Critical Review

The use of software models is nothing new to software engineering techniques. Many methodologies have defined their own modeling infrastructures[1]. While the notation differs between them, the one thing that they have in common is that they're depicting scenarios for a given problem.

A model is an abstract representation of the system[2]. They can be constructed to suit a given task scenario, and where applicable help developers that work on a problem to conceptualise ideas. But the model should help depict the system overall. Visual representations of what the system are also helpful when verification takes place between the client and what is being produced. It's important to remember also that models are abstract for a reason – they're not real. They're a tool that model behavour and state of the system as it is built.

Of course, structural methods are important in the development of the models. Software modeling relies on them. The waterfall method[3] was one of the first structured approaches in producing software (and hence software models), but there were limitations in doing so. The fact that one is unable to go back to a previous step in the life cycle meant that the product at the end was most likely going to fail. Although what this model does show is that the *compartmentalisation* of information is the best way forward – if the problem can be broken down into stages, a set of logical instructions can then be drawn up.

Yet despite the different software life cycle techniques that can be applied, there is a commonality between both of them. All of them use a means of specifying stages that events take place. Of course, the end means is the same – the overall product will be produced.

It is the job of developers to express systems that they are building in two ways. One is to define program *structure* and the other is to define program *semantics*. The structure is very important since without that, the semantics can never be applied. This is where using a *methodology* is applied, since these are guidelines which can be applied when building a piece of software. Commonality and experience of solutions are all encompassed into a methodology so that a generic template can be formed from which software can be developed.

There is nothing to say that Yourdon, as a methodology, is any better than SSADM[4]. But Yourdon's methodology was applied to the development of the *encoded data systems* project. This allows for the development of various models at each state of the specification given. A *context diagram* is used to get an overview of the system that is to be developed. Perhaps more importantly, it also establishes a boundary between that which is to be modeled, and that which is not. This is essential early to define this – since it means we can filter out what is actually being modelled.

But a context diagam also shows entities – those beings which interact (or will do) with the system but are not as a direct consequence of it. Entities are extremely useful to us to define in a context diagram, since often they are the starting place for inputs of data to the system being modelled. For the encoded data systems design, the boundary was defined from the fact that the system being

1 SSADM, Yourdon, etc.
2 http://en.wikipedia.org/wiki/Model_%28abstract%29
3 http://en.wikipedia.org/wiki/Waterfall_model
4 The argument that either one is “better” is subjective at best, even if he was the founder of the SSADM methodology.

modelled was only interested in the software of the inception computer. The entities defined in this case are those which interact with the inception computer only. This diagram then forms the building blocks to continue further.

The diagrams themselves are derived from the *requirements specification*, the set of guidelines which tell the developer what the intended system is to do. The context diagram therefore gathers these in a very highlevel overview. It was interesting in that the requirements specification had a lot of extraneous detail in it that simply was not needed for the overall design of the intended system.

From the context diagram, one is then able to (in part) derive the *data flow diagram (DFD)*. Depending on the complexity of the system, these could go many levels deep; but it is at this point that the flow of data is depicted in the system. Perhaps even more so, the idea of tranforms are used. These will eventually become functions that perform a service, with flows of information in and out of them. For the encoded data systems, two levels of DFDs were used. Level 0 shows a high level abstraction of likely transforms and data flows, whilst the level 1 DFD shows those transforms broken down into lower-level transforms. In the case of the former diagram, this establishes three things. The first is that tranforms are used to receive data and transform data. The second shows the data from the external entities, and to which transforms these will be passing, and perhaps more importantly they then show the output of data from transforms. This output of data might be to a data store, or simply back to an external entity depicted on the context diagram.

Although all the while it is important to realise that the flow of information out of a transform is different from the input it received, At this level we're defining what happens to the data and how it changes. This is cruitial since it allows a developer to accurately model this, so that it can be verified and validated as part of the design process. The level 1 DFD is no exception to any of this[5].

So the DFD is drawn to show the flow of data. It's needed to track where the data goes, to identify any data stores, and then to define the transforms which act upon the data in some way. What these DFDs do not show are the order in which something is done – for that to happen, a *state transition diagram* is produced.

A state transition diagram shows the order in which events occur – that is to say, the order in which the transforms take place. They are also useful in determining the responsibility of a method. For instance, in the encoded data systems design, it was obvious that there was a transform which switched between other transforms. This would later become a *control transform*[6].

The *integrated model* goes on to really pull of the diagrams together to co-ordinate the flows and actions events of before. The control transform is used to determine the execution order of each of the transforms and to ultimately depict the system as an integrated whole. It's perhaps from this that code can be developed. By this stage, each of the transforms should be adequately explained such that the code can be produced. The *data dictionary* and structure chart are means of helping with this – especially the structure chart as that defines the order also that the tranforms are invoked and with what data (and return flags, return data, where applicable, etc).

5 http://yourdon.com/books/msa2e/CH09/CH09.html
6 Ibid.

It has already been mentioned how modeling supports the design process by providing a visual representation for both the developer and the client. But it is also useful in the delegation of tasks to individual teams of developers where necessary. Using models in this way should allow for the disambiguation of the flow of data. The design process is supported via modelling by allowing this key factor. But as previously mentioned, it does also allow a visual check between customer and developer.

Of course, the intended outcome of modelling is to be able to produce functional code that will all fit together into a coherent, working sytem. The design as it is for the encoded data system is far from perfect. Firstly, because there is no right or wrong way of designing the system[7] it is hard to gague whether the diagrams are accurate at each stage. Although as it is designed currently, the encoded data systems design has both its own strengths and weaknesses.

The design is such that overall there is only one data store which houses all the information that the inception computer uses. While the centralisation of data in that way is perhaps preferrable in some instance, it might not be the case for this system. Especially where the data is subject to change in volume, given certain product batches[8].

The other disadvantage in using one filestore is that it could be subject to file locks, where a process might request data from it while another process tries to update the data being read. This is not good for all kinds of reasons, not least of which for race conditions. Corruption is another possibility.

Furthermore, the design only has three main transforms overall. They do too much in terms of functionality, and would greatly benefit from being sub-divided further into more manageable tasks. While decomposition was defined in the Level 1 DFD, even that is too abstract, in my opinion.

This is then also has a knock-on effect for the data flows. Because of the way the requirements specification is ordered, there is a tendency to group data together. This makes the data flows somewhat abstract in trying to encapsulate the information being conveyed, rather than separating it out onto the diagram. But then the reason for aggregating the information in this way is necessary – both for ease of understanding (when applied to modeling techniques), and for the need to keep the diagram uncluttered.

Whether the entire system could be coded from this design, is unclear. The modeling techniques applied, although correct notationally, are lacking in terms of implementability. There is an unbalanced flow of information to and from the main process store[9], this could create a bottleneck of information flow when the implementation stage is gathered.

The coded example for the encoded data systems, defines the printing of the jobcard. The accompanying test table demonstrates how one aspect (transform) can be coded from the design process undertaken. This shows in part that the design implemtation used did at least work for one aspect.

---

7 Other than to ensure that it meets the requirement specification.

8 Assuming that the products change.

9 This is to be expected where there is only one file store – again, something which can be addressed with future revisions. Yourdon, however defines this here: http://yourdon.com/books/msa2e/CH09/CH09.html

It is interesting to note the difference between procedural and object-oriented (OO) methods of designing software. The thing to bear in mind, is that both methods, although different are just a means to an end of developing a software product[10].

Procedurally, assuming one uses a software engineering lifecycle such as the waterfall model, a product is eventually produced by looking at the flow of data, doing some sort of functional decomposition, and producing a coded artefact at the end. But this approach assumes that all of the details about how the product operates to begin with.

OO on the otherhand, takes a different approach in that OO builds a model framework, and then fits the data to suit that framework. The two techniques are quite similar to each other in many respects they're just different ways of building a system – working code is still produced.

For example, the *entity relationship diagram (ERD)* for the encoded data systems design specification shows how the entities of the system are related. This model is not unlike that of the class diagrams present in OO. But unlike the OO class diagrams, ERDs only show the attributes. OO class diagrams however, show the attributes and methods present.

10 The argument about whether one technique is "better" than the other is something that is not for discussion here.