



# Software Engineering: TMA 02

MATTHEW MASON  
C6122243

# Contents

- Question 1..... 3
  - a..... 3
  - b..... 3
  - c..... 4
  - d..... 5
  - e..... 7
- Question 2..... 8
  - a..... 8
  - b..... 8
    - i..... 8
    - ii..... 8
    - iii..... 8
    - iv..... 8
  - c..... 8
  - d..... 8
  - e..... 9
- Question 3..... 10
  - a..... 10
    - i..... 10
    - ii..... 10
    - iii..... 10
    - iv..... 11
  - b..... 11
- Question 4..... 12
  - a..... 12
  - b..... 12
    - i..... 12
    - ii..... 12
- Question 5..... 13
  - a..... 13
    - i..... 13
    - ii..... 13
    - iii..... 13
    - iv..... 13
  - b..... 14



## Question 1

a.

Candidate class name	Keep (yes/no)?	Reasons for keeping or discarding candidate class
Password, E-mail Address, Name, Payment details, Card Details	No	Attribute of advertiser or attendee
Advertise Event	No	An event is created by the advertiser so this could be an association
Advertiser	Yes	A person is required to be a registered user
Attendee	Yes	Details need to be kept for those attending an event and to confirm tickets match
Browse Event, Cancel Event, Check History, E-mail Ticket, pay within 5 days, Refund, Add Payment Method, Change payment method	No	Behaviour of the system
Category of Event, End Time, Free to Attend, Location, Max Number of Attendees, Paid Event, Start Time, Text description of event, Ticket price, Title of event, date	No	Attribute of event
Commission, Software System, Website	No	Outside scope of iteration
Event	Yes	Abstract entity
Ticket	Yes	Has a lifespan

b.

Having only the software description available means it's difficult to get a sufficient understanding of the problem and doesn't provide enough information to discern a detailed representation of the problem domain for the first iteration. If use cases or user stories existed then it might be possible to identify additional classes, associations and attributes. Additional artefacts would also help determine and identify the lifespan of objects through sequence diagrams and the potential discovery of problems that require the model to be altered. The description of the problem doesn't specify how payment to advertisers is managed, for example, is it part of a separate system or does the new system need to handle it? and not having a domain expert to talk to means assumptions have to be made about things. This also means that certain words or phrases can't be defined and clarified as these might have specific meanings within the domain being modelled. However, for the purpose of the BillyDoo Software System, conducting a grammatical parse of the problem allowed me to establish a list of possible classes and their attributes and asking myself questions such as 'would this element have a lifespan?' along with modelling object diagrams, I was able to create a very basic model of the domain.

(211 words)

C.

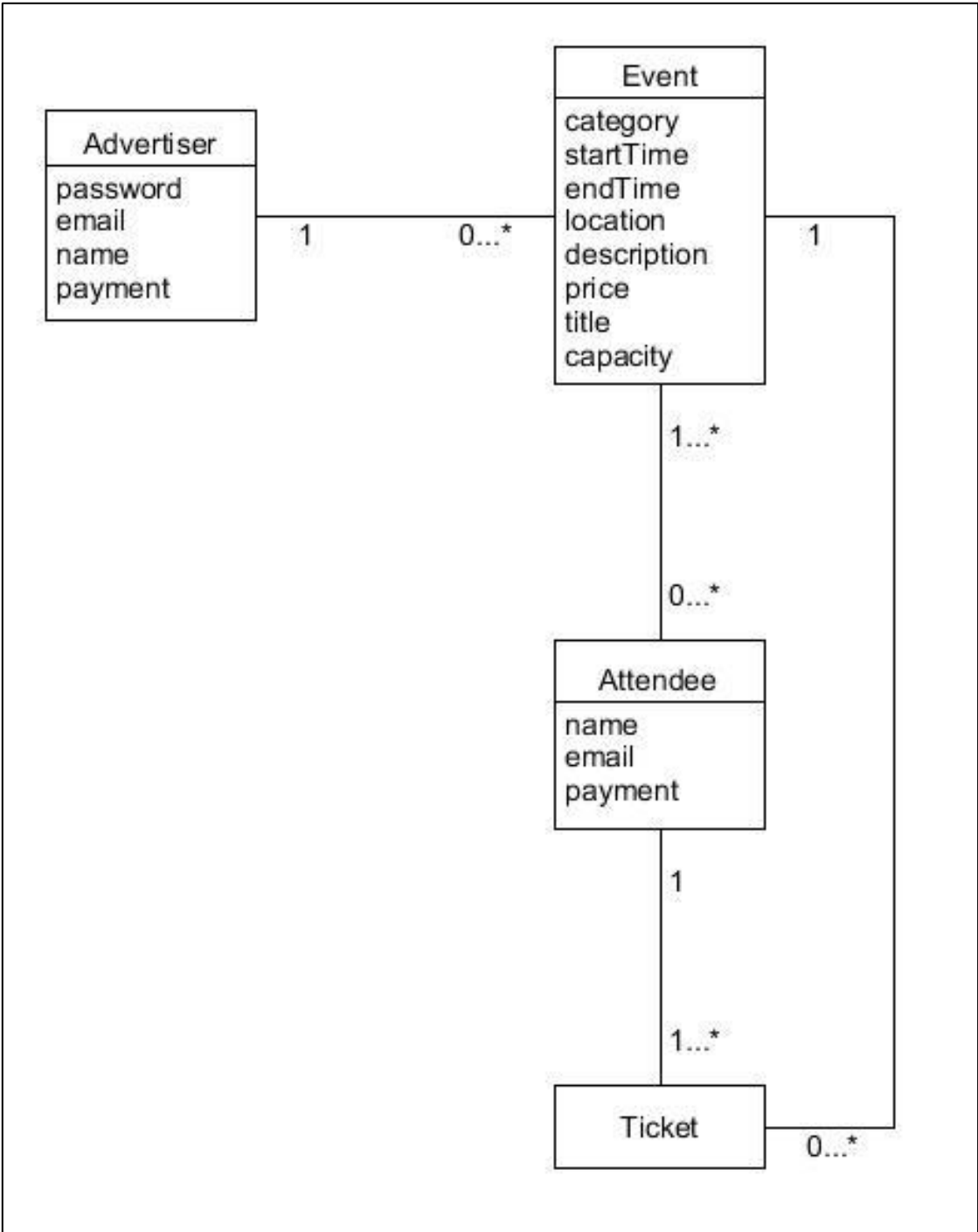



Figure 1: My initial class model

d.



**Matt Mason**  
25/02/20, 18:46

██████████

good job on the first attempt, it shows a good understanding of the problem. Most of the classes seem appropriate and give a general view of the problem from the description. Adding a cancelled attribute to the event class shows that you've considered the lifespan of the class and how it relates to an advertiser viewing the history of events, you could even add an attribute to count the number of people who purchased tickets as this is also part of viewing the history.

In your updated model I'd consider the following:

Potentially removing the BDSS class as it's outside the scope of the problem.

I'd also remove the category class and have it as an attribute of the event class as the advertiser simply needs to select an option from a pre-set list. I like the idea of the Email class but it might also be possible to exclude this too as its more of a behaviour of the system and doesn't say in the description that the email needs to be recorded in any way.

I would include ticket as a class as it has a lifespan from when its purchased to when the event is and finally i think the event-user link should have a multiplicity of 0...\* at the user end as even though an event might exist, there is a possibility that no-one buys a ticket for it.

Hope that helps and good luck.  
Matt


[Reply](#)  [Like comment](#) [Delete comment](#)

Figure 2: My comment to ██████████

**Matt Mason**

25/02/20, 19:02

Hi [REDACTED]

well done on the first attempt, from looking at your diagram its very simple to understand the links, how each class relates to others and gives a good understanding to the domain in general. I think the addition of the cancel attribute in the event class is something i missed and is a good addition to your diagram as it relates to the history section of the description. I'd also add an attribute to show tickets purchased as this was specifically mentioned in the description too. Personally, I'd change the attributes for the Ticket class to simply be 'name' and 'email' as the problem doesn't specify ticket numbers and it would be easier to check these against the users details. I'd also remove what the type of attribute is as that relates more to the designing phase, for example startDate could be a string or integers and at this point its difficult to know which is the best choice. I'd add a link between User and Event and finally i think the multiplicity at the ticket end of ticket-user should be 1...\* as if a user is attending an event they must have a ticket so it can't be 0, hence the need for a link between event and user.

Hope the feedback helps you with your updated model and good luck!

Matt

Reply



0 Like comment

Delete comment

Figure 3: My comment to [REDACTED]



e.

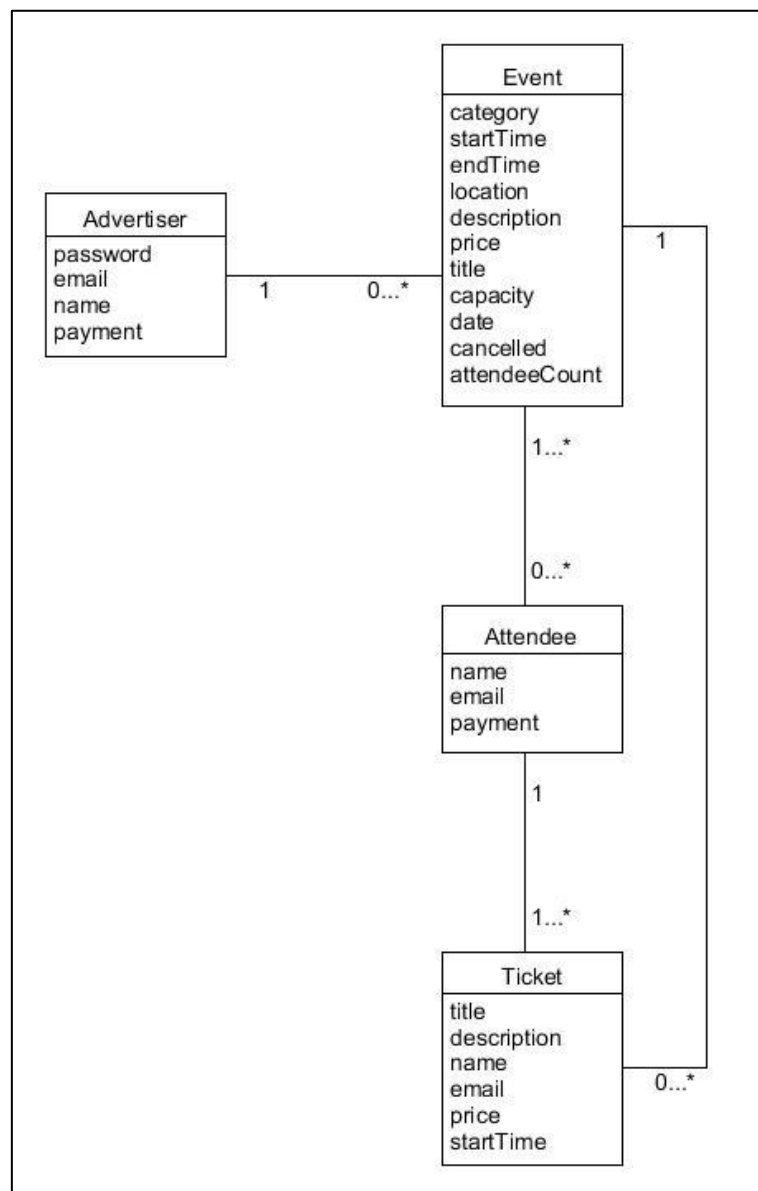


Figure 4: My updated class model

### My Updated Model

Based on feedback I have only made a small number of changes from my initial model. Firstly I added a cancelled and attendeeCount attribute to the Event class. This is due to the fact that advertisers need to view the history of all their past events and the problem description specified that they need to see if an event has been cancelled and how many people purchased a ticket for each event. I also added name, email, title, description and price attributes to the Ticket class as this will help to identify and match the attendee to the ticket upon entering the event and make sure no-one is accessing the event who doesn't have the correct ticket. I chose not to model the generalisation of advertisers and attendees as this is something that should be done when designing the implementation.

Archive post



## Question 2

a.

An alternative representation would be to replace the type attribute in the session class with three separate classes to represent sessions for mothers and babies, children and autistic children. The advantage of this is that attributes could be added during a later iteration determining what defines each type, for example, a child could be defined as being between a lower and upper age limit however a domain expert would need to be consulted to determine this. The disadvantage is that it complicates the diagram. Each class would need to convey what type of session it is in the class name which needs to be unique for each one and this name could end up being long or not clearly representing what the class does. It also creates more links between classes which need to be considered along with their multiplicities. Making each type a class also makes assumptions about the meaning and in turn the diagram could be interpreted incorrectly. Finally, as it's not a tangible object and nothing needs to be recorded about it during this iteration, it would be better to have type as an attribute of session.

b.

i.

```
context FilmForYou
self.member.ticket.s2.type
```

ii.

```
context FilmForYou
self.m1.ticket.session.film.title
```

iii.

```
context FilmForYou
self.s2.film.title
```

iv.

```
context FilmForYou
self.session -> select(s | s.date = "d1" and s.time = "t1").film.title
```

c.

For the set of sessions, the capacity of the room the session takes place in must be equal to or greater than the number of tickets associated with the session.

d.

To represent the need for members to be able to leave feedback about sessions and films we can add an additional class to the diagram with the name 'Feedback'. We would add this as a class because information needs to be recorded about either the session, the film or both and will need to be reviewed at some point. The Feedback class would link to the class Film with a multiplicity of 0..1 at the Film end, and 0..\* at the Feedback end. It would also link to the Session class with a multiplicity of 0..1 at the Session end and 0..\* at the Feedback end. The final link would be between Feedback and Member classes with the multiplicity of 0..1 at the Member end and 0..\* at the Feedback end. This configuration allows members to leave feedback about the session or film while choosing to remain anonymous if they so choose. The class model below shows this new configuration.

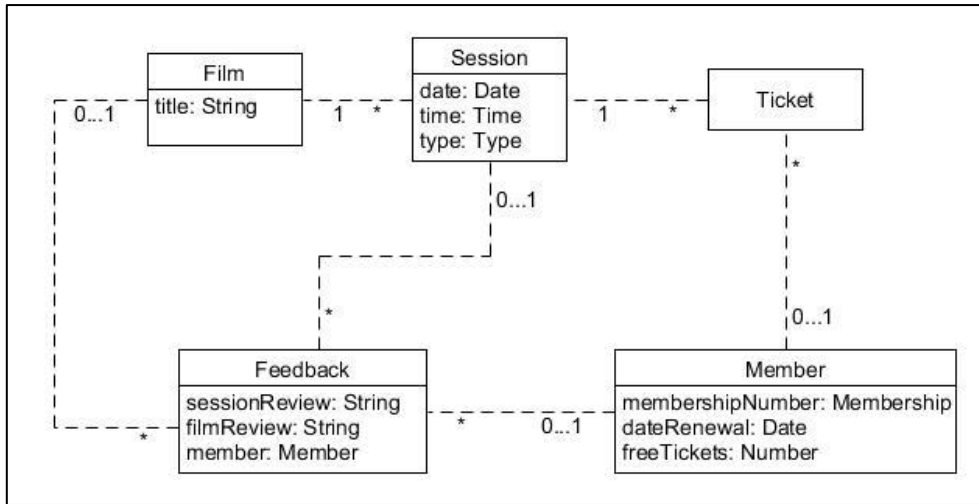


Figure 5: Class diagram including Feedback class

The diagram below shows an object model depicting a snapshot of the domain where multiple users are leaving feedback for different sessions.

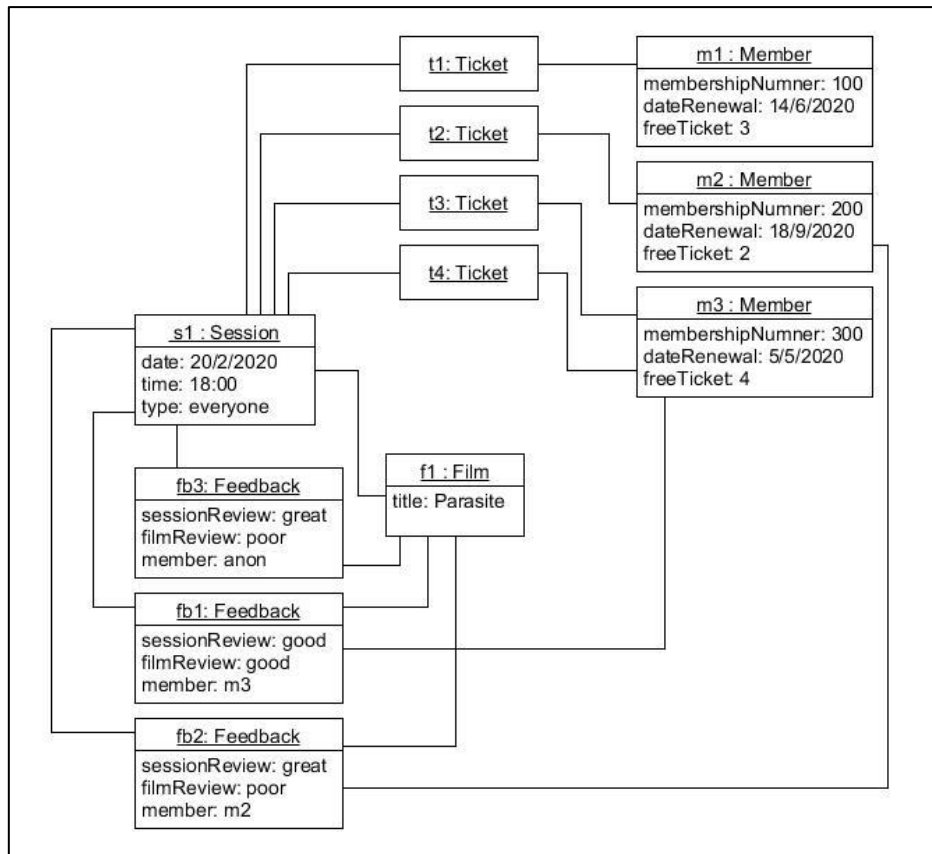


Figure 6: Object diagram of the Feedback Class

e.

An additional class could be added, for example 'PurchaserInfo', that links to both the Ticket and Member classes with attributes for any relevant information required. If a customer doesn't wish to opt in, then no instance of the class is created however if they are happy for their details to be recorded an instance of the class is created and linked to the specific member. If they are not a member then there is no link between the new class and the Member class.

## Question 3

a.

i.

pre-conditions:

- Session is not in the past and is linked to a room and film
- Tickets number must be less than room.capacity
- Member must be active and dateRenewal is not in the past

Post-conditions:

- Member will be linked to ticket
- Ticket will be linked to session

ii.

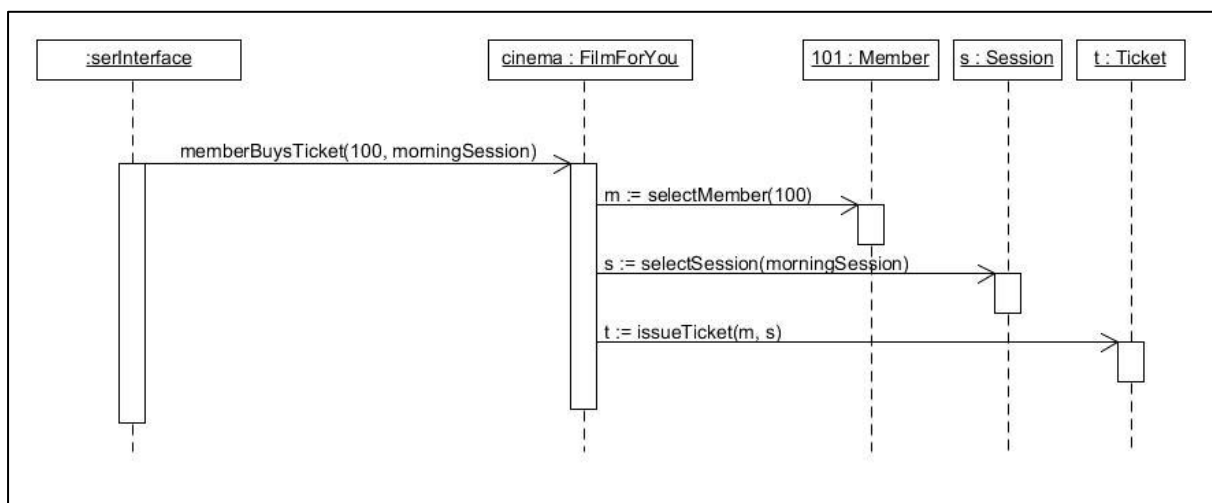


Figure 7: A fork sequence diagram of the use case – memberBuysTicket

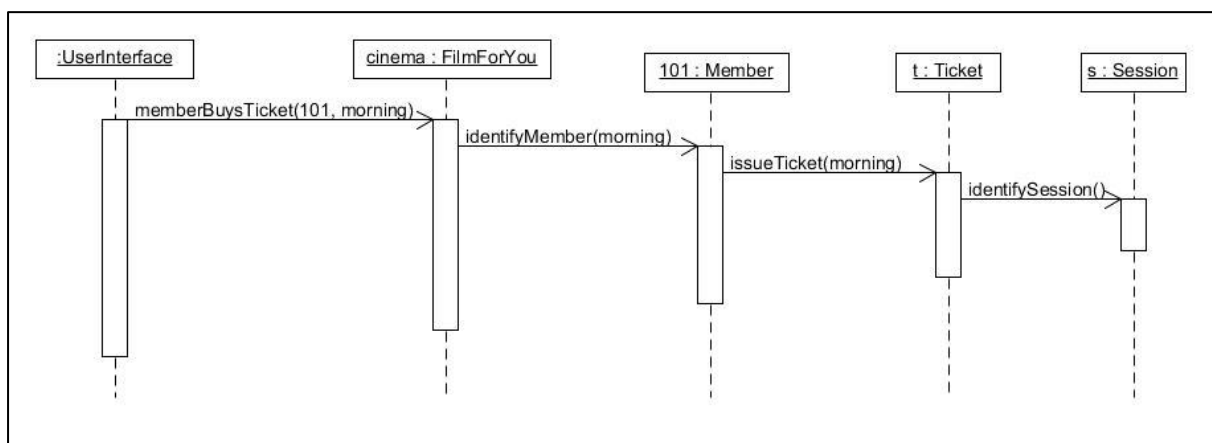


Figure 8: A cascade sequence diagram of the use case - memberBuysTicket

iii.

Neither of the two diagrams violate the Law of Demeter. In the diagram in figure 7, even though the `FilmForYou` class doesn't have a direct link to the `Ticket` class, a new instance of this class will need to be created and the Law states that a method can send an object a message if an object has been created as part of that method. In figure 8, although messages are sent to objects who are not

parameters of the current method, each object has a direct link to the next object in the sequence so there is no violation.

iv.

The sequence diagram in figure 7 shows that there is a low degree of coupling between classes however it means that the main responsibility of functionality is handled by the system class "FilmForYou" due to the sequence diagram being designed in a fork pattern. Because of this design, applying the GRASP creator pattern would lead to the system class also being responsible for creating instances of the Ticket class despite the fact there is no direct link between the two. This does however mean that the system class could end up containing too much specific detail making it less reusable. Assigning the system class "FilmForYou" as the creator is a more appropriate solution than the one outlined below.

The sequence diagram in figure 8 demonstrates a cascade pattern which in turn yields a sequence of events resulting in a high degree of coupling. It means that most classes are dependant on each other to send and receive messages resulting in less reusable classes. Applying the GRASP creator pattern to the cascade sequence diagram, we can assign responsibility of creating an instance of Ticket to an object of the Member class as it precedes it in the sequence diagram. The detriment of placing responsibility here is that the creator class, in this case the Member object, doesn't have the required data available to initialise a new Ticket object. Ideally, the Session object would do this once it has been determined that the capacity of the room the session is linked to has not been met or exceeded.

b.

In designing sequence models, we might discover that it could be beneficial to have a direct link between classes where one previously did not exist. Take the sequence diagram in figure 7 as an example. The class model has no direct link between "FilmForYou" and "Ticket", yet the sequence diagram shows that one may be required so the class model will need to be updated. Another kind of change might be that an additional class is needed to represent something that had not been considered or required during the initial modelling. This new class would need to be added and associations would need to be established before returning to finish the sequence diagram.

## Question 4

a.

### One Central Class

Looking at the analysis diagram, the “FilmForYou” class would become the system class for current iteration of the software system. This would mean that the “FilmForYou” class would be required to understand a number of use case operations which could lead to the class becoming overloaded. This would mean that no additional classes are required and there will be less to change regarding links, but it limits the flexibility and makes changes difficult to maintain. The operation that would need to be implemented in the “FilmForYou” class would be as follows: `memberBuysTickets(ticket : Number, member : Member, session : Session)`. This would mean that the first messages sent from the interface could be something equivalent to the following: `memberBuysTickets(2, morning, jack)`.

### Actor Class

For the scenario where an actor class is used to manage the use case, an extra class would not be required as the initiator is the member for which a class of this type already exists within the system. This means that additional operations would need to be understood by the class “Member”. This solution would require a possible redesign of the analysis model as there is no direct link between Member and Session and so other classes may be required to carry out additional operations such as the “FilmForYou” class returning a list of all possible sessions before a ticket can be purchased. Alternatively, an additional link could be used to connect “Member” and “Session” classes. The operation that would need to be implemented in the “Member” class would be as follows: `memberBuysTicket(ticket : Number, session : Session)` meaning the first message sent from the interface could be: `memberBuysTicket(1, evening)`.

### Use Cases as Classes

This scenario would require a new class to be added to define each new use case. In this situation the class could be called `MemberBuysTicket` and would have the operation `run(ticket : Number, session : Session, member : Member)` which the use case class would be required to understand. Therefore, the interface might send the message `run(2, midday, frank)` to the newly created object of the use case class.

b.

i.

Using external identifiers affects the design of the business model by strongly coupling the rest of the system to the user interface. As each object already has its own identity, rather than using the external identifiers to force the system to identify specific objects, it makes more sense to identify an object based on its already existing identity. If identification is localised to the interface, the business model only deals with business objects and is insulated from alterations to how object identification is performed.

ii.

A type of operation where using external data is the only option is during a situation where object creation is required such as creating and recording information about a new guest or reservation in a hotel software system. The external data can then be assigned to variables of the newly created object and any identification for the new object can be achieved by using the objects identity.

## Question 5

a.

i.

There are three different states that the Session object can be in: The first is when the session is created and no tickets have been sold, the second is where one or more tickets have been sold but the number of tickets sold is less than the room capacity and the final state is where the tickets sold is equal to the capacity, i.e. the session is sold out.

ii.

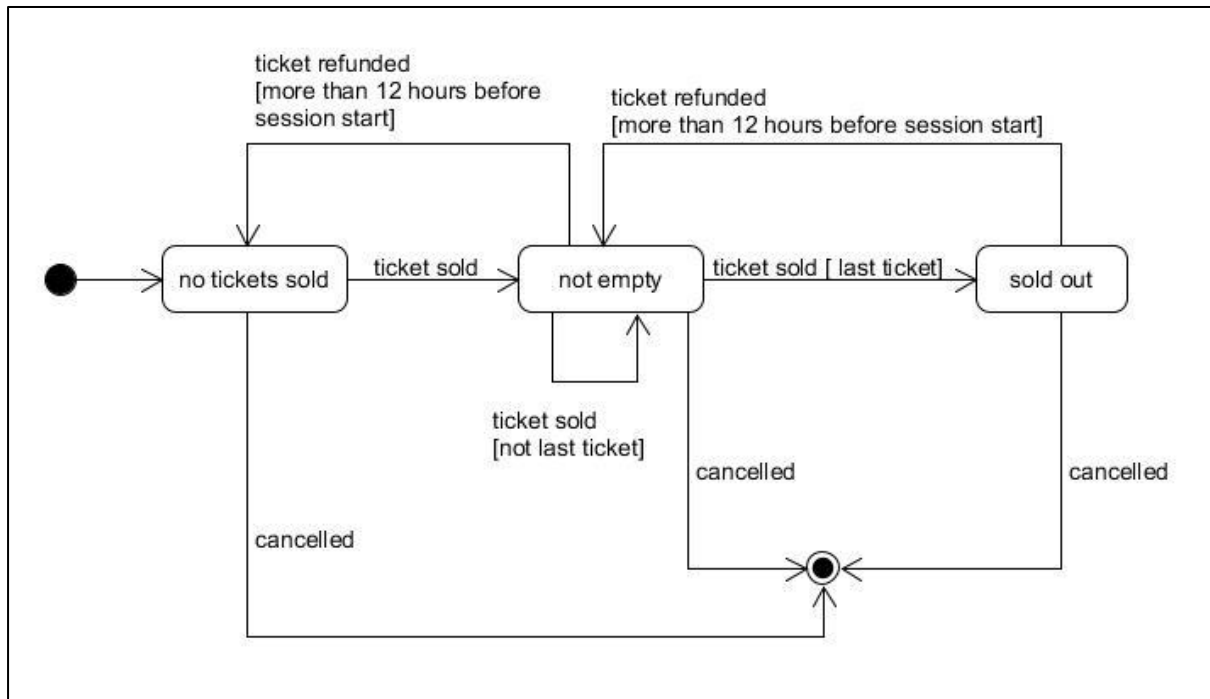


Figure 9: A state machine showing the life of a Session object

iii.

The information that would need to be added to the analysis diagram is a way to manage refunds and cancellations. A new class could be added to represent and manage the refunding of tickets for each member and whether they paid or used a free ticket, it would also need to manage refunds of tickets for those customers who are not members and it would also need to be able to handle a way to refund multiple customers in a single instance should a session need to be cancelled.

iv.

Once the session time has passed, there are several different decisions that could be made regarding the object's life history. The first is that session object and its links are destroyed and broken after a set period, for example, after the length of the films run-time. This would allow anyone who is running late to still purchase tickets for it, even if they miss a portion of the film and providing capacity hasn't been reached. The second option would be to allow the object to stay on the system for a longer period of time, for example, three months and then be removed from the system. The reason for this being that information pertaining to the film's popularity might be required by box office charts at set periods, of which the number of attendees and other information may need to be passed on. It also means that any complaints or issues regarding a specific session can be dealt with by finding the relevant session information on the system.

b.

A namespace is a unique identifier for a package which contains groups of different classes or even more packages. This partitioning helps with being able to manage the size and complexity of a software system by allowing different teams to work on different sections of the software simultaneously, for example one team could work on a section of the software that deals with customer details and another team could work on the section that deals with accounts which could then be decomposed further into another team working on an accounts payable package that lies within the accounts package. It also assists with information hiding and reuse becomes possible in certain areas. Packages could be redistributed or sold as a component to slot into another piece of software without the need to understand the details of how the classes and methods defined within that package work.