

Context Modelling

Chris Britton

October 2006

Web site: www.itmodeller.co.uk

This paper describes what context modelling is, why it is important and my approach to context modelling.

What is Context Modelling

Context modelling is about showing how the IT application exists in the context of the organization. In other words, it expresses at a high level what the IT application does.

A major purpose of a context model is explanatory; to explain to everyone who needs to know, how the IT applications support the organization's operations. This includes business managers, IT staff and external consultants. The context model should express as clearly as possible all the major elements of the system and the dependencies between the elements. This needs diagrams.

Another major purpose of a context model is analysis. Analysis can be manual. For this to be effective you need diagrams, and, if possible, diagrams that let you easily see characteristics of the system that may be a cause of concern. Analysis can be automated. For this to be effective the model should be expressed as structured data (data with a schema) as this is much the easier for an automated tools to analyse than diagrams. Ideally, the structured data expression of the model should be linked to the diagrams so that if you change one, you change the other.

A third major purpose of a context model is to explore future possibilities. You can build a context model for a proposed new system. Perhaps you might want to build several context models each for alternative new systems. Any tool that supports context models, needs to support versioning and scenario building.

Context models are the link between IT strategizing and planning on the one hand and application development on the other. Context models should be built during every Enterprise Architecture study, every IT strategic analysis review and every IT planning exercise. Ideally of course, a context model should be built once and kept up-to-date so it can be used in all of the above. Context models of existing and proposed systems can then be handed over to application development where it forms the basis for building the project requirements and for defining phases of project deliverables. Rigorous analysis of the context model gives the application development team confidence that they are developing something that will work and there are no nasty surprises waiting in the wings.

Today context modelling is done largely by drawing business process diagrams, supplemented by narrative text. Often – especially in enterprise architecture studies – the existing or proposed application architecture is presented and you – the reader – is left to infer the business context from whatever you can glean from the text. There are three serious problems with today's approach:

- *Lack of precision.* It is all too easy to gloss over difficult issues, avoid difficult decisions and obscure inconsistencies and incompleteness.
- *Lack of analysis.* There is virtually no analysis done on high level designs in IT. It is not surprising that complex IT implementation projects find themselves uncovering tricky business problems late in the development cycle.
- *Missing the data view.* Information systems are called information systems for a reason – they are largely about storing and providing information. A high level description of information structures and how information is used by the activities is almost entirely missing.

This paper is about how to address these issues.

My Approach to Context Modelling

I have developed a new kind of diagram, called box bag diagrams, that allows us to represent the relationship between activities and information. This is described in detail in the next section.

Box bag modelling is used in conjunction with process modelling. A tool has been developed called Polyphony (see the web site for more details) that can generate both box bag diagrams and process diagrams from a common set of data. The beauty of this approach is that diagrams can be redrawn many different ways. For instance, browsing by diagrams is supported. You can select an element and generate a new diagram focused on that element.

This tool is an information centric tool and the structure of the model data is defined by a schema. This structured data approach has made it possible to build analysis routines to check for flaws in the model.

Context models are valuable used standalone, for instance if the rest of application development uses UML or extreme programming. The context model becomes input that helps define the scope of the development project and outlines the basic requirements.

But context models can be integrated into the development cycle. Context models are the topmost models in a framework of models as illustrated in figure 1.

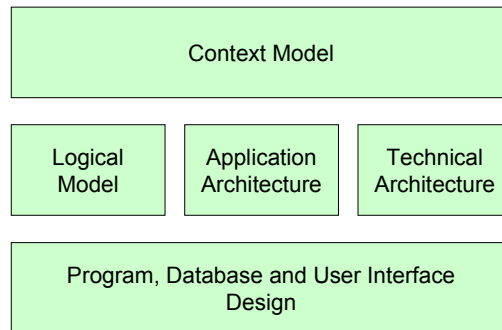


Figure 1. The three levels design model.

I have a long-term desire to make IT application development more like an engineering discipline and a rigorous context model is a necessary part of this programme. The vision is described in the paper “Making IT Application Design an Engineering Discipline” which can be found on the web site. A more detailed description of the other models can be found in the paper “The Three Level Framework” also found on the web site.

Even if application development goes its own course, for instance uses extreme programming, the context model together with the application architecture and technical architecture are usefully developed for architectural studies. This is particularly useful when strategizing or planning what to do about existing applications. The context model lets you quickly identify which areas where you want to change functionality. It helps you define the integration requirements between old and new. If you want to leave the functionality untouched, you may still want to change the underlying technology. The technical architecture and application architecture tell you which applications are affected.

The Model

When I first devise the box bag diagram it was originally designed to show how activities interact with data in the context of a business process. Since then I have found a context model needs more than that. I will describe the model in six parts – activities, information, contexts, processes, sessions and external elements.

Activities

The fundamental unit of activity we call a *task*. In most cases, a task is done by one person, doing one activity, at one time. Sometimes tasks are fully automated so there is no person. Occasionally a task will be suspended while the user does something else and returns to it later. This happens in many web system where half way through ordering your goods the system insists you create a new customer record. I would treat ordering goods and creating a customer record as two different tasks. A task usually corresponds to a Use Case in UML modelling.

Tasks are grouped into *functional areas*, and functional areas can be further grouped into larger scale functional areas, thus forming a hierarchy.

Tasks have inputs and outputs. These include reading and updating the information boxes and bags that are described in the next subsection. We also record tasks sending messages to each other. The messages can be one way or two way, in other words, request and response. One way messages I call *activity messages*, and two way messages – *calls*. The messages can be deferrable or real time. By default, activity messages are deferrable and calls are in real time. A deferrable message does not necessarily mean there is a delay. It just means that it is more important that the message arrives at its destination sometime in the future than it arrives in the next micro second. For instance, if the destination application is unavailable a deferrable message should be sent when the application is next ready. Real time messages are the opposite – they must be sent now (usually because someone or something is waiting for the response) and, if there is a delay, the transmission is cancelled.

Of great interest to context modellers is – how are tasks started? I recognise 5 types of task.

- *UseCase* tasks – these are tasks done and started by a person.
- *AutoCase* tasks – these are tasks initiated by an electronic message such as a web service call (there may be an ultimate user but it is invisible to the system.)
- *TimeTriggeredCase* tasks – these are automatic tasks triggered by some time condition, for instance – the next night, the next weekend. These tasks are typically implemented by batch runs.
- *InternalCase* tasks – these are tasks called by other tasks. For instance, you can debit your bank many ways but ultimately the debit transaction is done one way. InternalCase tasks allows you to capture the fact of common logic, for instance, allows you to define the rules about debiting an account in one place.
- *ExternalActivities* – these are tasks done outside the system by other organizations or people or tasks that are entirely manual.

This is not the same as what drives process control flow which is discussed below.

In the logical model, the task description (except ExternalActivity of course) are expanded to show the details of input and output data structures and to list the task's business rules. The idea is to pin down where all the information comes from and where it goes. Knowing such detailed information is itself an excellent check on the context model.

Information

To show how activities use data, I have devised the box bag diagram. The words boxes and bags are metaphors. Tasks can create information boxes and, when they do, the only information they need to provide is a label so the box can be found (uniquely) by other tasks. Information boxes are used to model the fact that some data is persistent. Information boxes can contain information bags. The purpose of information bags is to model information sharing. Tasks can place bags into boxes. Tasks can find an existing box and read an information bag contained in the box. The nature and structure of the information in the bag is unspecified in the context model,

because the purpose is to record how activities share data. The data structure in the bag can be thought of as a negotiation between the designers of tasks that provide the bag and the designers of the tasks that use the bag. The purpose of the context model is to identify what tasks use what bags.

A simple example of a box bag diagram showing one box, two bags and three activities is shown in figure 2. The diagram shows an expense process. The box is a rectangle. The bags are the bag-like things in the box. The ellipses are activities and the arrows to the bags show provide bag and read bag relationships. The circle with a “C” in it and with a line to the Create Expense Form means that this activity creates the box. There is also a delete box operation (shown in figure 9) but these are only usually used in error recovery activities since in normal processing there is usually a requirement to keep historical records of important business events. Some lines are red and some are grey. The red lines show process flow and is discussed the below. The grey line to the manager is an activity message but in this case it is going directly to the task user. It could be implemented by an email message. The clock like sign above “Pay expenses” shows this is a TimeTriggeredCase task.

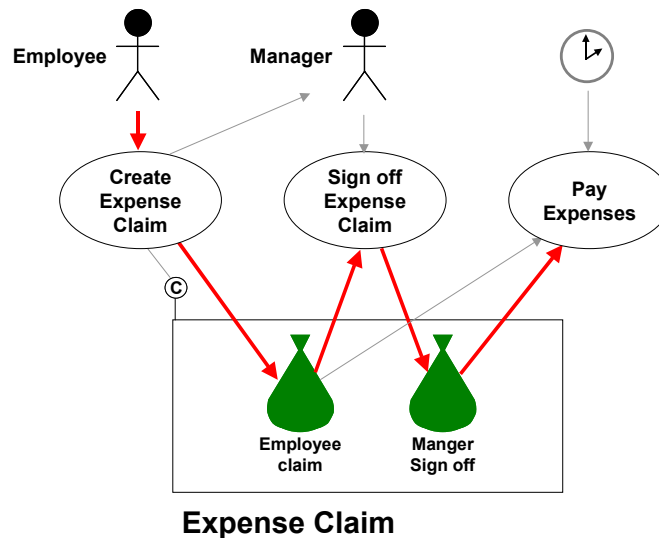


Figure 2. Expenses application.

Note that this diagram would be identical if the expense claim was a single figure with no break down or whether the expense claim split out all different types of bills and taxes. In other words the context model shows the structure of the application while avoiding the detail.

Information boxes and bags are normally implemented in database systems. How boxes relate to tables is unspecified in the context model (it is specified in the logical model.) How boxes are implemented in databases is unspecified in the context model. It is specified in the application architecture but note, a box could be spread across several databases.

There are four principles that drive the design of boxes and bags.

1. If a task reads a bag then **all** of the information in the bag is relevant to the task. If you find a case where a bag is used by applications A and B and some data in the bag is only significant for A and some for B, then you should split the bag into three bags – one bag for both A and B, another for A only and a third for B only. This keeps data flow down to a minimum.
2. If several bags in the same box are used in an identical way by all task, then the bags should be merged. This keeps the context model as simple as possible.
3. It is easier to analyse a system if only one task provides a bag (i.e. creates the bag) because there are rules associated with providing a bag and life is much easier if you define the rules in one place. If it looks like this rule is to be broken, it can usually be enforced by making an InternalCase task that provides the bag, and have the other task call the new InternalCase task. In practice the only times I know when this rule cannot be enforced is when modelling an existing system, and the fact it is broken highlights a problem of maintaining consistency in the existing system. As an aside, this rule pushes you to define a SOA (service oriented architecture) as InternalCase tasks are natural services.
4. One of the reasons for developing this model is to analyse the impact of failure and how to do error recovery. It is much easier to define undo actions if bags are provided and used but never updated. There are cases where this rule cannot be enforced but it is a good target to aim for.

Note, a bag can be empty. This is used to represent that something has been done – the bag is a state field. Empty bags can be read – this is useful to show that a task cannot start unless the previous task has been done.

Boxes may contain counters as well as bags. They are drawn by a small box with an “n” in it. An example is shown in figure 14. For instance, the amount in bank accounts can be modelled by a counter. Note, that the undo action of a task that changes a counter field is really easy – subtract where you added before, add where you subtracted.

Bags can hold relationships to boxes and you can have the relationships draw directly on the box bag diagram. The real meaning of a relationship is that it indicates a dependency – should the box be deleted some action must be taken as the bag is probably invalid. Relationships are directional (unlike in a database system) meaning that if bag A points at box B then action must be taken if box B is updated but no action need to be taken if bag A is updated. For instance, if an order depends on a product, then the product is unaffected if the order is cancelled, but the order may be highly affected if the product is updated or deleted. Relationships are about actions not about static data structures; we are interested in what can happen, not developing a data model that is best for queries – that is the job of the logical model.

Contexts

Tasks often work together. This is modelled by introducing the notion of context; a task can belong in several contexts. At this moment in time I have identified three contexts:

1. *Processes*. A series of task that start with an event and end with one or more outcomes.
2. *Sessions*. A group of tasks that can be performed by one user.
3. *Standalones*. The task does something meaningful just by itself. Currently identified kinds of standalone tasks are:
 - o Data service – for instance, updating a bank account.
 - o Track External – for instance, updating information about a customer like change of address.
 - o Report – an task that does nothing but provide information.
 - o Other – a cop out because I’ve probably forgotten something.

Processes

I define processes in a more restricted way than many as I exclude single task processes and circular processes – all my processes have one or more starts and one or more ends. For instance, in a banking example, I would define a process for opening an account and one or more processes for multi-step operations like clearing a cheque, but I would not try to encapsulate the whole overarching provision of an account service in a process. The reason for this decision is analysis; with a not-too-open notion of process you can analyse it for errors to ensure the process rules are adhered too. If there aren’t any rules you cant check them. If you need to group things together, use functional areas.

Processes are usually represent in activity or process flow diagrams (for instance using the activity diagram standard from UML or BPMN). An example is shown in figure 3. I am going to assume such diagrams like this are familiar to my readers.

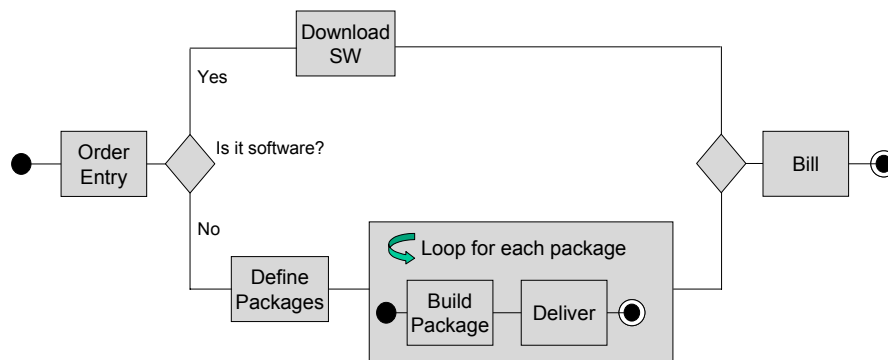


Figure 3. Process for Order Processing.

Process flow diagrams are used in several areas like process optimization studies and modelling workflow as well as within context modelling. There are many good things about process flow diagrams but for a context model you need to be careful that you understand the mapping between process flow on the one hand and tasks on the other. In other words, you need to understand how process flow diagrams relate to box bag diagrams. Two points need to be made.

First, every activity and decision in the process flow diagram is associated with a task. Note, the context model does not define a workflow router because such routers are an implementation decision, not a functional decision. We define how the process flow works and we define how outputs from one task become inputs to another task. We don't define how the messages are routed.

Second, a task may be associated with more than one node in the process flow diagram. In the example above, "order entry" and "Is it software?" correspond to one task and maybe "download SW" does as well. One of the problems with using process flow diagrams for high-level design is that it can be hard to know when to stop. With box bag diagrams it is much clearer – it is at the task level; one person, doing one thing at one time.

So if the nodes are implemented by tasks how are the lines between the nodes implemented? Clearly if the end nodes of a line are part of the same task, this is not an issue, but in all other cases there is a potential delay between the completion of one activity and the start of the next. The line can be implemented by:

- *A message.* If the next task is an AutoCase task then the message could be activity message sent by some middleware such as message queuing. If the next task is a UseCase then the message should be to a user. This can be in the form of an email or as a prompt on the screen. It is important that the context model designer thinks about what should happen if the user ignores the message.
- *Data becoming available.* In the order processing example, the delivery staff probably scan the list of completed packages and decide which ones to put on the next delivery van. We model this in the context model by showing that the line in the process is implemented by reading an information bag.
- *A resource becoming available.* The resource could be a person, a machine, money, or a room – pretty well anything. We model this by indicating that the process flow is implemented by an external entity.
- *The actions of an external party.* For instance, in an aircraft boarding process, the process flow step from check-in desk task to departure task is implemented by passengers walking from one desk to another.

The process flow diagram cannot show any of this, but the box bag diagram can.

The context model analysis checks that the process flow view and the box bag view are consistent. They might also consider the opportunities for errors and problems like messages being ignored, resources being lost and people not turning up at the departure gate.

The other area for analysis is the consideration of the effect of errors on the data – what boxes need to be deleted, what bags need to be removed and what counters updates need to be reversed? Errors need to be handled and the error recovery routines need to reverse the data updates correctly.

Sessions

The session context lets you consider the system from the viewpoint of a single user.

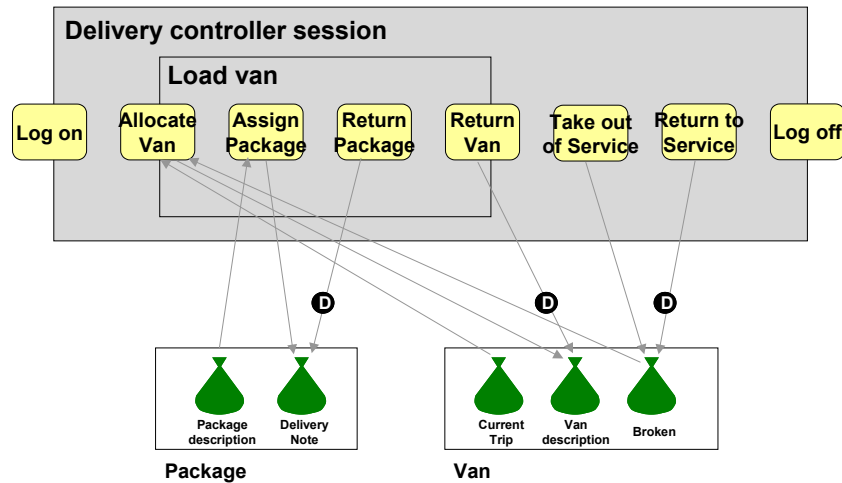


Figure 4. Session context for delivery controller.

Figure 4 shows an example for a manager controlling package deliveries. At the top of the diagram are the activities and at the bottom are the boxes and bags used. The bigger rectangles indicate state – the biggest of the rectangles boxes is the session state, the smaller one indicates that a van is selected and the assign package task is associated with the van. The tasks on the edges of the rectangles are for going into the state and leaving the state. The boxes in between can be done in any order. If you wish to indicate a specific order you connect them by lines.

Session diagrams provide a way of doing functionality checks with management and operational staff. Present them with a session diagram and they will quickly identify many of the tasks that you the designer had not considered.

Session diagrams are particularly useful when looking at the allocation of resources. Normally, resources can be allocated for an activity – like the vans are allocated to make deliveries in figure 4. If a resource is allocated it must be released when the job is complete. New resources are acquired and old resources are removed or are temporarily taken out of service. Resource management is a pervasive activity common to many situations.

Session diagrams should also be reviewed for two technical issues. One is security – the session diagram is the starting point for identifying the security requirements of the system. The other is assessing the impact of application failure on the users. If the system fails is there a backup, say pen and paper? If so, when the system comes back, how are the boxes and bags synchronized with the state of the real world as captured on paper?

External Elements

The final group of elements in the context model are external entities. These fall into two categories:

- *External parties*. These are people or organizations that are external to the system, for instance customers, suppliers and partners.
- *External resources*. These are resources that are external to the IT application (but normally internal to the organization.) They can include people, machines, goods, space (e.g. rooms or buildings) and money.

The main reasons for including these in the model are first to be able to find out the impact of the system on the outside world and second, to model process flow that depends on the actions of an external entity.

Order Processing Example

This example is about using a box bag diagram to analyse a process. The process is well known one; submitting an order, supplying the goods for the order, delivering it and billing the customer. It is illustrated in figure 5.

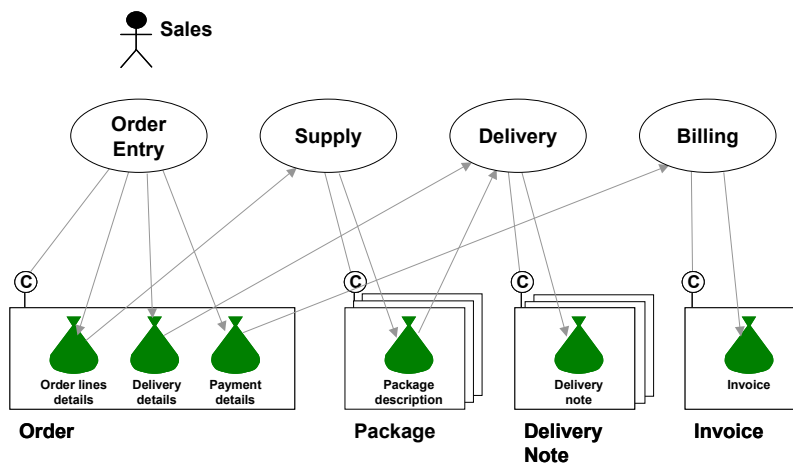


Figure 5. Order processing example.

The four main activities behave as follows:

Order entry – creates a box to represent the order and puts in the box three bags:

- Order details – the order lines and quantities.
- Delivery details – delivery address and conditions.
- Payment details – invoice address and method of payment.

Note we have defined three bags not one in compliance with our rule that all data in a bag must be relevant to all consumers of the bag.

Supply – builds, procures or takes out of stock the order items and packages them for delivery. It reads the order details bag. It produces one or many package boxes.

Delivery – assembles the packages delivery runs and takes them to the customer. An order made up of more than one package might end up on more than one delivery run. It reads package boxes and the delivery bag in the order box. It produces a delivery note box.

Billing – sends an invoice for the order.

The Supply and Delivery activities are complex and, in most real life instances, could be described by box bag diagrams of their own.

With this basic picture we can start extending and analysing the system. The first step is to examine how the data flow implies an ordering to the activity. The main flow that controls the order is shown in Figure 6

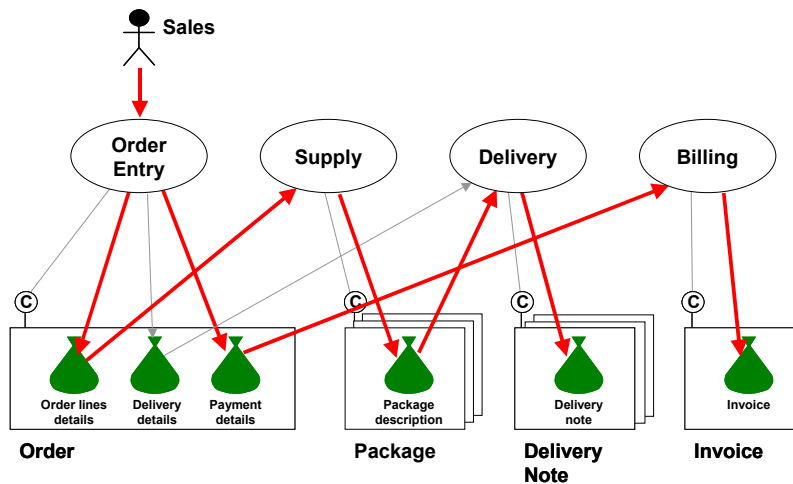


Figure 6 Activity Flow.

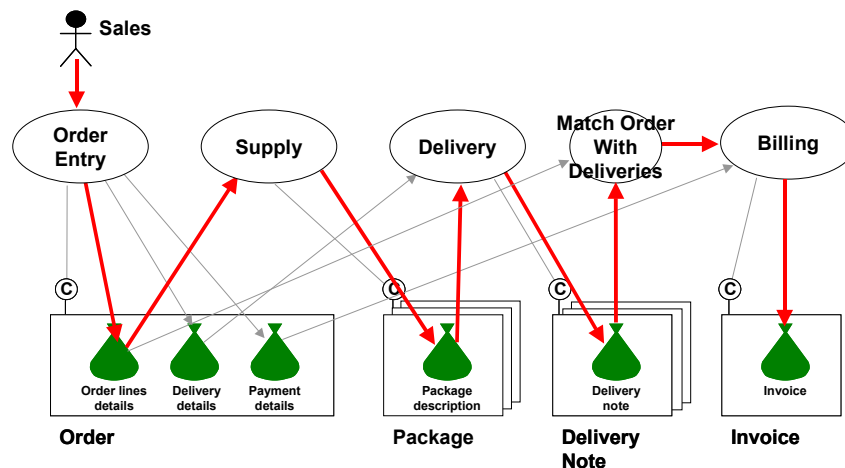


Figure 7 Matching order with deliveries.

Immediately one point becomes obvious – the billing activity is left dangling and there needs to be a way to ensure billing is done after delivery. We cant put a simple line from a delivery activity to the billing since a single order may give rise to multiple deliveries. So we come up with a solution illustrated in figure 7.

The next step is to analyse external dependencies. The purpose of this step is to understand error recovery. The external dependencies in the order process example are illustrated in figure 8.

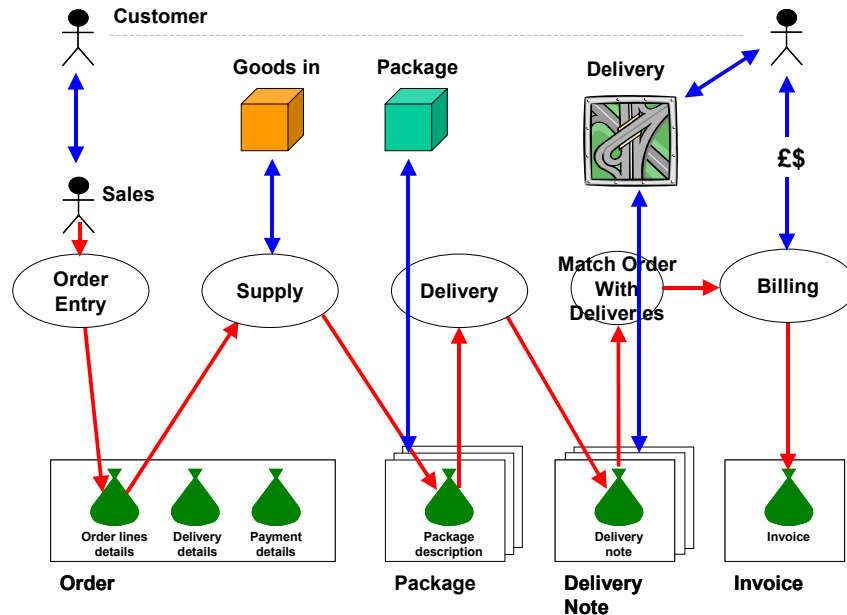


Figure 8 External dependencies.

The dependency between sales and customer just indicates that the sales person has recorded the customer’s desires correctly, for instance, the sales person has received a letter of intent. If something changes – the customer disappears or they change their mind – the order will either have to be amended or cancelled. This is illustrated in figure 9.

There needs to be an additional activity for doing this. At the very least, there needs to be the ability to cancel an order at any stage. This does not mean deleting all the boxes and bags from the system because these may be needed for retrieving the goods or unpack the packages.

The Return to Supply activity figures out what has been sent to the customer and hence what needs to return. Note that there is no requirement for the Return to Supply – or indeed any activity – to be automatic. It can be manual or semi manual. What is absolutely required is that the data is updated in a consistent manner and that the data at the end of the activity accurately reflects the external dependencies.

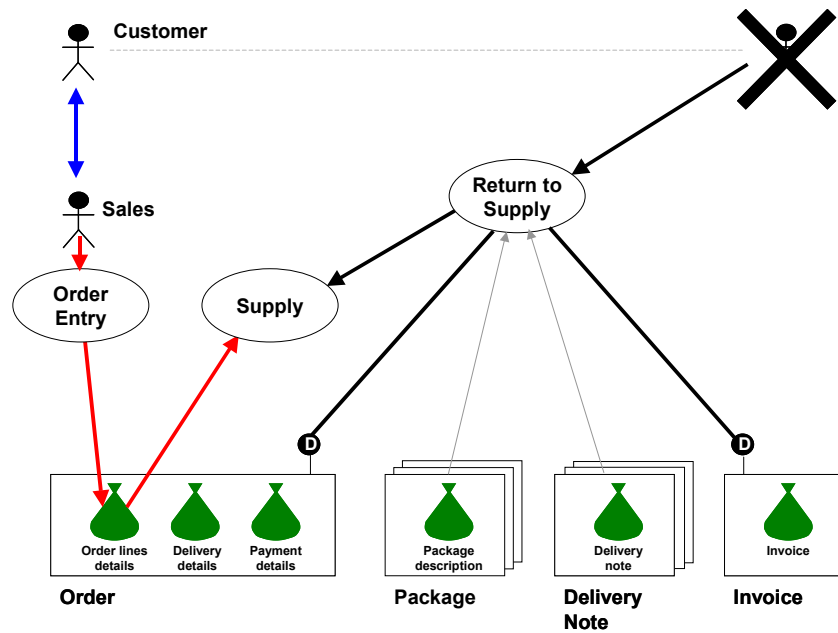


Figure 9 Customer rejection.

The dependency between Delivery Note and Delivery may break if the delivery fails – for instance there is a traffic accident. There is then a requirement for a re-delivery of the same packages. This is illustrated in figure 10.

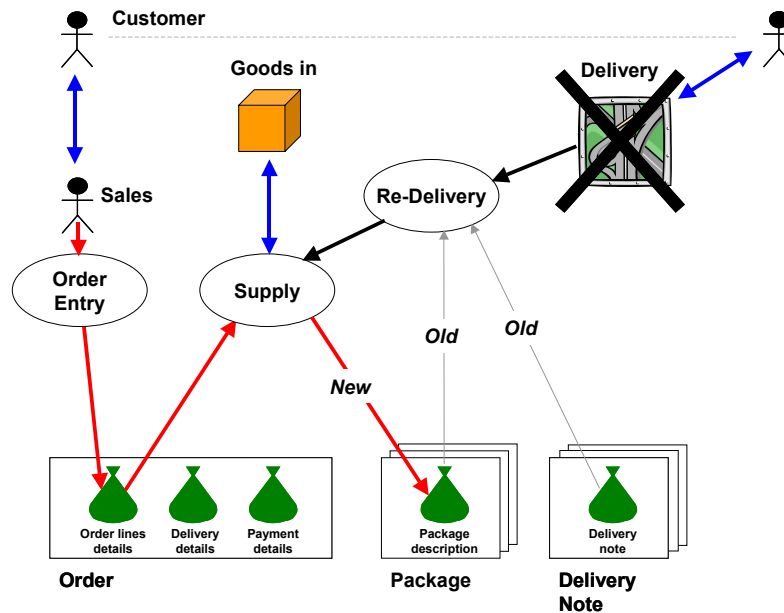


Figure 10. Delivery failure.

The basic idea of this re-delivery activity is to reorder the missing goods and create new package and delivery notes. This means reading the old package and delivery boxes and having the supply activity create new packages. Note there is a timing

subtlety. The Delivery activity must not trigger the billing activity until it is sure that the Delivery has reached its destination. Otherwise the customer would be billed twice – once for the delivery that failed and again for the re-delivery. This is a general point about external dependencies. You must be sure that there is proper synchronization between the internal data and the external thing or action. Also note that the re-delivery activity did not delete the old packages and delivery records since these are still important records of business activity.

The dependency between external packages and the box called package simply indicates that the package box accurately reflects the contents of the package. Two things can go wrong – the package can be completely lost or destroyed or the in can be incorrectly packed. The great difference between these two cases is that they are detected at different times. A package lose is detected by the delivery activity. A package being incorrectly packed – or packed with faulty goods – is detected by the customer. The two cases give rise to two new activities. These are illustrated in figure 11.

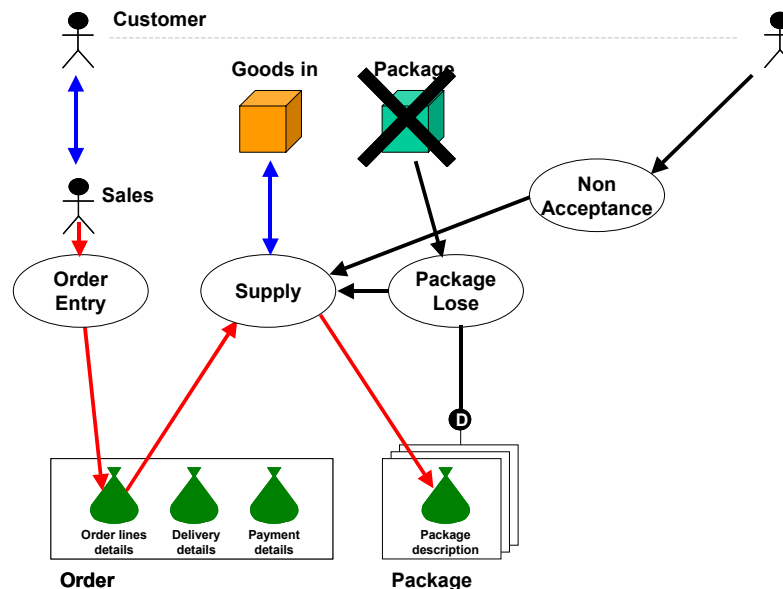


Figure 11. Package failure.

The dependency between supply and goods-in is important since if the goods-in becomes unavailable all or part of an order may not be able to be fulfilled. Three actions are possible – go on regardless, renegotiate with the customer, or cancel the order. Going on regardless or renegotiating with the customer means fixing up the order and reprocessing. We don't need a new activity for this. Instead we can ask for a rework of the order entry activity.

The diagram for a failure of goods-in is shown in figure 12.

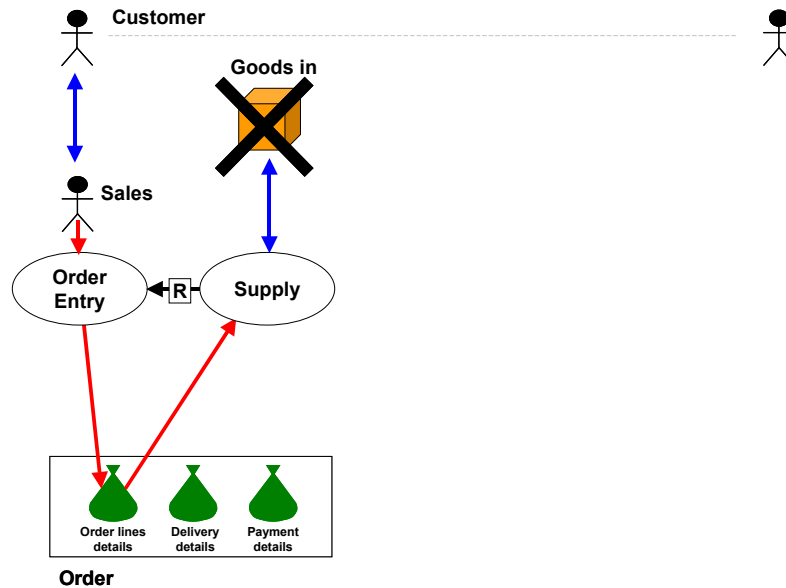


Figure 12. Goods-in failure.

In general the process of analysing errors, is about going through all the external entities and asking three questions:

- What if the whole entity is destroyed/lost/doesn't get created?
- What if the entity is incorrect vis-à-vis the box and bags that are meant to be tracking its state?
- At what point in the overall activity does the error become apparent?

The easiest form of recovery is to undo all the activities until a point is reached where everything up to that point was correctly done. A more complex form of recovery is to try to reuse as much as possible. This can be enormously complex but of course you can always resort to manual fix ups. Even so it is good design to give the manual fixer as much guidance as possible, which means making clear what boxes and bags have been invalidated.

Unfortunately external dependencies are not the only concern for error recovery. There may be internal dependencies. The internal dependencies in the order processing are illustrated in figure 13. There are two kinds – dependencies within the boxes that directly participate in the activity and dependencies to other boxes. I have shown two other boxes – a product box and a customer box. Note that strictly speaking the order entry process is a consumer of the product and customer boxes, but since there is a dependency we don't bother to draw this because it is implied by the dependency.

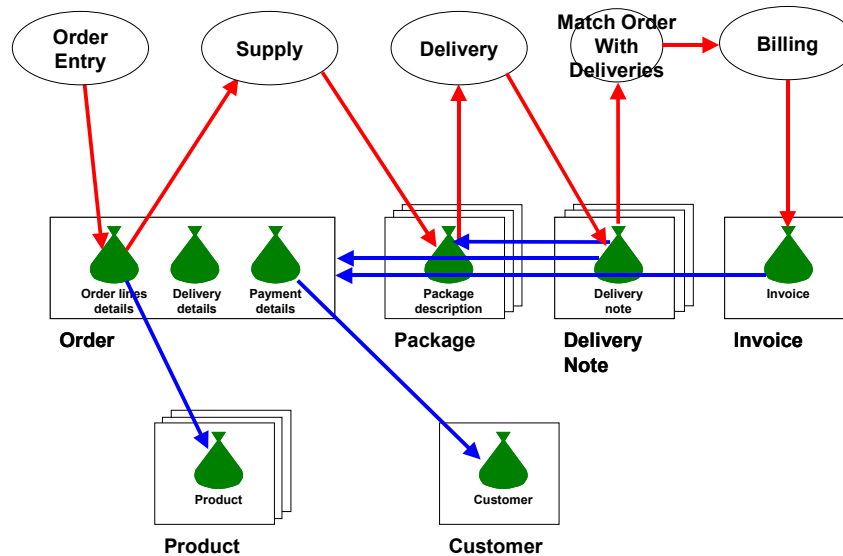


Figure 13. Internal dependencies.

Of course what we are describing is referential integrity and a database system that supports referential integrity is a great help in implementing such dependencies. But database referential integrity is not sufficient because:

- The boxes might be physically implement in another database.
- The dependency is often to data not just the fact a box exists hence, if the data in the bag changes, we need to ask ourselves the question – does change to the data have an impact? For instance, if the address in the customer bag changes should the invoice address change?
- We need to think about the consequences of deletions. What the database does automatically for maintaining referential integrity may not be the behaviour we require. For instance, if the product or customer boxes are deleted, what happens to the order?

Note these questions are the same as the questions we asked about external entities.

Having completed the model to one level of detail, the next step is to take it down to the next level of detail. I will do this with only one of the activities – the supply activity. This is illustrated in figure 14.

We are only concerned with the path highlighted in red since the rest of the diagram shows asynchronous activities. The biggest problem is what to do if the stock isn't available to process the order. The simplest solution would be for the Process Order activity to make sure there is enough stock before starting the Build Package activity. But if this is done, there is still a problem if someone else is ordering the same stock at the same time. In other words, the Build Package could fail because the other order has taken the stock. It might be better to use the alternative solution illustrated in figure 15. This has the Process Orders create the package box and fill it with preliminary information. Stock can be assigned to packages at this stage and the Build

Package activity is less likely to fail. In reality the stock levels recorded on the system may be wrong or some stock is found to be faulty in which case the Build Package will fail. This can be sorted out by the Undo activity.

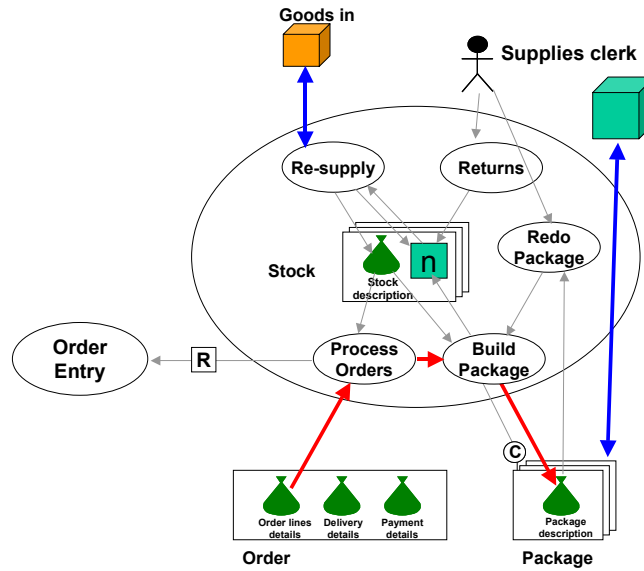


Figure 14. The Supply Activities.

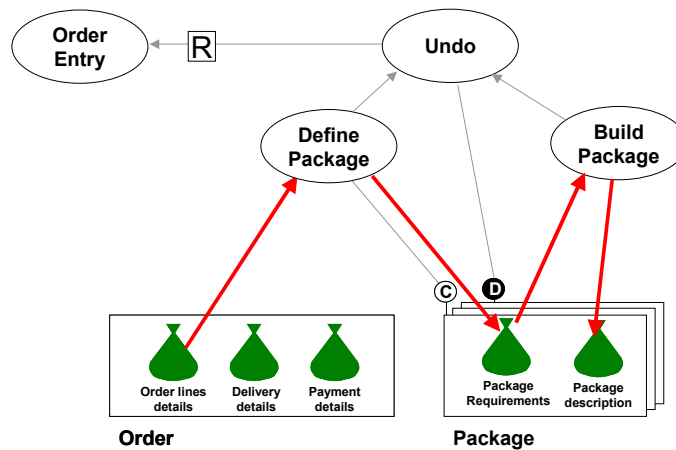


Figure 15. Undo of package activity.

This was an informal run through the thought processes in building a context model using box bag diagrams. In the next section I discuss the issue of correctness with a bit more rigor.

Correctness Criteria

When we look at a model we are often take it for granted that certain things are true. For instance, if you look at a model of a car you know where “down” is. There are rules that are usually unsaid, but are essential in interpreting the model properly. I call these rules correctness criteria. For computer models the correctness criteria are often not so obvious so we need to make sure we understand what they are.

The purpose of the correctness criteria is to provide:

- Criteria that can be tested to ensure the model are complete and consistent.
- Criteria that the implementation must consider to implement the model correctly.

I will present the criteria in categories and subcategories as I find it easier to use that way. The full list is:

1. Criteria that can be tested.
 - 1.1. Data requirements.
 - 1.1.1. *All information boxes are created.*
 - 1.1.2. *All information bags are provided and used.*
 - 1.1.3. *All information counters are used.*
 - 1.1.4. *Information boxes have contents.*
 - 1.1.5. *External dependency tracking.*
 - 1.2. Process requirements
 - 1.2.1. *Process flow exists.*
 - 1.2.2. *Process path exists.*
 - 1.2.3. *Resource assignment.*
 - 1.2.4. *Timeouts.*
 - 1.2.5. *Undo tasks.*
 - 1.2.6. *Error reporting.*
2. Criteria that the implementation must consider.
 - 2.1. Data requirements.
 - 2.1.1. *Information bags are sufficient.*
 - 2.1.2. *Information box are findable.*
 - 2.1.3. *Messages are sufficient.*
 - 2.1.4. *Message transmission.*
 - 2.1.5. *Durability.*
 - 2.1.6. *Dependency on other data.*
 - 2.2. Task requirements.
 - 2.2.1. *Tasks rule consistency.*
 - 2.2.2. *Tasks must be atomic.*
 - 2.2.3. *Tasks must be isolated.*
 - 2.3. Process requirements.
 - 2.3.1. *Once and once only.*
 - 2.3.2. *Processes must be atomic.*
 - 2.3.3. *Processes must be isolated.*
 - 2.3.4. *Synchronization with external actions.*
 - 2.3.5. *React to data dependencies.*

- 2.3.6. *React to external dependencies.*
- 2.4. System requirements.
 - 2.4.1. *Data accuracy.*

Please note that I expect this list will evolve over time. This is typical of engineering solutions. As people understand the problems better they discover more factors to consider.

I will now describe each point in the above lists in more detail.

All information boxes are created. Note, there is no requirement to ensure the information boxes are deleted since there may be a need to keep records of business transactions for a lengthy period of time.

All information bags are provided and used. Otherwise they are pointless.

All information counters are used. Likewise otherwise pointless.

Information boxes have contents. All information boxes have at least one information bag or information counter.

External dependency tracking. If a box is used to track the state of an external party or resource, then there are one or more tasks that update the box if the external party or resource changes.

Process flow exists. If task A has a process flow to task B then something must implement that flow.

Process path exists. All processes have one or more start points and one or more end points, and have a path of process flow from all start points to all end points. (If not all start points reach all end points you have at least two processes not one.)

Resource assignment. All resources that are assigned in a process are returned in the same process (or subprocesses / superprocesses).

Timeouts. Timeouts on tasks and subprocesses are less than the timeout of the calling process.

Undo tasks. Processes and subprocesses have one or more undo task(s) because failure is inevitable. The exception to this is processes that have only one task (which I would prefer not to call processes at all.)

Error reporting. All tasks and subprocesses, except the first in the process path report errors to an undo task or sends a rework message to a task earlier in the process flow.

Information bags are sufficient. Information bags must have data that is complete and consistent sufficient so that the bag's readers can do their work.

Information boxes are findable. Information boxes must be uniquely identifiable so the box can be found by later tasks.

Messages are sufficient. All messages (activity messages and calls) must have data that is complete and consistent sufficient so that the destination task can do its work.

Message transmission. Messages are sent once and once only. They do not lose data in transit. Note, in practice if the destination task reads a message and the task fails (e.g. because of a power cut) it may be necessary to read the message again when repeating the task so you have to be careful about read and delete.

Durability. All bag and box updates must be durable, that is, all data updates made should persist – they cannot accidentally disappear. This is requirement for IT systems to be properly backed up and secure.

Dependency on other data. References between bags and other boxes must always be consistent.

Tasks rule consistency. Task rules are consistent with themselves and consistent with the inputs and outputs defined in the context model.

Tasks must be atomic. This translates into three sub-criteria.

- Every task must always either terminate successfully or fail.
- If a task fails, all outcomes must be undone.
- Every task should have a timeout (to ensure it does not remain suspended indefinitely.)

In other words, there is no half way house between success and failure. The most common reason why a task may not finish properly is probably when there is a failure and it is ignored.

Tasks must be isolated. Tasks don't mess with other tasks running at the same time they are running. This means, among other things, that:

- Information produced by a task must only be made available to other tasks only when the task is finished (since the task may abort – in which case the information would be invalid.)
- Failure of one task should not have knock on effects on other activities in progress at the same time.

Once and once only. When there is a flow in a process from task A to task B, task B must be executed once and once only. For instance in figure 2, expenses should not be paid twice.

Processes must be atomic. A process must succeed or fail but not half succeed or be suspended indefinitely. In other words, a process ends with success or an error. Furthermore if a process fails the outputs associated with the process must not be produced. Typical reasons for failing to meet this criteria are:

- A message is missed.
- A box output from one task is not found by the next task in the path.

- The activity goes into a loop – one activity calls another that calls the first, etc.

Note, loops always occur if there is a rework call or an undo task so they cannot be eliminated entirely.

If a process fails, the steps that have already been done must be undone. This means not only data but also external things like taking undelivered packages and putting the goods back into stock. The simple part of undoing a task is removing the boxes and bags that were created by the task and reversing any counter adds or subtracts. The difficult part of undoing include:

- Undoing the effects of passing a message;
- Undoing an information bag update;
- Giving enough information to the preceding activity to do its undo.

These are the areas that need concentrated examination. Finally, in some business activities external entities may want to be reused. For instance, in our order processing example, the packages might be reused for a different order rather than unpacked.

A fallback solution is to have a timeout on all or part of the activity path. Any time an activity depends on an external agent there is the possibility that the external agent may not do their job and may not report the failure. People leave or go on holiday, messages get lost and so on. There needs to be a mechanism to escalate problems (preferably before the customer does).

Ultimately you probably cannot cater for all eventualities – like failures in the failure handling processing - so ultimately the system may rely on manual intervention.

Processes must be isolated. This means:

- Steps in the process must not get muddled with another parallel process. For instance, a person must not get paid someone else's expenses.
- Failure of one process should not have unwanted knock on effects on other processes in progress at the same time.

This is not to say that processes cannot interact with other activities happening at the same time. For instance, in our order processing example, it may be reasonable for a delivery to handle two customer deliveries at the same time. Hence the failure of a delivery might affect more than one order.

Synchronization with external actions. External actions must be complete before a subprocess is finished and the next step starts (for instance a customer should not be billed until the delivery is complete.)

React to data dependencies. If a process depends on information that is updated outside the process, then there should be a mechanism by which the process can react to changes to that information. For instance, suppose an order uses product information and the product is cancelled. Action must be taken on all outstanding orders for that product to cancel or rework the orders.

React to external dependencies. If information is used to track external entities (e.g. packages), actions (e.g. delivery), or party (e.g. customer) there must be steps taken to ensure if the external entity is changed then the data tracking it is changed and appropriate action taken.

In all cases the dependency to an internal or external entity is only relevant for a certain period of time. The dependency relationship can be subtle. For instance the relationship between order and product is mainly significant from the period of time the order is created to the point in time when the product is packed because from the customer's point of view, if the product is discontinued after that point, the order is still fulfilled. But the data for the discontinued product must persist, firstly for reporting reasons, and secondly in case the order is rejected and has to be redone. The dependency relationship between order and customer is longer – it persists from the time the order was made to payment.

Data accuracy. All tasks that create or update information have a responsibility to ensure that data is accurate as much as that is reasonably possible to do. The system design should not duplicate information without having a proper plan to ensure the information is synchronized at the time it is used.

Today, most of these criteria are enforced by manual supervision and manual intervention. There is usually ample scope of more automation and control.

Using Context Models

As noted earlier in this paper, context models are used for developing IT strategies and plans and as the first step in application development. This means that in the overwhelming number of cases, the first step is to build a context model for the existing system. When this is done, the model can be analysed. Points that are likely to arise out of the analysis are:

- Data duplication – separate boxes that contain the same information;
- Inefficiencies like information being passed by email and retyped rather than being sent directly;
- Missing error reporting routines;
- Failure handling that leaves the data in an inconsistent state.

Of course, analysis will also reveal mistakes in the context model so will require several iterations to get it right. Building a context model is a great tool for uncovering the reality of how the system works. You also uncover complacency, comments along the lines of “we don't bother having a procedure for that event because it will never happen”. To counteract this and because it's a useful exercise in any case, it is desirable to document and understand a sample of problems that have occurred, and to analyse them in terms of the context model. This way you have concrete evidence on which parts of the system are not working well.

If you are doing an enterprise architecture study (or actually more or less anything with “architecture” in the title) you may find yourself building a context model in parallel with building an application architecture model and a technical architecture model. There is a temptation (especially for people with a technical background) to do the context model rather superficially. This is a mistake. The more you understand how the organization works and how the technology supports the business, the easier it becomes to underpin your recommendations with a solid business case.

A context model is an essential tool for deciding how to move an existing application forward. The issue is discussed in more detail in the paper “The Three Level Framework”.

Hardly any application development projects these days use context modelling as an integral part of the development process. This is one reason, I believe, why application development has such a poor record of success. I believe a context model should be developed before doing anything else. There are several reasons for this. The context model will:

- Help eliminate business process design flaws;
- Provide information that will help you decide between application architecture and technical architecture design options;
- Help identify all the tasks that need to be done and thereby help understand the size of the project;
- Help partition the tasks into groups that can be implemented together;
- Help ensure tasks that interact are designed co-operatively.

The larger the project the more important these points become.

As the project progresses, the context model becomes more background documentation rather than something actively being developed during the project’s progress. There may be occasions when you will uncover flaws in the context model. This is to be welcomed; if there is one thing worse than uncovering a flaw in the context model, it is having a flaw that has not been uncovered. The advantage of maintaining a context model is that you can work out the ramifications of correcting the flaw and you can communicate the changes to the affected parties.

So how do you develop a context model for new business area? In the paper “Making IT Application Design an Engineering Discipline” I describe a four step activity for design – understand, select design hypothesis, clarify and analysis. Applying these four steps to building a context model we have:

Understand – find out about the organization, the external context of what they want to do, the activities they want to change, how the activities work currently, and why the organization wants to change.

Select design hypothesis – brainstorm different approaches and select one. Maybe draw a few informal box bag diagrams for alternative solutions.

Clarify – take one solution and develop a first cut box bag model. You need to:

- a. Identify the processes and break them down into tasks.
- b. Identify the boxes and bags.
- c. Identify the events that trigger each task.
- d. Ensure the main path of the flow has a clear beginning and end.
- e. Identify external dependencies.
- f. Identify internal dependencies.
- g. Develop session contexts.
- h. Check that resources are assigned and returned.

Analyze – try to pick holes in the design.

The analysis phase is done by taking every activity and checking through the criteria for correctness for each.

Not having a context model is not an option. If you haven't designed a context model as discussed in this paper or put it into a computer model or a document, then you still have a context model, but you have it in your head. Unfortunately so does everyone else and you cannot guarantee that all the different context models are complete and consistent. It's far better to make all such thoughts concrete, open and understood by all.

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 "The Three Level Framework" that describes the other models shown in figure 1.

A modelling tool has been developed for context models that covers many of the issues in taking the box bag models into implementation design. It generates drawings of box bag diagrams, as well as providing a certain amount of automatic analysis. The documentation describes the data that underpins the context model. The tool is available from the web site www.itmodeller.co.uk/models/.