Tips and Tricks for Project Task 1

Data Storage Paradigms, IV1351

1 Avoid Entities Without Attributes

While both a domain model (DM) and a conceptual model (CM) are descriptions of the reality, the important difference is that a DM is intended as a basis for *program* development, and a CM is used as a basis for *datastorage* development. This means that a *CM has more focus on data*, since that's what's stored in a data storage. This data is present in a CM in the form of attributes. The entities (classes) in a CM is more of a grouping of attributes, to make them easier to understand and manage. A *DM, on the other hand, has more focus on classes* (entities) than attributes. Associations, finally, are important in both a DM and a CM, but for slightly different purposes. Associations in a DM can, at least to some extent, be seen as comments that explain the purposes of the classes. In a CM, the associations are more of actual relations, that show how data is joined to give all the desired information.

These differences between DM and CM can be discussed and evaluated for a very long time, but for now let's just focus on one obvious consequence. **A CM mustn't contain any entity without attributes**, even though that's perfectly fine in a DM. An entity without attributes is an entity without data, which makes it meaningless, since in the end data is all that matters in a CM. This is illustrated in Figure 1, which shows both a DM and a CM for a simple hotel booking system. Much can be said about these two models, but the purpose here is only to show an example of a useful, although not perfect, DM without attributes, while a CM with an attribute-less entity would only raise questions.

2 Actions Are Normally Not Important

It's not forbidden to let a CM show who does what, and where it's done, but it's **often best to avoid illustrating such actions**. The reason is that actions seldom include data, and are thus not of interest in a CM. Say for example that we are modelling a database for a grocery store, and the description of the business says something like "the cashier scans the items the customer is buying". That might make us create a CM which shows that *Cashier scans Item*. Such a relation is, however, often meaningless. One reason it could being meaningless is if the data storage isn't really supposed to

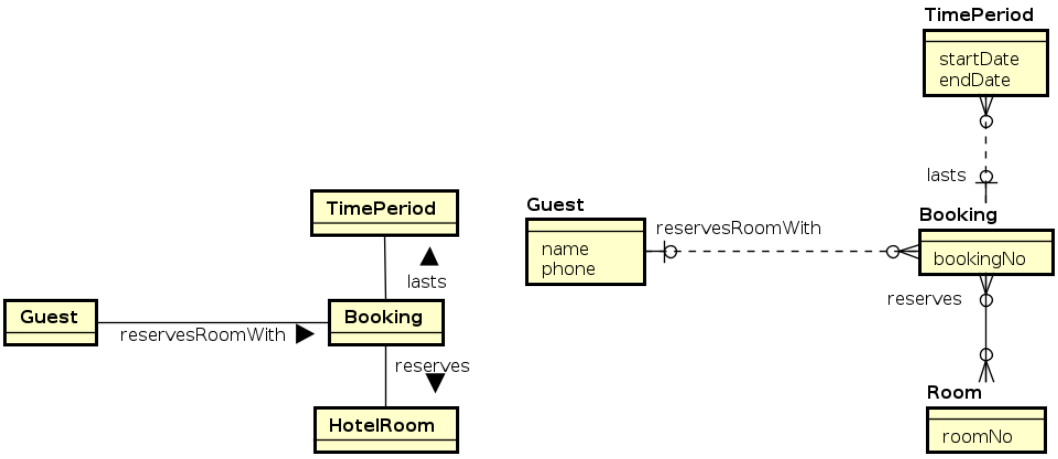


Figure 1: As shown in (a), it's fine to have classes without attributes in a DM, but as shown in (b), there mustn't be any entities without attributes in a CM.

handle cashiers at all, and the business description doesn't mention any data related to a cashier. This would make `Cashier` an entity without attributes. However, even if the data storage, and thus the CM, was supposed to handle cashiers, the relation *Cashier scans Item* would still be quite meaningless. The reason is that it's highly unlikely that we would ever be interested in finding out which cashier scanned a particular item, or which items a particular cashier has scanned. The only reason there can be for including *Cashier scans Item* in a CM is if this relation is clearly asked for, maybe in the unlikely case that the cashier's payment depends on the number of scanned items.

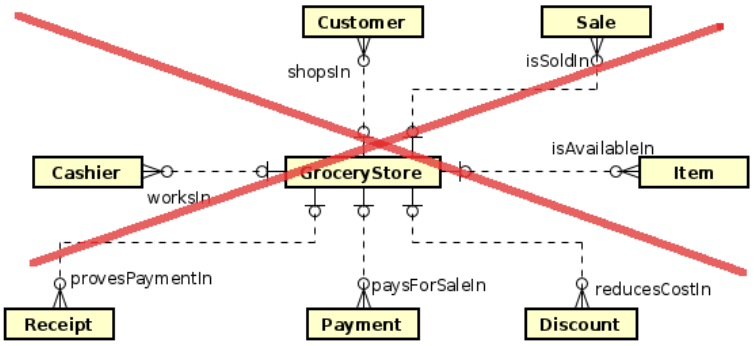


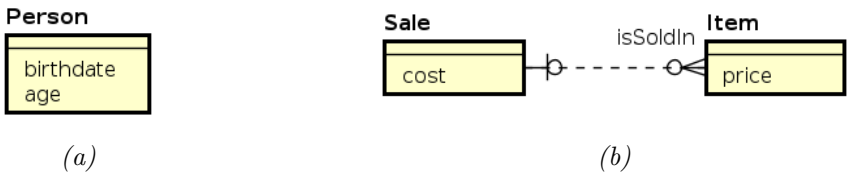
*Figure 2: Partial CM where the location (GroceryStore) is, **erroneously**, included and associated with many other entities. Note that attributes are excluded from the diagram, and also relations not involving the location. Both should of course be there in a complete CM.*

Now let's have a look at the location where the action, in this case the scanning of an item, takes place. We shall be reluctant also to include this location in the CM.

If we do include the location, the CM will show that *Item isSoldIn GroceryStore* and that *Cashier worksIn GroceryStore*. This might seem fine, but the problem is that the CM will be completely cluttered with relations to the `GroceryStore` entity, as is the case in Figure 2. More or less every entity in the CM is located in the store, since the CM is in fact describing exactly the business in that store. The solution to this problem depends on whether the database is actually supposed to include the location. If it's not, we of course omit the location completely. This happens for example if the company has only one store, in which case it's obvious that everything takes place in that store. The situation is more complex if the company has more stores, because then they probably want to now in which store a particular sale took place. In this case we have to include the location in the CM, but we still don't want to relate every other entity with `GroceryStore`. The solution here is to make `GroceryStore` a quite peripheral entity, related only to one (or very few) other important and central entity. An example of such a central entity in a grocery store CM could be `Sale`, since the business is all about sales. This means we could create a relation *Sale isMadeIn GroceryStore*, and then, no matter in which entity we have to start, we can hopefully find the store through `Sale`.

3 Be Careful With Derived Data

Derived data is data that can be calculated from other data in the data storage. An example is an entity `Person`, that has an attribute `birthdate` and another attribute `age`, Figure 3a. `Age` is a derived data since it can be calculated from the birthdate, instead of being stored in an attribute. A less obvious example is found in Figure 3b, which models part of the grocery store application mentioned above. There's an entity `Sale` with a one-to-many relation with an entity `Item`. The items represent all that's bought in the sale. If `Item` has an attribute `price`, which tells how much that item costs, and `Sale` has an attribute `cost`, which tells the total cost of the sale, then `cost` is derived, since it can be calculated from the prices of all bought items.



(a) *(b)*

Figure 3: Two examples of derived attributes. Age is derived in (a), and cost is derived in (b).

Now, even though a CM models the reality, let's instead for a moment consider databases, since that's where the pros and cons of derived data are obvious. In an actual datastorage, the disadvantage of derived data is that writing to the data storage is slower, since more data must be written and more entities might be updated than if there wasn't any derived data. Also, derived data is against the spirit of relational data storage, which has data integrity as it's main goal. One important way to ensure data integrity is to never store duplicated data, which will in fact be done if we have

derived data, since the derived data is a kind of duplication. The advantage of using derived data, on the other hand, is that reading is faster and maybe also easier when there is derived data. The bottom line is that we shall **not include derived data in a database unless we're very sure it's needed**. The reason is mainly the risk of loosing data integrity. Also, the read speed is very seldom a problem, and we must never optimize without being sure it's needed. That is, don't create derived data to improve read performance unless there's a measurement clearly showing that reads really are too slow. Also, the simplicity of extracting data can be achieved in better ways than derived data, more on that when database queries are covered later in the course.

Now we know that derived data shall be avoided in the actual datastorage, but what about the conceptual model? There isn't any obvious reason to ban such data from the CM, but there's also seldom a good reason to include it. A rule of thumb can be to allow derived data if it clearly clarifies the CM. The downside is that we have to remove it when creating the database, which might be extra work and might easily be forgotten.

4 How to Know That the Domain Model Contains All Relevant Data

The CM can't be considered complete unless, as stated in the headline above, it contains all data that's relevant for the business. How to verify that this requirement is met is a difficult task. Most important is, however, to just remember that the requirement must be met, and critically evaluate if it can be argued that the CM meets it.

In a real project, it's mandatory to evaluate the conceptual model together with the customer. Here, we have no customer, but are instead forced to do what we can on our own. It's not possible to say much about how to do this, the point here is rather to emphasize that it must be done. We just have to **carefully read the description of the Soundgood music school once we've created a CM, and do our best to verify that all data required by the text does in fact exist in our CM**. Below follows some examples, to illustrate how we might reason. Note that this reasoning isn't something that's done only last in the process, when we think the model might be complete. We do similar reasoning when creating the model, in order to decide which entities and attributes to include.

- As a first example, the Soundgood description says that *Person number, name, address and contact details must be stored for each student. It must also be possible to store contact details for a contact person* What are these *contact details*? We can't be sure without asking Soundgood, which of course is impossible here. Thus we just have to consider how such a school might want to contact their students and the students' contact persons. Will they want to reach someone as quickly as possible? Will they want some kind of signature from the recipient?

- It's also stated that *it must be possible to see which students are siblings, since there is a discount for siblings.* Note that they want to see which students are siblings, not just if a student has siblings. This should however be clear even if

it wasn't explicitly stated, since, if we only recorded if a student had siblings and not who the sibling was, it wouldn't be possible to change this information if the sibling quit. If two students are siblings, and there's just a boolean value saying they both have a sibling, there's no way of knowing if the student that quit was in fact the sibling.

- As a final example, let's consider instructor payment. We know that the model must contain all data required to calculate payments, since it's stated that *There are no instructors with fixed monthly salaries, instead they are payed monthly for all lessons given during the previous month*, and it's also stated that it shall be possible to calculate *what sum shall be payed to or by who*. From this we can conclude that, as a minimum, the model must contain the instructor payment for all types of lessons, and also that it must be possible to see which lessons each instructor has given. However, it doesn't say that there must be an entity like `InstructorPayment`, that's for us to decide. Such an entity might be derived data, which isn't the best of ideas, as discussed above.

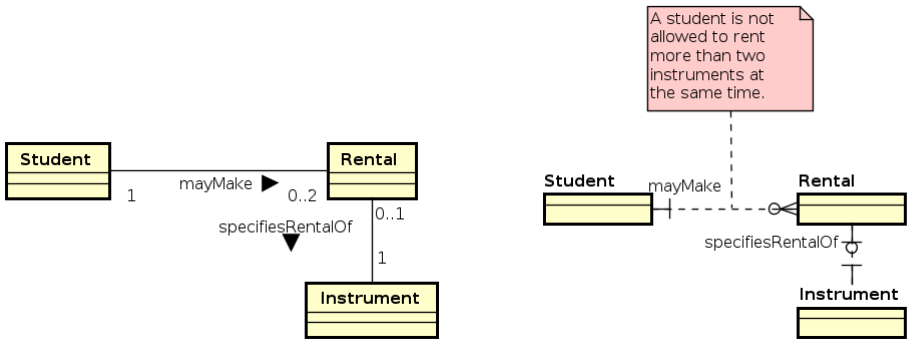
5 How to Model Business Rules

There are a few different definitions of exactly what *business rule* means, but that's not really important for the reasoning done here. Let's just say that there are things in a description of a business that's hard to model. Before looking at such things we can ask if everything in the business description really has to be included in the model, and, if not, what can be omitted. It's hard to give an exact answer to that, but a good guideline is that **everything related to data, including allowed and forbidden values of data, must be mentioned**. Now let's have a look at how to model some kinds of rules about data.

First, we can see that the organization who's data we are modelling may have **rules that can quite easily be modelled**, at least in some diagram notations. Such business rules are related to cardinality, being optional, and uniqueness of attributes and relations. We can for example specify that the cardinality of an attribute or relation representing rented instruments in the Soundgood music school is 0..2, meaning that a student is allowed to rent 0, 1 or 2 instruments at the same time. Such business rules shall be included in the diagrams, as is illustrated in Figure 4.

Then there are categories of business **rules that are harder to model**. This can for example be domains, which are limitations on allowed values of attributes. An example of this from the Soundgood CM is that there's an attribute `skillLevel`, which must be exactly 'beginner', 'intermediate' or 'advanced'. In task two, when looking at the actual database, we'll consider ways to include this in the diagram. Now, in the CM, however, we'll just write a note describing such domain information, see Figure 5a for an example.

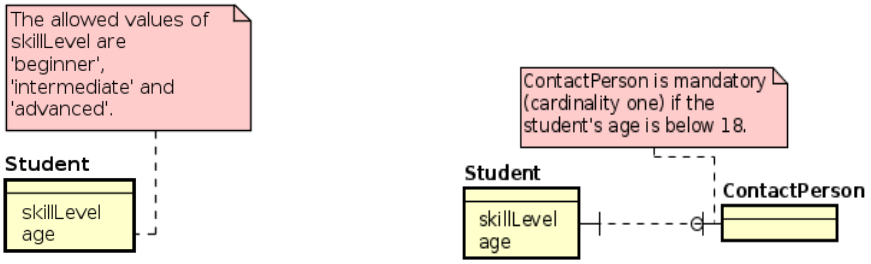
Finally, there are categories of business **rules that are impossible to include in a diagram**, at least with the notations considered in this course. An example is rules related to values of one or more attributes, imagine for example that Soundgood requires a contact person for each student under 18, but not for students above 18. This means



(a) *(b)*

Figure 4: Partial CM, with attributes omitted, illustrating that cardinality of a relationship is easy to show in UML notation, as is seen in (a). Using IE notation it's instead necessary to make a note, as is the case in (b).

that contact person is optional in some cases, and mandatory in other, depending on the value of an attribute telling the students age. In this course, and also often in reality, we have to be pragmatic and model such business rules using either plain text accompanying the diagram, or a note in the diagram, as illustrated in Figure 5b. Before leaving the topic, it's worth mentioning that this kind of rules can in fact be included in the actual database, using types, triggers and functions. We'll come back to that in task 2, to a small extent, but it's mainly outside this course.



(a) *(b)*

Figure 5: (a) Using a note to model a business rule specifying domain.
(b) Using a note to model a business rule that depends on an attribute value.
Note that both diagrams are intended only as illustrations of business rule modelling, ***not as hints on how to model the Soundgood music school.***

Quite a lot has been written about modelling business rules, below are links for those who wish to learn more. Just be aware that all on those links is far outside this course.

- https://tdan.com/modeling-business-rules-what-data-models-do/5174
- https://tdan.com/modeling-business-rules-what-data-models-cannot-do/5190
- https://etutorials.org/SQL/Database+design+for+mere+mortals/