

The Three Level Framework

Chris Britton

October 2006

Web site: www.itmodeller.co.uk

This paper is about the diagram shown in figure 1.

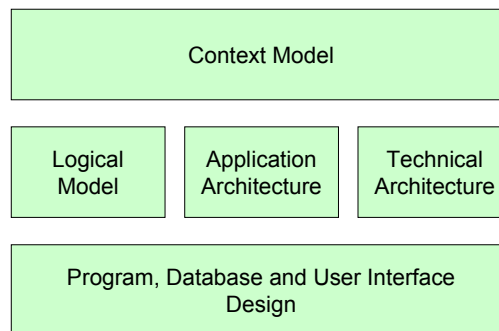


Figure 1. The Three Level Design Model

In fact it is mainly about the middle level of this diagram since the context model is discussed in a paper called “Context Modelling” (see the section on Further Reading at the end of this paper) and the Program, Database and User Interface Design level is discussed at length in many other books and papers.

Models, in general, are a simplified representation of a real or imaginary system. Models are used for describing and/or analysing systems. A computer model is a collection of structured data (like a database), describing a system in terms of its components and the relationships between components. Such models are used for drawing diagrams, for analysis and for simulation. Modelling is an ideal way of representing computer application designs.

The context model gives a high-level overview of the proposed system. It describes the system in terms of activities, information boxes and bags, and external elements like people, external actions and entities. Information boxes and bags are a metaphor for data handling. An information bag is used to model data sharing, information boxes are used to model persistence. An information bag is simply some data, the structure of which is undefined in the context model. An information box contains information bags. To store a bag you place it in an information box – remember place bag in box and you have got it. Other activities can then find the box and read the contents in the bag.

The structure of the data in an information bag is defined in the logical model. The logical model also describes all the input to and output from the activities and the

business rules implemented by the activities. The data structures for the information boxes and information bags are consolidated into logical databases.

The application architecture describes the application's tiers and the databases that support the activities. Thus the logical model and the application architecture together tell developers which rules are implemented by which applications and which databases.

The technical architecture describes the proposed hardware configuration, and identifies the software that will implement the applications. It also outlines the security strategy and systems management strategy.

These models and architectures together provide a way of describing an IT system in logical and consistent terms. It does not mean that when you develop the IT system you start at the top left and work your way down to the bottom right, though it is usually best to start off with a completed context model and you need a first cut application architecture and technical architecture before you can write any code. But the logical model and detailed design are usually best developed incrementally. As you build the system you will uncover issues that should be addressed in the context model, application architecture and technical architecture, hence these models must be continually under review.

The next four sections describe each of the boxes in this diagram in more detail except for the context model because that is described in the Context Modelling paper. These are followed by a section about handling existing applications, and a section about design processes.

The Logical Model

The logical model contains the following:

- Bag data attributes and verification rules.
- Task functional descriptions – input verification, decision making rules, processing rules.
- Logical data model for each database – data structure and data verification.
- Additional process monitoring and control functions – choice of facilities for monitoring business operational progress and assistance with manual intervention in the process (note: does not include trend analysis, data mining and ad-hoc analysis of the data. I would treat these as different kinds of applications.)

A task is the lowest level of activity – an activity you do not want to decompose into sub-activities. Normally it is one person doing one thing at one time, but there are occasions when a user wants to suspend a task and return to it later. A task is described in eight parts.

1. Input sources and data browsing requirements.

2. Output data structure – the data structure of any information bags or messages the task provides.
3. Output data verification rules.
4. Mapping of input to output – in other words, for each output attribute you need to define where the information comes from.
5. Event handling rules – how the process is initiated.
6. Find rules – how the information boxes are found. If a key is used where does the key data come from?
7. Decision rules – decide what output to create or what errors to raise.
8. Error handling rules – what recovery actions are taken after an error is reported.

On this last point, errors can be of three kinds:

1. Errors detected by the application. Like an order for a product that has become unavailable.
2. A system error. Like a power cut or software bug stopping the program in its tracks.
3. A time out error. Like an order that has been waiting too long for stock to become available.

For each case, we must work out how the error is detected and how the recovery process is initiated.

The criteria for correctness that need to be observed if the model is to work correctly are:

1. Tasks must be atomic. If there is a failure there must be no output. Remember a task can have a conversation with the end user so it is unlikely to be a good idea to rely on database facilities to handle atomicity and isolation for you.
2. Tasks must be isolated. Tasks must not get muddled with tasks working in parallel. This is normally implemented by having the task take ownership of a key box.
3. All tasks must be started once and once only. For every event that starts a task, only one task must be started.
4. All tasks must not be forgotten – that is half finished but not going anywhere.
5. All output data is created somewhere. For every output there must be a rule whose consequence is creating the output. For every data attribute in the output there must be a source – the data must come from somewhere.
6. All information boxes must be uniquely identifiable – that is they can be found.
7. All errors must have a destination task, person or device.
8. All information bag data must map to somewhere in the logical database.
9. The logical database must not have data in it that does not have a function in the tasks.

Some of these criteria of correctness (like 5 and 6) can be checked either manually or automatically. Others become requirements on the implementation. I anticipate that this list will lengthen in the future as we understand more about logical modelling.

A diagram convention used to illustrate the logical model for an activity is the Input Output Activity diagram. An example is shown in figure 2.

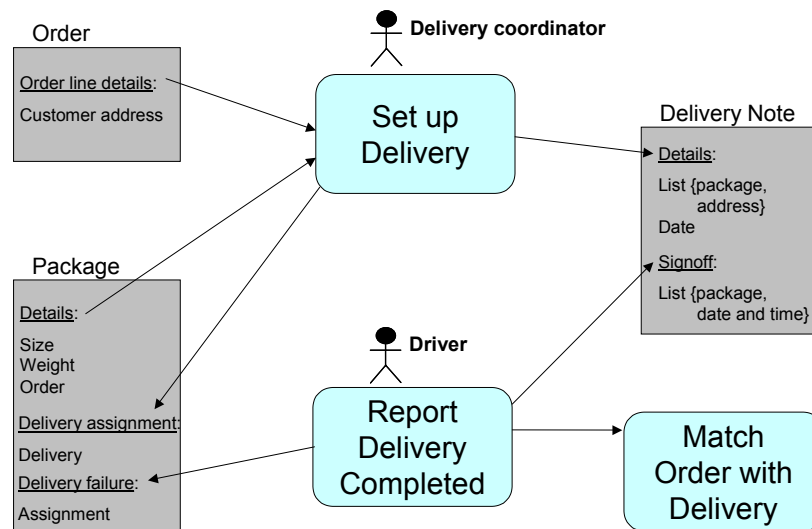


Figure 2. Logical Task Diagram

This shows three tasks but is only focused on the two running down the middle of diagram. One assigns packages for a delivery to one or more customers. The other reports on what happened to the delivery. The grey boxes are information boxes. The underlined text fields are the names of information bags and the lists underneath them are attributes of the bag. The Match Order with Delivery in the bottom right is another activity initiated by a message from the Report Delivery Completed activity.

A task covers much the same ground as a Use Case. It is more formal than a typical Use Case but says nothing about the user interface or the flow of the user dialog. This is deliberate. The reason is that the focus of the logical model is on the integrity and consistency of the system. It is quick to develop (so long as all relevant parties are involved), mainly because there is little consideration of the end user interface.

Task analysis can be used to present alternative solutions. In our example of a delivery activity, there are two ways of handling delivery failure. One is to add additional information recording the failure. This is illustrated in figure 2. But there could be a series of attempts to deliver the package which all fail. As shown this would mean multiple Delivery Assignment bags and Delivery Failure bags added to the Package box, creating a history of delivery attempts. The other approach is illustrated in figure 3. Here we create an additional task, an “Undo” task that would delete failed delivery assignment bags.

As an aside, Undo tasks should not be mixed with normal tasks and if necessary you should create an undo task with no end user dialogue as in this case (called an Auto Case in the context model). The reasons are: first, it is quite common for undo work to be called from more than one place, and, second, mixing undo rules with forward processing rules is very confusing.

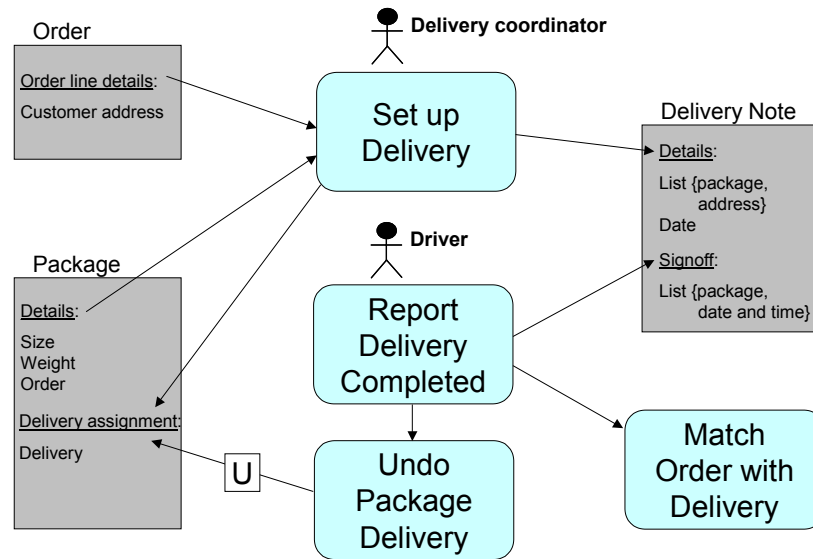


Figure 3. Alternative delivery task design.

During task analysis it is likely that, as you investigate a task that reads a bag, you realize you need information in the bag that is not provided by earlier tasks in the process, and therefore have to revisit the task design that provides the bag. This can be avoided if you start by designing the end tasks in a process first.

The second main activity in building a logical model is creating the logical databases. This can only start when you have developed the application architecture to the extent that you have decided what databases to have. How to make this decision is discussed in the next section. From the application architecture you will know which boxes are in the database and which activities use the database.

The starting point of the logical data model development is to merge all the bag attributes and create a first cut table design. You should look for duplicates during this process and for data attributes that are calculated values from other attributes. You can see an example of this in figure 2. The Delivery Note box has two bags – one for specifying the packages in the delivery, the other for recording the package has been delivered. It would be sensible to merge the fields into one package list. In database terms, an information bag is a view on a logical database, not the underlying data structure. But you should still examine the box bag model to check whether any duplication is indicative of an error in the model caused by the same information being provided twice.

The merged table is likely to be un-normalized, so normalization is the next step. I would recommend normalization even if you are implementing in an object database because when you identify an extra table you should ask yourself the question – is this an information box I missed? Normalization is a check on the design.

So far what we have discussed is entirely concerned about how activities work. In practice a large part of implementation is to do with monitoring the activities. This additional functionality needs to be added to the logical model.

Operational monitoring can take many forms, for instance:

- Looking for problems in the process like orders being forgotten.
- Looking for process optimization opportunities by monitoring error rates, process times, and external resource use.
- Looking for trends, for instance change in order patterns.

Deciding what monitoring needs to be done of course requires input from the user community. You need to divide the requirements into three kinds:

- Information pushed to users, like alerts or summaries.
- Online, pre-constructed queries on the production data.
- Queries on historical data.

Missing in this list is ad-hoc queries on the production data, but since they are ad-hoc it is difficult to design for them.

Queries on historical data should probably be implemented by data warehouses or data marts in which case they should be taken out of the production system design.

Application Architecture

Developing an application architecture is primarily an attack on two problems:

1. Developing a data distribution design. I think of this as splitting the application domain vertically.
2. Developing an application tier design. I think of this as splitting the application horizontally.

Data Distribution Design

There are four basic patterns. The four patterns are:

- Single centralized
- Multiple centralized
- Pass through
- Copy out/Copy in

These are illustrated in figure 4.

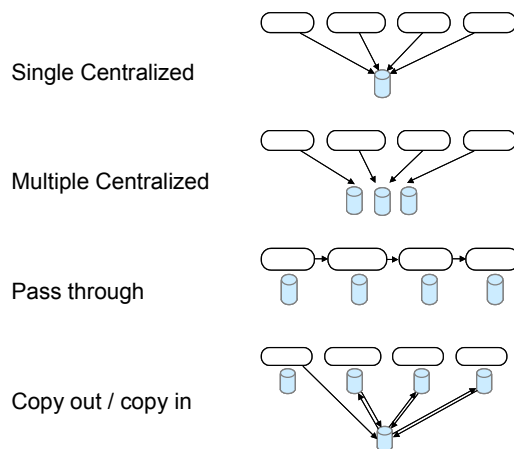


Figure 4. Patterns of data distribution design

These patterns can be combined.

Some of the factors to consider in choosing which pattern to follow are:

- *Cost and performance* – the optimal solution from a cost/performance perspective is usually to download a chunk of data to a PC, work on it locally and upload a chunk of data back. In other words, a few big messages rather than many small ones. But this may be awkward to implement.
- *Parallelism and scalability* – parallelism usually costs because it needs extra hardware resources but may give scalability benefits. Parallelism is hard when there are data sources that have no obvious ways of dividing them up (customer data is usually a typical example) but works well on data that is naturally associated with a location (e.g. warehouse data).
- *Local database design* – in an order processing example, supply might need a lot of data that is irrelevant to the other processes. The Pass Through and Copy out/Copy in patterns allows each database to have its own design.
- *Ability to mix technologies and use package applications* – it may be possible to buy an off the shelf application that does supplies or delivery. In which case it will almost certainly have its own database and the Pass Through or Copy out/Copy in pattern are the only options.
- *Sharing data* – the more data that is shared across the processes the more a centralized solution becomes desirable.
- *Business process status* – the disadvantage of the Pass Through pattern is that it is inherently more difficult to find out what happened to an order because you have many systems to interrogate.
- *Ease of doing reversals* – the Copy out/Copy in pattern may make doing reversals easier since the data can be reloaded.

Developing a data distribution design is a highly iterative process. A major factor is whether the application naturally splits into several self-contained parts or whether it is highly integrated. This is often evident from the context model simply by looking at it. Another major factor is the use of 3rd party and/or existing applications. These patterns are discussed more in our architecture book (see the section on Further Reading at the end of this paper).

Application Tier Design

The application tier architecture I (and I think most others at the moment) most favoured consists of three tiers:

- External interfaces tier.
- Optional middle tier for integration.
- Service tier for data handling.

The middle tier is optional because many applications don't have integration requirements or only have trivial integration requirements that can easily be put in the external interface tier.

The relationship between the tiers and the processes and boxes in the box bag diagram is unfortunately not straightforward. An application in the external interface tier might handle several processes. You might have a portal-like or workbench-like interface that provides a common look and feel to a disparate collection of functionality. On the other hand a single process may be useable from several different kinds of device (e.g. web and mobile phone). So the relationship between application tier entity and process is many to many.

Except for data validation, it is generally a good idea to drive as much business logic out of the external interface and into the middle or service tiers where it can be reused by different user interface. The external interface tier can be added to, changed, rewriting with a minimum of disruption to the underlying system.

The service tier components are more closely aligned to the processes than is the external interface tier. A good approach is for a service to manage one or more boxes and encapsulate as much of the data handling logic as possible so it can be more easily reused. This means services are largely transaction services not just simply a shell on the database.

What is left goes into the middle tier. The amount of code here is largely dependent on how much decision making is being done by the IT application rather than by the end user.

Application Architecture Relationships

The Application Architecture, as befits its location in the middle of the diagram in figure 1, is important because of its relationships with other elements in other models. It is the place where everything meets.

The application architecture should be related to the logical model so that you can find out which applications implement which activities and which bags belong to which databases.

The application architecture should be related to the technical architecture so you can find out what technology is used to implement each application.

The applications architecture should be related to the program, database and user interface designs so you can go into more detail into the implementation.

Technical Architecture

The technical architecture contains the following:

- The physical configuration – the locations, hardware resources and network.
- The software configuration – the operating systems and middleware.
- The development software configuration.
- A security strategy.
- A system management strategy.

While the application architecture allows you to say where every rule is implemented, the technical architecture allows you to say what technology implements it.

I will not discuss technical architecture in more detail because all the topics for discussion have already been covered in our architecture book (see Further Reading.)

Program, Database and User Interface Design

I will not describe program and database implementation in any detail because there is so much material elsewhere, but I want to make a few remarks about transitioning between the higher-level models and the nitty gritty of writing code.

An early part of the project should be to develop the end user interface. The reason why it should be early is that by showing the user interface to potential users you will get feed back on the logical model.

An important point is that is many tasks will use similar techniques. In the section on the logical model I list the criteria for correctness and note that many of these become requirements on the detailed design. I call such requirements *implicit requirements* because they are common across many tasks and it is kind of tedious to spell it out every time. Implicit requirements can be implemented the same way – or a few different ways – across many tasks. This gives rise to the notion of having a few patterns; each pattern implementing one or more of the implicit requirements. Thus detailed program design is about developing patterns and reusing the patterns to implement different tasks.

For example, in the delivery processing example (see figure 2 and 3) one criteria is that the deliveries for different orders don't get muddled up and that might be implemented by ensuring that only one person works on defining how the order is split into packages and deliveries. This might be enforced in the application by storing this person's id in the database table for orders. If the person handling the order goes to lunch leaving the task unfinished, no one else will accidentally start working on the same order. This is a general pattern for any situation where you want one person to be attached to one task – it is a generalizable solution, in other words a pattern.

For both patterns and for tasks you can draw a task/message diagrams. The purpose of a task/message diagram is to validate the technical and application architectures. A task/message diagram is a slightly extended form of a UML sequence diagram. The idea is to examine the message flow from user to application, application to application, application to database for a single task. Time goes down the page and message flows start from the left. An example is shown in figure 5.

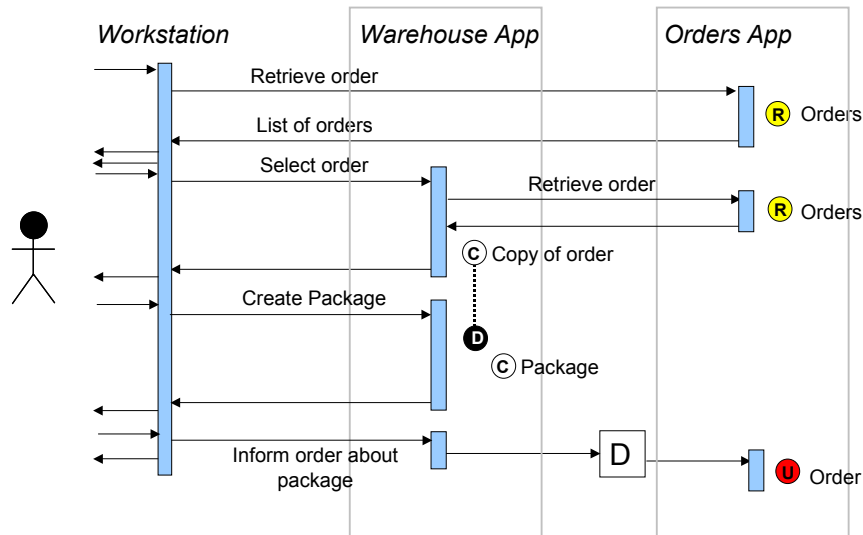


Figure 5. Task/Message diagram.

The strength of the task/message diagram is to analyse how a processes uses the IT infrastructure. Some to the requirements to the technical architecture will be what are often called non-functional requirements. These cover aspects such as performance, reliability and security goals. The task/message diagram is a technique for testing these requirements. We use a task/message diagram to do the following:

- Examine network performance problems can be found by calculating the numbers of messages and estimating network data volumes.
- Look for problems of data loss by studying the recovery procedures for errors at every point in the flow.
- Examine access control security by seeing where access control checks take place.
- Assess system monitoring by checking that faults and performance problems anywhere in the flow are reported.

In practice task/message diagrams do not have to be drawn for every task. Perhaps draw them for every pattern, and perhaps a few performance critical tasks.

At this point, after developing the end user interface design and the patterns for implementing the tasks, I personally don't believe you need to any more modelling. In spite of this being a paper about modelling, I don't believe in modelling to death. My experience in writing large programs (such as the modelling tool WeaverBird, see www.weaverbird.org which is all my own work) is that detailed program designs that

look good on paper don't survive long after coding starts. So long as there is clear relationship between objects in a high level model and components in the implementation, there is no need for design artefacts in between. Thus there is no point in modelling just above the code level.

Existing Applications and 3rd Party Applications

With existing applications and applications from 3rd party vendors you get an implicit logical model, an implicit application architecture and an implicit technical architecture. The question is the degree to which they match what you want. I shall discuss in this section how to approach design when you want to keep existing or 3rd party applications. The implication is that much of the existing infrastructural architecture is also retained, though possibly extended and updated.

I suggest that the starting point for design that incorporates existing applications is to develop a context model and an application architecture diagram that covers the functional areas of the existing applications. If you are not using a modelling tool at least draw the diagrams like the box bag diagrams.

You can use the context model in three ways.

First, it can help you judge whether the existing application supports the process well. Once the context model is complete it can be analysed as if it were a new application, as described in the paper on Context Modelling. This will help you identify holes in the existing applications that should be filled and weaknesses in the existing design.

Second, the context model can help you understand the requirement for integrating the existing application with new applications. For instance, you may find that the data needed by the new application is in certain information bags. You may find that the data is not collected until too late in the business process so a process redesign might be in order.

Third, the context model tells you which parts of the existing application are important to the new application and which parts can be ignored. If a box is only used by tasks that you do not want to change, any further analysis of those the data is unnecessary. Knowing what not to look at is vital if you are going to finish the project this century.

When considering whether to change the application there are several scenarios.

The context model and the application architecture are fine. You may still want to change the technical architecture. I will not discuss this scenario further, except to ask – are you sure there is a financial case to do this?

You don't like either the current context model or the application architecture. You may be able to salvage some code, or at least some tasks, but it looks like you want to do a complete rewrite of the applications. I suggest you still do a detailed analysis of the current application by building a logical model and go through it in detail with the end users. The reason is that, when a rewrite is done, a common experience is that

many of the features of the existing system are left out (because the end users setting the requirements assume they will all be done so they don't bother to specify them).

You like the box bag diagram but don't like the application architecture. The most common situation is that you wish to create a service interface to your existing application so you can use the functionality from other programs. This is described in detail in our architecture book so I won't discuss it here.

You may need to extend the context model with new functionality. There may be:

- An activity supported by an existing application that is initiated by a message from a new activity.
- An activity supported by an existing application that should be initiated by data in a new bag becoming available.
- There is a new activity consuming an existing bag that is provided by an existing application.
- A new activity that is initiated by a message sent from an existing application.

In each of these cases it is almost certainly possible to build integration hocks to or from the existing application to the new application. If you don't want to extend the existing databases and wish to have a new database then you are looking at a Pass through or Copy out/Copy in pattern for data distribution. In which case you need to develop an application for passing the data around. The need for integration may change the application architecture of the existing application. You may want, for instance, to make the existing application provide a service interface so you can integrate it through a middle tier.

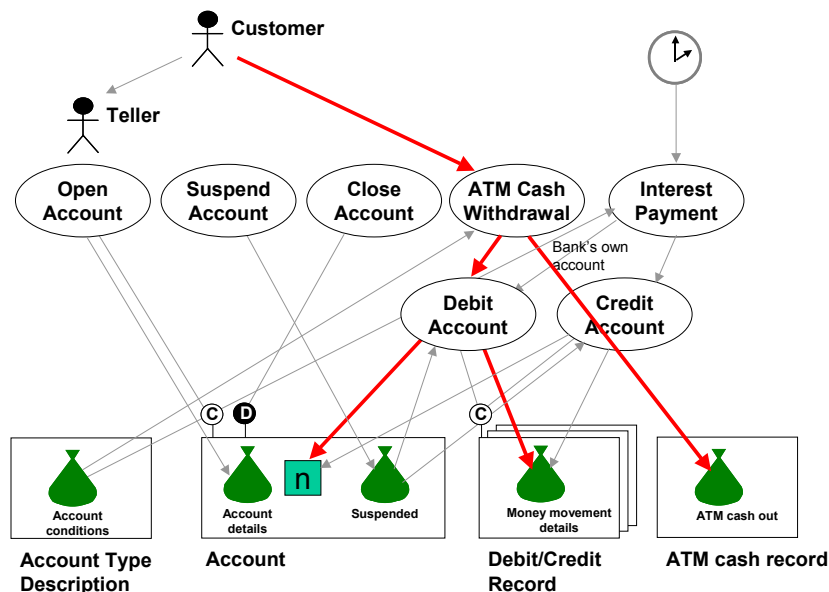


Figure 6. Box bag diagram for a bank

If you are unlucky, you will see in the context model a mix of existing and new activities using the same boxes and bags. Sometimes with a sleight of hand you can turn the existing activity into an internal task that has no external interface but is called by the new external activity. This is illustrated in figure 6. This example shows a bank system where there are numerous processes all doing debits and credits on the accounts. The debit and credit processes are internalized so that they provide common logic. This can be implemented by turning the existing debit and credit applications into service applications.

During new development you may want divide the task into a series of incremental deliveries. The problem of integrating a new increment with a released increment is similar as integrating a new application with an existing application. The main difference is that you can anticipate the problems of integration and design the application architecture accordingly.

Development Processes

I firmly believe that whatever development methodology you use, building a context model should be your starting point. Developing a context model lies somewhere in the hinterland between IT Planning and application development. It is not really part of an application development project because you need the information in a context model to stand a chance to scope the application development project or projects.

For most organizations the discussion from the previous section will be relevant because they will need to decide whether to replace, extend, integrate with or re-engineer existing applications. In a massive project, the context model will be a preliminary to splitting the project into a number of reasonably sized sub-projects. Hopefully you will be able to define sub-projects that only share a small subset of data. But there is usually some data that is particularly pervasive, for instance, shared customer and product information. You will then have to define an application architecture and a technical architecture for the shared data such that it can be made available to all sub-projects.

Even with an extreme programming methodology you will have to decide on a first cut application architecture and technical architecture before writing your first line of code. But even in the most documented heavy project, you don't have to have a complete application architecture or technical architecture before starting coding so long as you expect to make changes to the code later to comply with the final architecture. The time consuming part of developing a technical architecture lies in designing the network, developing the security architecture and developing the systems management architecture. I would not delay business functional development while these parts were being developed. Of course, in many instances this is not an issue since your new project simply complies with the existing application and technical architectures.

Defining good iterations for a project is an art. Ideally you want something that is time bound, short (agile project often work on two or four weekly iterations) but delivers something that is coherent that can be tested and reviewed as a logical unit. The ideal is not always achievable. The context model is a superb tool for helping you

decide on good subprojects. You should be guided by the notion that subprojects should own tasks and boxes, so allocate subproject tasks that have a high coupling through usage of the same boxes. An advantage of this approach is that, if you develop box by box, you are likely to be adding to a logical database rather than changing what is there. I suggest a subproject build the logical model for their assigned tasks and then call in an expert to help them on build the logical database. (I would also suggest that subprojects call on a specialist to help them design the end user interface.) Some subprojects will end up owning a box that is used by everybody so largely end up as a database development subproject.

How does this style of development differ from existing methodologies? Observe that I have not presented a methodology. I have said practically nothing about vital important subjects such as size of teams, functions within teams, how to communicate with stakeholders, project management and so on. In fact the only subjects I have addressed at any length is how to express a design and how to validate your design. Many methodologies these days are flexible and need little of no changes to take these ideas on board. For instance, some agile methodologies like SCRUM can be applied to any situation where the project can be split up into tasks small enough to be implemented by one or two people in a single iteration. It can easily be applied to development as described here.

The bigger question is whether there are practices what I propose that go against other approaches.

Some people might feel that the approach used in the logical model is contrary to the way they tackle Use Cases. There is some truth to this. In theory my notion of logical modelling of tasks is complementary to Use Cases since Use Case focus on the flow of work within a task rather than the details of the input and output from a task which is the logical model focus. As such a Use Case is bridge from the logical model to the user interface design. But if you have built a logical model the Use Cases practically write themselves and it is not much effort to go directly from logical model to user interface design without another design artefact in between. But if Use Cases is what project stakeholders want to see and what programmers expect, then they should be written.

People who are using agile development methodologies like extreme programming may distrust any form of design up front. I think they are mistaken, especially for large systems where the interaction between activities is complex. I also think the effort in building a context model, an application architecture and a first cut technical architecture is not that great, especially if they use a tool that generates the diagrams so they don't get lost in prettifying diagrams mode. I suspect that most good programmers develop a logical model. The only difference between what they do and what I propose, is that I would like to see it captured in a tool so it can be analysed and animated while they want to rush on and develop code.

However on one point, I have proposed something that will go against the grain of many designers – I haven't cast the artefacts in the jargon of object orientation. I could have done. I could have called information boxes, objects. I didn't because I believe it would have been misleading. First information boxes are persistent things and objects aren't necessary persistent. Second, it is the information bags not the

boxes that are unit of data sharing. This point would be lost if the boxes were called objects. Third, if they were objects what would be the methods? The simple answer would be to make the bag provider the method. But the data structure of the bag and its verification constraints are there to meet the demands of the bag users not the provider – the provider is just the slave to the demands of the user. Arguably then the binding between user tasks and bag is stronger than the binding between provider tasks and bag. Since activities, more times than not, consume multiple bags, arbitrarily picking one as “the owning object” is nuts. In any case I see a retreat from using object orientation in design. UML has class diagrams but used in practice they are used to define the data structure of persistent object not behaviour (look at the UML documentation itself – it hardly has any behaviour defined.) If there is no behaviour, there is no data hiding, and no encapsulation. This is sham object orientation, not the real thing. Object orientation concepts used in integration design is also on the decline, being replaced by XML and SOAP. The only place where object orientation is healthy is in programming languages. Even there, describing so many languages under the single umbrella term object-orientation obscures some fundamental differences in features and approach.

In the long term, I believe development will change. I see the detailed design being little more than the development of patterns, the association of patterns with tasks and the generation of code from the logical model being guided by the patterns. Programming will be largely a technical exercise, developing code that supports the patterns that can be tailored by the generator to create the final product. This is close to OMG’s vision of MDA – Model Driven Architecture. When model driven development is truly a reality, the tensions between agile and traditional development will largely disappear since development of the logical model runs along agile lines.

Further Reading

The other papers in this series can be found at www.itmodeller.co.uk/papers. The include a paper called “Making IT Application Design an Engineering Discipline” that describes the background thinking to the structure of models, and a paper called “Context Modelling” that describes the context model.

More information on many of the topics discussed in this paper can be found in our book “IT Architectures and Middleware: Strategies for Building Large, Scalable Systems” (Chris Britton and Peter Bye, Addison-Wesley, 2004). In particular this book looks at task/message diagrams, system integration design and modernising existing applications. Follow on reading is found on the web site for the book at www.unisys.com/books.

A modelling tool has been developed for all these models. It generates drawings of box bag diagrams and task message diagrams, as well as providing a certain amount of automatic analysis. The tool is available from the web site www.itmodeller.co.uk/models/.