. 1 (showing only class associations) and Fig. 2, Fig. 3 and Fig. 4 (showing the fields and operations that would go with the classes). Please

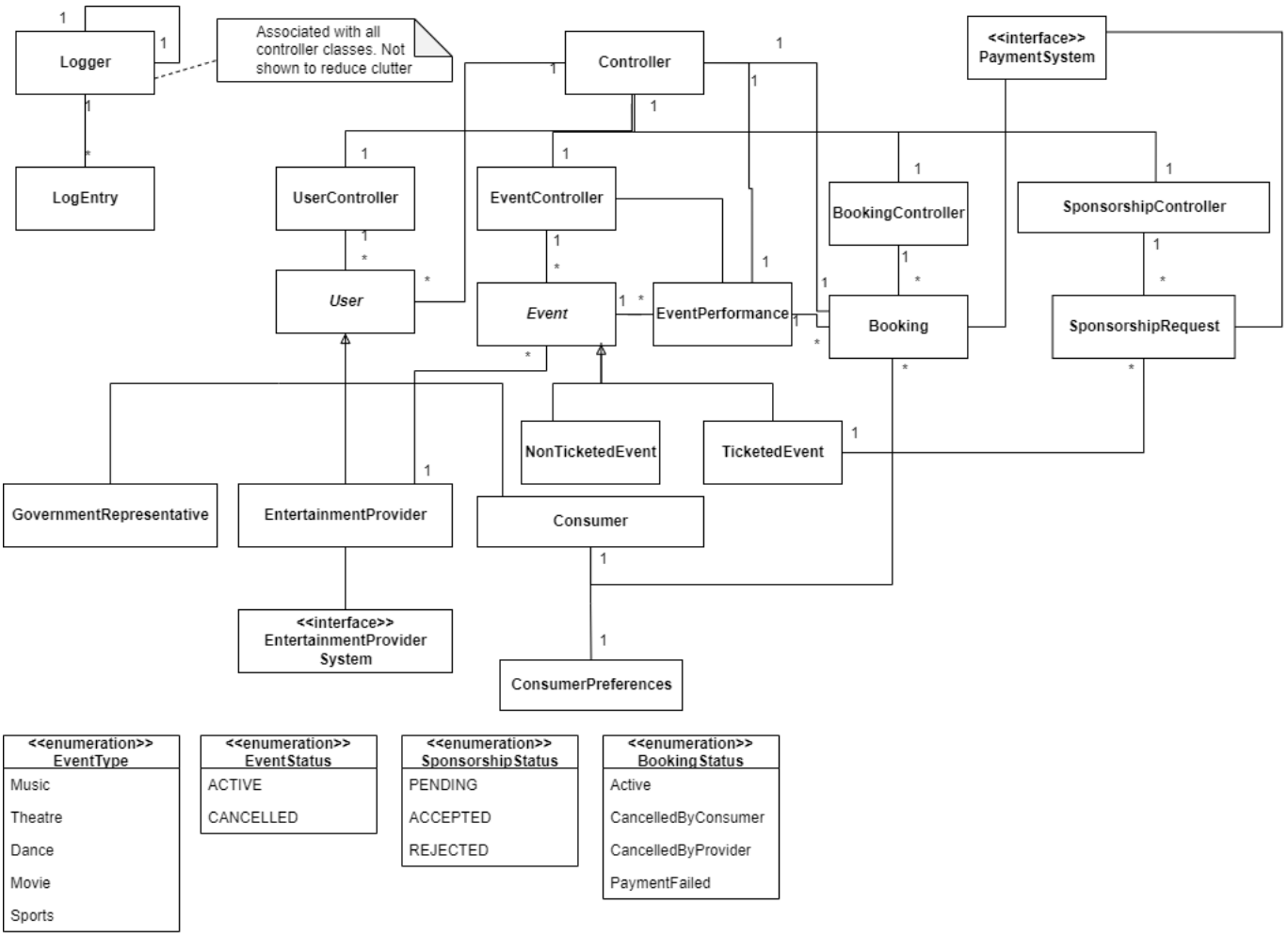

Figure 1: UML Class Diagram- high level view

note that the fields do not contain those which are objects of other classes, which are represented through associations.

Controller

+ registerConsumer (name: String, email: String, phoneNumber: String, password: String, paymentAccountEmail: String): void

+ registerEntertainmentProvider(orgName: String, orgAddress: String, paymentAccountEmail: String, mainRepName: String, mainRepEmail: String, password: String, otherRepNames: Collection<String>, otherRepEmails: Collection<String>): void

+ updateConsumerProfile(password: String, newPreferences: Preferences): void

+ createTicketedEvent(title: String, type:EventType, title: String, numTickets: int, ticketPrice: double, requestSponsorship: boolean): void

+ createNonTicketedEvent(title: String, type: EventType): void

+ addEventPerformance(eventNumber: long, venueAddress: String, startDateTime: LocalDateTime, endDateTime: LocalDateTime, performerNames: Collection<String>, hasSocialDistancing: Boolean, hasAirFiltration: Boolean, isOutdoors: Boolean, capacityLimit:int, venueSize: int): void

+ listEventsonGivenDate(searchDateTime: LocalDateTime): void

+ viewEvent(eventNumber: long): void

+ bookEvent(eventNumber: long, performanceNumber: long, numTicketsRequested: int): void

+ cancelEvent(eventNumber: long, organiserMessage: String): void

+ cancelBooking(bookingNumber: long): void

+ listEventBookings(eventNumber: long): void

+ respondSponsorship(requestNumber: long, percentToSponsor: int): void

UserController

- currentUser: User

+ addUser (user: User): void

- registerGovernmentRepresentatives() : void

+ checkCurrentUserIsConsumer(): void

+ checkCurrentUserIsGovRep(): void

+ checkTicketsLeftwithEP(eventNumber: long, performanceNumber: long): User

- getEntertainmentProviderOwningEvent(eventNumber: long): EntertainmentProvider

- getEntertainmentProviderWithSubmittedRequest(requestNumber: long): EntertainmentProvider

+ notifyEPofNewBooking(eventNumber: long, performanceNumber: long, bookingNumber: long, consumerName: String, consumerEmail: String, numTicketsRequested: int): void

+ notifyEPofCancelledBooking(eventNumber: long, bookingNumber: long): void

+ notifyEPofPositiveSponsorshipDecision(requestNumber: long, percentToSponsor: int): void

+ notifyEPofNegativeSponsorshipDecision(requestNumber: long): void

+getAllUsers() : Map<String, User>

EventController

- nextEventNumber: long

- nextPerformanceNumber: long

- getEventByNumber(eventNumber: long): Event

+ createNonTicketedEvent(organiser : EntertainmentProvider, title : String, type : EventType) : NonTicketedEvent

+createTicketedEvent(organiser : EntertainmentProvider, title : String, type : EventType, ticketPrice : double, numTickets : int) : TicketedEvent

+createEventPerformance(event : Event, venueAddress : String, startDateTime : LocalDateTime, endDateTime : LocalDateTime, performerNames : List<String>, hasSocialDistancing : boolean, hasAirFiltration : boolean, isOutdoors : boolean, capacityLimit : int, venueSize : int) : EventPerformance

+ checkEventandPerformanceBookable(eventNumber: long, performanceNumber: long, numTicketsRequested): EventPerformance

+ getEntertainmentProviderAcccountEmailforEvent(eventNumber: long): String

BookingController

- nextBookingNumber: long

+ findBookingsByEventNumber(eventNumber: long): Collection<Booking>

+createBooking(booker : Consumer, performance : EventPerformance, numTickets : int, amountPaid : double) : Booking

+ checkBookingWithNumberActionable(BookingNumber: long): Booking

- getBookingByNumber (bookingNumber: long): Booking

SponsorshipController

-nextSponsorshipNumber: long

+addSponsorshipRequest(event : TicketedEvent) : long

- getRequestByNumber(requestNumber : long) : SponsorshipRequest

+ checkCorrectSponsorshipRequest(requestNumber: long, percentToSponsor: int): bool

+ acceptSponsorship(sponsorshipNumber: long, paymentAccountEmail: String): void

+ rejectSponsorship(sponsorshipNumber: long): void

Figure 2: UML Class Diagram- classes in detail 1

Figure 3: UML Class Diagram- classes in detail 2

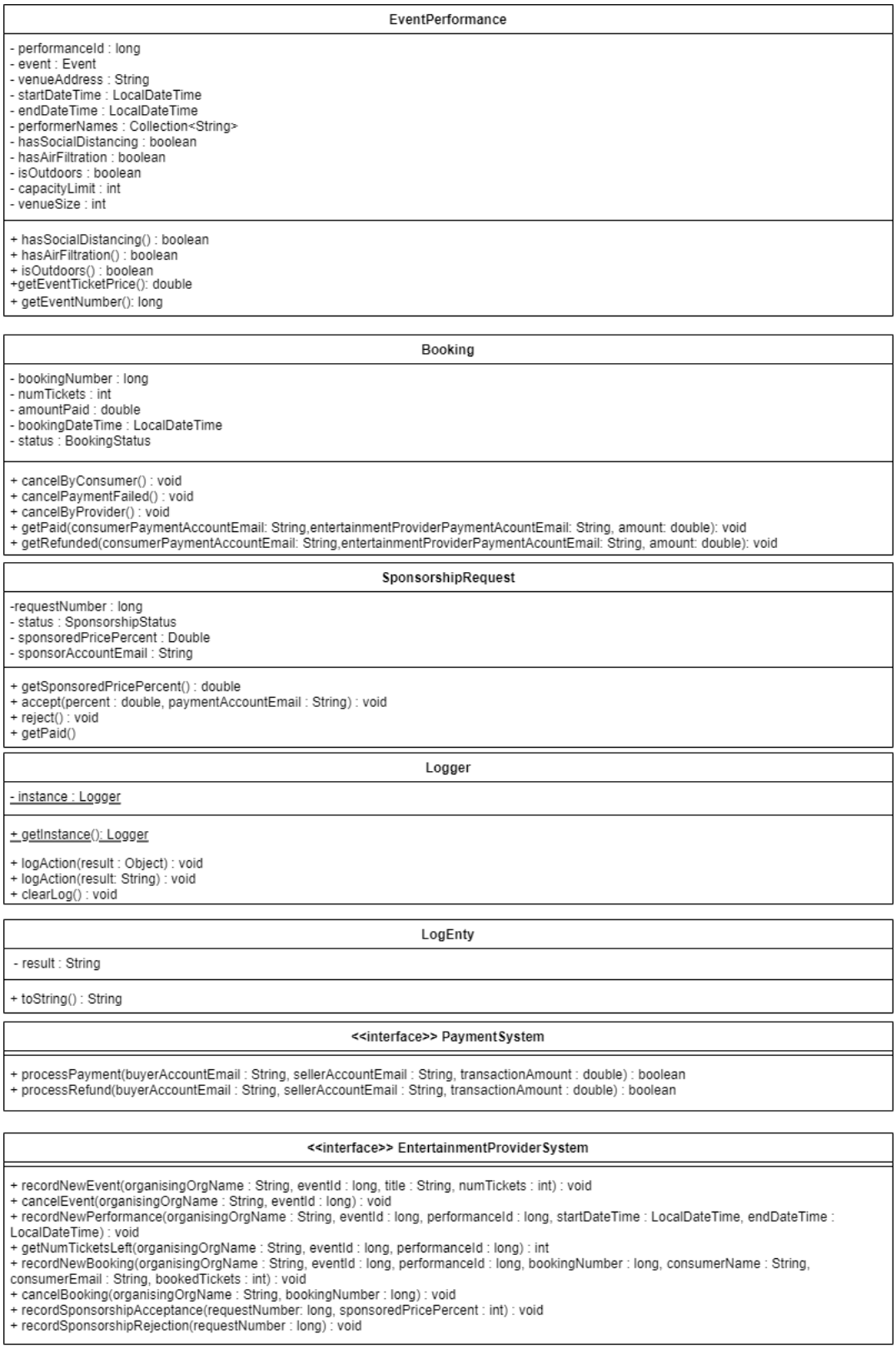

Figure 4: UML Class Diagram- classes in detail 3

3 Task 3: High-level description of the UML class model

The class model includes:

- The Controller class which launches the operations involved in the required use cases and as part of them delegates responsibilities to the other controllers. It must also be associated to User, Booking and EventPerformance because objects of these classes must be passed on to it when handling bookings: e.g. to create a new booking, one must know the event performance and its consumer which are needed for the constructor, moreover the booking must then be passed to the consumer to add it to its collection of bookings.
- Controller classes which manage collections of the different core entities in the system, such that the data could be used effectively for different use cases. They are delegated roles by the main Controller.
- Classes corresponding to each of the entities presented in Task 1, and associated as conceptually logical in the real world. Their attributes mostly correspond to the information included in those entities, with the difference being a few attributes which facilitate the implementation.
- Interfaces to the external systems (respects separation of interface and implementation design principle): While for the payment system the bookings and sponsorship requests would be logically the ones that lead to payments and refunds being made, there can be several entertainment provider systems and so their interfaces are associated to the EntertainmentProvider class (see next item regarding such 'associations').
- Most correctly, the relationship between our system's classes and the interfaces needed to be denoted with a dotted line with `<<use>>` on top, but as this notation was not taught in this course we accept associations like in the figure.
- The Event and User classes are abstract classes, extended by classes for specific types of users and events. This is because while there are things in common between them, there are also attributes and operations that are specific for each type of user and event. The controllers can use the generic User/Event types instead of needing associations with each of the extending classes, and this reduces coupling.
- Each class has distinct responsibilities and well-defined interfaces given by their public operations (decomposition, modularisation design principles).
- An attempt is made at keeping high cohesion by delegating responsibilities to the right classes (e.g. controllers to handle collections of objects, objects which are entities to handle information about themselves).
- Even if sometimes being able for objects to communicate would have simplified the implementation, associations were kept to those that are conceptually correct and

those that are absolutely needed for the functioning of the system. Moreover, when only certain information (e.g attribute values) about an object several steps in a chain of associations was needed, that information, and not the entire object at the other end, is passed between the objects, to avoid creating additional associations (Law of Demeter, also see sequence diagrams). This is all meant to keep coupling low.

- Attributes are always private and only the needed (public) getter and setter operations for them (not shown in class diagram) would be used. Moreover, only those operations that must be accessible by other objects are public, while operations for internal processing are private. These help respect the design principles of abstraction and encapsulation
- A Logger class with log entries is used to log all the outcomes of an operation, which can then be all displayed at once to the user rather than having to send different types of notification through operation returns.
- The Logger class is using a singleton pattern, which makes sure this class only has an instance which is globally accessible and thus sharable by all other classes which need to write to the log. While singleton is problematic for testing, this is addressed here by having an operation for clearing the log which can be used between tests.
- Singletons could have also been used for keeping event, booking and sponsorship numbers which are correctly kept unique system-wide. However, these would have posed difficulties when testing (objects would have had numbers associated to them after each test), and moreover can be easily handled by the controllers storing as an attribute the next number to use.
- Other design patterns which could have been used were facade and command, however these were not taught in the course so there was no expectation to use them.

4 Task 4: UML sequence diagram

The 4 required sequence diagrams from the coursework instructions are provided in Fig. 5, Fig. 6, Fig. 7 and Fig. 8.

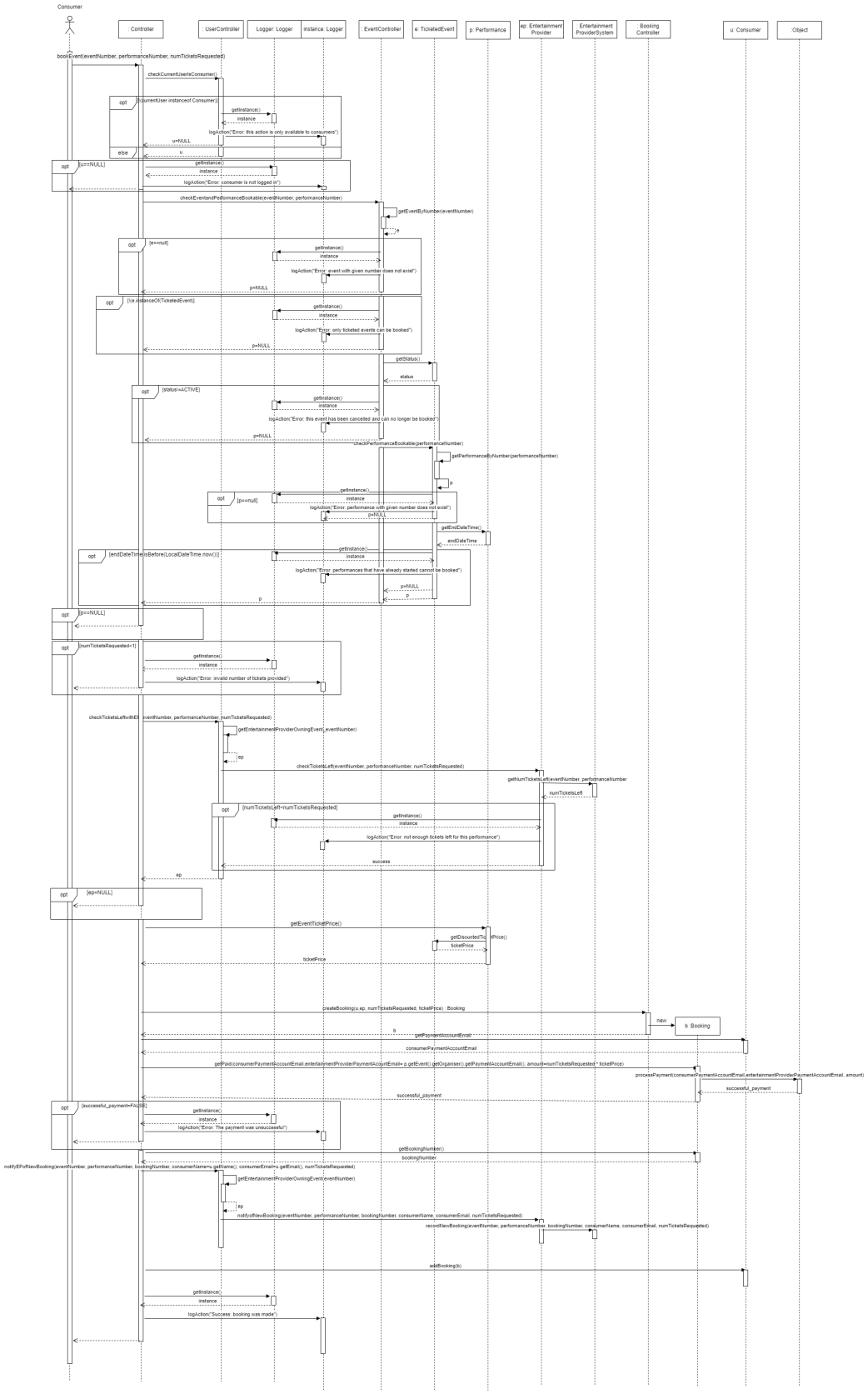

Figure 5: UML Sequence Diagram for “Book Event”

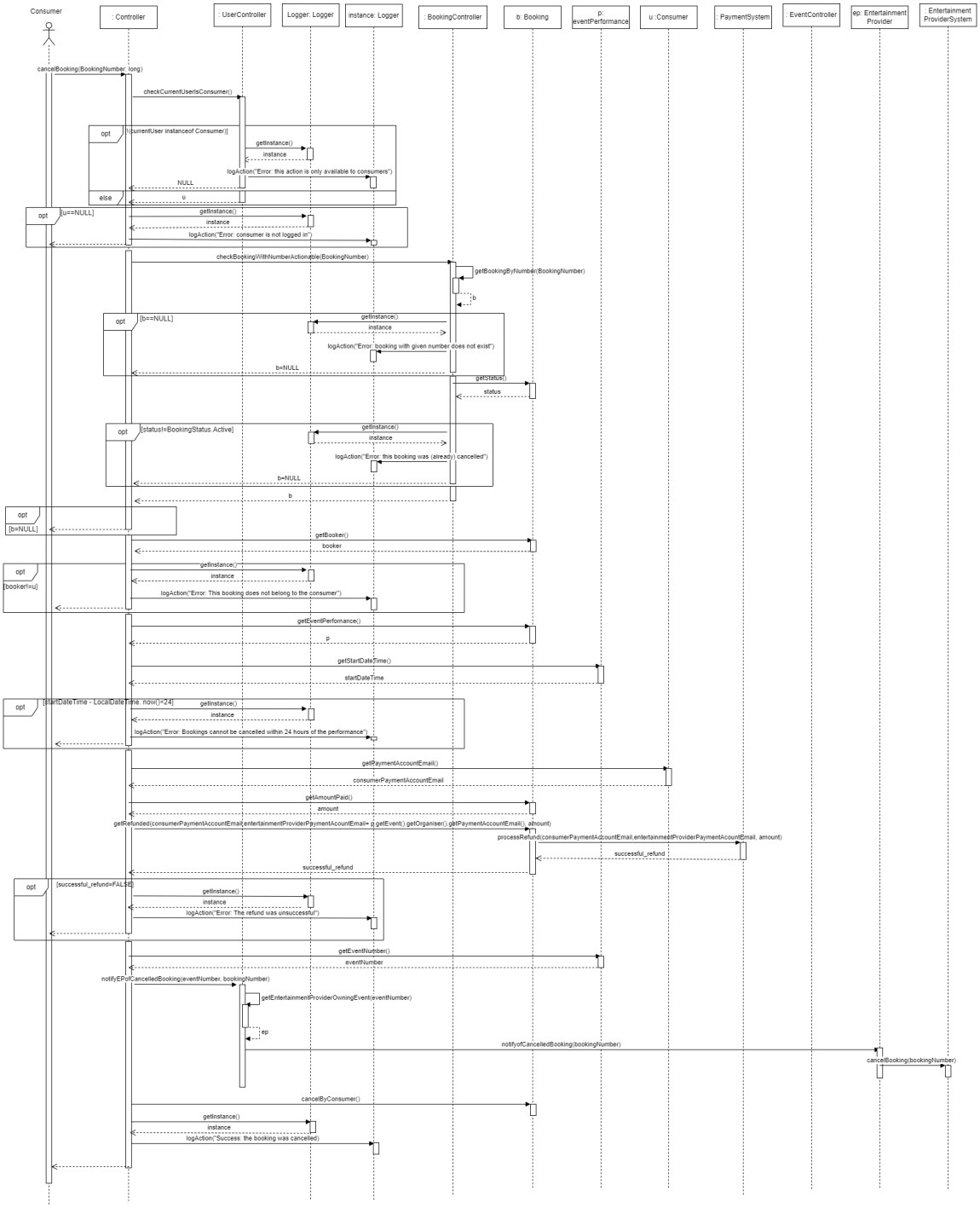

Figure 6: UML Sequence Diagram for “Cancel Event”

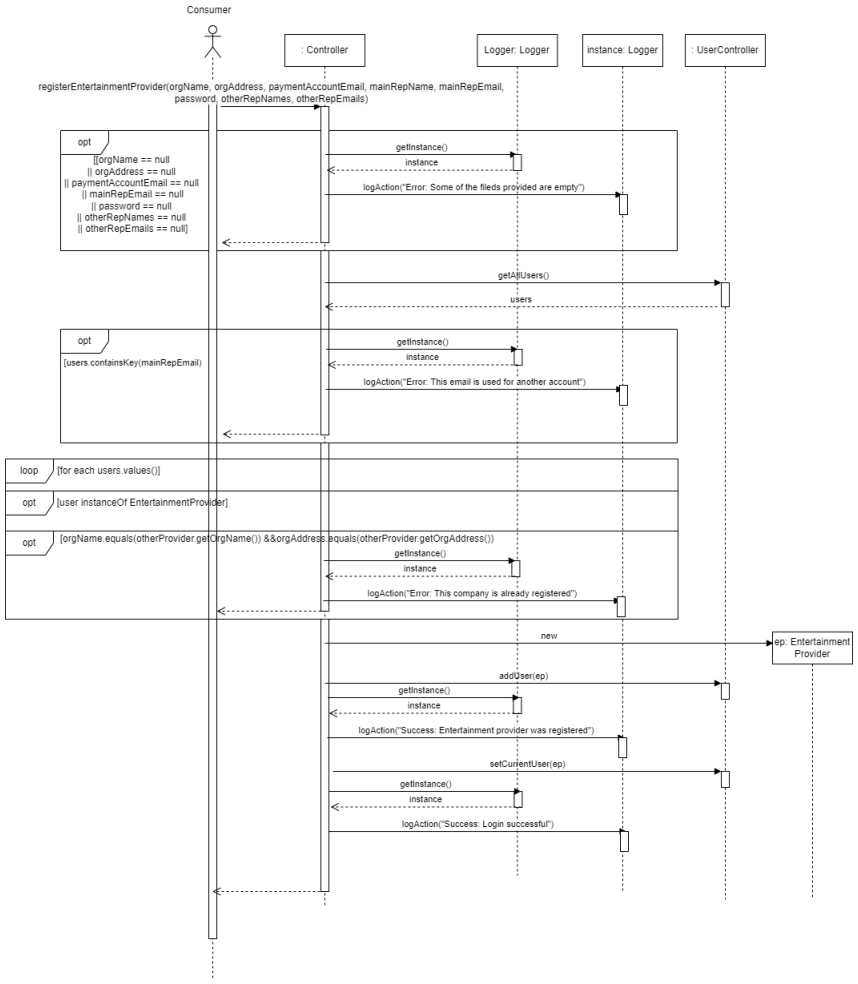

Figure 7: UML Sequence Diagram for “Register Entertainment Provider”

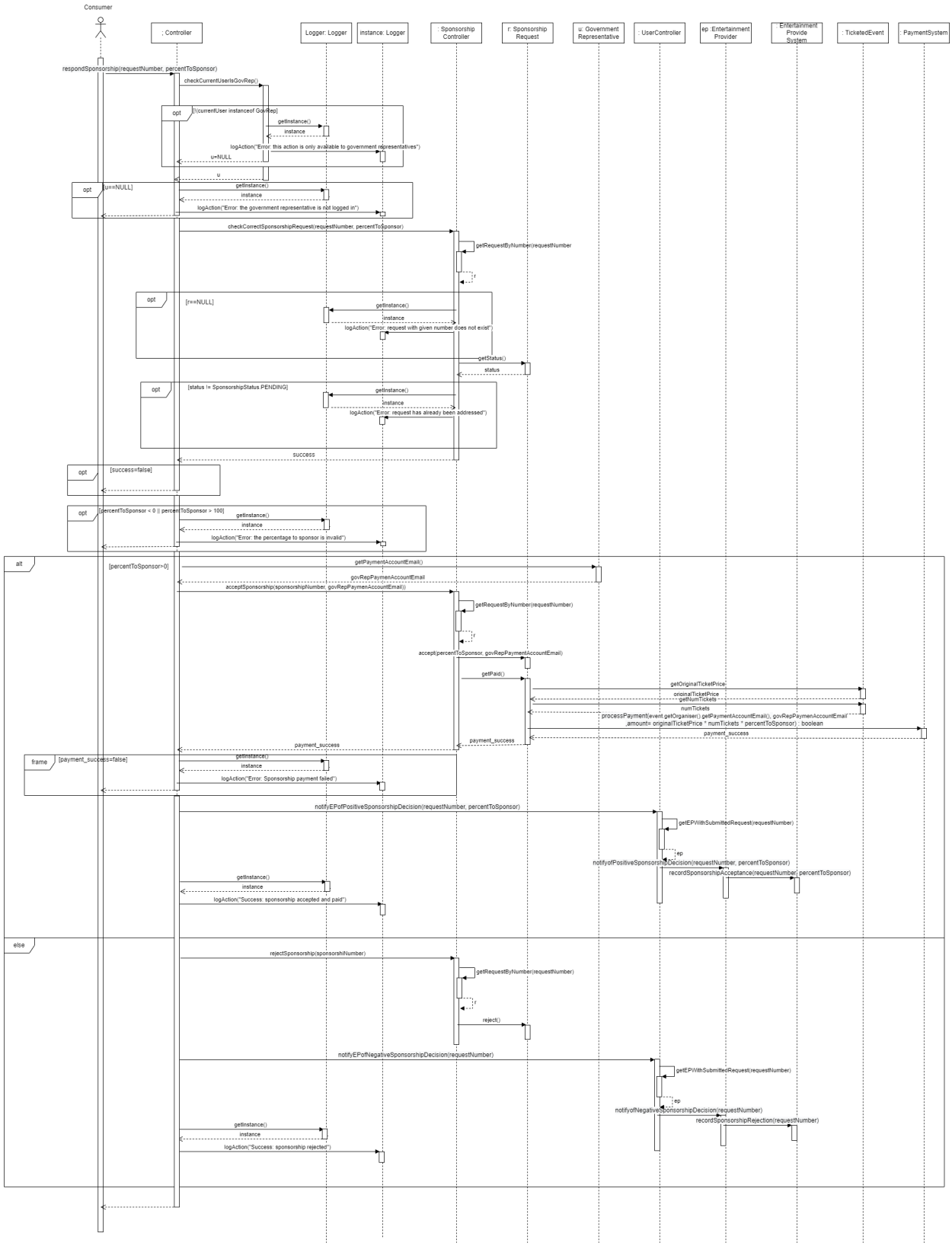

Figure 8: UML Sequence Diagram for “Accept/reject sponsorship”

5 Task 5: The software development

a) The answer is plan-driven software development process. Reasons: Design is a separate stage, and not interleaved with implementation and requirements like in agile; 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 more heavyweight than in agile; we use UML formally and not just for communication like in agile.

b) 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 (e.g. in this solution, we have foreseen a need for producing reports which is why we kept ticket original prices), 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.

c) Here, a discussion is expected. There are arguments for using YAGNI and/or DTSTTCPW for being able to deliver the solution quickly, especially considering the Covid-19 context and the need for such a system. There are also arguments for using them so that we could prototype a solution and get feedback from the customer, 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 also depend on the customer: if the government considers the requirements stage completed and only want us to deliver a final solution which matches these requirements in detail, and are unavailable for providing feedback on intermediate prototypes, BDUF can work better because it allows focusing all efforts on building a good design. On the other hand, 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.

6 Task 6: The system architecture

Client-server or (even better) service-oriented architecture expected (also microservices architecture accepted). Reasons:

- Need for different views of the system via different types of interfaces, which can be ensured by different clients

- There is a clear need for services provided by different providers like restaurants, hospitality providers, taxi and other transportation providers.
- Need to easily add more system components corresponding to new providers.
- Need for scalability (needs to be used by a large number of users and systems).
- Need for resilience, as the system is to be tracing the COVID-19 spread and notify the NHS and consumers if anybody got infected, which can have vital implications.