

# Making IT Application Design an Engineering Discipline

Chris Britton

October 2006

Web site: [www.itmodeller.co.uk](http://www.itmodeller.co.uk)

Ever since I started working in IT – a very long time ago – people have been trying to make IT application development more like engineering. We in IT have looked with envy at the ability of civil engineers to design buildings that don't fall down, and at the ability of electrical engineers to reuse more and more complex components to make more and more effective devices. It has often been remarked that IT program design is more like a craft than an engineering discipline, in spite of the profession often calling itself “software engineers”. We work with rules of thumb and pattern books, and we create most of our components from scratch every time, all characteristics of craftsmen not engineers. But I believe that an engineering approach is possible and I believe I have made a start, albeit on a long road.

But first, should we bother? Almost everything made before the 19 century was made without an engineer and this, of course, includes many impressive buildings and many ingenious devices. Furthermore, engineering cannot guarantee a project will be on time and on budget; engineering is no substitute for good project management. Nor can engineering guarantee the buying public will like the product and make it a commercial success. So what exactly does engineering deliver? What engineering has done is enable us to build bigger, and more safely. You can build a cottage without an engineer but not a sky scraper. You can build a stone, arched bridge but not a suspension bridge. There are many people in IT who claim they build good applications to time and budget and, if they are building the IT equivalent of cottages that may well be true. It is the large projects and the technically demanding projects that suffer from a lack of an engineering discipline. And the evidence points strongly to the need; IT has a very poor record when it comes to large projects.

Another important advantage of engineering is that it gives us a much more accurate concept of project success. If an engineer builds a bridge and it falls down in the first winter storm, the engineer will most likely take the blame. If an IT contractor builds an application and it fails when the load gets high or loses some messages while recovering from a power cut, the IT contractor is likely to say that working under these conditions isn't in the specification and they will only fix it with a large injection of money.

One reason why it has been so difficult to apply engineering ideas to IT is that IT development is simply different from classic engineering disciplines like civil engineering, electrical engineering and mechanical engineering. It is common to view programming as the production phase of software engineering but it isn't. Programming is much more like a design task than a mechanical, albeit skilled,

production task. Thus, the real equivalent to production in IT is little more than installation. IT development is also unique in that you must expect the users to lack skills. You don't need a license to drive a software program. Many of the users are naïve or incompetent and some of them are malicious. Furthermore an IT designer must design in the knowledge that components will fail suddenly and unexpectedly not only because of software error but also because of human error, hardware error, power failure and network failure. You cannot anticipate failure like you can check an airplane for fatigue. Finally, changes after production are more prevalent in IT applications than in other engineering tasks (engineers might disagree with this statement) because of the difficulties explaining to all interested parties exactly what the application will do.

While IT is unique, is it so unique that engineering does not apply? There have been many attempts to put engineering practices into IT and these attempts have given rise to many useful ideas and approaches over the years. The notion of frameworks – like the Zachmann framework – has helped IT development to become more comprehensive by at least viewing, if not analysing, IT systems from multiple perspectives. Notions of components, abstraction and layering (like the ISO seven layer model) have been useful, not to say essential, techniques for organizing large programs. Test-driven development is an example of a discipline that has made major improvements in programming quality. But none of these amounts to engineering. The single most important characteristic of engineering is the analysis of design starting early on and continuing throughout. We do very little of this in IT; almost all testing is done on the completed application or completed chunks of the application.

Unfortunately the notion of engineering in IT application development has been linked in people's minds with "big design up front" – the notion of gathering a large numbers of requirements into a large, unwieldy document, followed by design that draws lots of diagrams and writes another large, unwieldy documents, all of which is turned over to an implementation team. This is unfortunate not only because "big design up front" has a history of failure but also because it is not engineering. There are two main reasons. First, we, humans, don't design by mulling over thousands of requirements. Instead we start at one point and expand outwards. Secondly, big design up front fundamentally misunderstands the relationship between design and requirements. Requirements gathering is part of design, not separate from it, as I shall explain in the next section.

Extreme programming and other "agile" approaches are more in tune with how design is actually done. Two fundamental practices (not the only two) that underpin agile techniques are designing and implementing in small increments and having rapid feedback from the users. This is good for refining how the interaction between user and system. It stumbles when many people are involved or when not all of them can give rapid feedback or when they simply disagree among themselves. It fails badly when it comes to analyse error recovery in a business context when the failure has a wider area of impact than is covered by the users giving feedback. It fails completely when developing an IT application that has a lot of invisible structure, a compiler for instance. Here you must have a design for the programming language itself, the tokenizer, the parser, the code generator, and possibly the intermediate code and the optimizer, all requiring complex, detailed design, before a single line can be compiled.

Fundamental to my whole thesis is a view on how engineering design is done and how IT design should be done. This I cover in the next section. Following this section are two worked examples applying these ideas to IT, and that is followed by the beginnings of an attempt to lay a theoretical foundation for this work. Finally there are some comments on progress to date and direction.

## What is Design?

The key point I want to make is that design is like a hypothesis.

The word hypothesis is most frequently used in science, and it was Karl Popper who introduced the idea that scientific endeavour was essentially about cumulative hypothesis building. Before Karl Popper, science already moved forward through hypothesis building, the difference was that scientists didn't explain it that way. They said that science worked by an accumulation of facts and then someone comes along and develops a theory that puts the facts in order. Sounds a bit familiar? Change the word "facts" to "requirements" and you have the classic description of big design up front. No one does design this way.

The hypothesis that design is like a hypothesis applies to all design; architecture, designing a chair, designing a car and business process design for instance. In other words, we humans do design one way and one way only. Although we start by looking at the requirements, in fact we only pay attention to only a few of them (because our brains aren't big enough to think about many facts simultaneously). This is analogous to a scientist looking at the facts and the experimental results they want to explain. We then create a rough design. This is analogous to the scientist developing a hypothesis so I will call it a *design hypothesis*. The next step is clarification; we check through the remaining requirements/facts and check whether the design hypothesis still works, possibly moulding it a bit to make it fit. Maybe at this stage the design hypothesis fails and we go back, probably a bit wiser and probably paying attention to more critical requirements. So we try again with another design hypothesis and go through the clarification stage again. When the design hypothesis is ready, the next stage is to try to knock it down. Scientists do this by experiments. Engineers do it by analysis; running the design through various calculations to check whether the design will break under load. A non-engineering discipline – like the cathedral builders in the middle ages – left out this step altogether and quite frequently their new, expensive and beautifully crafted cathedrals fell down. This is the state of IT application development today.

To summarize, to build a design requires 4 steps:

1. *Understand*. Gather the basic requirements.
2. *Select design hypothesis*. Brainstorm the options and select one design hypothesis.
3. *Clarify*. Add detail to the design hypothesis. Go through the design checking off all the requirements, possibly adding to and changing design details. If this is found to be too difficult, go back a stage, select another design hypothesis, and try again.

4. *Analyze*. Check the design is complete and works. If it isn't, try to amend the design or go back to step 2 and select another design hypothesis.

While humans have designed many wonderful things, there is one flaw in the human psyche that can derail the whole process – people become wedded to their hypothesis. There are numerous examples of scientists refusing to abandon their big idea. Doctors continued blood letting for a hundred years after the circulation of the blood had been discovered. Many miscarriages of justice are rooted in the authorities holding a hypothesis and refuse to budge even in face of an inconvenient fact. The analysis phase is therefore not just a nice-to-have but an essential part of the process.

But there is one major problem. How on earth can this be made to work on a large scale IT application where the number of requirements run into the thousands? In fact these four steps look horribly like a waterfall methodology, sugar coated by a bit of new terminology. Even worse, the major form of analysis we have today is testing, and to go back to brainstorm a new design hypothesis because a test failed, would be disastrous. The solution is hierarchical design.

Think again of design in other engineering disciplines. Design is organized into a number of levels. For instance, when Boeing designed the B777 their top-level requirement was to design a commercially successful airliner that seated about 300 people. The top-level requirement led to a top-level design that outlined the basic shape of the airline. The hypothesis was that the design was best with two engines and the wing attached to the bottom of the fuselage. The top-level design defined the airplane as consisting of a number of components, some of which were very big and complex. From the top-level design came the requirements for the second level design. For instance, the requirements for the engine – thrust, weight, noise, fuel consumption, etc – were set by the top level designers as they would have been very different if the design had called for four engines not two. The notion of levels of design is illustrated in figure 1.

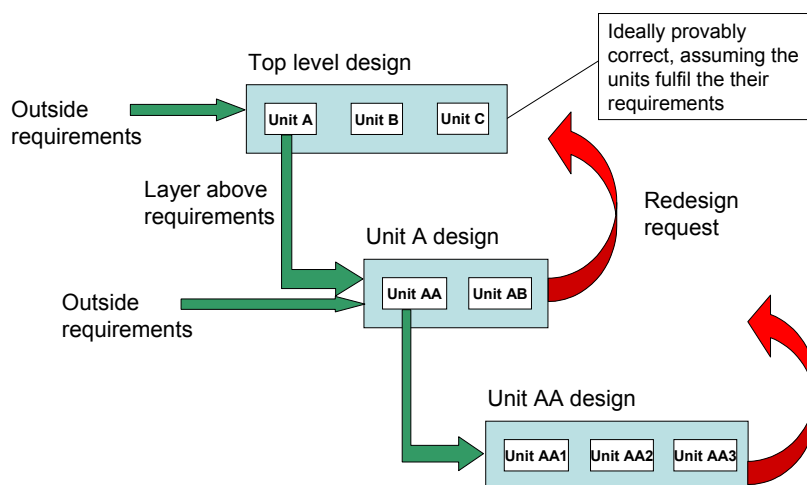


Figure 1. Levels of Design

The great thing about levels of design is that both the design and the requirements can be understood at various levels of detail. It is not a single hypothesis; it is a hierarchy of hypotheses. Not only that, analysis can be done at each level. This, in more concrete terms, is what I mean by an engineering discipline. For instance, it is possible to do a series of calculations on the top level aircraft design which prove that, so long as the components work to specification, the aircraft will fly. In fact they can do much more, they can say how it can fly. On the A380 they built the cockpit simulator before they built the first plane and the test pilots found the real plane behaved much like the simulator.

What would have happened if the engine manufacturers for the B777 had turned round to Boeing and said they couldn't build an engine that met their requirements? Boeing would have had to change their top-level requirements (e.g. reduced the total size) or searched for an alternative design (e.g. three or four engines). Clearly if this had not been found out until the testing phase, it would have been disastrous. The problem can be alleviated by analysis of the engine. The engine too had a top-level design and numerous component designs, and by working through the level design the engine manufacturers can reach a point at which they can make a good estimate of project feasibility.

It is clear that this notion of levels of design is enormously important to engineering design. But the levels of design in an aircraft are dictated by the structure of the thing you are building. The levels are obvious, and people don't go round talking about the wonders of levels of design. The levels in IT application design are not so obvious. This is one of the key questions I shall tackle in the following sections.

Before leaving this subject there is another point to consider – the nature of requirements. There are two kinds of requirements; explicit and implicit. Suppose an engineer is designing a bridge. The simple explicit requirement will include things like the position, the load to carry and the size of the space underneath. The implicit requirement is that the bridge does not fall down. The engineer may go to the client and tell them that the cost of building a bridge that survives Richter 7 earthquakes is X while building a bridge that survives Richter 8 earthquakes costs Y, hence leading to a discussion on trade offs and risk. What a professional engineer is not meant to do is (a) not know what level of earthquake would impact the bridge, (b) deliberately build a bridge to a low specification to save money or (c) point out at a late stage that the requirements said nothing about earthquake risk and demand extra money to fix it. The major point about implicit requirements is that the engineer should know more about them than the client and if nothing is said then the implicit requirements are included in the requirements to the degree that is necessary (e.g. level of earthquake protection is sufficient.)

In the next two sections I will look at examples to see how we can apply these ideas to IT application development. In the interest of space, I shall be concentrating on top-level design.

## Example One – Order Processing System

The example is a customer ordering something, the goods being delivered and later followed by an invoice.

The first point to note is that the system is not an IT system. It is a business process supported by IT; the ultimate test on whether the IT applications work is whether they support the business process. So if we are going test for correctness we have to start there.

The explicit requirements for the order processing system, from the customer's perspective are: what's ordered is delivered and what's delivered is billed. Sometimes there will be problems – deliveries will go missing, product will become unavailable, warehouses could burn down. So there has to be a system for amending the order. You may also allow order cancellation. So there are complexities, but not many.

The implicit requirements include:

- Good record keeping (they can tell you what happened to your order);
- Timeliness (the process does not take excessively long);
- The information at the time of ordering is current;
- Customer details are not to be given away to unauthorized 3<sup>rd</sup> parties.

If the customer orders directly via the web, you should add to this list, acceptable performance, ease of use and availability of the order service.

Observer that the implicit requirements are generic, by which I mean that if you drew up a list of implicit requirements for a different business process application the list would be more or less the same.

The top most design hypothesis I call a context model. Other people call it a conceptual model or sometimes the software architecture (though there is not a uniform agreement what is included in a software architecture.) I like the words “context model” because it says what it is.

The aim of a context model is to show how the activities, the data, the people and any other resources implement the process. Today we do context modelling using business process modelling (for instance using UML activity models or BPMN both standardized by OMG, see [www.omg.org](http://www.omg.org) ) supplemented by Use Cases to flesh out the detail of the tasks. Checking that the design meets the requirements is an informal, error prone activity, especially if the people who really understand the business are busy or reluctant to spend the time. Some aspects of the design are hardly ever reviewed properly like timeliness, record keeping, the need for information being up-to-date, the impact of errors on the data, and the ability of the processes to recover properly.

There are three problems with today's practices.

First we shouldn't have to rely on narrative. Engineers and architects don't. Yes, they might write some guff explaining their proposal, but the purpose of the narrative is to sell, not to specify. In IT we should specify using structured data from which we generate diagrams, possibly animated to show how processes flow through time.

Second, there should be much better analysis. Instead of relying on review, the designers should be able to identify issues and concerns. Even if the stakeholders don't do their review job properly, the designer should be able to ensure that the processes work and work well. The only consequence should be that the design doesn't quite do the job quite how the stakeholder envisaged. If you ask an engineer to design a car and you are rather vague about what sort of car you want, you might not get the car you want but you should get a car that works. It should be the same in IT.

Third, today's context models miss out a crucial element – data. While they sometimes show data flow between activities they rarely distinguish between shared persistent data (data stored in a database) and message data. Furthermore they are poor at modelling the dependencies between data flow, persistent data and control flow (control flow being how the steps in a business process are ordered.)

My solution is a new form of context modelling I call "box bag" modelling. A box bag diagram for order processing is illustrated in figure 2 (the supply and delivery processes have not been expanded into tasks).

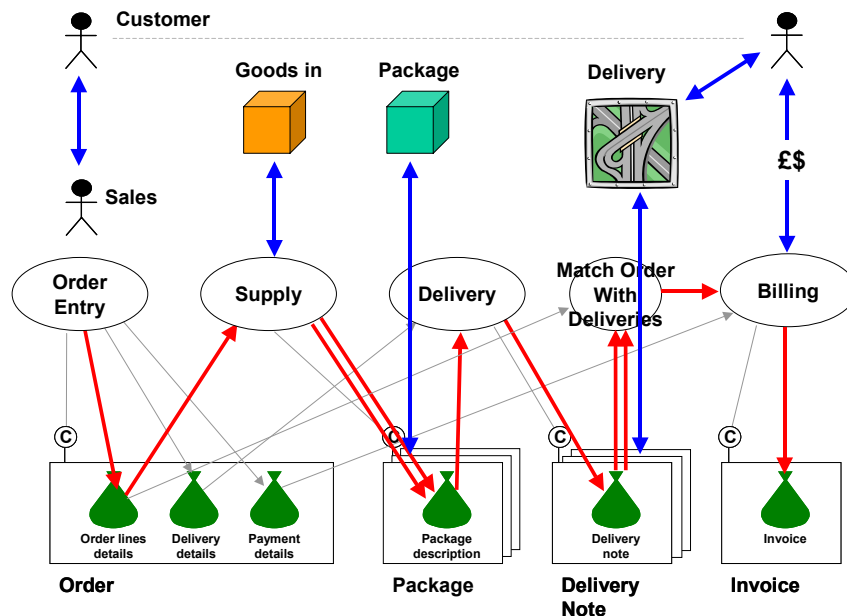


Figure 2. Box bag model of Order Processing System.

To explain it, imagine that behind your screens are demons (hot from a physicist's thought experiment) or house elves (if you are a Harry Potter fan), one demon for each task. The context model would explain how the demons cooperate to implement the application. To have the demons model persistent data, we give them the ability to

create information boxes and bags. An information box is simply a container for information bags with a label so other demons can find it later. Information bags hold information. Just remember – put the bags in the boxes – and you've got it.

The rectangles near the bottom are boxes. The bag-like things are bags. Thus we have:

- Order details – used by the create package activity.
- Financial details – used by the billing activity.
- Location details – used by the delivery activity.

There are various ways of drawing the diagram and I have chosen to show more or less everything. The grey lines and the red lines indicate which activities create or read the bags. The red lines are on the control path – they control the movement from one step in the process to another. The blue lines show dependencies between boxes and external entities.

Note that information bags do not document the structure of the data. Their purpose is to document how activities share data. The definition of the structure of the data that goes in the bag is a negotiation between the designer of the activity that provide the bag and the designers of the activities that use the bag.

One of the aims of using boxes and bags in context models is to have a good handle on error recovery. Even at the customer level, the process is simple until something goes wrong. This is typically the case all the way down – most of the application is there to detect and handle errors. So design of error handling at the topmost level has enormous ramifications on system. The context model allows you to define what the error recovery routine (e.g. to handle a lost delivery) must do to the boxes and bags. An observation – error recovery in a process is much easier if activities only create and read bags rather than update them, so you should try and eliminate bag updates where possible.

Recall the explicit customer requirements – an order is delivered and the delivered is billed. Using a box bag diagram these are met if:

- There is a process path from order entry to billing;
- For every order item there is a package (though a package can cover more than one order line);
- For every package there is a delivery (though a delivery can handle many packages to many customers);
- For every order there is one billing;
- The data is in sync with the outside world – i.e. the package and delivery data is accurate;
- If there is a failure, the relevant actions are repeated or the customer's order is adjusted.

The implicit customer requirements (timeliness, etc) are met if:

- Activities are either completely done or completely undone;
- Failures are always handled;

- Every time the process waits (e.g. waiting for supplies) there is a timeout so the wait does not last forever;
- The product information used by order entry and packaging is consistent;
- The packaging and billing activities don't process the same order twice;
- The applications are secure and, at least if the customers use them directly, are easy to use, perform acceptably and are available.

Most of these points are what I call “criteria for correctness”. They are assumptions you have to make about the box bag model that lead you to a correct implementation of the model.

The context model is just the top level in a hierarchy of designs. I have developed a framework for positioning the various design models. This is shown in figure 3.

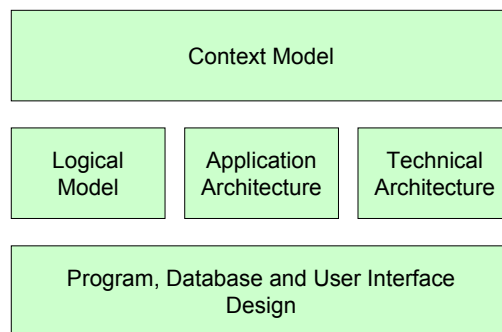


Figure 3. Framework for IT models.

The designs in this diagram are:

*Level 1: Context model.* Outlines the relationship between data, activities, users and external entities. It provides the framework for the whole design.

*Level 2: Logical Model.* Takes the context design elements and additional user requirements and elaborates them with rules providing a complete functional description of how the system works. This is where the logical data structure for the information bags is defined. The box and bag data structures are merged to create a logical database. Information bags can be thought of as a view on the logical database.

*Level 2: Application Architecture.* This maps the logical design elements to the implementation elements – the applications, databases and network connections, etc.

*Level 2: Technical Architecture.* The choice of hardware and software technology, the high-level network design, the high-level security design and the high-level system management design.

*Level 3: Program, database and user interface design.* This is the detailed design of the program components, database physical design and user interface.

In my demon analogy, the context model shows what demons there are and what actions they can take, the logical model provides the demons' precise instructions. The rest of the models show how the demons are implemented. The application architecture shows how the functions are divided between programmable elements, and the technical architecture explains the platforms on which all of the above is built.

Of course, just because it is described this way it does not imply you need to implement a project starting at the top left hand side and working downwards and rightwards. I would suggest the context model should be complete before embarking on anything else, because the context model enormously helps in specifying the size and scope of the development projects. But the tasks can be implemented one at a time – build the associated logical model, develop the code and test – before embarking on the logical model for the next task. This makes an incremental delivery but there is always a chance that designing the logical model uncovers flaws in the context model. One of the advantages of using boxes and bags in the context model is that you can identify groups of tasks that share a great deal of data versus sets that don't. Thus it helps you define a phased project delivery plan.

The top-level design has requirements from the outside world and as noted earlier there are both explicit and implicit requirements.

The lower level designs takes requirements from both the outside world and the design above. Again there are both explicit and implicit requirements. Each model or architecture has criteria for correctness which are rules that must be obeyed if the model is to be understood or implemented correctly. Criteria for correctness normally end up as implicit requirements on the next level down design down in the hierarchy of designs.

The lowest level design is program code. The code's criteria of correctness are the semantics of the language, the middleware and the database. For instance, the transaction semantics known as ACID (atomicity, consistency, isolation and durability) are four of the criteria for correctness the code depends on to work correctly.

It is not only the logical model that takes additional outside requirements. The detailed design takes requirements for the look and feel of the end user interface. The technical architecture takes the non-functional requirements – performance, scalability, availability, security requirements and so on. As an aside, rather than setting blanket targets it is better to look at the different applications and develop non-functional requirements for each. The context model helps enormously in assessing the business impact of not meeting a non-functional requirement.

## **Example Two – An Email Implementation**

I chose this example because it is well understood rather than because I expect anyone to implement a new email product.

The external top-level requirement is simply to send and receive emails and manage email stores. The implicit requirement is that the emails are sent in reasonable time, sent in their entirety, and are sent once and once only.

I used the same framework of models as in the previous example – context model, logical model, application architecture, technical architecture and detailed design. I also decided to reuse the diagrammatic conventions in the context model. Not surprisingly this wasn't straightforward as box bag diagrams were developed for business operational systems and I found I needed to add some additional elements, shown in figure 4, to the diagram otherwise it would be too cluttered.

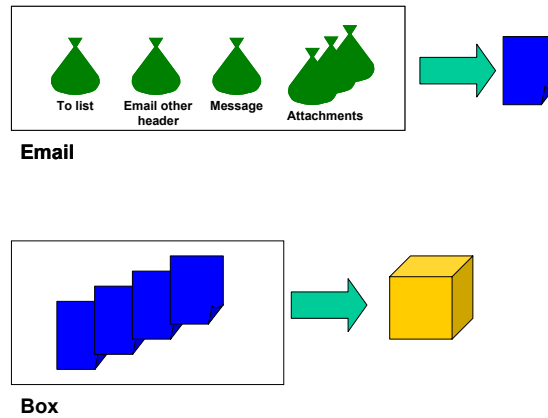


Figure 4. Extra drawing elements.

Using these new conventions I drew the top-level context diagram, figure 5.

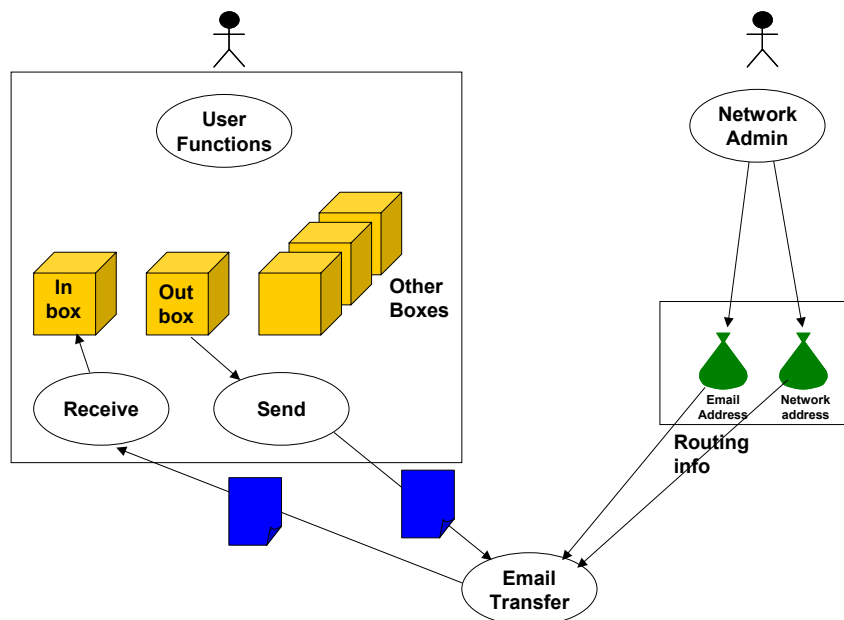


Figure 5. Overview context model.

This shows someone working on a device (as yet unspecified) sending and receiving messages. The Send and Receive functions are drawn beneath the email boxes to indicate that they are not user driven functions but happen automatically in the background. Notice that the specification has already changed. I have added the concept of email boxes that wasn't in the original requirement. Already there is a discussion point with the project stakeholders.

The next step is to elaborate on the activity called "User Functions". To do this I draw what I call a session diagram, see figure 6.

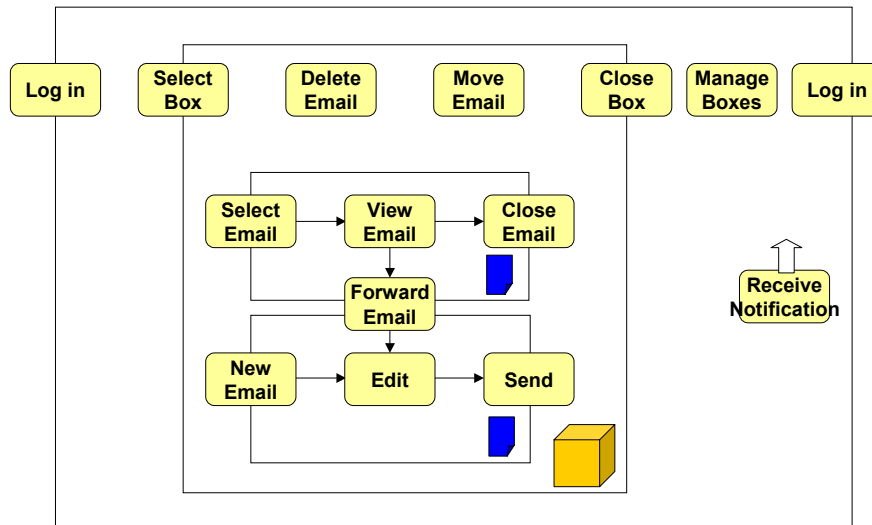


Figure 6. Session diagram.

The little rectangles are functions. The big background rectangles indicate session state. Thus the biggest background rectangle indicates the state of being logged on to the email system. The functions that overlap the edge of the rectangle – in this case the functions called “log on” and “log off” – are the functions used for entering and leaving the state. The next biggest rectangle records the state of having a current email box open from which the user selects emails. The two smaller rectangles are for grouping the functions that are available when an email is open. The functions within a state can, by default, be run in any order. If you want to indicate that the functions run in a predefined order then you draw arrows between the tasks. This is done within the processing emails rectangles. Note it shows that when you select an email and start viewing it, you can ask to forward the email and, in doing so, the state changes from a viewing state to an editing state. Finally, the arrow above the Receive Notification function indicates that it is initiated by the system not by the user.

The purpose of the session diagram is three fold.

First it helps clarify what functionality we need in the system. It is something to discuss with the project stakeholders. It also is beginning to clarify the ambitions and size of the project.

Secondly a session diagram is the starting point for the user interface design.

Thirdly, it can be used for analysis. To do the analysis you need first to document how every task uses the information boxes and bags. Some checks you can do are:

- Ensure that every box and bag is created, used and deleted by some activity somewhere.
- Analyse errors by considering the impact on the boxes and bags and hence what the user sees when they next sign on to the system. In our example this becomes particularly pertinent when looking at what happens when there is a failure around the time an email is sent.
- Create a first cut at the security requirements.

However note that we have said nothing about what end user device to use, how many and what databases (or file system) or even what applications need be written. For instance, it is perfectly possible to implement the above using a central store of emails and a lightweight client (e.g. a web browser) as the user interface. At the other extreme the whole thing could be put in a workstation. In short, this is a context model not an application architecture or a technical architecture. When we come to develop the application architecture and technical architecture we will be seeking additional detailed requirements to inform us about the end user devices and the configuration. Why ask such a basic question so late you might ask? Well it only took an afternoon of work to get to the position represented in figure 6. A few days work will complete the context model. You are now in a much better position to discuss the implications of hardware and network choices.

In both these examples I have concentrated on the first step of taking basic user requirements and developing a context model. I have also shown how detailed requirements are conditional on the context model design. I have not gone all the way through, pursuing the design all the way to code, but the ideas are the same. Every design at every level implements the design above it and takes additional requirements from the outside world. At every level there is analysis for completeness, consistency and checks on whether the design implements the requirements.

## **Towards a Theory of Hierarchical Design**

So it works in practice, but does it work in theory? (Readers of a sensitive disposition who dislike theoretical discussions might want to skip this section.)

To begin with, I have rather casually started calling designs “models”. What does this mean? I expect most readers are familiar with IT designs documented by diagrams and narrative text. To take an engineering approach, we are going to have to be more precise. The word model is used rather loosely in computing, but underneath all this sloppiness there is core definition that should (in my opinion) hold for all models anywhere: models are a simplified representation of a real or imaginary system and are used to help describe and/or analyse the system. With this definition, any design except the most detailed, should be a model.

While models of ships or buildings may be made of wood or modelling clay, models in computers are made of data. The data can be designed either to be just the most convenient way of storing a diagram on disk or to represent the model subject. I call these two alternatives *diagram centric* and *information centric* models. Information centric models are by far the most powerful because you can always generate diagrams from the data but it's much more difficult to generate the information from the diagram. Furthermore once you have mastered the art of diagram generation you can generate many different kinds of diagram at will – diagrams with different numbers of objects in them, diagrams focused on different elements, diagrams drawn from different angles, or diagrams including or excluding different kinds of objects or linkages. If you wish to automate the analysis of your model or animate them then an information centric model is the only approach that will do. Diagrams are fun, interesting and useful but, ultimately, not essential. It is the data objects, the attributes and the relationships that are core of the model. Thus the foundation of any work that will further a move towards engineering design of IT application is to specify the schema for the information-centric models for design. This is illustrated in figure 7.

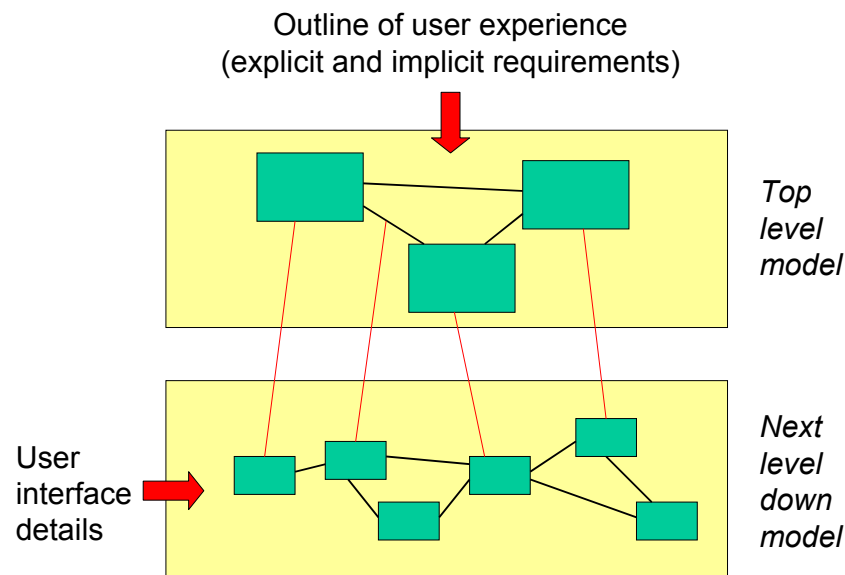


Figure 7. Relationship between models

The little boxes are the elements and the black lines are the relationships within the model. The red lines between the models are relationships that indicate the element(s) in the lower model implement the element or relationship in the upper model.

It is also possible that there are relationships between models at the same level. Thus in the framework shown in figure 3, elements in the logical model should relate to elements in the application architecture (i.e. the program or database that implements the logical requirement) and every element of the application architecture should relate to an element in the technical architecture (i.e. the technology used to implement the program).

The notion of modelling in computing as I have described it is fairly well known, albeit still not widely used. The notion of a hierarchy of models is not so accepted, or at least not so discussed.

For a hierarchy of models to work, there must be a way of asserting that, what is defined in a model is implemented in the next lower level model or models. In other words, as we move down the hierarchy of models:

- The lower level models are complete with respect to the upper level model;
- The lower level models are consistent with the upper level model.

At each level we have objects as in figure 7. Say we have objects at level  $n$  that belong to types  $t_1, t_2$  to  $t_m$ , and at level  $n + 1$ , we have objects that belong to types  $s_1, s_2$  to  $s_k$ . The notion of “implementation” described in the preceding paragraph is a transformation function that converts objects of type  $t_x$  (and maybe a combination of object, e.g.  $t_y$  and  $t_z$  as well) into objects of type  $s_r$  (and maybe  $s_p$ , etc as well.) I will call transformation functions, transforms.

The idea of transforms is interesting. In practice, today we have the notion of patterns and a pattern is really an idea of how something could be transformed and some guidelines on when to apply different patterns. With design model transforms, we can make this idea more concrete. Ideally there should be a library of predefined transforms and design would be a matter of selecting the transform. But note, it is possible to update the objects in the level  $n + 1$  model in a manner that invalidates the transform. Therefore one task of analysis is to check that the level  $n + 1$  model is consistent with applying transforms on the level  $n$  model.

Today design usually means developing the transforms from scratch. The best we can do is then to have analysis routines that check whether the result is internally consistent.

There are other reasons for analysis besides checking transforms, for instance:

- We need to check new requirements have been implemented correctly in the model;
- The transforms may not be independent.

A simple example of the last point is choosing technology which can be thought of as transform from a logical model to a physical model. You may want some technology for the network, some technology for middleware and some for security, but the technologies you select for each function does not work together.

Checking transforms is very much easier if the object in level  $n + 1$  have relationships back to the objects in level  $n$  that were the source of the transform (as illustrated in figure 7.)

The ideas as presented look very complex but in practice there are some simple cases. There is one very simple transform – an object is converted to another one that is identical except that it has additional data. An example is the objects representing activities in the context model. In the logical model, these objects are not changed in

any way except by adding to them, giving the information bags a data structure and the activities rules.

And also some objects that aren't involved in transforms at all. For instance, with the context model as described earlier at level 1, objects like the ones representing organizational groups have no level 2 equivalent. Some objects at level  $n + 1$  are new as well, by which I mean that they have no equivalent at level  $n$  because all their requirements come from external sources not from the level above. An example is the objects representing the outline network design. In the context model it is assumed that the network is sufficient to send messages. In the technical model, the outline network design must take into account the geographic spread of users and the message volumes. While the volumes are related to the activities, the geographic locations are new information.

Given the idea of transforms, what do we need to prove to show an application is correct, that is, implements all the explicit and implicit requirements. To do this we need to prove four points:

1. All the transforms work correctly, creating a level  $n + 1$  model from any correct level  $n$  model.
2. The transforms have been applied to all relevant objects in the level  $n$  model.
3. The transforms used do not clash.
4. The level  $n + 1$  model also implements all new level  $n + 1$  requirements correctly.

Note, we don't have to prove a level  $n + 1$  model implements all level  $n$  requirements because that is implicit assuming the transformations are correct.

The most difficult of these points to prove is the first one. I think that developing transforms will be a slow process, and we will make mistakes. But over time we will build a richer and richer suite of well understood, used and tested transforms. This is exactly what happens in other engineering disciplines; new techniques are being developed all the time. For each transform we will have to convince ourselves it can correctly implement the model above. For instance, I might have a transform that turns an activity from the context model into a three tiered application structure in the application architecture – presentation layer, a process layer and a data layer. But that isn't good enough. The criteria for correctness in the context model includes things like atomicity – if there is a failure, the application undoes all data updates associated with the activity. So as well as the three levels of application I need something in the application architecture that captures the correctness criteria of the context model; I can't just forget it. The simplest approach is to document the requirement by having an attribute in the application that indicates that atomicity is required. This must be picked up by the transforms that turns the level 2 application architecture into a level 3 program design. In other words, we are implementing a requirement from a level  $n$  model by passing it on as a requirement to the level  $n + 2$  model.

We are judging the effectiveness of the transforms against the criteria of correctness of the higher-level model. Not only will we over time develop more and more transforms we will also identify more and more criteria of correctness. This too is like existing engineering disciplines. For instance, understanding how tall buildings

behave in high winds came along later than understanding how buildings behave under load. As the criteria of correctness are refined so the transforms will need to be re-examined to ensure they still work.

On the final point above, to show that a level n model implements its requirements correctly means casting the requirements into a language that can be understood by analysis routines. This can be done by including in the models, objects that represent things that are external to the IT application, such as organization structure, user groups, goods in, and external actions (like driving a truck to go from A to B) that might be relevant to the statement of requirements. All the research that has been applied to business rules can be used to specify exact requirements and check their consistency. As an aside, the main differences between what I am discussing here and the traditional (if you can use that word) business rules approach is that I am calling for a hierarchy of rules and claiming that rules that define requirements for a level n + 1 model are dependent on rules that were used to define the requirements for the level n model.

So in summary we need a program to establish the following:

- A schema to define the types of all objects in all models;
- A set of correctness criteria for each model;
- A set of transforms;
- A set of analysis routines.

For each transform we need to define:

- The input object types;
- The output object types;
- The details of the transform;
- Reasons why the transform correctly implements all the correctness criteria in the model above;
- Any incompatibilities with other transforms at the same level.
- The associated analysis routines to check the transform has been done correctly.

The set of analysis routines is needed to:

- Check the transforms remain valid.
- Check the requirements for the level n model are implemented in the level n model.

If we can do this for all levels from context model to code, we will have good reason to believe the application is correct.

## **Status and Direction**

So where are we today (October 2006)?

I have developed a PC tool for context modelling, logical modelling, application architecture and technical architecture (see Polyphony at [www.itmodeller.co.uk](http://www.itmodeller.co.uk).) A first working version of the model schema is available.

No doubt some will be dismayed I have not taken UML as my starting point. The reason is simple – the context model is essential to this program and UML is weak in this area. They are working bottom up, I am working top down. I get the sense (perhaps erroneously) that the people working on the UML standard have the same goals as I do, but they are rather hoping an engineering approach to design is going to emerge out of their work standardizing the data structures that underpins each diagram. I think that's optimistic. For instance, UML still has pocket of diagram centric thinking. An example is a class called "Pool" to support activity diagrams; a pool makes sense in an activity diagram but makes no sense in the world being modelled. Finally, since the standard is organized around diagramming, even if there is a model trying to emerge, it's nearly invisible.

At the moment in my work on Polyphony automatic analysis has largely been checking for missing information in the context model and I feel I have barely scratched the surface of what's possible.

All analysis does not have to be automatic – perhaps cannot be. We need ways of helping people check the model. I'm looking at ways of animating the box bag model so people can visualize the process flow through the elements.

Comparing this work to other work in the same area you will notice that I have said little about program design (level 3 design). This is not because I don't think it is important. I'm heavily involved in writing a large modelling program (WeaverBird – see [www.weaverbird.org](http://www.weaverbird.org)) and feel the lack of good modelling tools in this area acutely (and have a long term ambition to fix it.)

However recall the points about transformation and patterns in the previous section. It could be that all that exists at level 3 are a set of patterns linked to applications and activities. These provide the information to transform level 2 designs all the way to source code. (It can't all be done as level 3 includes user interface design.)

This looks like the MDA programme from OMG and is in a sense similar (see [www.omg.org](http://www.omg.org).) OMG have this notion of PIM and PSM and I have yet to be convinced this works well. The idea is that you generate the PSM (product specific model) from a PIM (product independent model) and that give you an opportunity to change the attributes in the PSM, tuning it for instance, before generating the final system. What concerns me about this idea (besides the fact I always get PSMs and PIMs muddled up) is that I don't see an easy way to keep them in sync. When you regenerate the PSM from the PIM a new set of objects are created and you have to change the attributes in the PSM all over again. I plan to explore an alternative approach that is, for each transform, to set up one or more profiles that holds additional parameter settings. All relevant level 2 object should be associated with both a pattern and a profile, and all three – object, pattern and profile – shall be used to generate the application.

Before this vision materializes it is still worthwhile building level 1 and level 2 models for several reasons:

- The programmer will have a more precise, and hopefully, more understandable specification to work to.
- The design will have taken into account some of the trickier aspects of the system such as error handling.
- The application is more likely to meet the real needs of the organization.

In short, I've made a start, but it is a long road.