B usiness
O bject
R eference
O ntology
Program

simplifying semantics

Working Paper

MA1

Methodology: Application - 1

Starting a Re-Engineering Project

Issue: Version - 4.01 - 01-July-2001

Contact

For queries regarding this document, or the BORO Program in general, please use the following email address:

contact@BOROProgram.org

MA1

METHODOLOGY: APPLICATION - 1

STARTING A RE-ENGINEERING PROJECT

CONTENTS

MA1
METHODOLOGY: APPLICATION - 1

STARTING A RE-ENGINEERING PROJECT

1 Introduction

The BORO *O—ONTOLOGY Papers* give you an understanding of what business objects are. The *BORO M—The BORO Re-Engineering Methodology* Papers show you how to apply this understanding, describing a systematic method for re-engineering an entity ontology oriented system into a business ontology object model. But there are also management factors that affect whether a BORO project to re-engineer an entity ontology oriented systems will be a success.

Inevitably a BORO re-engineering project works in a different way from traditional system building projects. Its success depends, to quite a large extent, on managing it in a way that recognises these differences. This paper outlines five management tactics that I have found help to make re-engineering projects a success. These are:

- Taking a re-engineering approach,
- Establishing priorities for constructing fruitful, general and so re-usable patterns,
- Taking care to manage large projects in a generalisation-friendly way,
- Producing a validated understanding of the business, and

- Object modelling the 'data' translation/migration.

2 Take a re-engineering approach

It is important to approach the project as a re-engineering exercise. People sometimes succumb to the temptation of thinking that they can only construct a radically new system by 'starting with a blank sheet of paper'. They want to ignore the existing system completely, so as not to taint the new system with its mistakes.

While this tactic may have its merits when working within a paradigm, it is not re-engineering. In fact, it is the wrong way of shifting to a new and better ontology paradigm. Apart from the actual difficulty of 'blanking out' all knowledge of the existing system's patterns, this approach ignores the nature of re-engineering and evidence of how successful re-engineerings have worked.

2.1 Salvaging investment in business patterns

A core feature of re-engineering is that it salvages the business patterns embedded in the existing system. The great 17th century physicist Isaac Newton used a striking image for this. He commented in *Mathematical Principles* (describing his re-engineered physics) that his work was only possible because 'he stood on the shoulders of giants'. A new paradigm may be radically different, but it normally takes full advantage of the investment in the patterns of the old paradigm. Building patterns from scratch takes a substantial investment of time and effort and is just not practical in most situations.

This is why a re-engineering approach actively seeks out and re-engineers the business patterns in the existing system, salvaging the investment made in them. We saw how this worked in the worked examples in previous papers. The re-engineering of the existing system's entity formats yielded radically different, useful, general patterns that formed a foundation for the business object models.

From an economic point of view, the big benefit of the salvage approach is that it needs much less investment than most other approaches. A systematic re-engineering approach, such as that described in the previous papers, salvages the existing system's investment in business patterns. The re-engineering project effectively starts with this investment in hand.

2.2 A well-defined scope

Basing the project on the re-engineering of the existing system's entity formats also provides a simple and effective way of scoping the project. From a management control point of view, it is important to have a reasonably clear idea of the boundaries of the project. The existing system's business patterns provide just that. We can define the scope in terms of the entity formats in the existing system that hold these patterns (things such as files and records). It is a simple matter for us to list these, giving the project a clear-cut boundary.

If we did not have a clear-cut boundary, there could be problems. Objects tend to be closely linked to a number of other objects. Their webby nature means that they have few, if any, natural boundaries. If we were to keep on analysing them until we found a natural boundary, we would end up constructing a model of everything.

You may want the project to include in its scope some business patterns that are not embedded in the existing system (in other words, new requirements). These can be associated with a related pattern that is embedded, and the patterns re-engineered together. Time zones (from the re-engineering of temporal patterns in *MW4—Re-Engineering Time*) could be considered an example. If they were within the scope, but not embedded in the existing system, then we could associate them with either the bank holiday or weekend patterns, which are. The two sets of patterns could then be re-engineered together.

However, it makes sense to try and schedule the development of specific new requirements after the re-engineering project. Re-engineering delivers significant amounts of extra functionality as well as a new way of seeing the business. When

this takes shape, the original 'new' requirements may already be satisfied or have taken on a completely new meaning.

3 Establish priorities for the construction of fruitful, general, and so re-usable patterns

An important benefit of the object paradigm that it enables us to construct fruitful, general and so re-usable patterns. However, to get these patterns, we have to actively exploit the paradigm. This means establishing priorities for the construction of re-usable patterns.

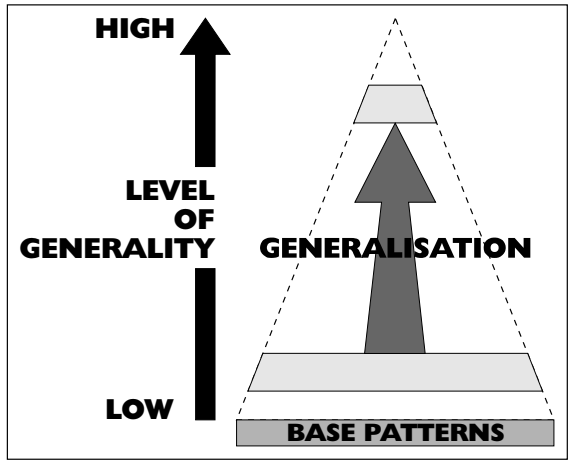
3.1 Establishing priorities for the construction of general, and so re-usable patterns

The worked examples in the previous papers illustrated how vital generalising is to successful business object modelling. The more general a pattern, the more potentially re-usable it is and so more useful.

3.1.1 Generalisation produces a compact system

Generalisation's usefulness comes, in part, from its ability to compact. It enables a large number of base patterns to be re-engineered into a much smaller number of simpler more general patterns. In essence, it fits more information into a smaller space (illustrated graphically in *Figure MA1–1*).

Figure MA1–1 Generalisation fits more information into a smaller space



Generalisation leads to less costly components

In computing terms, generalisation's compacting means fewer, simpler components. Compacting works its way through the development life cycle. Compacting during business object modelling leads to fewer components in the business model, and this translates into fewer components at all the later stages of system building. There are fewer and simpler components to specify during systems analysis and design and so fewer and simpler components to code and test. This in turn means fewer and simpler components to maintain and fix. Furthermore, when people start to learn the system, there are fewer and simpler components to master. Overall, the effort required to build and maintain the system is reduced. This translates into a reduction in time and cost. So systems with generalised components are quicker and cheaper to both build and maintain.

Generalisation means no inherent complexity

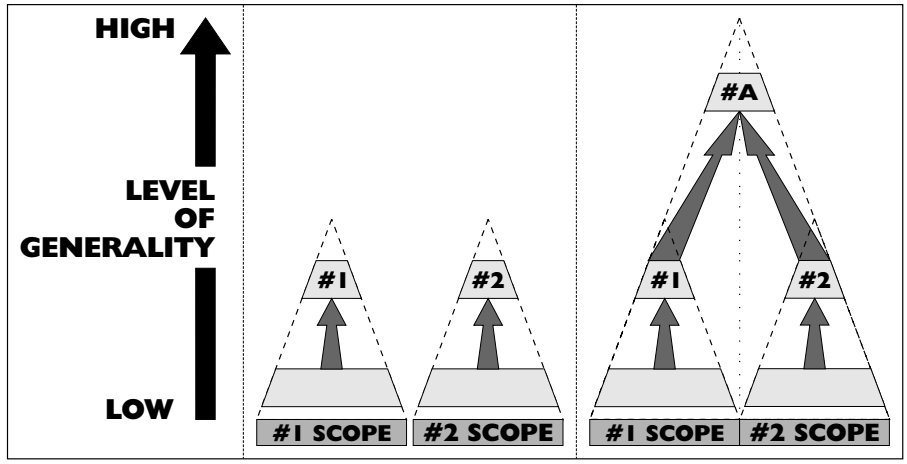
An important consequence of using generalisation as a core tool in system building is that we can no longer think of a pattern having an inherent complexity. We are used to thinking that a complex set of business patterns needs a computer system of equivalent complexity. With generalisation, this rule of thumb no longer works. The worked examples have shown us how, using a combination of re-engineering and generalisation, we can transform complex patterns into simpler, more powerful, general patterns. This may initially seem slightly counter-intuitive, but it is a natural way of working. It is, for example, the way in which science

accumulates knowledge. If we look at its history, we see again and again a complex theory being superseded by a simpler, more powerful, theory.

Part of what is going on is that as we generalise the patterns, their scope increases. This is shown schematically in *Figure MA1–2*. Here, the patterns, #1 and #2, are generalised into pattern #A, which covers the same scope as #1 and #2.

The number of patterns needed for the full scope has halved from two to one. However, this is only part of the story. In actual re-engineerings, the generalised pattern often turns out to be simpler. We saw this in the model for spatial patterns. After re-engineering the first group of entity formats, the final model not only has fewer objects (at the application level), but is also simpler. The power of the spatial model was revealed in re-engineering of bank address in *MW3— Re-Engineering Bank Address*, where no new business objects were needed.

Figure MA1–2 Generalisation creates patterns with increased scope



The impact on estimating

This lack of inherent complexity has a serious impact on the way in which we estimate re-engineering projects. In traditional estimating, a sensible rule of thumb is that there is a reasonable correlation between the complexity of the requirements and the effort (and so cost) of building the system. This assumes that complexity is unaffected by the system building process. The complex pattern that goes into the process inevitably leads to complex code. This made estimat-

ing relatively easy. The resources needed to build a system could be calculated from the complexity of its requirements.

With generalisation, this is no longer true. The business modelling process can significantly reduce the complexity of a pattern. Two patterns, one complex and the other simple, may both be re-engineered into the same simple pattern. This happened in the worked examples with the simple country pattern (in *MW1—Re-Engineering Country*) and the more complex address pattern (in *MW3— Re-Engineering Bank Address*). We re-engineered both entity formats into the same general pattern – which will end up as the same computer code. The complexity of the requirements no longer correlates well with the resources needed to build the system.

It might seem that the correlation still applies within the business object modelling stage. It is certainly true that a complex pattern can take longer to model than a simpler one, other things being equal. However, other things are not often equal. For example, the re-engineering of the simple country pattern took much longer than the re-engineering of the more complex address pattern, because we could match the address patterns with the re-engineered country patterns. The estimation of effort can no longer be based simply on the complexity of a requirement.

With experience, one acquires a rough feel for how much compacting to expect from a group of patterns, and can translate this into a rough estimate. I suspect it will be some years before there is enough hard data to work out a formula. This makes accurate estimating difficult. The bright side is that as generalisation compacts and simplifies, so building a system from a generalised business object model takes less time and effort than building it from an ungeneralised model.

3.1.2 A generalisation friendly environment

Generalisation brings benefits, but how do we encourage generalisation? People naturally and unconsciously generalise patterns. However, without a framework to help them, this instinctive tendency does not normally lead to very general patterns. The object paradigm remedies this by providing a generalisation

friendly environment, within which a systematic approach can encourage more general patterns.

Traditional methods of system building do not have access to the compacting power of generalisation. They do not provide an environment that is conducive to producing general objects and so have no reason to make generalisation a priority. A good, and widespread, example of this is common subroutines. Analysts and programmers naturally recognise their potential. They naturally construct reasonably general subroutines during system building. However, traditional approaches to modelling processes, such as functional decomposition, hinder rather than help this natural process. Analysts have to rely on their instinct and initiative and ignore the approach.

One particular experience of this sticks clearly in my mind. Many years ago one of the projects I was managing was a large development, using the popular SSADM method. Towards the end of the analysis stage, a lead analyst told me he was going to start identifying common subroutines. After some discussion, it became clear that he was in uncharted (though familiar) territory. This task was not identified by the method; so, he had not put it in his plan. Yet, he realised it needed to be done. The method had no systematic techniques to help him, so he had to rely on his intuition.

When he discovered common subroutines, he found that they could not be described either in the method's (data flow diagram) functional decomposition structure or in the CASE tool he was using. The root of the problem was that neither of them could model the way two processes use the same common subroutine—what could be called re-composition. They could handle de-composition's tree structure, but not re-composition's lattice structure. Given that common subroutines depend on re-composition, this meant functional decomposition actually hindered this type of generalisation.

By contrast, a re-engineering project not only supports generalisation, but also has a systematic approach that actively encourages it. In this environment, it makes sense for project managers to make generalisation one of the top priorities. However, old habits die hard. Modellers used to a traditional environment do

not naturally push generalisation as far as it should go. Project leaders can help ensure successful generalisation by actively checking how much of it is going on and persuading modellers to do more when it is needed.

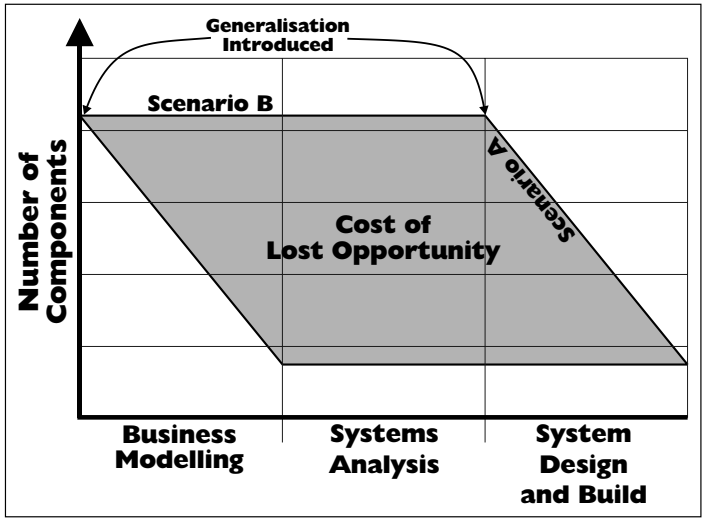
3.1.3 Introducing generalisation during business modelling

A general theme running through many approaches to building computer systems is that the earlier in the system development life-cycle we introduce a good technique, the greater the benefits. This not only feels intuitively correct for generalisation, it is correct—the best time to generalise is during business modelling. However, as the tale of the lead analyst identifying common subroutines above illustrates, system builders often generalise later in the life-cycle.

Economically sensible to generalise business patterns early

It is reasonably obvious that the earlier generalisation is introduced into the development life-cycle, the greater the reduction in overall effort. We can visualise this by thinking in terms of the reduction in the number of system components. *Figure MA1–3* shows a simplified schema of this.

Figure MA1–3 Introducing generalisation at different life-cycle stages



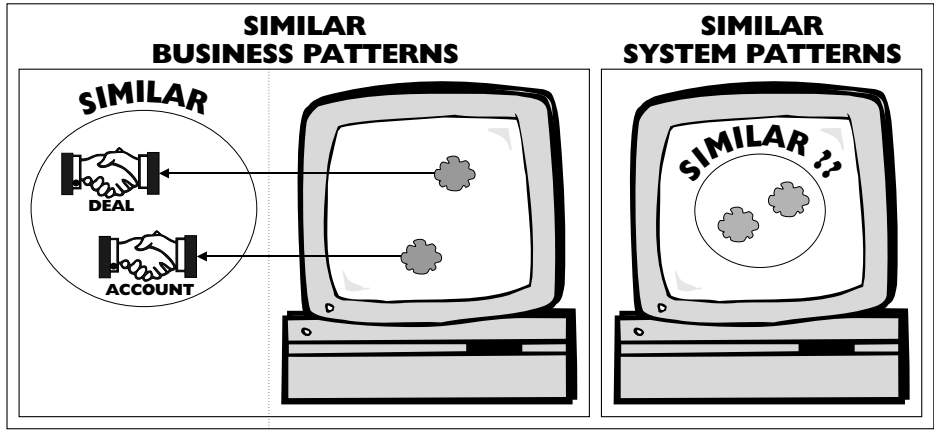
If generalisation is introduced at the business modelling stage (shown as scenario A in the schema), the benefit of compacting is delivered from the beginning of the life-cycle. The compacted business components are used in system analysis and on into system design and build.

If, however, generalisation is introduced at the system design and build stage (shown as scenario B in the schema), then the reduction in components only occurs then. (For simplicity's sake, it is assumed that the final reduction in scenario B is equivalent to scenario A.) The reduction in costs associated with compacting only starts appearing then; it does not happen during business modelling or systems analysis. The cost of this lost opportunity is shown in the schema by the shading between scenarios A and B. The earlier we compact the components, the greater the reduction in costs. The later we compact the components, the greater the cost of the lost opportunity.

The natural stage to generalise business patterns

As well as the 'economic' cost reasons for generalising business patterns during business modelling, there is also a sound practical reason for generalising them at this stage—it is the natural stage to do it. We use similarity to generalise business patterns. Finding this similarity is more naturally done at the business modelling stage, while we are looking at the objects in the business (illustrated in *Figure MA1–4*). In the later stages, when we turn our attention away from the business and towards the system, the business patterns are not so visible.

Figure MA1–4 Similar business patterns



3.2 Establishing priorities for the construction of fruitful patterns

Generalisation and fruitfulness can be seen as two sides of the same coin. A fruitful pattern can either be sufficiently general to be re-used frequently or sufficiently similar to many other patterns to be generalised into a common pattern. While we can systematically generalise, fruitfulness is more elusive. It is a kind of potential for generalisation—an ability to deal with future patterns that have not been modelled as yet.

3.2.1 Fruitfulness during and beyond a project

This fruitfulness shows itself in two ways—within the scope of the project and outside it. While a project is in progress, the team can see how a fruitful pattern leads to high levels of generalisation. Its potential fruitfulness becomes actual before their eyes as the re-engineering reveals generalisations. But does this potential only extend as far as the agreed scope? If the scope were widened, would the pattern's fruitfulness suddenly dry up?

Our experience is that it does not. Where a pattern has been fruitful within a re-engineering project, its fruitfulness always seems to extend well beyond the scope of the project. One way this reveals itself was mentioned in the *AS1—An Overview of the Strategy*, which describedthe experience of users finding that their re-engineered system could handle situations not in the scope, including situations they did not even envisage when the system was developed.

The naming and spatio-temporal patterns in the previous papers provide another example. They were fruitful outside their initial scope. For example, the fruitfulness of the naming and the nesting geo-political area patterns was clearly shown in the re-engineering of address. They effectively matched all the address patterns. In my experience, the naming and spatio-temporal patterns are also fruitful outside the scope of the worked examples. I have found their patterns re-appearing in many re-engineerings. (You might remember it re-appeared in the bank holiday example in *MW4*.) This often means that future business requirements are either already catered for by the system or can be relatively easily dealt with.

3.2.2 Building fruitful patterns from complex entity formats

Seeking out fruitful business patterns is a sensible goal for a re-engineering. However, taking this as a goal, overturns a rule of thumb in traditional system building. We touched on this point when capturing the conceptual patterns for country in *MW1—Re-Engineering Country*. There we noted a tendency to favour dropping complex patterns when setting the scope of a project,. In a traditional environment, this makes sense because they take more of an effort to build. In an object-oriented environment, it does not. It is not that complex patterns no longer necessarily take more resources to build. It is that complex patterns are more likely to have fruitful patterns embedded in them. These fruitful patterns are the Holy Grails of business modellers; the more that can be found the better.

3.2.3 More accurate patterns are more fruitful

There is another way to increase the fruitfulness of the business patterns, and that is to make them more accurate. In *AS3—What and How we Re-engineer*, we looked at how increased physical accuracy was essential to the introduction of interchangeable parts in manufacturing. We observed that a similar revolution in accuracy—this time, accurately reflecting the world—was necessary for the introduction of interchangeable parts in business modelling.

The *0—ONTOLOGY Papers*, describe how this accuracy has increased as information paradigms evolved. They describe , for example, how modern literate western culture has more accurate notions of sameness and signs than the Huichol Indians, who see corn and deer as the same (this is discussed in *OP2—Substance Ontology Paradigm*. They discuss how western culture is now developing an understanding of the logical paradigm's more accurate distinction between the whole–part, super–sub-class and class–member patterns. And how it has started to absorb the object paradigm's notion of sameness for four-dimensional objects. It is in the process of providing an accurate explanation of how something now is the same as it was yesterday; it no longer has to be both the same and different, much like the Huichol's corn and deer.

When we build a business ontology object model, it is important that we take advantage of the object ontology paradigm's ability to produce more accurate patterns by using them to construct more general and reusable patterns. Just as physical accuracy enabled interchangeable reusable parts, so referential accuracy encourages the information paradigm's counterpart—generalisation and reuse. Increased accuracy reveals the patterns more explicitly, taking the guesswork out of whether patterns are similar or not. The general patterns constructed from more accurate lower level patterns inherit their accuracy and so are able to operate at higher levels of generality.

So it makes sense for the manager of a re-engineering project to try and determine whether his modellers are being referentially accurate. And if they are not, to take remedial action. This should help to ensure the fruitfulness of the patterns.

Object model's lower granularity

Increased referential accuracy also leads to a lower granularity in the descriptions of patterns—which initially means more objects. If you look at the early version of the spatial model in *MW1—Re-Engineering Country*, you can see that the re-engineering increased the number of operational items. From this, it might appear that we have to weigh the benefits of increased accuracy against the cost of handling a larger model. But, it turns out we do not. The increase in accuracy leads to a corresponding increase in generalisation, which reduces the number of application objects significantly.

In the region example in *MW2—Re-Engineering Region*, the generalised patterns made almost all the patterns from the re-engineering of the country example redundant; this was only the second file re-engineered. In the first stage of the address example in *MW3— Re-Engineering Bank Address*, no new application objects were re-engineered. It is these application objects that the system builders construct. Once the re-engineering gets beyond a few entity formats, at the application level, the compacting effects of generalising more accurate patterns more than outweighs the expanding effects of their lower granularity.

Learning to see accurately

Learning to see with the referential accuracy demanded by the object paradigm is one of the most challenging aspects of business object modelling. It is important for the success of the project that the people undertaking the business modelling have mastered the challenge and learnt to see in this new way. It also helps if the people carrying out the systems analysis have at least a broad understanding of business objects.

Managers should ensure that the people working on their project have the right level of understanding. For the people that need training, *The BORO Working Papers* is one way of providing a useful grounding in business ontology object modelling. It can be usefully supplemented by formal training courses. However, once people have mastered the foundations, there is no real substitute for experience on a real project. At this stage, the easiest way to progress is by working with expert practitioners, learning by example.

This generally means that IT departments starting out on a project have to either buy in or grow experts. If the decision is to grow, then it is only sensible for inexperienced people to cut their teeth on a trial project in a non-critical area of the business. Their learning can be speeded up considerably if the team is beefed up with an expert mentor.

4 Taking care to manage large projects in a generalisation-friendly way

To take full advantage of the benefits generalisation brings, we need to manage it. In large projects, it is particularly easy to stifle the potential for high levels of generalisation. To create a stable generalisation-friendly environment, we need to ensure the careful management of the balance between the increased opportunity for generalisation that comes with widening the scope and the increased risks associated with large projects.

4.1 Widening the scope increases the opportunities for generalisation

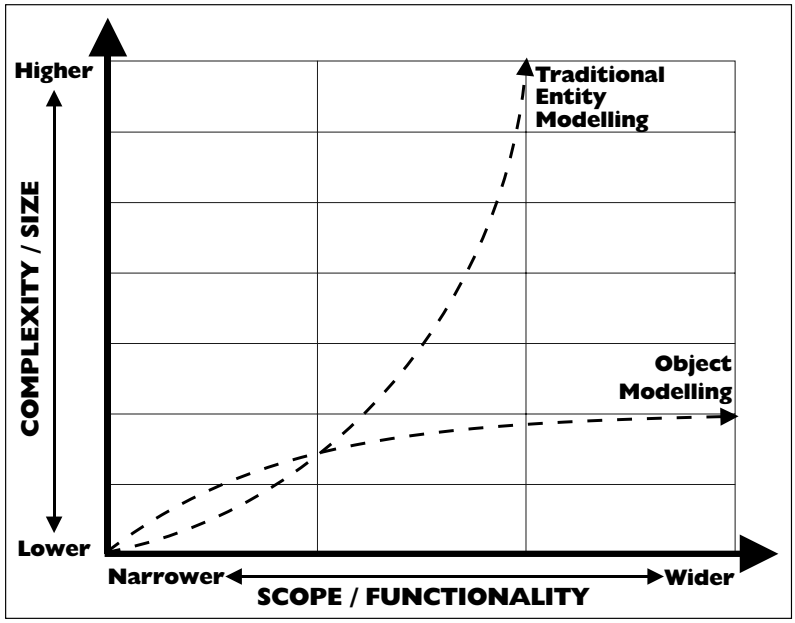
Each new business pattern added to the scope of a re-engineering project brings an opportunity for generalisation. We saw an example of this in *MW3— Re-Engineering Bank Address's* re-engineering of region. Extending the scope of the re-engineering from country to region enabled us to generalise both patterns to geo-political area. Though it may seem counter-intuitive, widening the scope led to a smaller, more general and powerful model rather than a bigger one.

The more conceptually powerful the model, the more pronounced this effect. The model does not have to grow in size or complexity as we add patterns, we can generalise and make it smaller and simpler instead. When we introduce new patterns to a generalised model, the likelihood of us being able to generalise them is higher. This is because the model's general patterns are more likely to match with the new patterns.

This also means that the more general the model, the smaller the cost of re-engineering each new pattern is likely to be. We saw an example of this in address's re-engineering in *MW3— Re-Engineering Bank Address*. The spatial model was sufficiently general that the new business patterns introduced in the address entity formats all matched with existing patterns. There were no new patterns; so, the cost of building new computer code for the address business patterns would be nil!

This is the complete opposite of what happens in traditional system building, where generalisation is not properly supported. There we have to harmonise each additional pattern with the existing patterns, adding to the complexity of the system. As the number of patterns in the system increases, the task of harmonisation gets more onerous. The traditional rule of thumb is that the more patterns there are, the greater the cost of handling each new pattern. The difference between traditional system building and system building using business object modelling is shown graphically in *Figure MA1–5*

Figure MA1–5
Correlation
between scope
and complexity



4.2 Balancing the economies of scope against the problems of size

This would seem to imply, at least in theory, that it is better to have as wide a scope as possible in a re-engineering project, because this will keep the costs of building the system down. To an extent this is correct, because the wider the scope the greater the opportunity for generalisation. But as the scope increases, so does the size of the project. And a large project brings increased risks.

The scope of re-engineering projects is defined in terms of the entity formats of an existing system. If the system is small, it has few entity formats and so can be re-engineered in one go. Doing it piecemeal would just take longer and demand more effort. When a large system is re-engineered the situation changes. The scope includes many more entity formats. Re-engineering them all in a single project usually takes either many years, a large team or both.

However, the longer a project lasts and the larger the number of people involved, the more difficult it is to manage. If it gets very large it is much more likely to end up out of control. Furthermore, you cannot guarantee that, after all the man years of effort, the system will actually work. So, when re-engineering large systems, we need to balance the benefits of a wide scope against the inherent risks in a large project.

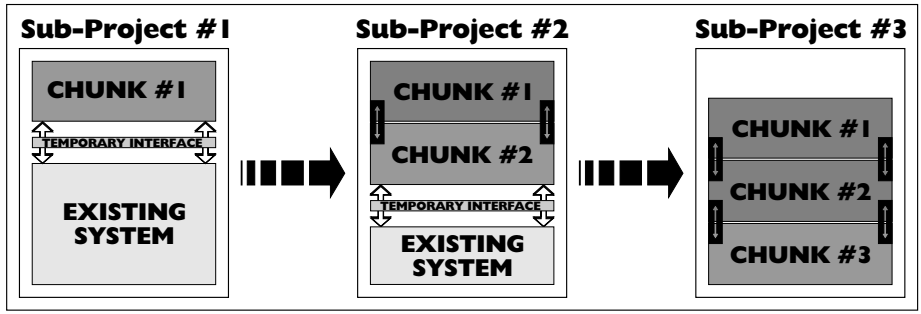
4.3 Chunking the existing system

In practice, people tend to redevelop large systems bit by bit. It is like the child's riddle—'How do you eat an elephant?' The answer is 'in bite-sized chunks'. By dividing the system into manageable (digestible) chunks, we can keep things under control. As we implement chunks at regular intervals, we are giving tangible evidence of progress. Management can see the results of their investment reasonably soon after they make it—instead of waiting until the end of the overall project.

One problem has to be overcome in this approach. A system, by its very nature, is an interconnected coherent whole. When we redevelop a chunk, we have to fit it in with the existing system, if it and the system are to work together. However, if we design the new chunk to work with the old system, it will probably inherit some of the structure of the old system.

The standard tactic in traditional system re-development for dealing with this problem is to design the new chunk to work unencumbered by the existing system. Then, to get the two to work together, a temporary interface is built that handles the connections between the old and the new. As each new chunk is redeveloped, it permanently links up with the other new chunks and connects to the old system through a new temporary interface. When all the chunks are redeveloped there is no longer a need for a temporary interface. This is shown schematically in *Figure MA1–6*.

Figure MA1–6 Eating the problem in bite-size chunks

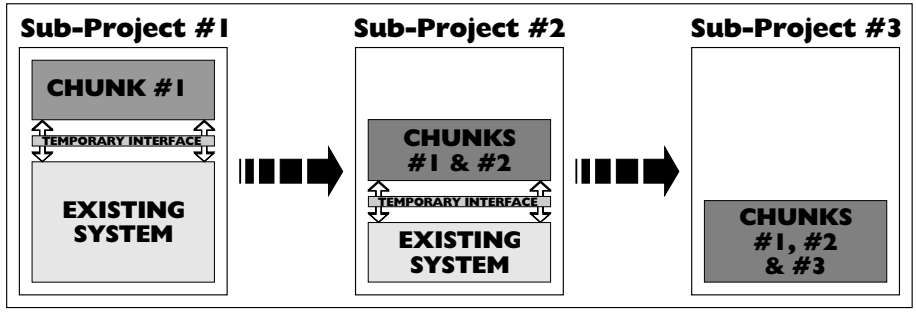


4.4 The object re-engineering approach to chunking

Working on the re-engineering large systems, we have developed a varaition of the standard practice of chunking. We divided the overall project into a number of subprojects, each dealing with a chunk of the existing system. But we did not re-engineer each chunk in isolation. If we had, this would have restricted the scope of each re-engineering to its chunk, losing the potential for generalising patterns across chunks.

Instead, we adopted a different approach. As we re-engineered the chunks in turn, we included the scope of all the previous chunks. This meant that by the time we came to the last chunk, the scope of the re-engineering was the whole existing system (illustrated in *Figure MA1–7*). In this way, we took full advantage of the power of generalisation across a wide scope, without the risks associated with a large project.

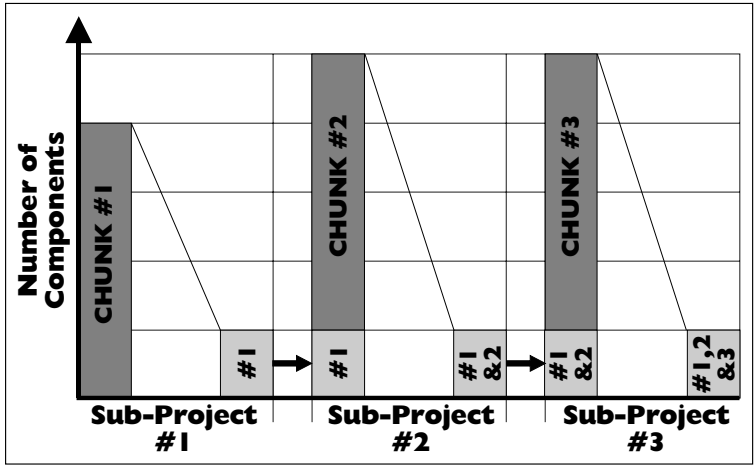
Figure MA1–7 Re-engineering combined chunks



In a traditional environment where there is little generalisation, this approach would make little sense. If, when the first chunk was re-developed, its requirements were included in the scope of the second chunk, then this would effectively double the size of the chunk. Adding in more chunks as they were re-developed to subsequent chunks would treble, quadruple, and so on, their size. This would lead to larger and larger projects, defeating the whole purpose of chunking.

However, in an object-oriented environment, the approach is sound. When we re-engineer a chunk, we generalise and compact its business patterns. So, when we widen the scope to include the re-engineered patterns from the earlier chunks, this does not lead to a substantial increase in the actual number of patterns. The earlier chunks' patterns are general; so, it leads to a substantial increase in the opportunities for generalisation. As a result, adding in earlier chunks does not substantially increase the size of individual chunks. In some cases it can actually substantially reduce the size of the re-engineered chunk (illustrated schematically in *Figure MA1–8*). By the time we get to the last chunk, the scope has widened to the whole system without any of the sub-projects being any larger than they would have been in a traditional system building project.

Figure MA1–8 Compacting combined chunks



4.5 The benefits of chunking

One of the big benefits of this kind of chunking is that there are multiple opportunities to get a pattern right. After each combined chunk is implemented, the modellers get a chance to see how their patterns are performing in a live system. This suggests improvements that they can incorporate into the re-engineering of the next combined chunk. Each redeveloped chunk, except the final one, can be treated as a prototype for the next chunk of redevelopment.

This encourages the development of fruitful patterns. People are unlikely to find the most fruitful general patterns at the first attempt. To some extent, they have to go through a process of trial and error. And chunking offers a controlled environment for trying out the patterns and finding any errors. The opportunity to have a second, third and even fourth chance to construct the right pattern, and to see each attempt in live operation, significantly increases the chance of constructing a fruitful general pattern.

This goes against the grain of the mind-set associated with traditional approaches. Because these normally only allow one attempt at constructing the right pattern, great store is set on finding a strong rigid fixed pattern that lasts the whole of the redevelopment and beyond. The object approach turns this value judgement on its head; in a re-engineering, fixed patterns are bad. If a pattern remains fixed, then this is probably because it is not being generalised. Under the object approach, the goal becomes generalising patterns rather than finding fixed ones.

4.6 Choosing chunks

One awkward management decision is deciding how to chunk up the existing system. There are many factors to weigh up, and these vary from system to system. One important factor is the type of information passing between the candidate chunks. If we choose chunks that have only a few types of information passing between them and the rest of the system, then we keep the 'complexity' of the temporary interface low.

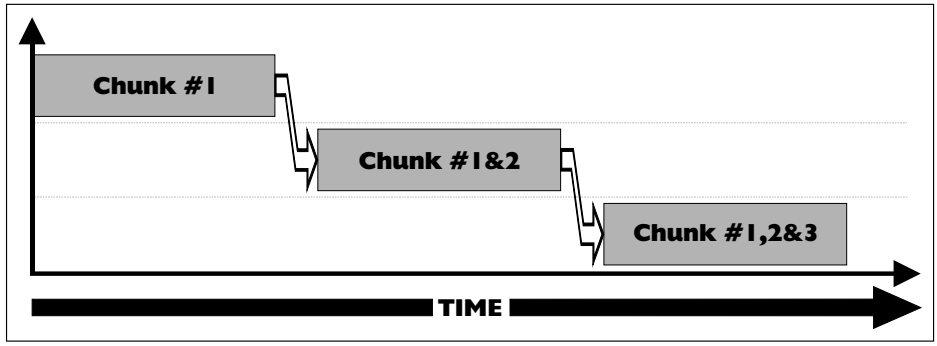
Another factor is encouraging generalisation by putting similar patterns together in the same chunk. For example, most modellers familiar with the financial sector would be able to guess that the securities and currencies patterns are similar. If we were to allocate these patterns to the same chunk, then this would encourage their generalisation to the financial asset pattern (illustrated in MA2's *Figure MA2–6*).

Most systems naturally fall into a number of modules; often, these are a reasonable basis for chunking. This still leaves open decisions on whether to have each module as a small chunk or group modules together into bigger chunks. However, someone with a working knowledge of the system should be able to have a good stab at chunking, once they understand the principles of the re-engineering approach.

4.7 Scheduling the sub-projects with the overall project

One management task is planning how the schedule for the chunked sub-projects will fit into the overall project. The simplest schedule has each chunk completely redeveloped and implemented before the next (combined) chunk is started (shown in *Figure MA1–9*).

Figure MA1–9
A simple sequential pattern for the overall structure

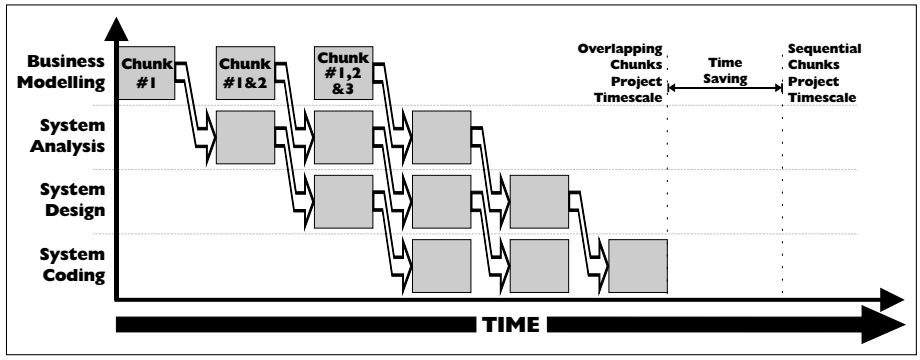


If there are tight time constraints on the overall project, then this is probably not the best schedule. In this situation, it is sensible to overlap the chunked sub-projects. One solution is to overlap them within a life-cycle stage (shown in *Figure*

MA1–10). When the business modelling stage is complete for the first chunk, the systems analysis stage is started. At the same time, business modelling starts on the second chunk, including the compacted business model from the first chunk.

The perceived benefit of overlapping the sub-projects is that the overall project takes less time than if the sub-projects were to follow one after another in sequence. However, when the sub-projects are overlapped, the experience from implementing one sub-project no longer feeds back into the business modelling of the next. Project managers need to weigh up the relative benefits of sequencing or overlapping the sub-projects for their particular overall project.

Figure MA1–10 An overlapping pattern for the overall structure



4.8 How to order the individual chunk sub-projects

It is important to consider the order of the individual chunk sub-projects as well as their overall structure. To some extent, this is dictated by the demands of the business. If a particular chunk contains new functionality that is critical to the business, naturally it is given a high priority.

However, the dependencies between business patterns also dictate, to some extent, the order of re-engineering for chunks and entity formats within chunks. For instance, transaction entity formats should, in general, be re-engineered after 'static data' entity formats. This is because the business patterns in transactions tend to depend on the patterns in 'static data'.

We shall see an example of this in the re-engineering of an accounting transaction in *MA2—Using Business Objects to Re-engineer the Business* . Its patterns depend on the 'static data' person and asset patterns, but not vice versa. In a 'real' re-engineering, it would make sense to re-engineer this static data before the accounting transaction.

When planning the order of the individual sub-projects (and the order of the entity formats within the sub-project), it makes sense to take account of the dependencies between the business patterns and to plan to re-engineer the dependent patterns after the patterns they depend on.

These dependencies between patterns are not just a feature of object systems. They are reasonably well-known in larger traditional systems. I have come across a number of package systems that explains the dependencies between data in their start-up documentation, saying, for instance, that company data depends on the correct country data being available. These dependencies are then used to suggest a schedule for setting up data in the system.

4.9 Ephemeral documentation

Because this approach treats all chunks, except the final one, as prototypes for the next chunk, this raises a tricky issue for system documentation. Unlike traditional approaches, the re-engineering approach expects the early (prototype) chunks to change significantly as their patterns are generalised. This means that the documentation for these chunks is ephemeral, going out-of-date when the next chunk is re-engineered. So, producing full documentation for each chunk seems like a costly waste of time. But this has to be set against the problems of running the implemented chunks in a live system without all the documentation. And the problem applies to all the types of documentation; both business model and system.

It is sensible when planning the project to specify the types of ephemeral documentation that will be produced during the re-engineering of the prototype chunks, balancing the cost of producing it against the benefit it brings. Also, the

plan for the final chunk needs to contain the task of producing the full set of documentation. Then, everyone can be clear about what documentation has to be produced when.

When assessing whether particular types of ephemeral documentation should be produced, we need to consider its uses, both in live operation and the system building process. It makes sense to produce documentation that is key to either of these. For example, it could be argued that producing tidied up versions of the object schemas is not particularly important to the live operation of the system. However, I have found that trying to produce presentable versions often brings to the surface useful insights, improving the quality of the business model. For this reason, I usually suggest that they are produced.

5 Produce a validated understanding of the business

Industry studies seem to show that a large number of the errors found when systems are implemented are because of misunderstandings about what the business required rather than errors in coding. These types of errors are, for the most part, avoidable in a re-engineering project. The re-engineering should produce a business model that reflects the business patterns accurately. And, in theory at least, these should not need to be changed during the system building process. Nor should they lead to errors in the implemented system.

However, I have found that object schemas on their own do not provide a complete enough check on the accuracy of the patterns in the business model. They are a potent tool for making visible the patterns we use to understand the world. They take advantage of the human brain's ability to spot any out-of-place shapes in the patterns. But, despite all this, they do not provide as complete a check on the accuracy of the patterns as required. I find that I need to build a validation system to give myself a reasonable confidence that the reflections are accurate.

5.1 Building a validation system

The validation system is the business model translated into a database and populated with a representative sample of operational objects. I use the existing system as my primary source for the operational objects, migrating its data onto the validation system. For small files, I migrate all the operational data; but, with the larger files, I usually only migrate a representative sample. Where possible, I migrate the data automatically. I also load up any new operational 'data' found during the analysis of conceptual patterns (an example would be England and the other nested countries in *MW1—Re-Engineering Country*).

The validation system does not require sophisticated technology and should not involve much effort. It can be built within a CASE tool (if one is being used) or constructed on a simple computer database. (I have found that non-object-oriented PC databases are a cheap and effective solution.)

I normally construct the validation system as I am doing the modelling, translating and migrating the data from the existing system as I re-engineer its entity formats. This usually brings up issues, which I can resolve there and then. When sufficient data has been migrated, I produce reports and enquiries from the validation system. This enables me to touch and feel each bit of the model as it grows; there is no real substitute for this. I can often see immediately whether something works and change it if it does not. In addition, most users have found these reports and enquiries a more accessible way of checking the model than object schemas.

The key benefit from constructing the validation system is that inaccurate reflections of business patterns are found and fixed before any time has been spent building them into the system. This helps avoid the frustrating experience common in traditional system building, that is, finding resources have been wasted building inaccurate business patterns. Using a validation system significantly reduces the level of these errors, minimising the wasted resources.

6 Object model the migration of business patterns

A re-engineering project, by its nature, involves the migration of business patterns from the existing system to the new object system. These will be application level patterns, such as countries, and operational level patterns, such as United States. If the business paradigm embedded in the final system does not accurately reflect the business model (and so the real world), then much of the model's power can be lost. One way of ensuring that this does not happen is having a sufficiently formal and accurate specification of how the business patterns are migrated. I do this by constructing an object model of the migration.

6.1 Tracing the migration of application level business patterns

The business model produced by the re-engineering process contains an accurate reflection of the business. It should be embedded in the final system. However, I have found that a common problem with the embedding is that some system analysts and designers treat the business model as a proposal rather than a formally defined input into the process. They assume that they can exercise their judgement to pick and choose what to embed and amend as they see fit.

This is, in my experience, a general problem for all business models—not just object models. But with an object model, it is a sure-fire way of losing the benefits of business object modelling. The object model is a tightly connected system. Fiddling with bits of it, particularly by someone who does not understand the business patterns, is almost certain to have a deleterious effect on the whole structure.

This is not to say that systems analysts and designers should be discouraged from finding inaccuracies in the business model—quite the opposite. But if they do find what they consider to be an inaccuracy, business modellers should check it. If it is a real inaccuracy, the more accurate pattern should be applied to the business model and it should work its way through normal channels to the systems analyst and not unilaterally applied to the system specification.

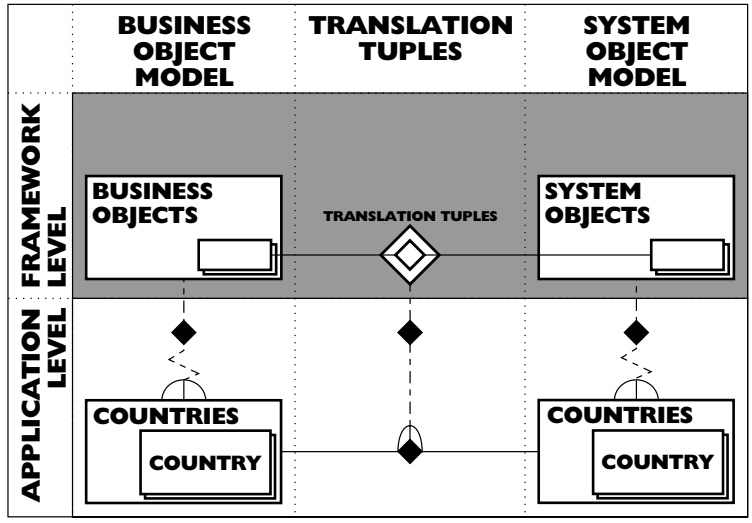
I have found that the simplest way to ensure that the business model's accurate patterns are embedded unchanged in the final system is to provide a system for confirming that this has been done. I model the business patterns' translation into the system model and onwards into the implemented system. This translation model provides traceability. We can trace the migration of the patterns from the business object model to the implemented system. Any unauthorised changes are brought to light. The formal nature of the translation model also means that the checking can be automated.

6.2 Modelling the migration of application level business patterns

The simplest way to model the migration of the application level business patterns is to extend the business object model to include the translation of the patterns into the implemented system. The first step is to extend the meta-model. Then as a second step, these two extensions are populated.

We need two extensions to the meta-model. The first is a system object model for the paradigm used by the implemented system. The second is a general translation tuple that has as members the tuples connecting the objects in the business model and the objects in the implemented system model. The result is illustrated in *Figure MA1–11*.

Figure MA1–11 Extended application level migration model



It is worth bearing in mind that the business object model is technology independent. This means, among other things, that it can be implemented on any technology. It can be implemented into an object database, a relational database or even simple flat files. It can be implemented in an object-oriented programming language, such as C++ or Smalltalk, or it can be implemented in a traditional language, such as COBOL. However, each of these implementations requires its own system meta-model in the migration model.

6.3 Modelling the migration of operational level business patterns

In a re-engineering project, we need to migrate the operational level business patterns as well as the application level patterns. I normally do this twice. I do this once during business modelling, when I populate the validation system with operational objects (described earlier); and then a second time at implementation, when I populate the implemented system with relevant (operational) static data (items such as currency and clients). These operational objects come, for the most part, from the existing system.

We normally re-engineer a few representative operational objects from the existing system to provide us with the basic patterns from which we generalise the

application level patterns. For example, in *MW1—Re-Engineering Country* we re-engineered the individual objects, the United States and the United Kingdom, and generalised them into the countries class. However, the validation and implemented systems need many more operational objects than the re-engineering.

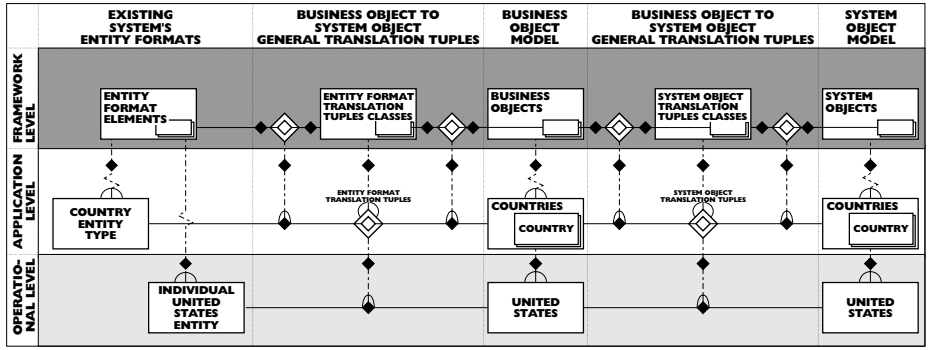
I have found that it helps me to manage the migration of this large number of operational objects, if I model its general patterns. I do not describe each individual operational object's migration pattern; instead, I use a representative sample to construct general 'application level' migration patterns. These then constitute a migration specification I can use for all the operational objects.

My first step is to extend the business object meta-model. I extended it for the target system (either the validation system or the implemented system), when I set up the 'system' for checking whether business patterns were properly embedded in the implemented system. (This was described in the previous section and illustrated in *Figure MA1–11*.)

So I now extend the meta-model to include the existing system, the prime source for operational objects. I also include a general translation tuple linking the existing system's entities to the business objects. *Figure MA1–12* provides an idea of what the extended meta-model would look like.

The second step is to model the migration of some representative operational objects and to discover the basic patterns for application level migration. These are migration 'rules'. I use them to specify how the operational entities from the existing system can be correctly embedded in the target system. I sometimes use them as a model for an automated migration process. Either way, they greatly simplify the migration of data from the existing system to the new system.

Figure MA1–12
Extended operational level migration model



7 Summary

This paper focused on how you can use business object ontology to help you re-engineer your legacy entity oriented systems. It stresses the importance of embedding them properly in the final system, if you want to harvest these benefits. The next *MA—The BORO Re-Engineering Methodology: Applications* paper (*MA2—Using Business Objects to Re-engineer the Business*) focuses on a different topic, using business ontology objects to re-engineer the business.

BORO Working Papers - Bibliography

The BORO Working Papers

Volume A
A—The BORO Approach

Book AS
AS—The BORO Approach: Strategy

AS1—*An Overview of the Strategy*
AS2—*Using Objects to Reflect the Business Accurately*
AS3—*What and How we Re-engineer*
AS4—*Focusing on the Things in the Business*

Volume - O
O—ONTOLOGY Papers

Book - OP
OP—Ontology: Paradigms

OP1—*Entity Ontology Paradigm*
OP2—*Substance Ontology Paradigm*
OP3—*Logical Ontology Paradigm*
OP4—*Business Object Ontology Paradigm*

Volume - B
B—Business Ontology

Book - BO
BO—Business Ontology: Overview

BO1—*Business Ontology - Some Core Concepts*

Book - BG
BG—Business Ontology: Graphical Notation
Constructing Signs for Business Objects

Graphical Notation I
BG1— *Constructing Signs for Business Objects*
Graphical Notation II
BG2— *Constructing Signs for Business Objects' Patterns*

Volume - M
M—The BORO Re-Engineering Methodology

Book - MO
MO—The BORO Re-Engineering Methodology: Overview

MO1—*The BORO Approach to Re-Engineering Ontologies*

Book - MW
MW—The BORO Methodology: Worked Examples

Worked Example 1
MW1—*Re-Engineering Country*
Worked Example 2
MW2—*Re-Engineering Region*
Worked Example 3
MW3— *Re-Engineering Bank Address*
Worked Example 4
MW4—*Re-Engineering Time*

Book - MA
MA—The BORO Re-Engineering Methodology: Applications

MA1—*Starting a Re-Engineering Project*
MA2—*Using Business Objects to Re-engineer the Business*

Book - MC
MC—The BORO Re-Engineering Methodology: Case Histories

Case History 1
MC1—*What is Pump Facility PF101?*

INDEX

METHODOLOGY: APPLICATION - 1

STARTING A RE-ENGINEERING PROJECT

A-F

A

B

C

D

E

F

G

H

I

L

M

N

R

S

V

W