Hotel Case Study Data Design and Object Oriented analysis

Modification history

Seb Coope 6/9/2020 Created

Noun identification

First we look for as many nouns as we can (remember a noun is something which can be preceded by the or a). A noun is a word or set of words for a person, place, thing, or idea.

Hotel
Owner
Room
Night
Guest
booking
Breakfast
Invoice
Availability
Payment
RoomRate
RoomType
Guest details
Username
Password
Login
CityTax

Data analysis

We take each element in the previous list it might be out of domain (so not relevant to the problem).
Once we have removed all redundant items we then need to work out if items could be classes or attributes of classes. Once we've done that we look for inheritance. Without inheritance your design may show poor re-usability and not be good as a basis for future products. Good re-use is considered critical quality when producing software designs.

At any stage from now one, we can also look for classes implied, i.e. not in the noun list explicitly, but useful to make the system flexible and useable.

Hotel At first glance this looks redundant out of domain (the system managed the hotel
bookings but in this context it is above the problem itself.)
However it might useful for future expansion, to include a hotel id as an attribute
this is so the system in future can manage multiple hotels with little modification.

Owner Redundant
Room Class
Each room class needs to store information on a given room
For the system given, there will be at least 30 instances of this class

RoomDescription Implied Class

Because multiple rooms may have similar qualities (type, occupancy and tariff) it makes sense to generate another class to store these in.
For the system given, there will be at least 3 instances of this clas

Night Part of the descriptive language, but not really an attribute or class
Guest Class stores details about the guest
Booking Class
Stores details about the booking for the guest

RoomBooking Implied Class
Since a booking can consist of a whole list of room bookings, we need this in a separate class/table.

Breakfast(Booking) Class
Stores details about the guests meal booking, we will look at this again in the analysis stage

Invoice Class
This is a tricky one, if the information in invoice can be totally generated by other information, for example in booking then the computer science principle
called DRY (Do no Repeat Yourself) would state this should not be stored again.

However, it might be useful to be able to re-issue the invoice or send to other parties for payment, therefore for this system we will add it in.

Availability Redundant, looks like a property of room at first glance but is really something that can be determined by checking the room bookings for a room

Payment Class, useful class can store details on payments (card or cash or Paypal) for example.

RoomRate Attribute of RoomDescriptor
RoomType Attribute of RoomDescriptor
Guest details Complex set of attributes of guest, will be analysed in OO development
Username Attribute of User (User is an implied class created by concept of Login)
Password Attribute of User
Login Class, used to control state of login
CityTax Attribute … tricky when should it go..? Store in table/class called SystemProperties

OO Analysis
This is where we take each and every class and identify possible attributes and also determine of these classes and attributes will need an associated database table. To help out with this stage we
will be putting the information in a tabular form.

When we store the data in the database table, when one class contains an instance or instances or another class we will use a unique id to associate the table rows. This id will be used by the OO language to distinguish instances of the classes within the table rows.

Hotel class

Attribute name	Type	Persists
hotel_ID	Int	Yes
hotelName	String	Yes

Room class

Attribute name	Type	Persists
roomID	Int	Yes
Hotel	Hotel	Yes
roomNumber	Int	Yes
roomDescription	RoomDescription	Yes

Relationships of Room

Many to One with RoomDescription
Many to One RoomBooking

This design allows rooms to belong to hotels, if there is more than one hotel. Whether a booking can involve more than 1 hotel is a business rules decision and can be made later in the design.

RoomDescription class

Attribute name	Type	Persists	Comment
roomDescriptionID	Int	Yes	
Description	String	Yes	
maxOccupancy	Int	Yes	
tariffInMinorUnit	Int	Yes	Smallest unit, for example, pence (GBP), cents (USD/EURO), fen (CNY)
currencyCode	String	Yes	ISO 4217

Notice the denomination of currency, by including this in the system at this stage, we improve the re-usability of the code.

Relationships of RoomDescription
One to Many with Room

Guest class

We have in this class an opportunity to start some inheritance modelling, so we will put some in. If we make a new system, we can re-use class Person.

Inheritance
Inherits from class **Person**

Guest class

Attribute name	Type	Persists	Comment
guestID	Int	Yes	
Email	String	Yes	

Relationships of Guest

One to Many with class Booking

One to Many with class Payment

One to Many with class Invoice

If may seem strange that this class does not contain many extra attributes. This is not a problem, it will act as a placeholder for attributes we might want to add in for guest in the future. The address id will allow you to reference the address in another table.

Person class

Attribute name	Type	Persists	Comment
personID	Int	Yes	
surname	String	Yes	
forename1	String	Yes	
forename2	String	Yes	
dateOfBirth	Date	Yes	

Relationships

Person is parent of class Guest

Address class

Attribute name	Type	Persists	Comment
addressID	Int	Yes	
houseNameOrNumber	String	Yes	
addressLine1	String	Yes	
addressLine2	String	Yes	
addressLine3	String	Yes	
postalTownCity	String	Yes	
postalOrZipCode	String	Yes	Format depends on CountyCode

countryCode	String	Yes	3 digit ISO 3166 code
Person	Person	Yes	Associates address table with Person, allows each Person to have address book

Relationships of Address

Many to one with class Person

In this class again you can see some potential for re-use, lots of applications will deal with customers' addresses. It will also be common for customers need to have an address book. It's a useful feature even if it's not used on this application

Booking class

Attribute name	Type	Persists	Comment
bookingID	Int	Yes	Unique for each booking, useful to find customers bookings
DateOfBooking	Date	Yes	
Guest	Guest	Yes	This is the primary guest

Relationships with Booking

One to Many with RoomBooking
One to One with Guest
One to One with Invoice

RoomBooking class

Attribute name	Type	Persists	Comment
RoomBookingID	Int	Yes	
Booking	Booking	Yes	Association with Booking
Room	Room	Yes	Which room is being booked
startDate	Date	Yes	Start of stay
endDate	Date	Yes	End of stay

The room booking class links the bookings with the rooms, so that each booking can manage multiple rooms and each room can be associated with different bookings on different dates. This type of class is a link association class, it associates two other classes in a relationship (in this case a room booking).

Relationships of RoomBooking

Many to One with Booking

Many to Many with Guest // We need to keep track of guest in rooms

MealBooking class (not Breakfast booking)

In the previous analysis we worked out we need to have a booking for breakfast, however for very little extra complexity, we can extend this to a meal booking. It is always useful to think of what a good system should be able to do, rather than the bare minimum.

We will simply connect the meal bookings, with the bookings in this model, as our only requirement in this example is to know who to invoice and how many guest will be expected to for the meal.

This system also will allow you to make a meal booking without staying in the hotel, this is added flexibility.

In this data model, you would be expected to create 3 MealBooking instances if you need to book 3 people for breakfast.

Attribute name	Type	Persists	Comment
MealBookingID	Int	Yes	
MealID	Int	Yes	Enum for meal, 1=breakfast, 2=lunch, 3=dinner
Booking	Booking	Yes	Which booking will be invoiced
Charge	Int	Yes	Cost in minimum currency unit
currencyCode	String	Yes	ISO 4217
Date	Date	Yes	Date of meal
Time	Date	Yes	Optional time for meal, for example for dinner service

Relationships

Many to One with Booking

Invoice class

Attribute name	Type	Persists	Comment
invoiceID	Int	Yes	
dateIssued	Date		
booking	Booking	Yes	Which booking will be invoiced
amount	Int	Yes	Amount in minimum currency unit
currencyCode	String	Yes	ISO 4217

Relationships

Many to One with Booking

Payment class

Attribute name	Type	Persists	Comment
paymentID	Int	Yes	
dateTaken	Date		
invoice	Invoice	Yes	Which invoice will this pay against
paymentMethod	int	Yes	1=cash, 2=card, 3=paypal
Amount	Int	Yes	Amount in minimum currency unit
currencyCode	String	Yes	ISO 4217

Relationships

Many to one to Invoice (an invoice can be paid, part cash + part card if needed)

User class

Attribute name	Type	Persists	Comment
userID	Int	Yes	
DateCreated	Date		
username	String	Yes	
password	String	Yes	
emailAddress	String	Yes	To help with forgot password, username or 2FA
role	Int	Yes	Role of login, 1=hotelManager 2=receptionist

			3=guest 4=systemAdmin

With the user class, this allows use to perform a basic Login procedure. The role attribute is important as it allows us to add different users with different capabilities. We don't need to add these all in now but it makes a class a lot more flexible.

Login class

This is used to control the complexity of the Login process, this might be very simple (as in a PAP or password access protocol) or more complex like CHAP (Challenge Handshake Access Protocol). All the complexity of the process is hidden within the login class.

Attribute name	Type	Persists	Comment
loginID	Int	Yes	
DateCreated	Date		
user	User	Yes	User who is trying to login
status	Int	Yes	Current state of login 0=started 1=pending (waiting for credentials) 2=ok (login successful) 3=bad (login failure)

With this class there is an implication that the attributes may be expanded later if the class increases in complexity.

Relationships

One to One with User

Notice this complexity is not stored in the User class, this is a different type of concern (security) than the user class which is for storing the user's details. This class **encapsulates** the security concern hiding from the user of the class.

SystemProperty class

Sometimes you need to be able to store some general global information that can be used by a range of modules. This data may well be static (not associated with a particular instance of a class) as well as persistent (it needs to be stored in the database). For this reason we are going to introduce a class which will store this data as name, value pairs.

One obvious application of this is to store the CityTax value, remember you should never hard wire values into your code that are liable to change (except in the extreme examples like there are 8 bits in a byte). All values need to be 1, stored in a file or 2. In the database (better).

Other examples of system properties are email service usernames and passwords, details for logging into payment systems etc. Also for this type of table, you should also consider encrypting the values, this helps to keep the information discrete.

Attribute name	Type	Persists	Comment
Name	String	Yes	Name of name value pair (e.g. CityTax)
Value	String		Value of the parameter, example 300

Summary class list

Hotel
Room
RoomDescription
Guest
Person
Address
Booking
RoomBooking
MealBooking
Invoice
Payment
User
Login
SystemProperty

Class diagram

Here is the class diagram, notice we are using this to simply model the relationships, the class diagram itself is of limited use to show all the attributes and methods as it can become very messy and hard to read. Usually the methods and attributes of each class can be handled better using something like a CRC card, this approach takes each class in turn and models it relative to its association with its interacting classes.