Chapter 8

Conceptual Modelling

Conceptual Modelling (sometimes called *Domain Modelling*) is the activity of finding out which concepts are important to our system. This process helps us to understand the problem further, and develop a better awareness of our customer's business.

Once again, the UML does not tell us how or when to do Domain Modelling, but it does provide us with the syntax to express the model. The model we are going to use is the **class diagram**.

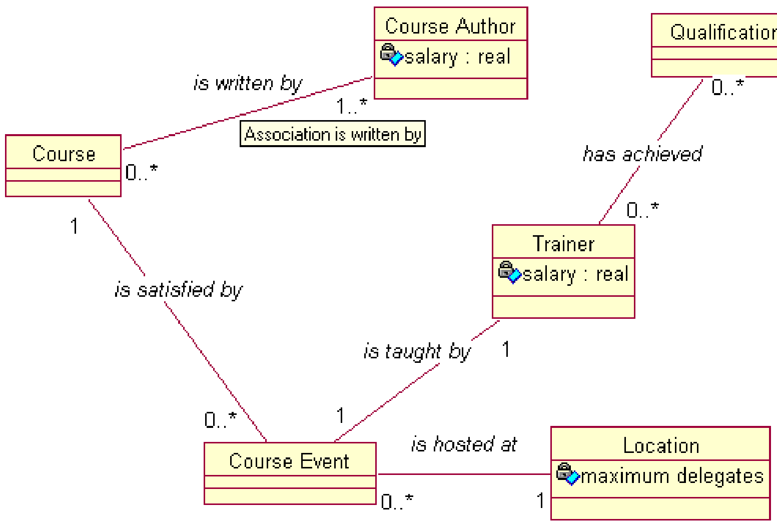

Figure 1 - A UML Class Diagram

Developing a Class Diagram is key to any object oriented design process. The Class Diagram will essentially provide the structure of the eventual code we produce.

At this stage, however, we are not yet interested in the system design (we are still analysing), so the class diagram we produce at this stage will be quite sketchy, and will not contain any design decisions.

For example, we would not want to add a "LinkList" class at this stage, as that is tying us down to a particular solution far too early.

We are essentially producing an **Analysis Class Diagram**. Many practitioners prefer to completely distinguish their Analysis Class Diagram from the Design Class Diagram by calling the Analysis Diagram the **Conceptual Model** – a term I shall use for the rest of this course.

On the conceptual model, we aim to capture all of the concepts or ideas that the customer recognises. For example, some good examples of concepts would be:

- **Lift** in a lift control system
- **Order** in a home shopping system
- **Footballer** in a football transfers system (or a PlayStation football game!)
- **Trainer** in a stock management system for a shoe shop
- **Room** in a room booking system

Some very bad examples of concepts are:

- **OrderPurgeDaemon** the process that regularly deletes old orders from the system
- **EventTrigger** – the special process that waits for 5 minutes and then tells the system to wake up and do something
- **CustomerDetailsForm** – the window that asks for details of the new customer in a shopping system
- **DbArchiveTable** – the database table holding a list of all old orders

These are bad concepts, because they are focussing on design – the solution, and not the problem. In the DbArchiveTable example, we are already tying ourselves down to a relational database solution. What if it turns out later that it is more efficient, cheaper, and perfectly acceptable to use a simple text file?

The best rule of thumb here is:

If the customer doesn't understand the concept, it probably isn't a concept!

Designers hate the conceptual step – they cannot wait to crack on with design. We shall see, however, that the conceptual model will slowly transform into a full design class diagram as we move through the construction phase.

Finding Concepts

I recommend a similar approach to finding Use Cases – a workshop is best – once again, with as many interested *stakeholders* as possible.

Brainstorm suggestions for concepts, capture **all** the suggestions. Once brainstorming is complete, work as a group to discuss and justify each suggestion. Apply the rule of thumb that the customer must understand the concept, and discard any that don't apply to the problem, and discard any that are touching on design.

Extracting Concepts From Requirements

The requirements document is a good source of concepts. Craig Larman (ref [2]) suggests the following candidate concepts from the requirements:

- Physical or tangible objects
- Places
- Transactions
- Roles of People (eg Customer, Sales Clerk)
- Containers for other Concepts

- Other Systems external to the system (eg Remote Database)
- Abstract Nouns (eg Thirst)
- Organisations
- Event (eg Emergency)
- Rules/Policies
- Records/Logs

A couple of points here. First of all, gathering concepts in a mechanical manner is poor practise. The above list are good suggestions, but it would be wrong to think that it is enough to run through the requirements document with a highlighter pen, pulling out some phrases and setting them as concepts. You **must** have input from the customer.

Secondly, many practitioners suggest extracting **noun phrases** from documents. This approach has been is common usage for almost 20 years, and although there is nothing inherently wrong with it, I hate the implication that mechanically searching for nouns will result in a good list of concepts/classes. Sadly, the English language is far too ambiguous to allow for such a mechanical approach. I'll say it again – **input from the customer is essential!**

The Conceptual Model in the UML

Now that we've seen how to discover the concepts, we need to look how to capture the concepts in the UML. We'll use the core aspects of the **class diagram**.

We represent our concept in a simple box, with the title of the concept (by convention, capitalised) at the top of the box.

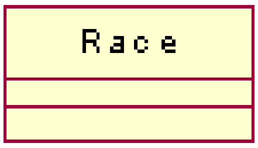

Figure 2 - The "Race" concept captured in UML (for a Horse Racing System)

Notice that inside the large box are two smaller, empty boxes. The box in the middle will be used shortly, to capture the **attributes** of the concept – more on this in a moment. The bottom box is used to capture the **behaviour** of the concept – in other words, what (if anything) the concept can actually do. Deciding on the behaviour of the concept is a complicated step, and we defer this stage until we are in the design stage of construction. So we needn't worry about behaviour for now.

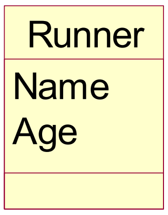

Figure 3 - The UML captures the attributes and behaviour of a concept

In the example above, we have decided that every runner will have two attributes – "Name" and "Age". We leave the bottom area blank until later, when we decide what a "Runner" is able to do.

Finding Attributes

We need to determine what the attributes of each concept are – and again, a brainstorming session with the stakeholders is probably the best way to achieve this.

Often, arguments arise over whether or not an attribute should become a concept on its own. For example, lets say we are working on a Staff Management system. We have identified that one concept would be "Manager". A suggestion for an attribute might be "salary", as follows:

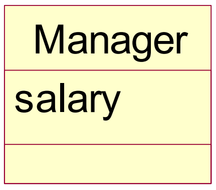

Figure 4 - Manager concept, with the attribute "Salary"

That looks fine, but someone might argue that "Salary" is also a concept. So, should we make promote it from an attribute to a concept?

I have seen many modelling sessions descend into chaotic arguments over issues like this one, and my advice is to simply not worry about it*: if in doubt, make it another concept.* These kind of problems usually resolve themselves later on anyway, and it really isn't worth wasting valuable modelling time on arguments!

Manager
Salary

Figure 5 - Manager and Salary, two separate concepts

Guidelines for Finding Attributes

The following rules of thumb may be helpful when deciding between attributes and concepts – but heed the advice above and don't worry too much about the distinction. *If in doubt, make it a concept!*

- Single valued strings or numbers are usually attributes[1]
- If a property of a concept cannot *do* anything, it might be an attribute - eg for the manager concept, "Name" clearly sounds like an attribute. "Company Car" sounds like a concept, because we need to store information about each car such as the registration number and colour.

Associations

The next step is to decide how our concepts are related. In any non-trivial system, at least some of the concepts are going to have some kind of conceptual relationship with other concepts. For example, back to our Staff Management system, given the following two concepts:

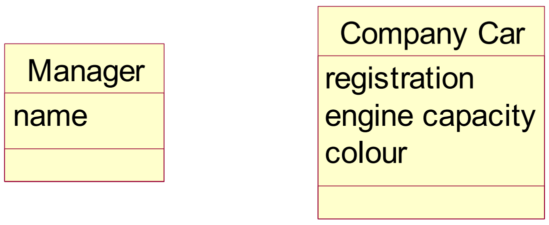

Figure 6 - Manager and Company Car Concepts

These concepts are related, because in the company we are developing a system for, each Manager drives a Company Car.

We can express this relationship in the UML by connecting the two concepts together with a single line (called an **association**), as follows:

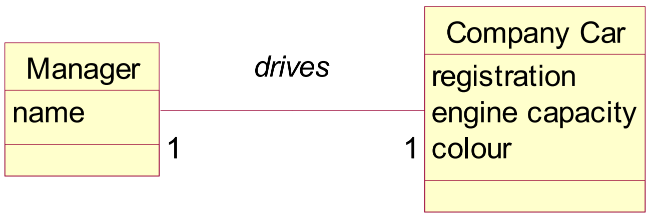

Figure 7 - "Manager" and "Company Car" related by association

Two important things to note about this association. First of all, the association has a descriptive name – in this case, "drives". Secondly, there are numbers at each end of the association. These numbers describe the **cardinality** of the association, and tell us how many instances of each concept are allowed.

[1] But not always – this is a rule of thumb and shouldn't be followed slavishly.

In this example, we are saying that "Each Manager Drives 1 Company Car", and (going from right to left) "Each Company Car is driven by 1 Manager".

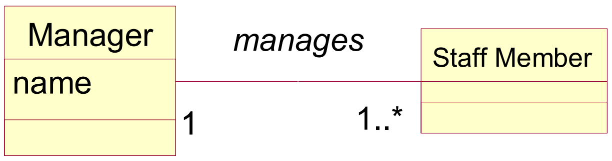

Figure 8 - Another Association Example

In the above example, we see that "Each Manager manages 1 or more staff members"; and (going the other way), "Each Staff Member is managed by 1 Manager".

Every association should work like this – when reading the association back in English, the English sentence should make perfect sense (especially to the customer). When deciding upon association names, avoid weak names like "has" or "is associated to" – using such weak language could easily hide problems or errors that would otherwise have been uncovered if the association name was more meaningful.

Possible Cardinalities

Basically, there are no restrictions on the cardinalities you are able to specify. The following diagram lists some examples, although this list is by no means exhaustive. The * notation denotes "many". Note the slight difference between "*" and "1..*". The former is a vague "many", meaning that perhaps any number of concepts are allowed, or maybe we haven't made the decision yet. The latter is more concrete, implying that one or more are allowed.

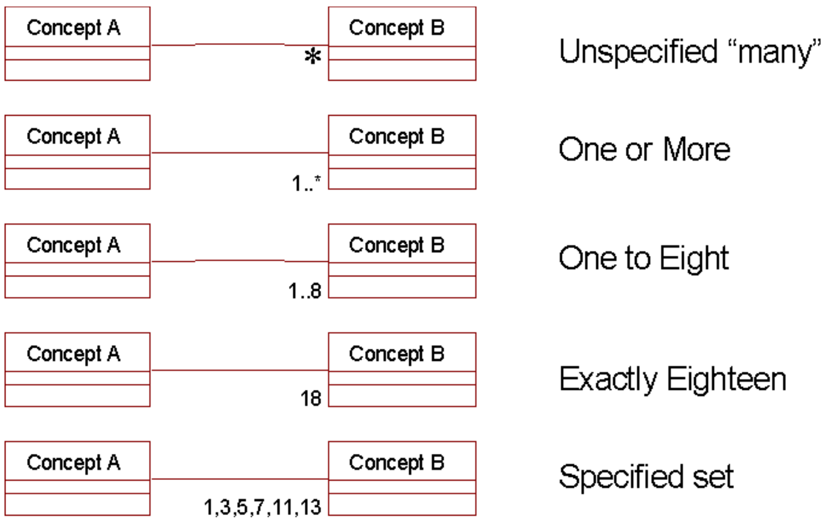

Figure 9 - Example Cardinalities

Building the Complete Model

Finally, let's look at a methodical system for determining the associations between the concepts. Assume we have completed the brainstorm session and uncovered several concepts for the Staff Management system. The set of concepts are in the figure below (I've omitted the attributes for clarity).

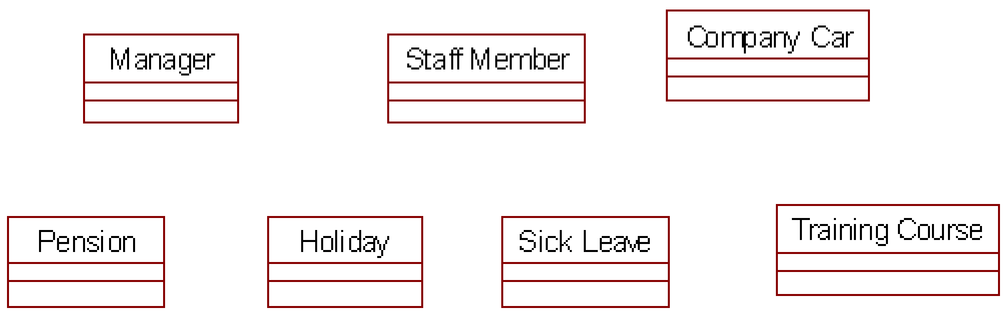

Figure 10 - Set of Concepts for Staff Management

The best way to proceed is to "fix" one concept, say "Manager" and consider every other concept in turn. Ask yourself "are these two concepts related?", and if so, immediately decide on the name of the association, and the cardinality…

> "Are **Manager** and **Staff Member** related? Yes, *Each Manager manages 1 or more staff members*.
> **Manager** and **Company Car**? Yes, *Each Manager drives 1 company car.*
> **Manager** and **Pension**? Yes, *Each Manager contributes to 1 pension*"

And so on until the model is complete. A common mistake at this stage is to decide two concepts are related, draw a line on the diagram and leave off the association name until later. This is making extra work for yourself – you'll find that once you've finished adding lines, you'll have no idea what any of them mean (and they'll usually look like spaghetti), and you have to start all over again!

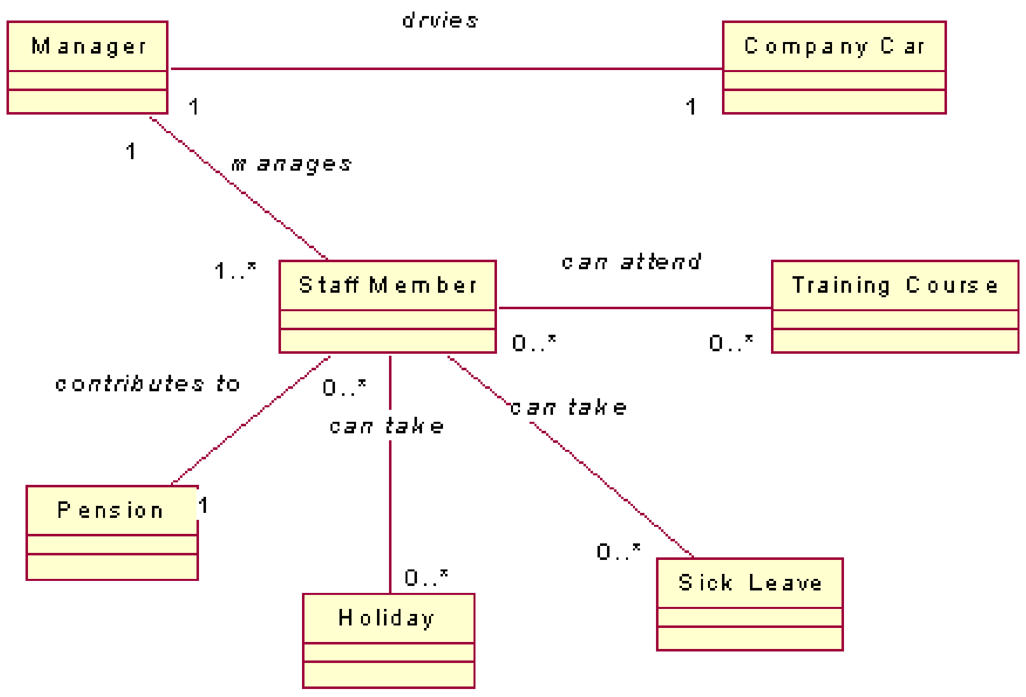

Figure 11 - The simple conceptual model, completed

When building the model, it is important to remember that associations are less important than attributes. Any missing associations will be easily picked up during design, but it is harder to spot missing attributes.

Furthermore, it can be tempting to overspecify the map of associations "just in case", and end up with quite a confusing and complex diagram. So a good rule of thumb is to concentrate on concepts and attributes, and try to fix the most obvious associations.

At the end of the modelling, the diagram should make sense to the customer when you "read back" the diagram in English.

Summary

Conceptual Models provide a powerful way of investigating the problem further.

Later on, we'll expand our model into the design aspects.

This model will eventually be one of the key inputs when we build the code.

To build the model, use a workshop technique as with the Use Cases.