Product Requirements Document

Submission 2 - Report

**Team LANS**

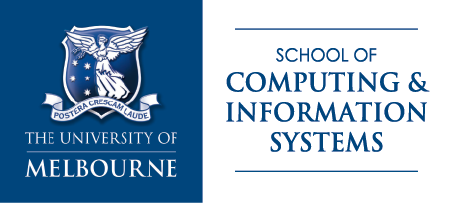
SWEN90007 SM2 2022 Project

In charge:  
Mohammad Saood **Abbasi** <mohammadsaoo@student.unimelb.edu.au>

Arman **Arethna** <aarethna@student.unimelb.edu.au>

Navdeep **Beniwal** <nbeniwal@student.unimelb.edu.au>

Levi **McKenzie-Kirkbright** <levim@student.unimelb.edu.au>



Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version | Description | Author(s) |
| 20/09/22 | 01.00-D01 | Created document outline | Levi |
| 21/09/22 | 01.00-D02 | Pattern descriptions and diagrams | Saood, Navdeep |
| 21/09/22 | 01.00-D03 | Unit of Work description | Levi |
| 21/09/22 | 01.00-D04 | Lazy Load | Levi |
| 21/09/22 | 01.00-D05 | Authentication and Authorization | Levi |

Contents

[Contents 2](#_Toc1820698803)

[1. Introduction 5](#_Toc1547537305)

[1.1 Proposal 6](#_Toc1320027451)

[1.2 Target Users 6](#_Toc1200489991)

[1.3 Conventions, terms and abbreviations 6](#_Toc1925520591)

[2. Class Diagram 6](#_Toc739989668)

[3. Pattern descriptions: context, implementation, and illustrations 6](#_Toc1135575884)

[3.1 Domain Model 6](#_Toc1668658146)

[3.2 Data mapper 6](#_Toc171248472)

[3.3 Unit of Work 6](#_Toc1682522139)

[3.3.1 Unit of Work: Design Rationale 6](#_Toc342129806)

[3.4 Lazy Load 6](#_Toc1517921861)

[3.4.1 Lazy Load: Design Rationale 6](#_Toc409744540)

[3.5 Identity Field 6](#_Toc473173654)

[3.6 Foreign Key Mapping 6](#_Toc1050453829)

[3.7 Association Table Mapping 6](#_Toc1367427407)

[3.8 Embedded Value 6](#_Toc1786349496)

[3.9 Inheritance pattern 6](#_Toc922306626)

[3.10 Authentication and Authorization 6](#_Toc775873465)

# Introduction

## Proposal

This document contains the second submission of the project for SWEN90007 Software Modelling and Design, 2022. It specifies the class diagram and pattern descriptions used in the solution.

## Target Users

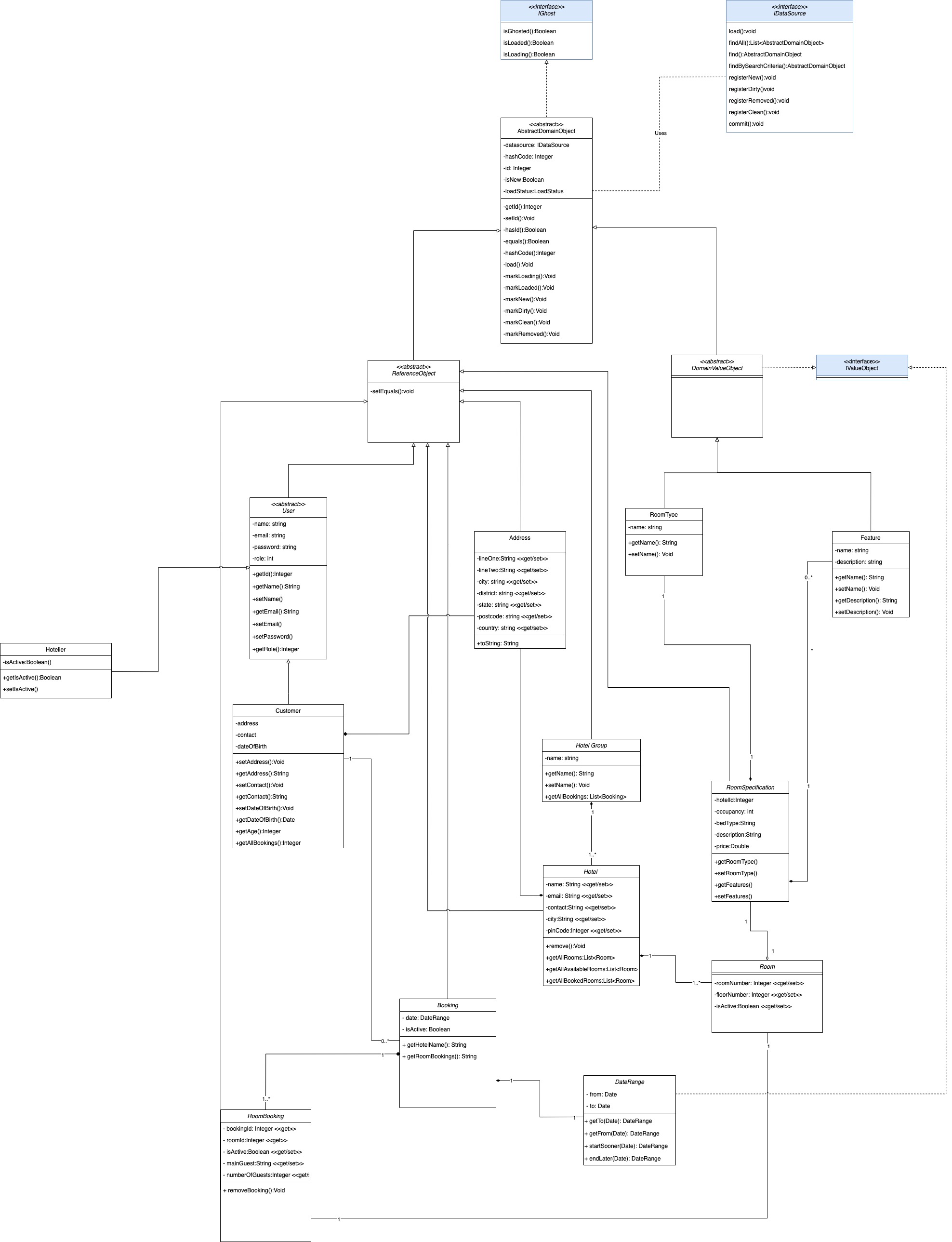
This document is intended for SWEN90007 teaching staff and other SWEN90007 students, particularly the LANS team.

## Conventions, terms and abbreviations

This section explains the concept of some important terms that will be used throughout this document. These terms are detailed alphabetically in the following table.

|  |  |
| --- | --- |
| Term | Description |
| “Team” or “the Team” or “LANS” | The project team. |

# Class Diagram (Please find an additional document as an attached PDF with the submission to view the class diagram better).



# Pattern descriptions: context, implementation, and illustrations

## Domain Model

In the realm of modern computer science, the focus on designing software has steered towards an approach to mimic human behaviour. Our approach to domain model implementation was to start by determining purely from a functionality point of view, the elements involved. Below is the domain model for our set of use cases compiled with reviewal comments from part 1.

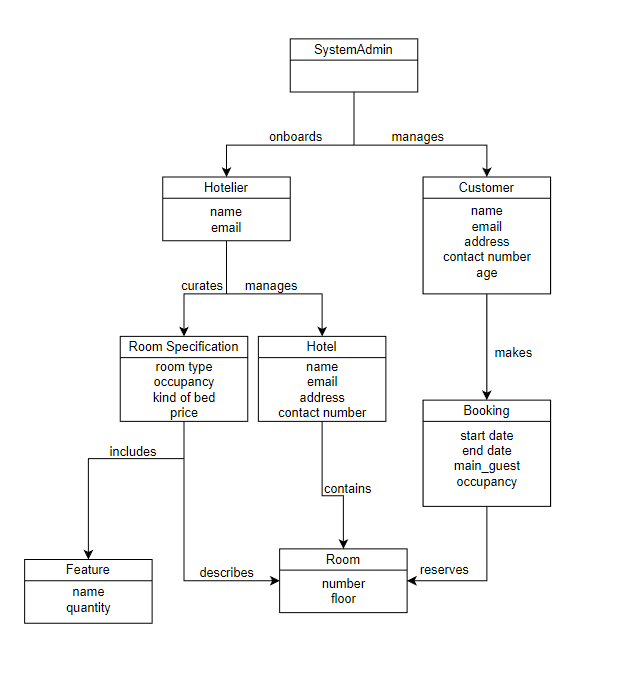
**Domain Model Summary:**

* Our system has a 3 level distinction of users: System Admin, Hotelier and Customer.
* One system admin can view all the users of the system.
* One system admin creates all hotel groups.
* One system admin creates all hoteliers and assigns them to hotel groups.
* Each hotelier is responsible for managing and creating hotels inside his designated hotel group.
* Each hotelier can create rooms for the hotels belonging to their hotel group.
* Each hotelier has modification access to bookings inside any of the hotels belonging to the hotel group he is a part of.
* Each customer can filter and view different hotels with rooms available for their itinerary.
* Each customer can make a booking for one or more rooms belonging to one hotel given the number of guests and max occupancy of the rooms.
* Each customer has access to their own sets of prior and future bookings.
* Each customer has modification access to any of the future bookings with respect to addition of people, changing dates or cancelling the booking in its entirety.

**Domain Model Gloassary:**

We have used different terminology from the project specification to increase the clarity and specificity of our model. Below are relevant terms that we wish to disambiguate.

* **System Admin:** System admin corresponds to the admin in the specification.
* Our efforts are to model real world situations taking inspiration from actual hotel booking processes currently in place.
  + **Hotelier** is a user appointed by the system admin that manages the hotels inside of a hotel group.
* Another narrative of modelling real world applications is to enable users to make bookings for multiple people often not themselves. As seen widely in the corporate industry where in company departments are responsible for booking work travels.
  + **Customer:** This is the user that has an account in our system and can place bookings for himself/ other people.
  + **Booking:** This is the actual transaction initiated by the customer with our application acting as the liaison to reserve a room in given hotel.

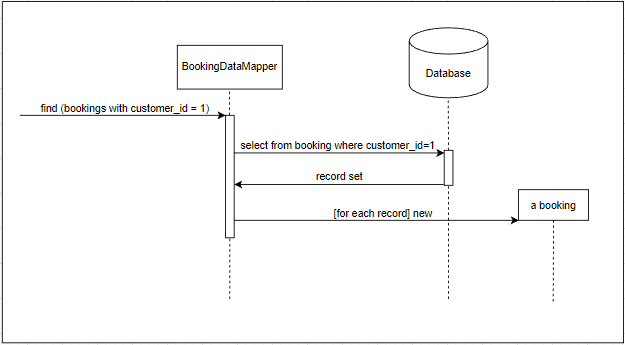


## Data mapper

Good design approach suggests that each layer in our solution needs to be separate from the other. The idea behind this lies in the fact that on change of the client serving that layer, the other layers should not be impacted. Data mappers belong to the data source layer that interact with the database. They are coded behind a façade so that regardless of the database client and its dependent queries, the domain layer will always have a defined view of it.

**Façade:** Each façade stands to serve a different function effectively masking any implementation that might occur on the data source layer from the domain layer. The following are the facades that we have created:

* **DataSourceFacade –** Implements the IDataSoruce interface and generalises the methods contained within it with appropriate Unit of Work declarations.
* **IDataMapper –** Provides an interface to define methods that can be incorporated by the AbstractDataMappers.
* **IIDentityMap -** Provides an interface to define methods that can be incorporated by the identity maps
* **IIdentityMapRegistry –** Provides an interface for the registry to store all used identity maps
* **IMapperRegistry –** Provides an interface for the registry to store all used data mappers
* **IUnitOfWork –** Provides an interface for the unit of work methods –
  + **RegisterNew –** Adds to the list of newly added object
  + **RegisterClean –** Add to the list of the objects that have not been changed nor have been created from scratch
  + **RegisterDirty –** Adds to the list of updated objects
  + **RegisterRemoved –** Adds to the list of objects that have been removed from the system.
* **PostgresFacade –** Façade to instantiate registry, set identity maps and establish connection with the postgres client.



**Data Mappers:**

* **AbstractPostgresDataMapper –** This data mapper is an abstract data mapper defining the general methods that all classes implementing it will have access to. This mapper contains the IDataSource interface which can be used to exhibit similar behaviour by all objects implementing it.

The functions contained within it are:

* + **getByID(Integer) –** Returns the id from the database
  + **getFromDB(Integer) –** Fires the query and calls the load function to extract data from the resultset.
  + **load(ResultSet) –** Extracts the data from the resultset in a domain object recognizable format.
  + **create(AbstractDomainObject) –** Creates a new domain object and adds it to a loaded hash map for further use.
* **CustomerDataMapper –** The customer data mapper is a specific mapper that implements the AbstractPostgresDataMapper with specialized queries for fetching objects in relation to the customer table in the database. The functions correspond to the generally defined function signatures in the IDataSource interface which is initialized within the AbstractDomainObject.
* **HotelDataMapper -** The hotel data mapper is a specific mapper that implements the AbstractPostgresDataMapper with specialized queries for fetching objects in relation to the customer table in the database. The functions correspond to the generally defined function signatures in the IDataSource interface which is initialized within the AbstractDomainObject.
* **HotelierDataMapper -** The hotelier data mapper is a specific mapper that implements the AbstractPostgresDataMapper with specialized queries for fetching objects in relation to the hotelier table in the database. The functions correspond to the generally defined function signatures in the IDataSource interface which is initialized within the AbstractDomainObject.
* **PostgresMapperRegistry –** This is used to keep track of all the mappers available within the system.
* **RoomDataMapper -** The room data mapper is a specific mapper that implements the AbstractPostgresDataMapper with specialized queries for fetching objects in relation to the room table in the database. The functions correspond to the generally defined function signatures in the IDataSource interface which is initialized within the AbstractDomainObject.
* **RoomSpecificationMapper -** The roomSpecification data mapper is a specific mapper that implements the AbstractPostgresDataMapper with specialized queries for fetching objects in relation to the room\_spec table in the database. The functions correspond to the generally defined function signatures in the IDataSource interface which is initialized within the AbstractDomainObject.
* **HotelGroupDataMapper -** The hotelGroup data mapper is a specific mapper that implements the AbstractPostgresDataMapper with specialized queries for fetching objects in relation to the hotel\_group table in the database. The functions correspond to the generally defined function signatures in the IDataSource interface which is initialized within the AbstractDomainObject.
* **FeatureDataMapper -** The feature data mapper is a specific mapper that implements the AbstractPostgresDataMapper with specialized queries for fetching objects in relation to the feature table in the database. The functions correspond to the generally defined function signatures in the IDataSource interface which is initialized within the AbstractDomainObject.
* **BookingDataMapper -** The booking data mapper is a specific mapper that implements the AbstractPostgresDataMapper with specialized queries for fetching objects in relation to the booking table in the database and all the tables that are noramlized from it. The functions correspond to the generally defined function signatures in the IDataSource interface which is initialized within the AbstractDomainObject.
* **RoomBookingDataMapper -** The roomBooking data mapper is a specific mapper that implements the AbstractPostgresDataMapper with specialized queries for fetching objects in relation to the room\_booking table in the database. The functions correspond to the generally defined function signatures in the IDataSource interface which is initialized within the AbstractDomainObject.

## Unit of Work

### Unit of Work: Design Rationale

Our Unit of Work (UoW) is located in the data source layer and is responsible for managing database transaction control of the domain objects over the lifetime of a business transaction. Our overall design resembles a Command pattern (GoF). Our data source façade implements the IDataSource interface defined in the Data Model, thus providing the services to the Data Model, including Uinit of Work capabilities. The domain objects are able to register themselves with the Unit of Work via the IDataSource interface, which delegates to the concrete Unit of Work instance. This separation of concerns allows the design and update of the Unit of Work without affecting the domain objects, as long as the behavior defined in the interface’s contract is delivered.

**Session**: our UoW is associated with a session. As such, multiple requests across the same session are handled by the same UoW instance.

**Concurrency**: servlets follow a one-thread-per-request model. As such, we leverage servlet’s thread isolation for each request we also associate an “active” UoW with the current thread.

By combining session and one-request-one-thread, different Units of Work will be isolated from each other whilst allowing a UoW to persist across user requests for an entire session.

## Lazy Load

### Lazy Load: Design Rationale

We follow the Ghost variation of the Lazy Load from Fowler. Like our Unit of Work, our IDataSource façade provides the Ghost functionality to the domain model. This loose coupling through dependency injection and interface segregation allows us to polymorphically select which data mappers are selected for competing the lazy load. As such, our design is easily extensible without affecting the domain model. If a particular domain object needs to use Ghost functionality, the data source the layer must recognize domain object’s concrete type and implement the necessary ghost functionality, however, the domain objects to not know about the implementation details of the ghost implementation in the data source layer. Thus, our domain model is protected and isolated from changes in the data source whilst having access to the lazy load service of the data source.

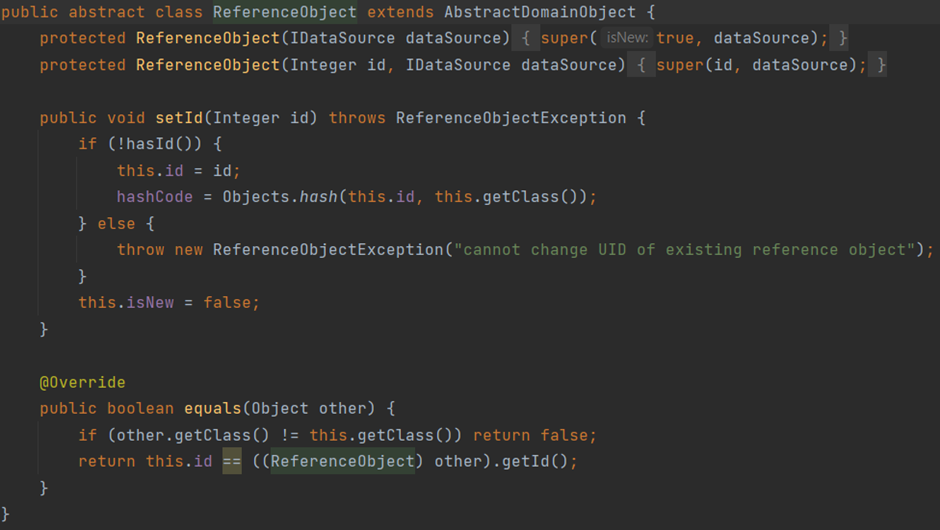
One key criterion for this design was the Open-Closed Principle. It is difficult to know ahead of time which domain objects will require lazy loading, especially as schema changes may affect the lazy loading strategy. Therefore, we prioritized decoupling the domain model from the data source. The IDataSource layer is extensible by adding concrete implementations of ghost functionality for specific domain object classes. This design has allowed us to add ghost functionality as the design evolved throughout implementation and, compared to other lazy loading variants, offers the most flexibility for changes as the project progresses by protecting the domain model from changes in the data source layer.

## Identity Field

Identity Field pattern is a great approach to ensure the specificity of an in memory object so that it can be linked with a database object which ensures exclusivity through its primary key.

Our approach towards the same was to have an Integer Id correspond to every database object that we instantiated as an in-memory object and the value of it was the same as the primary key of that database object.

The following are the instances of the contextualised usage of Identity Fields which in our case is set in the ReferenceObject:



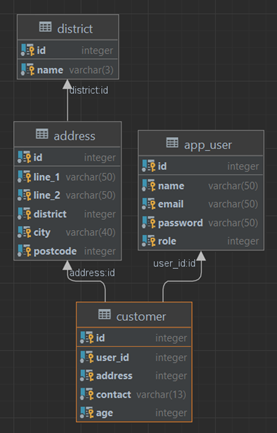
As can be seen from the above code base, the reference object, which further gets extended by every domain object in our system, sets the identity field ensuring exclusivity of every in-memory object. This value is absolute and cannot be changed.

## Foreign Key Mapping

In memory objects need to replicate association maintained in the database through foreign keys. This is done through object references. Our contextualized approach of the same was to initialize an instance of the foreign key class inside the primary key class.

The domain package in our code base consists of all the packages that help instantiate domain objects. An instance of foreign key mapping in our structure is visible through the Customer object which in its constructor requires an Address object. This association mimics that of the foreign key relationship between Customers and Address tables.

ER Diagram



Implementation in Codebase through object reference

public class Customer extends User{  
 Address address;  
 String contact;  
 int age;

public class Address {  
 String line\_1 = "";  
 String line\_2 = "";  
 District district;  
 String city;  
 int postcode;

There are various other counts of such implementation as can be seen in the following files

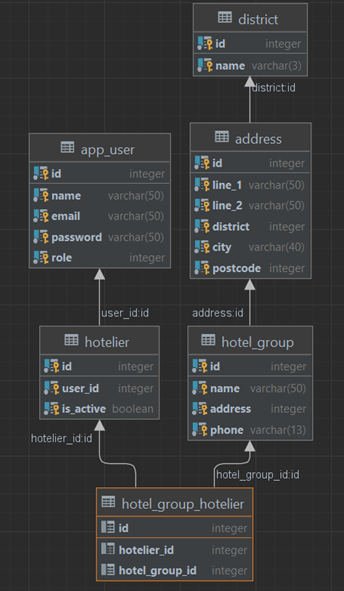
* lans.hotels.domain.hotel.Hotel.java
* lans.hotels.domain.room.Room.java
* lans.hotels.domain.room.RoomSpecification.java
* lans.hotels.domain.user\_types.Customer
* lans.hotels.domain.utils.Address.java

## Association Table Mapping

To handle relationships between entities where one entity has multiple counts of the other entity referenced within it is intrinsically difficult to handle in relational databases. To ensure these relationships are best handled, we use an association table in between the two tables to capture their one-to-many relationship. Replicating this in object-oriented programming, however, is not as challenging a task because of the ability to instantiate collections or hash maps of objects within other classes.

Below is an example of association table mapping in our implementation

ER Diagram



The relationship between hotel\_groups and hoteliers is one to many since there can exist more than one hoteliers assigned to a single hotel group. Capturing this information as a collection inside our relational database was not possible, hence we achieved this relationship through addition of a hotel\_group\_hotelier table which captures the primary keys of both, the hotelier and the hotel group tables.

Another such instance in our implementation was through the addition of a room\_booking table to capture the relationship of having multiple rooms inside a single booking.

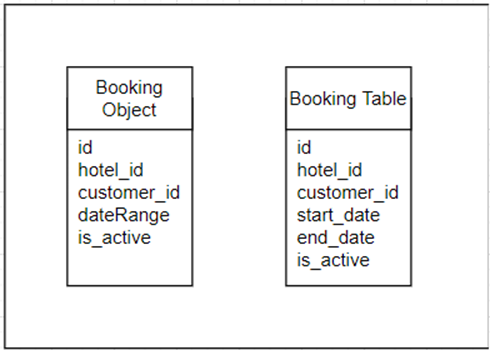
## Embedded Value

The embedded value pattern is useful for implementing cases that capture an object which cannot possibly have a specialized table for itself in the database. A common notion of this implementation is capturing counts of date ranges. A date range, while treated as a singular object in the domain layer, exists as a dissected piece in the data source layer as start\_date and end\_date.

We have used this concept to capture the date range in the booking process. The booking table in our current implementation consists of a start\_date and an end\_date. The way the object in the domain layer has been referenced is in the form of a date range.

The domain layer consists of a java class inside the utils package that packets the start and end date as one element.

The specific class where the above implementation has occurred is lans.hotels.domain.utils.DateRange.



## Inheritance pattern

Inheritance patterns are means of showcasing inheritance inside of relational databases which inherently do not represent it. There are 3 primary means of designing a schema with inheritance and they are

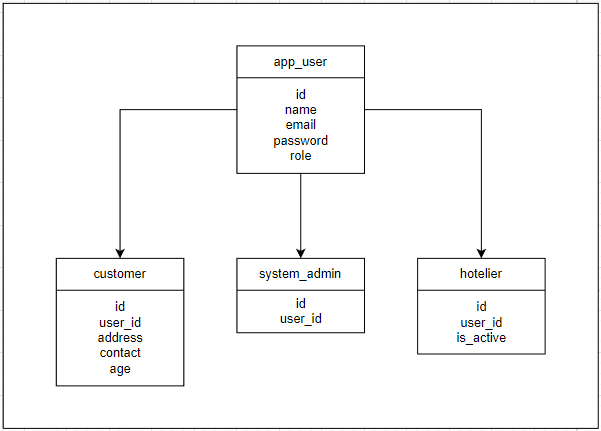
* Single Table inheritance – In this form of an inheritance, a single table is responsible for storing all elements of dependent classes.
* Class Table Inheritance – In this form of inheritance, there exists a unique table for every class.
* Concrete Table Inheritance – In this form of inheritance, there exists a unique table for every concrete class. A concrete class is ones which has direct use in the domain logic and is not used primarily to store details.

The inheritance followed in our database design was class table inheritance because of the following reasons:

* The design was meant to be as scalable as possible with little cohesion between elements in any layer. With class table inheritance, each class would have its own unique table making the readability of the database easier without wasting space.
* Visualizing the domain model through the data source layer would be easy to do.

The primary developmental challenge we faced through this pattern of inheritance was that the level of normalization made the query generation cumbersome and filled with multiple joins.

This inheritance pattern in our specific case is best observed in our implementation of the users distribution.



As can be seen from the diagram above, there exists a relation between app\_user -> (customer, system\_admin, hotelier). This is modelled by having a class for every table in the domain layer such that the class app\_user is extended by the customer, system\_admin and hotelier classes. This is an example of class table inheritance since every table object in the data source layer has a corresponding class for it in the domain layer.

## Authentication and Authorization

We use auth0 for Authentication and Authorization as it is suitable for single-page applications. Users authenticate with auth0 and are allocated access tokens for authorization. The access tokens are JSON web tokens (JWTs) that include basic auth information about the user. When the user makes requests to the backend API, they may include a JWT. The JWT is verified using the asymmetric public key schema provided by auth0. Authorization is not fully implemented at the time of submitting this report; however, our goal is to associate user emails in the JWTs on each request with a specific user profile and its associated role-based authentication privileges. For example, only the admin user should be able to call certain endpoints that change the roles of other users. From a UI point of view, our app follows a “Rich UI” (Fowler) pattern. The UI is fully downloaded, the user authenticates, and receives a token for per-request authorization. Our plan is also to update the UI according to the authorization of the user. For example, redirecting away from the admin section of the app if they are not admin. Thus, we have two layers of security: our user interface and our backend, both of which enforce security constraints.