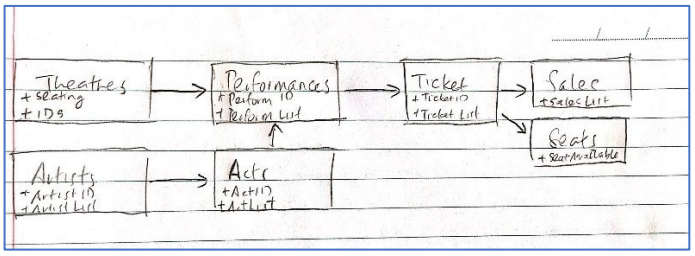
Domain Model – Description and Design

The Objective: To develop an implementation to the Server system of a Theatre Booking System where: *"A theatre booking system has to manage performances of acts performed by artists in different theatres. Theatres have seats. Tickets are issued for seats for a performance. Any act has a single artist (which may be a group), but an artist can put on many acts, and each act can have multiple performances".*

Given the description we must make the Server implementation class that will be used with the Client system of the Theatre Booking System, we can specify which methods have relationships and clear operations that will use other key classes that will help with the implementation. First, we must answer the questions: How do we initialise information? How would we manage the theatres, artists etc? What inputs are required for each theatre, artist, act etc? Due to user input, would we need to handle error exceptions? Thus, from the description above we can create classes specified to the appropriate literal nouns that will also complement the methods that will be in the implement Server class and other classes that complement the Server class:

Creating a class from the Implement, with methods of *initialise, getTheatreIDs, getArtistIDs, getActIDsForArtist, getPerformanceIDsForAct, getTicketIDsForPerformance, addArtist, addAct, schedulePerformance, issueTicket, seatsAvailable and salesReport;* we can create specific classes (related to the appropriate literal nouns) that will help with the implementation. These additional classes include: *Acts, Artist, Performance, Sales, Seats, Theatre and Tickets.*

Thus, we can see a relationship between each of the GET methods and ADD methods in the Implement class, so it's best to only provide a source of managing the main attributes (Theatre, Artist, Act, Performance, Tickets). In terms of managing these main attributes, we need to first realize that there is a link between initially creating attribute IDs. So, creating Artists first are prioritized to be able to create Acts, thus Performances and thus Tickets, but since specific attributes such as Performances and Tickets require Theatres to be assigned then we must initialise Theatre Information so that the user can assign available theatre locations to Performances. By managing these types of data, we can use unique IDs that will be created from each additional class and lists of the IDs that associate with each attribute (such as Artist ID can get all Acts assigned to artist). Also since there is a Ticket system, we must also create a seat availability system; where if ticket has been issued with specific row and seat input, then that seat has been taken up for that specific theatre. For this exception, we would need to link Theatre and PerformanceID with TicketID, so for the specific PerformanceID we can see which Theatre it is played at, and thus account for the ticket availability in terms of seating. This link of stored information will be stored within the Server implementation class so that each method can be provided with already stored information from previous initialised values from the User:



(Arrows represent order of availability – e.g. Sales requires initialisation of Ticket first)

We also need to handle error exceptions, in case the User Client inputs invalid inputs (such as initial artist name input already exists! You cannot add the same artist name) and since these errors will most likely to occur frequently in each Implement method then we will need to create a unique Exception Error class that uses an extended Enum class which contains all common errors that the User might run into.