Theatre Booking System

Domain Model

The implementation of the Theatre Booking System required managing performances of acts performed by various artists in various theatres. Tickets could be issued for performances held in theatres with different seating dimensions. Acts have a single artist but an artist could have various acts and each act could have various performances. The domain model described resulted in various design concepts, namely: Artist, Act, Performance, Theatre and Ticket. An **Artist** has a (Name) and an (Artist ID). An **Act** contains a single (Artist), a (Title), (Duration) and (Act ID). A **Performance** contains a single (Act ID), (Start Time), (Prices: Premium & Cheap) and (Performance ID). The Performance also contains a single (Theatre) in which the Performance was being held. A **Theatre** contains a (Theatre ID), (Rows) and (Floor Area). Finally, a **Ticket** is issued for a specific (Performance) and has a (Price), (Seat: Row & Position) and (Ticket ID).

The Design

The concepts identified in the domain model became initial classes for the implementation. Generations for IDs for each of the classes was done using a random UUID generator implemented in each class. This ensured each object created had a unique ID associated with it.

For instances such as where an Act was by an Artist, the Artist object itself was stored in the Act class. This could have been problematic if values of the Artist such as their names were editable, however, due to the lack of the edit feature, it was a viable option to store an Artist object inside of an Act object. The use of this method also allowed for the prevention of multiple duplicate fields in some classes. The same analogy applied for any other cases where this was required such as where each Performance had a Theatre, the Theatre object was stored in the Performance object.
All fields for the classes (Act, Artist, Performance, Ticket and Theatre) were declared final due to this very idea that there was no editing allowed.

To store the information on the server, collections such as Lists and HashSets were used. Instead of leaving all uses of collections(such as a List) out to exploit, the idea of the collection was encapsulated in various classes such as ArtistCollection, ActCollection, etc...
This allowed to create abstractions for various things a collection could do such as get specific items from the list by a field in the item itself. Ie. Getting an Artist by name. This create duplicate method types in the collection class as well as the TBSServerImpl class however to create the abstraction, this was necessary.

Edge cases such as NullPointerExceptions that could be thrown if a parameter in the request was null have also been considered and for such problems and other failures such as missing artists, Theatre .csv files not being available, inputs not being in their correct format (eg. the time not being in the ISO 8601 standard), appropriate messages are returned beginning with the word 'ERROR' to indicate a failure in

the request. Errors are also thrown for creation of duplicate objects such as two artists with the same name, acts with the same title,artist and duration, etc...
Parameter formats such as prices beginning with the $ sign and the time being in ISO 8601 standards were checked using custom regex patterns to ensure the correct values were being passed.