# ECM2419: Database Theory & Design
# Continuous Assessment

James Thomas

## 1.1 Conceptual Model Design

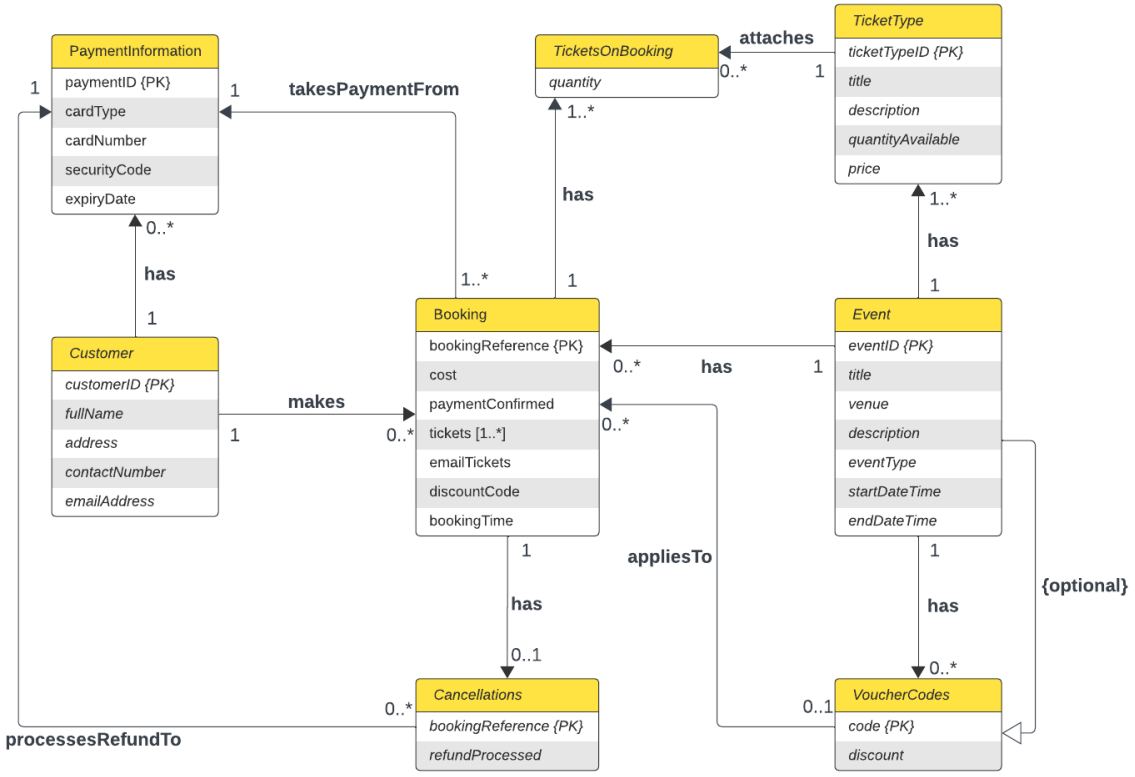


Figure 1.0: Entity Relation Diagram for the Event Booking System

## 1.2 Justification & Assumptions

### Customer

For the Customer entity I have assumed that the only required information is the name, address, contact number and email address. The name being important for general communication, address is required for billing information as per convention, contact number is required for notifying customers of potential changes to their booking and cancellation of the event and finally email address is required for sending booking confirmation as well as sending tickets if the digital ticket option is selected on the booking. I have assumed that Customers must have a phone number, email address and home address as without these, appropriate communication can't occur. I have also assumed that the age of a customer is irrelevant and if an event or ticket type requires an age restriction it is the job of the organiser to enforce this and not the booking system. I have given the customer entity a unique customerID attribute which can be used to distinctly identify records for ease of searching.

The relationship here to PaymentInformation is one-to-many with 0..* multiplicity on PaymentInformation since a customer may exist but may not have paid for anything hence no card details, but if they have paid for things they may have many cards attributed to them. The multiplicity to Booking is again 0..* with a one to many relationship since a given customer may not have made a booking and if they have, they may have made many different bookings so this endpoint is unrestricted.

## Booking

For the Booking entity I have included information only relating to a single booking in order to keep entities as encapsulated as possible. The bookingReference here is the primary key for the entity as each bookingReference is unique as per convention. The cost of the booking is important since it allows us to charge the payment card the appropriate amount, the ID of said payment card is also stored here. A confirmation boolean is also an attribute on the Booking entity as it ensures that the booking has gone through, although this is implied by the existence of the record anyway. Tickets contains information about the ticket types on the booking which is important when distinguishing between which ticket tiers have been booked. Discount code is also here as an attribute since a not null here will indicate that a code has been applied which lets us search for the validity of the code etc. bookingTime is useful for tracking the booking and the boolean emailTickets indicates if tickets are to be emailed or collected, running with the assumption that no more options will be added and one of the two must be selected. I have also assumed that only one discount code can be applied per booking, this is a standard and regular functionality of booking sites in the modern age so I will follow that. Another assumption I have made is that bookings are concrete once they have been created and they can only be cancelled, not updated. Under this assumption the tickets, cost and booking reference are concrete after inserting the record.

With PaymentInformation there is one to many relationship with 1..* multiplicity on Booking due to the fact that if a booking exists and is confirmed then it can be assumed that payment has been taken, under this I have made the assumption that events may not be priced to be free of charge, most systems in the modern day charge a booking fee so I have extended that norm to the assumption for my model. Similarly towards TicketOnBooking the 1..* multiplicity is due to the fact that if there are no ticket types on a booking then a booking cannot exist hence it being mandatory participation on the one to many relationship. Cancellations has a multiplicity of 0..1 since there either is or isn't a cancellation for a record and there are no further options from there. VoucherCode has optional participation on each end of the relationship since a voucher code may or may not be used for a booking, but if a code is used it will only be one code at most.

## Event

The Event entity has the following attributes: eventID, title, venue, description, eventType, startDateTime and endDateTime. The eventID is the primary key as it is a unique integer attached to event records; the title may be repeated between events for example if there was a regular event the title may be the same therefore that cannot be used as a primary key for the entity. The venue and description fields are self-explanatory as are the event start and end times, I have assumed there will be some form of categorisation or filtering on the booking site hence the eventType field is useful for the search feature.

The relationship from Event to Booking is described as one to many with 0..* multiplicity on Booking since no bookings being made is possible, as is many bookings. The booking cap cannot be set here on multiplicity since this varies for each event and bookings can vary in capacity therefore cannot reflect on the total event capacity. Events have 1..* multiplicity on

TicketType since tickets can't be bought for an event with no ticket types, the many side is required in order to satisfy the specification's request for many ticket types per event. On VoucherCodes from Event I have chosen a 0..* multiplicity constraint since voucher codes are optional for the event but if that option is chosen there is no limit on instances of codes.

## Cancellations

I made the decision that Cancellations required it's own entity in my model. Under this, if a booking is to be cancelled the record can be erased from the booking table and a record can be made in cancellation for that booking. The bookingReference from Booking can still be the primary key here since in the model there is no chance of a reference being duplicated for booking hence if they're unique in booking then passing it to cancellation will keep the unique property. The only other important attribute on this entity is the refundProcessed boolean which denote if the customer has been refunded. Cancellations connects to the PaymentInformation table in order to process the refunds so in a Logical model there will also be a foreign key attribute for that relation.

In regard to multiplicity, one cancellation will have one refund card hence a multiplicity of 1 on PaymentInformation, the many side of the one-to-many relationship here can be justified by the possibility of many cancellations refunding to the same card. I have made the assumption that refunds for bookings must be processed to the same account the payment was taken from, this may lead to issues in cases where card details change in between booking and refund however this situation would be so rare that one could assume the system administrator could feasibly deal with these case by case, especially considering making it easy to change the refund card would leave a security risk allowing dishonest users to manipulate the system in order to get refunds processed to them as opposed to the real customer. Cancellations has an optional participation to Booking since there is only a relationship if a cancellation occurs.

## PaymentInformation

PaymentInfomation is a relatively simple entity with few assumptions being made. A paymentID is given which in a physical system may simply be an incremented integer or perhaps a composite key of a few of the other attributes, which, due to the nature of payment details, can guarantee uniqueness. The cardType, cardNumber, securityCode and expiryDate fields are all essential for being able to charge payment to a card and as such they are stored here. I have made the assumption that the database system is using an appropriate cryptographic system in order to guarantee integrity and protection of this data. If left unencrypted there may be serious security implications so in physical instances of this model this table must be secured.

The relationship between PaymentInformation and Customer is one-to-many since one set of payment details can only be relevant to one customer, however one customer may have many card details stored. These two entities must be related when it is considered that billing information needs to be called upon when charging and refunding cards. Similarly there is a one-to-many relationship with Booking since a booking can be charged only to one card, but many bookings may have been charged to the same card. There is a 1..* multiplicity on Booking since a booking cannot have happened if a card wasn't charged. In the case of a refund as opposed to a charge, Cancellation has a one to many relationship which again is down to the fact that one refund can only be processed to one card but multiple refunds may have been processed to the same card, there is 0..* multiplicity on this since a card may have no associated cancellations.

## TicketType

In order to allow for a range of ticket tiers as per the specification, TicketType gets its own entity. For this the fields we need are ticketTypeID which is the primary key for the entity allowing for a unique point to search to, a title and description which will inform the customer of the ticket they're interested in and a price and quantityAvailable in order to cap sales and enforce pricing. Here I have made the assumption that any constraints on who can by tickets (e.g. child only, adult only, etc) will be enforced by the event vendor and not the sales site.

The relationship between Event and this entity is one to many with mandatory participation due to the 1..* multiplicity, justified by that if there are no ticket types a booking cannot be made thus ruling out zero, the many end simply allows for flexibility on how many tiers are required which will change between events. Each TicketType will only connect to one event so new records will be created for each new ticket type on events. Whilst some events may share the same format of ticket tiers, the quantity and price is something that needs flexibility to be set and updated on a per event basis. TicketType to TicketOnBookings has a one to many relationship since there are possibly multiple ticket types per event but bookings for each type are not guaranteed hence the zero in multiplicity.

## TicketOnBookings

TicketOnBookings is a small yet crucial entity that is required to track the type of tickets that have been purchased on bookings. Each booking will have a number of ticket tiers, each of these will be recorded as a new record in this table with the booking linked to it. For this entity the only field we need is quantity so we know how many tickets have been bought on a given booking for a given ticket type, allowing for data insight and capacity limiting. A link to event isn't required here since ticket types are unique to events so TicketOnBookings and Events are transitively connected through TicketType. There is no primary key on this entity making it a weak entity, this is fine since the combination of the two foreign keys will make the records uniquely identifiable.

The one-to-many link to booking is due to the reasoning that each record will have only one booking it relates to, however in the other direction, a Booking may have one or more ticket types contained, hence needing many records in this entity. The other one to many relationship is with TicketType which again can be justified with similar logic. Due to this being a linking table, no complex assumptions have been required.

## VoucherCodes

The final entity is VoucherCodes which has a unique code field which is the primary key for the entity. For this I have had to assume that a voucher code can be defined once and specifically generates a discount for one event, hence not applying to other events. This allows more control for where discounts are able to be placed, I justify this since it's a ticket sales system and the event organisers are likely distinct from the system owners; therefore discounts will be set by organisers to apply to their event as opposed to being set to the whole system. I have also used the assumption that voucher codes have no restriction of use based on ticket type on the booking. The only non-key attribute is discount which is a field holding the discount amount for the corresponding code.

The one-to-many relationship to Event with optional participation is here since events may have multiple codes but codes can only apply to one event. A one-to-many relationship with Booking enforces my assumption that bookings may only have one discount code each and they may not stack, there is optional participation on both sides here since there is no guarantee a voucher code will be used.

## 2.0 Logical Model Design

**Customers**(customerID, fullName, address, contactNumber, emailAddress)
Primary Key: customerID

**Events**(eventID, title, description, venue, eventType, startDateTime, endDateTime)
Primary Key: eventID

**Bookings**(bookingReference, customerID, eventID, cost, paymentConfirmed, paymentID, emailTickets, discountCode, bookingTime)
Primary Key: bookingReference
Foreign Key: customerID references Customers(customerID)
Foreign Key: eventide references Events(eventide)
Foreign Key: paymentID references PaymentInformation(paymentID)
Foreign Key: discountCode references VoucherCodes(code)

**VoucherCodes**(code, discountMultiplier, eventID)
Primary Key: customerID
Foreign Key: eventID references Events(eventID)

**PaymentInformation**(paymentID, cardType, cardNumber, securityCode, expiryDate, cardHolderID)
Primary Key: customerID
Foreign Key: cardHolderID references Customers(customerID)

**TicketTypes**(ticketTypeID, title, description, price, quantityAvailable, eventID)
Primary Key: ticketTypeID
Foreign Key: eventID references Events(eventID)

**Cancellations**(bookingReference, refundProcessed, refundCardID)
Primary Key: bookingReference
Foreign Key: refundCardID references PaymentInformation(paymentID)

**TicketOnBookings**(bookingReference, ticketTypeID, quantity)
Primary Key: bookingReference, ticketTypeID
Foreign Key: eventID references Events(eventID)