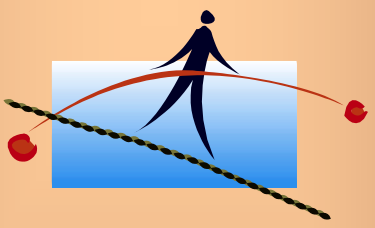
The Pattern Organization



Designed for change

Max Stewart

Max Stewart

The Pattern Organization

Designed for change

Published by Decomplexity Associates Ltd
First published 2004

Set in Times New Roman

US Acrobat edition - **ISBN 0-9540062-8-3**

also available in European Acrobat edition - **ISBN 0-9540062-7-5**
and
European printed edition – bound with colour illustrations – **ISBN 0-9540062-6-7**

This book complements the author's *The Coevolving Organization – poised between order and chaos* which is available through booksellers:
ISBN 0-9540062-0-8 (European edition – bound with full-colour plates)

or copyable free from www.decomplexity.com with different line illustrations and without colour plates:
ISBN 0-9540062-1-6 (European Acrobat edition)
ISBN 0-9540062-2-4 (US Acrobat edition)

and

The Robust Organization – highly optimized tolerance which is available through booksellers:
ISBN 0-9540062-3-2 (European edition – bound with colour illustrations)

or copyable free from www.decomplexity.com with black-and-white line illustrations:
ISBN 0-9540062-4-0 (European Acrobat edition)
ISBN 0-9540062-5-9 (US Acrobat edition)

AUTHOR

Max Stewart was educated at the Universities of Wales and Cambridge. He wrote the first and widely praised non-specialist account of the application of relational database principles to systems design – something that later became better known as Data Analysis. He was at one time IT Director for the Scottish operations of Leyland Vehicles and later spent many years with Mars, Incorporated. He is a Principal with Decomplexity Associates and lives in Rutland, England's smallest county.

COPYRIGHT AND TRADEMARKS

The smith also sitting by the anvil, and considering the iron work, the vapour of the fire wasteth his flesh, and he fighteth with the heat of the furnace: the noise of the hammer and the anvil is ever in his ears, and his eyes look still upon the pattern of the thing that he maketh; he setteth his mind to finish his work, and watcheth to polish it perfectly.

Wisdom of Jesus Son of Sirach 38 v28
King James version (Apocrypha)

CONTENTS

Preface

Acknowledgements

Chapter 1 – Introduction

Chapter 2 – Patterns

Chapter 3 – Decomposition patterns

Chapter 4 – Organization and business processes

Chapter 5 – Buffering

Chapter 6 – Buffer placement

Chapter 7 – From IT to organization

Chapter 8 – Reference material

Chapter 9 – Questions and answers

Bibliography

Index

Figures

Figure 1 - scope of simulation 19
Figure 2 - class and object diagrams 32
Figure 3 - Adaptor pattern diagram 33
Figure 4 - Adaptor pattern OMT 34
Figure 5 - Facade pattern diagram 35
Figure 6 - Facade pattern OMT 36
Figure 7 - Mediator pattern diagram 37
Figure 8 - Mediator pattern OMT 38
Figure 9 - Chain of responsibility diagram 39
Figure 10 - Chain of responsibility OMT 41
Figure 11 - Bridge pattern 'before' diagram 43
Figure 12 - Bridge pattern 'after' diagram 43
Figure 13 - Bridge pattern OMT 44
Figure 14 - HOT with a. equal and b. centred probability of sparks 48
Figure 15 - Trees and semi-lattices 63
Figure 16 - Military commands form a language 64

PREFACE

This book is the third of a series by the present author on business organization. The first of the three – *The Coevolving Organization* – was published in 2001. It tried to answer one fundamental business question – how decentralized should an organization be? – using developments in physics and theoretical biology which emerged during 1988-1995. It described how businesses could be positioned, poised and reactive, on the boundary between stability and anarchy, using the concepts of 'edge of chaos' (EOC) and 'self-organized criticality' (SOC), and tried to show what benefits might accrue from attaining this nirvana. The question of whether the edge of chaos was the optimal point *under all conditions* to which to decentralize was left unresolved. If, in particular, instead of relying on a random self-organization process to manage decentralization, we actively *designed* the organization, could the optimal point be shifted even more in the direction of decentralization without compromising the stability of the organization? In the late 1990s, the complete answer was simply not known.

But between 1998 and 2003, something new and related was discovered and then explored: *highly optimized tolerance* (HOT). HOT does not supersede EOC and SOC. Instead, it allows us to exploit the idea of decoupling parts of an organization (divisions, departments, even individuals) such that the decoupled parts can be even more responsive than with EOC/SOC. More significantly, HOT also highlights the role of deliberate *design* – the antithesis of self-organization. Self-organization or, alternatively, restructuring using a simple and limited amount of management intervention, can be attempted following the EOC/SOC principles outlined in *The Coevolving Organization*. But if a business is decoupled further using HOT principles, it is possible for the decoupled parts to be even more responsive than would be possible with the EOC/SOC ideas alone. It implies minimizing how the decoupled parts can affect one another and having a good understanding of the likely business risks to which each part is subject.

The first two books thus described how to position an organization at an optimal level of decentralization and what could be gained from doing so. But to those needing to implement the restructuring of a business, this may have sounded like airy-fairy nonsense. How could any fanciful theory take into account real business processes, for example?

This next book fills the gap. The processes of a business and its organization staff structure are, or should be, very closely related. Some businesses even rightly pride themselves on having transformed their organization structure into one which is closely in line with their business process structure. Their organization charts and business process charts look very similar. But business processes themselves will change. Some will evolve smoothly in a planned way as supply, manufacture and distribution evolve. Others will be forced to change rapidly in response to competitors' threats. Amending business processes in a hurry can be perilous, particularly if the business is accustomed to gradual change. If we want to build an organization which is decentralized to some optimal point arrived at via edge of chaos and highly optimized tolerance considerations,

- how do we put together the new organization from the bottom up so that the organization and business processes are aligned?
- how do we ensure that, when business processes themselves change, the organization and IT systems are not left flailing around and unable to keep up? The aim of applying EOC and HOT concepts to organizations was to engender responsiveness without instability. How, therefore, can we ensure that when business processes are changed, the various parts of an organization continue to work and communicate with each other effectively?

These questions inevitably raise a further one: when building from 'bottom up', how far down is 'bottom'? In other words, to what level of granularity do we descend in order to have the foundation on which to build upwards: individuals, teams, small departments, business processes or what? The organizational foundation on which the material which follows is constructed is roughly the size of a small team. One characteristic of such a team is that it is responsible for running a single discrete business process; further decomposition of this process and its supporting organization into smaller semi-independent pieces would be pointless since each such smaller sub-team would not be able to make decisions without reference to the others.

The final book in this series, *The Emergent Organization*, will cover *true* bottom-up construction – the evolution of an organization from rudimentary business process fragments. It will describe how to grow an organization from seed using a selection of elementary business-process building blocks. The growth of each process must take account of its future neighbours; it must not merely evolve to meet its own selfish ends. The processes and their supporting teams also need to 'grow towards the light': some long-term business policy or statement of ethics like the Five Principles of Mars plus some intermediate goals such as Balanced Scorecard objectives. In other words, we want to create a living business organization from scratch, or following the dismemberment of its failing predecessor, using long-term policies as attractors (desirable patterns). This emergent organization must then continue to evolve of its own accord. Since business policy can specify the degree to which decision making should be decentralized and the degree to which different parts of the organization compete with each other or otherwise, these attractors can mould a coevolving organization.

As with the previous two books, the background material is not readily accessible to most managers. But unlike the previous two, the present book draws on ideas from architecture and from object-oriented IT system design rather than from theoretical physics and evolutionary biology. The first detailed exposition of the usefulness and ubiquity of patterns was made by practising architect and mathematician Chris Alexander in the 1960s. His ideas were later picked up by IT program designers who were seeking ways to design reusable chunks of programming so that subsequent changes did not necessitate wholesale redesign or inelegant fudges.

Max Stewart
Rutland, UK - October 2004

ACKNOWLEDGEMENTS

The present book would not exist without the pioneering work of architect Christopher Alexander and of the 'object-orientated Gang of Four' – Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides – who recognised the value of Alexander's ideas to revolutionise IT systems design. Their ideas have been used, built upon and are freely acknowledged in the text. Apart from brief extracts for comment, no copyright material has been reproduced but the names and diagrams of the various buffer patterns have been made consistent with those of the Gang of Four's definitive work "Design Patterns - Elements of Reusable Object-Oriented Software" (Addison-Wesley 1994) in order to help IT practitioners already familiar with this latter book.

The prefatory extract from the Authorized Version of the Bible (The King James Bible), the rights in which are vested in the Crown, is reproduced by permission of the Crown's Patentee, Cambridge University Press.

CHAPTER 1

INTRODUCTION

The thread underlying all four books in this series is *flexibility*. The first two demonstrated how to split an organization into discrete parts – which could in principle even be down to the level of individual people – such that *decisions* could be made and implemented fast. This third book tackles the problem of putting an organization together such that organization *structure* can change quickly and without loss of effectiveness. In other words, until now we have been trying to identify exactly where and how an organization can be split such that the resulting pieces ('objects') are as autonomous as possible consistent with the overall stability of the business. Three issues were left outstanding:

- how can organizational objects be insulated from each other such that internal changes in one have minimal effect on any of the others
- hitherto, the connections between objects have been considered at a superficial level as links (C-couplings) with varying strengths. But what happens when several links conspire to work together?
- how can we catalyze an organization to evolve by growing small fragments of business processes in such a way that the growth upwards and sideways is guided by business policy

What follows addresses the first two issues, the first in particular. Its aim is to improve our ability to change an organization easily and quickly in response to external stimuli or internal decisions. Hitherto, we have used a 'language' based on Stu Kauffman's NKCS landscape modelling ideas in order to describe the dynamic behaviour of coevolving organizational objects. We also need a language – a different one – in order to describe the *building* of organizational objects which ideally can behave as autonomously as possible. This will be a different language: a combination of architect Chris Alexander's *Pattern Language* to provide the definition of a business object (or collection of linked objects) plus the object-orientated design concepts of *classes* to describe the internal structure and behaviour of each pattern.

In 1964, Alexander first described how abstract 'things' interact, and how misfits between these 'things' and their environment can be minimized. Alexander's work spawned considerable interest from other areas, notably object-orientated software design. He introduced the idea of 'patterns' which can be used at a local (decentralized) level to create structures, which in our case contain the internal processes (not necessarily the formal business processes) of organization units each of which has the most appropriate fit for its purpose.

With the discovery of highly optimized tolerance (HOT) in 1998 onwards (see *The Robust Organization*), it became clear that the placing of barriers between business areas, or more precisely designing where to buffer one business process from another, could be undertaken in a much more precise way. Alexander's aim was

to minimize the knock-on effect of a change. HOT showed how to use information on the likelihood (i.e. probability) of a possible change in order to place buffers around those areas where this was most likely to happen. The best analogy is the placement of firebreaks in a forest, where areas near campsites for example are closely ringed with firebreaks because of the greater probability of sparks occurring.

IT system designers have a similar challenge: to design systems such that subsequent changes do not introduce unwanted side-effects. One way to do this is to attempt to identify those parts of systems which are most likely to change. These are usually the programming nuts and bolts used in its construction rather than the higher level design (the architecture) which is typically more stable. Such areas vulnerable to change are buffered – hidden within black boxes ('encapsulated') – as far as possible.

The aim of this book is to pull together apparently unrelated concepts from architecture and object-orientated IT systems design such as:

- decentralization and decomposition
- buffering
- encapsulation
- barriers

in order to show where business processes (and their attendant staff) should be buffered (cushioned) from one another. The way in which business processes are linked – and in particular any buffering between them – will be defined by *design patterns* and elaborated as linked *classes*, linked *objects* or a mixture.

CHAPTER 2

PATTERNS

From templates to patterns

There are many types of template loosely called patterns. The familiar knitting pattern is a list of detailed instructions on how, for example, I can knit myself a sweater. It is more than just a generic set of instructions covering all sweaters: the pattern will be for males of a given chest size and will specify a particular wool thickness. I can probably choose the wool colour, but even this might be prescribed if the sweater is to be multicoloured. This knitting pattern is not generic in any way: it does not describe how to construct sweaters in general, merely ones for men of a particular shape. This *construction* pattern is *not* the type of pattern we are looking for.

The person who creates the patterns will, on the other hand, have some more general *design* pattern for sweaters of a particular type: 'heavy winter sweater with frontal cable-work and crew neck', for example. This design pattern is then used as a template to create construction patterns for knitting male and female sweaters of a set range of sizes. This design pattern is getting closer to the type of pattern we seek: it can be applied to generate many solutions – many knitting patterns – which have some readily identifiable things in common (shape; motifs and so on) and are appropriate for a particular context (cold weather). And the phrase 'heavy winter sweater with crew neck' may well be used as a convenient shorthand description between experts who create knitting patterns. Furthermore, our designer might have an even more generic pattern – a 'crew-neck sweater' pattern for example – to call upon which was used as a base to develop the design pattern for heavy winter crew-neck sweaters. The latter design pattern will inherit many of the characteristics of the 'crew-neck sweater' pattern but with variations to make it suitable for winter use. The 'crew-neck sweater' pattern may conceivably have an even more generic predecessor – 'sweater' pattern from which it inherits some basic shape. This is getting closer.

Engineering and construction inevitably have many concepts which we might recognise easily as some form of pattern. The simple arch bridge, the suspension bridge and the box girder bridge all have the same aim: to cross a gap. But the engineering principles upon which each works are different. Each represents a form of design pattern from which a construction pattern – the detailed design and construction details for a particular bridge – can be derived. But, unlike the various forms of sweater, they do not inherit a common ancestry even though they fulfil the same purpose. If we wanted to cross a gap with some form of bridge, we would, perhaps, first examine alternative bridge types. A catalogue of alternative bridge patterns – suspension bridge, cantilever bridge and so on – would be useful, particularly if each type were well proven and the circumstances (the *context*) under which it was most appropriate (long single span; high winds;...) were documented. Let us elaborate this pattern for a suspension bridge in a slightly more formal way as follows:

Name: "Suspension bridge"

Problem: Need for a road or rail crossing over a gap in the terrain

Context: Appropriate for gaps of between 500 and 2500 metres when there are substantial rock abutments at each end in which to anchor the cables

Success criteria: Elegance of design is important. Cost matters but is not an overriding factor.

Solution: A flat or slightly arched deck (set of carriageways) suspended longitudinally at regular intervals by cables made of twisted steel wire which are attached vertically to other similar but much stronger cables which fall in an inverted arch (catenary) either side of the deck. These chains pass over tall towers near each end of the bridge and are then firmly anchored in the rock abutments or in massive concrete blocks. Because suspension bridges are light and flexible, they are vulnerable to strong winds. The towers may need additional pendulum-like devices to stop them swaying, and the deck may need stabilizing fins

Rationale: Stranded steel wires are, for their weight, very strong in tension (i.e. when pulled).

This simplistic example is sufficient for me as an engineer to decide, provisionally at least, whether a suspension bridge – as opposed to other types of bridge – is likely to solve my problem. The keywords used: *problem*, *context*, *success criteria* and so on help to give some structure to the pattern definition so that we can compare this pattern with ones for other types of bridge. They summarise in a consistent way:

- the **problem** – for which we need a solution
- the **context** or environment with which any acceptable solution needs to contend – type of anchoring available at each end and so on. The context is black-and-white in the sense that the solution *has* to work within it (a bridge which spans *most* of a gap is not a solution)
- the **success criteria** (Alexander's *forces*) which must be satisfied if the solution is to be regarded as successful (or, following Alexander, what 'forces need resolving'). Success criteria are often shades of grey in the sense that the greater the degree to which they are met, the better is the solution. Success criteria may conflict; when 'low cost' is a criterion, it will, for example, conflict with others which imply high-quality materials or individually designed components
- the **solution** (Alexander's *configuration*)

- any **rationale** (optional: what makes this solution particularly appropriate)

It is worth quoting Alexander's definition of a pattern in its architectural context:

"…[a] rule which establishes a relationship between a context, a system of forces which arises in that context, and a configuration which allows these forces to resolve themselves in that context"

Outside architecture, and occasionally within architecture, it can sometimes be unclear where 'context' stops and 'forces' start. For example, in the suspension bridge example above, the context is a geographical one of gap size and rock abutments. But if the bridge is to be regarded as a success, it will also look elegant and not be too expensive.

The pattern format gives us a language to describe almost any generic design.

'Suspension bridge' is, to bridge builders, a very basic and high-level concept. An engineer would hardly need to refer to a book of bridge-type patterns of this simple type. But at a lower level, where designs become more detailed, the number of such concepts becomes very large.

Such a definition looks like formalization for formalization's sake – like over-complexing something which is actually simple. This is not true although the significance and power of patterns will not become apparent until we examine some more difficult design problems.

Outside engineering, there are two areas where the introduction of patterns has had a profound effect:

- the architecture of buildings and their surroundings
- IT system and program design

Chris Alexander laid the foundations for both. Engineers and IT people cottoned on to the elegance and ubiquity of his ideas quicker than the majority of architects. Or perhaps architects, particularly those who promoted the brutally sharp rectilinear shapes in grey concrete popular in the '60s and '70s, saw only too well that Alexander's analysis had sounded a death knell for their pet schemes. To see the true significance of what looks superficially like a trite concept, we will home in on the concept of patterns from three somewhat different directions:

- Alexander's first widely-published foray into this area (his *Notes*)
- Alexander's *Pattern Language*
- The Gang of Four's object-orientated system design

The simplistic example of a suspension bridge pattern may give the impression that a pattern is merely a description of an 'object which solves a problem' – like a pill taken for a headache. Apart from being a proven solution to a problem, *a pattern describes both objects and relationships between objects* – in other words *structures*.

This will become clearer when the buffer patterns are described, and is illustrated graphically in the Bridge pattern class diagram on page 44.

CHAPTER 3

DECOMPOSITION PATTERNS

"Notes on the synthesis of form"

Chris Alexander published a summary of his PhD thesis in book form with this arcane title in 1964. He later published two series of books on architecture which have been widely read and very influential. The first was on the definition and use of patterns to design rooms, buildings and spaces which were 'alive' – places which inhabitants enjoyed rather than tolerated. The second series, of which one book remains (as at October 2004) to be published, proposed the far more fundamental concept that architectural forms which were 'alive' could be created by repeating simple growth operations – 'structure-preserving transformations' – on fifteen basic geometrical properties.

Perhaps because of its title or analytical content, his *Notes* took some time to be appreciated for what it represented: an entirely new approach to designing buildings and collections of buildings. Why was it, for example, that buildings designed in the conventional way by groups of engineers specialist in particular disciplines were either dysfunctional – they failed to do what they were designed to do – or did not fit their external environment, or both.

Alexander started by trying to define 'design'. He suggested that every design problem was an attempt to make whatever we wish to design – the *form* – a good fit into its surrounding environment – the *context*. This context includes any mandatory requirements from the architect's design brief such as 'south-west facing' or 'single storey'. The form thus represents a solution to the design problem and the context is the problem itself. Design therefore is a process of analysing an *ensemble* - the combination of form and context – and trying to identify how well or badly the form was aligned with each part of its context.

He gives a simple example (which is one of construction rather than design): the machining of a flat piece of metal so that it is smooth and level. After some preliminary grinding, the piece is placed on a guaranteed-flat reference sheet of metal which has been covered in ink. Any high points on the piece being machined will appear as traces of ink. These traces are ground down and the process repeated until there are no high points indicated. The ensemble is the piece being machined (the form) plus the inked reference sheet (the context). The ink traces graphically represent the misfits (in this case high points) between form and context and, in this example, there is only one division between form and context: the two metal surfaces being compared.

Take now a slightly more complex example. If we wish to design 'something to heat small quantities of water quickly', the context is everything a kettle or pan designer needs to worry about: it must be safe to hold when hot, electrically safe (if powered by electricity), spill- and leak-proof, must raise water to boiling point acceptably quickly and so on. If the resulting form, a kettle for example, meets each

of these criteria well, it is a good solution to the problem. This example is more complex than the first in two significant ways:

- there are several different types of potential misfit (degree of electrical safety; speed of boiling; …)

- there may also be more than one division between form and context. For example, if the challenge is to design something which heats small quantities of water, we may focus attention on the source of heat or power – the stove or electricity supply. In this case, a kettle becomes part of the context and the stove or electricity supply is the form.

This second point is subtle but very significant for our purpose.

Many ways to split form and context within one ensemble

Assume that the outer limit of our 'design space' – the area within which our attention is focused and outside which we can ignore everything – is a house. Within the house, there are many ways to split the form and context. For example, when ignited, gas (the solution, i.e. the *form*) supplied to the kitchen (part of the *context*) is an efficient, relatively safe and cheap way to provide heat to water (another part of the context). The context is everything surrounding the gas flame: a pan or kettle, the water within it, the stove, the air supply to the kitchen (needed to keep the flame alight) and so on. The context also implicitly defines the criteria we will use to see how well the gas flame heats water. Part of the context is 'safety', so how safe is a gas flame and gas itself? Another part of the context is 'efficiency', so how efficiently does a gas flame transfer heat to whatever it is heating?

Since there are many ways to split form and context, this suggests two obvious questions:

- are contexts hierarchical like Russian dolls? Since kitchens are part of houses, gas flames (for cooking) part of kitchen stoves, pans are used on stoves, and water to be heated is contained in pans, do we have a hierarchy of forms and contexts in which one context (a house) contains many forms (rooms which need designing to fit the house in some best way). Another 'smaller' context – a kitchen – contains the usual kitchen facilities, one of which is a stove, which must be designed to serve the kitchen optimally in some sense. And so on, down to the smallest individual utensil.

- whether there is one best way to split any ensemble into form and context?

and one less obvious one: are these two questions contradictory?

It is worth emphasizing one point which was not mentioned explicitly in Alexander's *Notes* but is a fundamental feature of his design patterns: the criteria which, if met, make a solution (a form) a good solution should be separate from the context. The latter is pre-ordained and cannot be modified. The misfits (above) are the forces

which need to be resolved in order that the solution is a good solution; we have called resolution of these misfits *success criteria,* i.e. a success criterion is the fixing of a particular misfit. As Alexander points out repeatedly, misfits are more obvious – they stand out far more – than successes. Context is black and white and non-negotiable. The success criteria are shades of grey and may mutually conflict, in which case not all of them can be satisfied adequately.

Size versus complexity

A relatively simple task, like machining a piece of metal so that it is smooth enough when measured against a truly flat reference sheet of metal, may take a long time if the piece being smoothed is large, but we have only one criterion of fitness: is the piece smooth?

A craftsman who operates a grinding machine and smoothes metal bars for a living has a simple job in the sense that there are no compromises to be made between different fitness criteria. His job is a skilled one, certainly. But grinding large numbers of metal pieces – or grinding a few large pieces – is a straightforward job. Smoothing does not have adverse repercussions on some other possible fitness criterion such as durability (the heat generated during grinding might, perhaps, lower the resistance of the surface to wear and tear) because we have only included one such criterion – smoothness. The number or size of pieces machined makes no difference. So the size of a design problem does not in itself create complexity.

The problem of designing 'something to heat small quantities of water quickly' is different. Here we have several fitness criteria to manage at the same time. If each criterion were totally independent of all the others, the designer's task is still simple; it may take considerable time to find a design which satisfies all fitness criteria – is it safe to hold when hot, leak proof, and so on, but if these criteria do not affect one another, the design process is easy to manage. The designer merely designs for each fitness criterion separately and then tests for how well that criterion is met. But the designer's job suddenly becomes complex *when the criteria are not independent of one another*.

One criterion which is almost invariably not independent of others is cost. A kettle or pan which must conduct heat quickly from the gas burner or hotplate of a stove to the water inside needs a base which is a good conductor of heat, which is one reason why pans for serious cooks have copper bases. But copper is more expensive than steel or cast iron, for example. So when designing a pan, the designer cannot design for each fitness criterion independently. A pan with a Grade-A copper base is expensive, and if maintaining an even temperature were important, a thick copper base would be used. But the thicker the base, the heavier it is and the more heat it will retain after use; it becomes more difficult to wield and a burn caused by accidental contact becomes more likely. The designer lives in a world of compromise.

Readers of *The Coevolving Organization* may now be recognizing a common thread. 'The different faces of K' in Chapter 5 of *The Coevolving Organization* described what it might be like to attempt to reach a peak of high fitness by adjusting gene values (cf our different design criteria). Where genes are independent, the landscape to be climbed was a simple one which gradually sloped upwards to a high peak – like Mount Fuji. Any improvement in fitness caused by adjusting one gene

value was always good: it never had the side-effect of adversely affecting the fitness caused by the settings of other genes. When genes were linked to one another – were not independent – climbing became a far more difficult task. The landscape over which we were climbing was no longer a simple smooth path to a high summit. It was instead a rugged landscape with lots of small hills with steep sides. It became all too easy to become marooned on the peak of one of the smaller hills which represent relatively low fitness, in our case a fairly expensive pan with moderate heat conduction and only averagely safe.

The number and strengths of links between design criteria is what in *The Coevolving Organization* we called K-complexity. Where design criteria – the *forces* – are mostly independent ('attractive colour' and 'efficient heat conduction' are probably independent, for example) the landscape is 'low K'. When the design criteria have lots of interdependencies (copper bases conduct heat quickly but bump the cost up, for example), the landscape is 'high K'.

Complexity and decomposition

When these cross-connections occur, with the resulting compromises and complexity, design becomes far more difficult. In his '*Notes*', Alexander proposed that this was why so many buildings in the developed world are dysfunctional. He contrasted the way in which houses were traditionally constructed in undeveloped countries with the way they are constructed in developed countries.

The simple hut in the undeveloped country was usually built – hardly designed – by one or two individuals. These builders were not taught house-building in any formal way; instead they absorbed ideas by watching others. And when their hut was under construction, passers-by would suggest better ways to do things. In other words, there was no guidebook, no specific general rules to be learned, and no formal tuition. But the resulting huts were simple and rarely changed in basic structure from one generation to another. They fitted into the local environment well.

The house or office block in the developed world is designed and built very differently. They are multifaceted (see page 51) and, as we shall see, almost always complex, even the highly-standardized houses built on large housing estates by a single developer. Architects and engineers are *trained*. This training is essentially the absorbing of a large number of general concepts of 'good design' plus a bit of theory. Inevitably, some of these concepts clash with others – and most of them usually clash with 'lowest cost'! When an architect is commissioned to design a house, he or she applies these general principles to the design brief, the local topography, any prescribed orientation of the house, local services (whether a foul drainage main is available to remove sewage, for example) and so on. What the architect, unlike the builder of the simple hut, is unable to do is to copy a design which has been proved successful by centuries of use in the same locality. He or she will, if necessary, modify the site where possible to suit the brief: wet clay soil? just cut down nearby trees (which absorb water in dry summers but not winters) and build deeper foundations to avoid subsidence or heave; windy exposed site? create an artificial earth bank and provide additional heating on north-facing rooms; and so on. These would, individually, not necessarily lead to dysfunctional houses. To see where

dysfunction arises, we need to look more deeply at how changes occur in the structure of the simple hut and of the modern house.

Changes to the structure of the simple hut occur gradually, and when they occur they are rarely radical changes. There are two counteracting forces at work: the builder usually lives in the hut he built. If there is something wrong – perhaps there is not enough ventilation in an unusually hot summer, he may well poke another hole in the wall or expand an existing window hole. But almost certainly he will not radically redesign the hut to improve ventilation; local tradition dictates certain hut shapes which must be adhered to. He may, in fact, be completely unable to design a hut from scratch if he were uprooted into a very different environment. His hut may not last long, a few seasons perhaps or much less if his is a peripatetic lifestyle following herds or flocks to new pasture. So he has regular experience of building new huts and making minor changes to existing ones. He may even never have reason to want to change the design of his hut: years of fine-tuning of the design by himself and his colleagues and predecessors have removed any real need for redesign if the local topography and weather remain roughly the same.

These two features:

- immediate response to fix problems
- the weight of tradition which prevents radical changes

together make the house structure adapt easily to changes in requirement (such as additional ventilation needed to cope with the unusually hot summer) without creating other problems: an additional ventilation hole is unlikely to cause side-effects such as structural instability. Each problem – a 'misfit' in Alexander's terms – can be fixed independently. In this simple hut, each misfit is independent of others and can be fixed independently of others. This can be inferred from the fact that the construction details are relatively unchanging. It implies that if, back in the mists of time, the various details of construction were linked such that minor changes to one (wall strength, for example) had a knock-on effect on others (coolness in summer, for example), these interdependencies had been gradually severed over the passage of generations. If not, each house would be different and
there would not be any uniformity in construction. In other words, the standard construction and its unchanging nature are evidence that the construction has reached equilibrium: there is no longer any need to make significant changes. This unchanging nature is evidence that various details of construction are independent of one another. If not, minor changes would for ever be upsetting other parts of the construction (our ventilation hole could weaken the wall; weakening the wall then might have the knock-on effect of making the structure sway in the wind; the swaying in the wind then might have a further knock-on effect of making the structure skew around the centre which further weakens the wall; and so on). Any structure – or indeed any system – in which different parts can *not* receive minor modifications without upsetting others, is for our purposes *complex*.

The architect designing a modern house has a fundamentally different problem. His or her requirements brief, in the context of the site on which the house is to be

built, is full of potential conflicts. Any errors in design may be found out too late and by the eventual occupiers and not by the architect. The error may have been repeated many times on a large housing estate. Architectural briefs insist on change for fashion's sake – what will sell rather than what best fits the locale. These conflicts make the construction complex in the sense defined above, where fixing one part of the design brief such as 'make the house cool in summer' (perhaps by providing air conditioning because the site is an exposed one) conflicts with the requirement for the house to have 'low running costs' and the need for 'quiet' (air conditioning in small houses can be noisy). Each attempt to resolve a misfit, perhaps by installing larger air conditioning ducts in an attempt to reduce the background hiss needs thicker ceilings to house the ducts, which entails lowering the ceiling height of the rooms, which ….and so on. The large number of individual issues which the architect needs to resolve plus the fact that these are not independent of one another means that the architect needs to either:

- consider all factors at the same time, which for even a small building may be impossible

  or

- divide the factors into groups (heating/cooling/ventilation; acoustics; room shape and height; …) and consider each individually. He or she might then subcontract the solution of each group of issues to an expert in that field. A heating/cooling and ventilation expert, for example, should be able to specify the most cost-effective solution to meet the architect's brief for those factors.

There is, however, one fundamental flaw in grouping factors into expert areas, and this is at the heart of Alexander's argument:

> there is no reason to suppose that the way in which a designer groups parts of the design into such 'expert areas' has any relationship to any *independent* groups which naturally exist in the building to be designed.

The building may conceivably have *no* groups of factors which are independent – in which case it will be extremely difficult to design successfully. But if it has such groups – for example the lighting, depth of the foundations, roofing material and so on are largely independent of the heating and cooling system chosen, then any such groupings which are independent of other groupings ('roofing' may be grouped along with 'outer wall construction' and similar items into a group called 'building fabric') can safely be designed as a group in the knowledge that there are no knock-on effects of any design decision upon any other group. And as Alexander pointed out, the groups we define for convenience into expert areas such as 'acoustics' may and probably will be out of kilter with the naturally occurring independent groups. Experts will thus make decisions about things which are best for their area but which upset decisions being made by the experts in other areas. Unless we design within naturally independent groups, we store up trouble for the eventual construction.

So how do these strictures apply to business? If the design of a business's organization structure does not reflect any naturally independent groupings within its business processes, any change to one part of the organization as a result of a change in a business process can have unpredictable repercussions throughout the organization. Since organization groups are rarely completely independent – they are merely more autonomous than if the organization were cut in other ways – we can tackle this problem with two complementary approaches:

- ensuring that the organization is structured around any naturally 'more-or-less independent' groupings of business processes

- then deliberately engineering *buffering* between the organization groups we have chosen such that changes within a group are as far as possible invisible to other groups. This would, of course, be unnecessary if such groups were *completely* independent. But this is a rarity, so we need to minimize the effect of such changes with some artificial organization constructs which make a group look the same to its peers even when it changes radically internally. Real organizations contain lots of groups which are independent of each other: for example, sales teams selling different brands in different countries are largely independent of each other, but each sales team will have continual contact with the relevant customer services team which processes orders resulting from their efforts.

A Pattern Language

Alexander's *Notes* may have been written for architects but was much better understood by those with a scientific and mathematical background. His later books are different: they are targeted squarely at practising architects, town planners and those who want to design and build their own houses. This does not mean that the content is any the less significant but it is accessible to a more general readership; its precise style is offset by an enthusiasm for buildings which are 'alive' – a concept which is difficult to pin down since it is rooted in people's perception and is extraordinarily difficult to define analytically (this search for an analytical definition and the profound consequences which emerge are the subjects of his second series of books).

Although Alexander's ideas predate much of the technical work of the past two decades on coevolution, his central tenet is that buildings, landscapes and towns should not be designed centrally in the conventional way – 'on the drawing board'. Instead, they need to evolve in a way which is driven by those who live there. Failure to do this results in buildings and townscapes which keep its inhabitants in a state of tension. This tension arises from the simple fact that people have evolved over many thousands of years to live in accommodation of a human scale which is not built to a precise plan: smaller parts of it blend seamlessly with others and into the wider landscape. Its antithesis – the concrete tower block – is tall, regimented and probably out of kilter with its surroundings.

Before we can identify the origin of this tension, we need to look more closely at how people live. When people use a house, office, garden, park, fields or even a roadway, their usage is a series of events. When I have a dinner party, my dining room is the focus of certain events which are repeated, broadly but not identically, each time I eat there. Those who are dining enter the room, sit down, chat, are served a first course, eat, chat again, then someone clears the plates away and brings on the next course, and so on to the end when the diners troop out to the drawing room. No dinner party is identical with any other but the series of events is similar. If we ignore the quality of the food and conviviality of the company, the success or otherwise of the dinner is determined partly by the room itself. Is it lit well enough such that diners can see what they are eating, but not lit with harsh lights or with people seated facing the window and directly into the evening sun? Is the size and shape of the room consistent with the size of the party? Is the décor consistent with the furniture: does my prized two hundred year old Georgian table-and-chair set match the wallpaper, cornicing and room height? Each mismatch – each lack of fitness for purpose – creates unease, however small, among my diners.

Alexander's point was that every place to which people go regularly is associated with a repeating series of events. This series of events in my dining room is inextricably linked with the way in which the room was designed. A well designed room with the right height, the right ratio of length to width and with natural lighting from at least two adjacent sides creates a series of events which makes for relaxing and enjoyable use. A badly designed room – one which perhaps has low ceilings (engendering feelings of claustrophobia), which is lit with a single large picture window on one side facing the setting sun, which is some distance from the kitchen (allowing food in transit to get cold) or too near the kitchen (allowing cooking smells to permeate) can mar a dinner party even if the food and wine are excellent. To take another even simpler example, most people are afraid of heights. Were I fortunate enough to own a luxury apartment in a high-rise block overlooking New York's Central Park, I would still be uneasy about floor-to-ceiling windows which were not recessed (i.e. were in line with the wallpaper) and, on the outside, had no ledge. Having walling or panelling for about a metre at the bottom such that the window stops short of the floor makes a difference. If in addition the window is in a recess, a small bay with window seats for example, my unease melts away. My logic tells me that heavy laminated plate glass will prevent my falling out of the floor-to-ceiling window. But my innate fear of heights makes me shun the area near such a window. We could thus specify designs for dining rooms or windows or any building component which make us feel at ease (and, alternatively, what designs to avoid and which create unease). If a house or apartment block were built using a collection of such successful designs and these designs were complementary and not clashing, they will reinforce each other. This principle does not only apply to buildings. The successful design of gardens and courtyards follows the same principle. One of Alexander's best examples is that of a porch. For most architects, it is somewhere to shelter when opening the front door from the outside or when locking the door on leaving. If it is an enclosed porch, it is somewhere to shed gardening boots and to keep umbrellas. But to Alexander it is part of an 'entrance transition' which prepares the visitor smoothly for a different environment. When approaching the house, the

level should change, the light should change (perhaps with a sweeping path between trees), the texture underfoot should change (perhaps asphalt switches to gravel) and so on. The successful design for 'path' should link seamlessly with that for 'porch' and prepare incomers for the larger transition from outside to inside the house (or vice versa).

Alexander described a collection of such designs. The principle was not to design from bottom up using the smallest designs (window; doorway; ceiling height; …) but to decompose whatever it was we wanted to build into many smaller designs. This was to ensure that the designs fitted together. Were we to cobble together a house using designs for window, doorway and so on, we would probably find that the resulting house lacked cohesion and was a misfit to its building plot. 'Top down' design avoids this. We first decide on the scope of what we want to build. This is mainly geographical: do we want to include a design for the approach road or do we have to accept what is already there? Do we want to design the house and garden together such that there is some unity between the two and such that going from one to the other is a seamless transition? Wherever we set the bounds (scope) on what we want to design such as "house and garden but excluding approach roads", we select the 'largest' appropriate designs – perhaps 'four-bedroomed house', 'courtyard' and 'grass lawn'. Sensibly however we will include any salient features of the neighbouring house which might affect our house-to-be such as neighbouring windows overlooking our garden. 'Courtyard' will in turn be composed of smaller complementary designs for courtyard features which promote use of the courtyard: a sunny corner with a bench; a shady corner for when the sun is at its height in midsummer; more than one entrance and paths to encourage their use; an entrance transition to prepare someone leaving the house to enter the courtyard, and others.

Our entire design will be made up from decomposing the highest level scope into smaller and smaller designs. These smaller designs are not of fixed sizes. Like the knitting pattern for a sweater, the designs represent shapes rather than sizes. A dining kitchen, for example, may have a length-to-width ratio of 6 to 4 approximately, but whether this is six metres by four metres or nine metres by six metres is irrelevant. There will be some absolute lower and possibly upper size limits: there is no point, for example, in specifying a dining kitchen too small to fit a dining table or even a breakfast bar. The designs thus define the geometry – the shape – of the result but not its size.

A collection of such designs may fit one culture but not another. A design for a dining kitchen will not be used by an ethnic group which never eats in the kitchen. A design for a roughly square dining room which was intended for use with a round table would be anathema to a very patriarchal group which sat in order of precedence: paterfamilias at the head of the table, with children in descending order of age, seen and not heard, at the other end. One can imagine a pool of all known designs of such forms from which a selection is made appropriate to each such cultural group. When the designs appropriate to a group are successfully used again and again by that group to build houses, blocks and even whole towns, the individual designers do not need to design from scratch: they know intuitively and from group lore which collection of designs to use. It becomes to them an indigenous language.

each design is a pattern, and patterns (e.g. 'dining room') are composed of smaller patterns. The collection of all known patterns is a **pattern pool**

the collection of all patterns appropriate to a particular culture is a *pattern language*

What makes a successful pattern? We can imagine a world full of unsuccessful building patterns which fulfil some commercial objective but which are unpleasant to inhabit. The high-rise concrete-framed tower block built to provide low cost housing for the masses could equally be specified using patterns. It typically has thin walls separating neighbours (who can thus annoy each other – unintentionally or otherwise), and an unpleasant dour exterior with sharp edges and extreme regularity of construction. It is a positive deterrent to neighbours who – culturally – would otherwise pass the time of day with each other: there is no street for them to walk down to do so, just an unpleasant shaded corridor also in dour concrete. And so on.

We evolved from a less developed world, and in the less developed world regularity of construction was non-existent. Where everyone builds their own house, each hones the layout to whatever they find most congenial but within the constraints of their ethnic custom. Why do we feel unaccountably at home in the towns which have evolved over many centuries and whose streets are narrow and winding? It is more than a feeling of "gee that's quaint". Similarly, why do we feel strangely at ease in old country hotels which have sprouted over the years in strange directions and on many different levels? And why do we not experience these feelings in modern towns and modern hotels even though the facilities may be incomparably better? Why is old concrete depressing whereas the similar use of natural stone, which takes on lichens and a patina of age, is welcoming? Brick is an even more telling material. Hard-faced brick never mellows; brick with a slightly more friable surface and some irregularity in manufacture becomes less harsh with age. It may never look 'natural' – red is not a colour common in nature – but eventually blends in. Anyone doubting this should visit London's Hampton Court Palace, much of which was built in the Tudor era and has had a few years to settle down.

Object-orientated system design

A later chapter will look in more detail at complex IT systems which went adrift during development because project managers and system designers ignored *side-effects* – the inevitable accompaniment of complexity where a change in one area has an unanticipated knock-on impact on another area. Size and complexity are usually – and wrongly – treated as synonymous. But, as *The Coevolving Organization* stressed repeatedly, the essential difference between them is one of *cross-connections*. A company's sales-force in one country will, for example, very likely have an identical structure of:

- sales director
- regional sales managers
- area sales managers
- sales territory men and women

in a strict hierarchy. It might be a large hierarchy with hundreds of sales men and women at the bottom of the tree, but is relatively easy to manage. There are no links – no cross connections – between a salesman in the north of the country and a salesman in the south. Setting sales targets is similarly easy because poaching customers (at the store level at least) from a colleague is impossible. A store is either in one sales patch or another. If a store in my patch has an unusually successful promotion of one of the brands I sell, it will be at the expense of the market share of competing brands. The worst which could happen within my company is that if this store were near the boundary of my sales territory, customers who usually patronized a store in the neighbouring territory were seduced into mine instead.

Large thus does not necessarily mean complex. But does complex mean large? Not necessarily. It can be diabolically awkward to run a relatively small but largely matrix-managed business – exemplified by the visible cross connections on the organization chart. If I am sales manager for my country *and* for the manufacture and marketing of low-cost Brand A globally, I and my colleague who is responsible for the sales in his country *and* for the manufacture and marketing of more-upmarket Brand B globally can have an enjoyable time frustrating one another. My colleague wants to promote his Brand B in my country which will steal some share from my Brand A by up-trading Brand A's usual customers. This will clear his embarrassing overproduction of Brand B but make no profit in my country which is what I am measured on, since import tariffs on Brand B are high whereas my Brand A is made locally. In return, however, I could arrange a quid pro quo for my colleague…

IT systems are prone to the same underlying problem. They can be very large and are certainly technically 'complex', but are not necessarily complex in the sense with which we are concerned. What matters is whether the thousands of *objects* – pieces of program code which 'do something':

- are insulated from each other as far as possible
- can make no assumptions about how the others work

and

- communicate when necessary in a regimented way which allows the caller to ask for something to be done, to print a line of text for example, but is strictly barred from finding out how the action is actually performed.

By no coincidence, our coevolving business objects closely resemble these IT objects, where each of the latter is a self-contained section of program (a process) with its own associated data. Such objects communicate with each other by passing messages using formal message formats and protocols as described in Chapter 7 of *The Coevolving Organization*, but *how* they perform their functions is deliberately hidden from others. This 'information hiding' for computing objects was introduced

as a way to protect an object from being tampered with by other objects or from suppositions being made on how it worked internally. These objects exist independently of others, hide internal information on *how* they do what they do from others, respond only to formal messages, have standard 'classes' of object (sales products; field sales territories;...) with 'instances' of each object (a particular brand being sold; a particular sales territory) and so on. The programming languages which provide these features of classes, objects and so on are unsurprisingly called object-orientated programming languages and are generally thought of as a recent invention. This is wrong: they had their origin in the 1960s simulation languages whose aim was to model the real world.

This book is not aimed primarily at IT experts and a summary of object-orientated programming in isolation would be a sterile experience for non-specialists. Fortunately, however, we can approach it from a slightly different angle via its origins in simulation:

from
- simulation of the real world

to
- simulation languages

to
- to object-orientated languages.

The present writer's first 'real' job was with the UK's former national rail authority writing computer programs which simulated and scheduled the movements of passenger trains in the most complex railway networks in England: 'complex' in the sense that one train movement could have repercussions on many others. The aim was to find out how to regulate the flow of trains better. The targets were to improve how well they kept to the timetable and to identify opportunities for better use of the existing track capacity: a better service, more trains or ideally both. So, if only for selfish nostalgia, the following examples are taken from railway simulation. This will lead naturally to the concepts underlying object-orientated programming without dwelling too much on the purely IT aspects.

Firstly, we need to draw some boundaries. We could in principle try to simulate the entire railway network but this would be a mammoth job and not very productive. Instead, following Alexander's decomposition principles (see page 1), we split the national railway network into sections which are as autonomous as possible. This usually means dividing the network midway down long sections of simple track as opposed to trying to split up the network in the middle of a station, marshalling yard or junction. We also need another type of division. Train movements occur round the clock, but are much less numerous in the early morning than in the 08:00 or 17:30 peaks. A late-running evening train *could* possibly cause a train the morning after to start out late, but this is unlikely: there is usually ample slack in the overnight timetable and such knock-on effects are only normally apparent to passengers if there has been major disruption caused by snow or labour strikes when the engines and carriages end up in the wrong places overnight. So, following Alexander, we have selected for simulation a slice of the railway network and timetable whose

performance will be as little affected as possible by the behaviour of trains in adjacent networks or by their behaviour the previous day.

The railway network under consideration then needs to be broken down into all its components: track, junctions, signals and so on, and the relationship between each defined. The lengths of track between junctions need specifying and the relationship between the track on either side of each junction needs to be codified. Large junctions look like spaghetti to the uninitiated but are made up of many combinations of a small number of simple junction types.

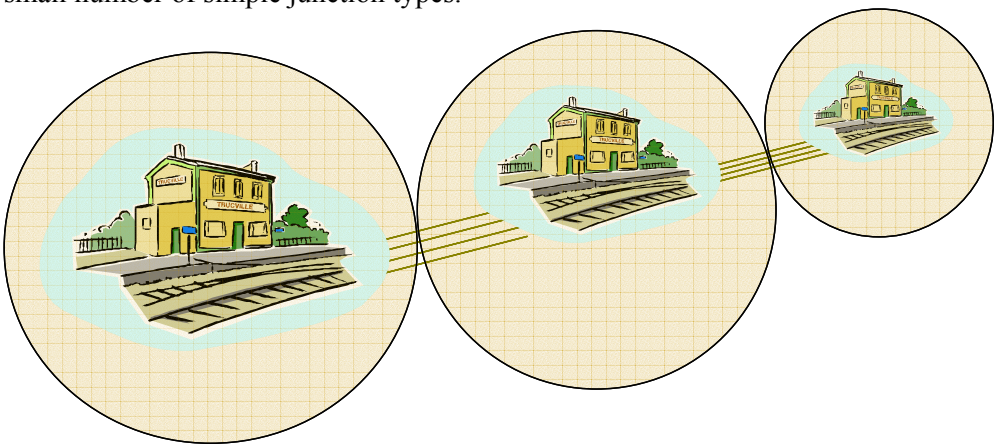


Figure 1 - scope of simulation

Some of this, such as the track, is unchanging, at least in the short-term. Some – junctions for example – change when a train movement is set up by the train regulator (signalman). Some – the signals themselves – change when the regulator sets up a route *and* later when a train passes in order to protect any following or otherwise conflicting trains. In summary, we have many different classes of 'thing' (such as signals – and trains themselves) to simulate.

To make the simulation programs reusable for other railway areas and timetables, it is essential that all the details of our timetable and slice of the railway network are presented to the simulation program purely as data. So what does the program itself contain? Firstly the basic mathematics of how trains start, move and slow down, together with the logic controlling the signals which prevent one train running into another. But the performance characteristics of different types of engine and carriage will be presented as data in order to cater for the numerous varieties of each which are present on a real railway. Secondly, the program will need to undertake the actual simulation of train movements: to move a train from A to B in a realistic and safe way.

To manage each type of item to be simulated, we could categorise them as in the following example:

locomotive
    electric locomotive
      type A
      type B
      type C

diesel locomotive
  type D
  type E
  type F

For simplicity we will assume that each locomotive type is permanently coupled to a fixed number of carriages and we will treat the combination as a single unit in what follows. The details for each type of locomotive would contain everything needed to simulate its movements: motive power, braking characteristics, weight (including carriages) and so on.

We thus have an abstract (high-level or generic) class of motive power: 'locomotive'. The details associated with this will be scant, and will be mainly the mathematics needed to calculate the movements of any locomotive, given some information about the track to traverse (up hill and down hill gradient, stopping places and so on).

At the next level down, we have two basic classes of motive power – electric and diesel – which work sufficiently differently for them to be treated as two species rather than as differently performing units of one basic design. The definition for each inherits from the parent 'locomotive' the mathematics needed to calculate its performance, but this is fleshed out by additional details peculiar to each class of motive power.

Finally, we have the individual locomotive types themselves. The definition for each inherits the mathematics from its grandparent 'locomotive' and the fundamental details of its motive power type (electric or diesel) from its parent. 'Locomotive' is less well defined than 'electric locomotive' which is in turn less well defined than the 'type A electric locomotive', one or more of which will have their behaviour simulated. The 'Type A electric locomotive' is said to have a *concrete* class because it is something which can run on a real railway and can be simulated. 'Electric locomotive' and 'Locomotive' on the other hand are one and two steps respectively removed from the real thing and are *abstract* classes.

At present, however, 'Type A electric locomotive' is merely a detailed definition of a particular locomotive. To simulate the movement of one, we need first to create it. We may have lots of Type A electric locomotives in our simulation, with various positions and with different speeds, and we need to create one or more 'instances' of each. When we create an **instance** of a (concrete) class, we are creating an **object** using the detailed definition (in our case the locomotive design manual and blueprints, plus position and speed on the track). An object is thus an implementation of a definition. The definition is static but the object will 'do something' – in this case move around our virtual railway.

From our point of view, what is equally important – perhaps more important – is how objects communicate with each other. We stated in passing that objects treat each other as black boxes and are unable to find out what happens within other objects. They can only make requests of one another with *messages*. These messages are very stylised: more akin to the format of a formal invitation to an English wedding ("Mr & Mrs X request the pleasure of Mr & Mrs Y at the wedding of their daughter Z ….") as opposed to an informal note. And the *only* way an object can

request services or data from another object is via a message. The structure of the message is determined by what the receiving object expects to receive. When a squad of soldiers is being drilled on the parade ground, the drill-instructor will shout a formal command – perhaps "By the left, quick march". The soldiers will obey this command if and only if it is in their repertoire of commands and in exactly the right format. If not they will ignore it. Similarly, an object will only carry out a request from another object if the request is in a format to which it responds. It defines – and notionally publishes or advertises – every request it will accept and the manner in which that request may be framed. A drill squad receiving a command "Left foot forward, stride out" will, if they are well-trained, be silently ignored. An object receiving a request which is not in a format it accepts will also ignore the request, or possibly send a courtesy message back saying it is unable to carry out the request. An object can accept many different requests (cf. "Stand at ease"; "Halt"; "Present arms"; …) and the collection of all valid requests is called its **interface**. Furthermore, different objects can accept requests presented in the same format. A British and an Australian army squad may both be legitimately commanded to "Present arms" but are entitled to perform the drill task somewhat differently. (Those who have read *The Coevolving Organization* will have, by now, realised that these messages are the formats and protocols which underlie C-couplings; one object C-coupled to another object effects changes in the behaviour of the other via a message).

We are now ready to run. Assume that we have the entire infrastructure – signals, track and so on – in place within the simulation program. To simulate the movement of a *particular* Type A electric locomotive (the '08:40 from Great Snoring to Houghton St Giles') we first of all must create an instance of one.

Each instance is a combination of:

data (position, speed, weight…)
operation (also called *method* – the mathematical process needed to simulate the movement of the train)

The operation used to move the train is invisible to the rest of the simulation program. Once an instance of a train is created, it will move under its own steam, respecting signals and traversing gradients correctly. In practice, the classes Locomotive, Electric Locomotive and Type A Electric Locomotive may only contain such things as acceleration and braking characteristics. Further objects such as Signal and Track will contain other settings needed in order to simulate the movement of the train. Simulating the movement of this train might then look something like:

a. *Create-instance-of* Type A electric locomotive at position X with speed Y (we now have a particular Type A electric locomotive object which can do something, as opposed to just its design or class)
b. *Simulate* [this] Type A electric locomotive [using] Signal, Track,…etc

But this is simulation and not mainstream IT. What about those more ‘normal’ systems which run business processes such as customer services? These ‘more normal’ systems are actually *simulations of the business processes.* The processes are (notionally) defined as **classes** in a business process handbook and the IT systems which run them are (roughly) collections of business process classes. The systems themselves, when being run, are nothing more than instances (implementations) of the business process classes although they probably look nothing like it.

CHAPTER 4

ORGANIZATION AND BUSINESS PROCESSES

Introduction

A business with well-thought-through business processes implemented consistently throughout the organization has an obvious advantage over its less well-structured competition. But it still has two further challenges:

- how can the business processes be engineered to evolve at the same pace as the moving target of competition and the changing requirements of customers? In other words, how can this very structuring be prevented from putting 'treacle' in the way of poise and responsiveness?

- how can exceptions be handled? These are either unusual events defined within a business process as 'to be handled manually' or events for which there is no process defined (and creating business processes is usually one of these!)

The advantages of patterns were recognized by many professions, notably IT program designers who saw immediately the connection between the autonomous (non-interfering) nature of patterns and the 'objects' of object-orientated programming. For the same reason, managers of large projects seized on the similarity of patterns with project tasks: any project is easier to plan and runs more smoothly when streams of tasks can run in parallel without interfering.

The first challenge was dealt with at length in 'The re-birth of growth' in Chapter 6 of *The Coevolving Organization*.

The second can be exemplified as:

> "To whom do I need to talk in order to understand the issue or get permission for me (or someone else) to take action".

In a large or complex organization, this is not easy to answer since, by definition, there is no business process extant to guide me. And the result is thus all too often either inaction or a reaction which is far too late. Say, however, that the business had been structured such that the role of each division, each department and even each individual is as autonomous as feasible in the sense that no other way of splitting up the organization could make them more autonomous. It then becomes easier for me to get information or make my decision since the information about my problem and the individuals I need to consult are probably clustered around me – organizationally if not geographically.

Note that this organization structuring is in addition to formal business processes (which also work better in such an organization). The designs for the organization units are patterns.

Formal business processes and such organization structuring are very closely related, but even a business which has ill-defined business processes can gain from a 'well-patterned' structure; indeed it may gain more that a business with good processes since, in the absence of good processes, it will handle more issues as 'exceptions'.

However, business processes themselves will change. Some will evolve smoothly in a planned way as supply, manufacture and distribution evolve. Others will be forced to change rapidly in response to competitors' threats (their new technology, new ways to market and so on). Amending business processes in a hurry can be perilous, particularly if the business is accustomed to gradual change. Patterns not only define objects but, more importantly, define how they communicate, and special patterns are now available which allow flexibility to be incorporated in the links between objects. A pattern can, for example, be an object or structure of objects which acts as an intermediary (buffer) between other objects, perhaps as an interpreter. Patterns can be objects and object structures but can also be more generic classes from which objects themselves are derived.

Processes for most businesses are usually grouped under three umbrella headings:

- purchase to pay (buying something through paying for it)
- order to cash (receiving an order through the customer's payment for it)
- record to report (roughly, all the remaining back-office functions)

To illustrate the introduction of buffering into an established business process and an organization designed around that process, consider the following simplistic example of a traditionally-structured business:

- customer services team
  - receives a telephoned order from someone in the sales force
  - checks customer's credit status
  - checks if stock will be available in the distribution depot either now or when the order will need to be shipped
  - earmarks existing stock for the order
  - requests the manufacture of extra stock if necessary
  - prices the order and applies promotional discounts
  - despatches the order details to the logistics team

- logistics team
  - allocates truck space
  - issues instructions to the depot to pick stock at the right time and then load the allocated truck

- sends a despatch note to the customer's receiving depot or store ('this is what we have sent you')

❖ customer services team (again)
- issues an invoice based on the despatch note (which may or may not reflect 100% of what the customer ordered; some items may be back-ordered; some might be on a later delivery that day and so on)
- receives the customer's cheque payment (which may or may not be a payment in full)

❖ finance 'accounts receivable' team
- banks the payment if not sent by bank transfer

There are several ways to map the bulleted (▪) tasks to teams. The split between customer services and logistics is often on the basis that customer services deals with individual orders from customers whereas logistics deals with aggregations of orders and trucking. However, following the principles described earlier, one acid test for whether the organization is out of kilter with the business processes is simply whether a lot of communication – particularly two-way communication – occurs between them. If it does, and in particular if this communication is between individuals who are checking and expediting rather than simply a result of systems passing information, then we need to see if there is some other 'cut' of the organization which will result in the groups who spend a lot of time communicating being part of the same team.

However, do teams matter; and what is a team? In principle, the business could be a collection of individuals subservient to computer-driven business processes. But this takes us back to the fundamental issue of whether we want a monolithic 'top down' business, and the contention in *The Coevolving Organization* was that there are better ways to structure a business than that.

If we elect to follow the principles outlined therein, we try to define areas which are as autonomous as possible. This means that they need to communicate with other areas as little as possible. This does *not* mean that information must be squirreled away within each coevolving object – the customer services team for example – but that each team must be free to fulfil its own objectives and make decisions without constantly needing decisions or approvals from another individual or team. It *does* mean that information which is purely about the internal workings of a team does not need to be passed on. Furthermore, such information should not be visible to the team's internal 'suppliers' such as those downstream – logistics for example, or internal 'customers' upstream – the sales-force, for example. Customer services are 'contracting' with the sales-force to arrange delivery and accept payment for all orders the sales-force manage to solicit. In turn, logistics are contracting with customer services to arrange for the loading and shipment of any orders sent to them by customer services. This implies – correctly – that the logistics team is invisible to the sales-force! (If I buy a faulty new car, I tell the dealer to fix it or supply a replacement; it may be the manufacturer's fault or shoddy handling in transit or even a fault in a bought-in accessory; but my contract is with the dealer).

Let us assume now that we have:

- well designed business systems for order-to-cash (as above)
- processes for accommodating exceptions, both real exceptions and possible exceptions: for example, a customer who, in response to the hard selling of an important impending promotion by the sales-force, has ordered slightly in excess of his credit limit
- teams whose grouping and objectives reflect the autonomy principle outlined above and described at length in *The Coevolving Organization.* These teams can – and probably will – be composed of smaller teams structured on the same principle which could be summarised roughly as "autonomy to fulfil their objectives". These objectives may (deliberately!) conflict with those of other teams as described in Chapter 4 of *The Coevolving Organization:* customer services wants to achieve on-time delivery with each order containing exactly what the sales person ordered for the customer (no short shipments; no item substitutions; no extraneous or damaged items shipped;…). Why? Because that is their 'contract' with the sales-force. Logistics on the other hand want to send out full trucks when trucks are available; they want to avoid part-loaded trucks, the need to buy additional emergency trucking, unbalanced trucks (ones which carry too many lightweight pallet-loads or too many heavy pallet-loads; ideally, each truck should be more or less at its volume *and* weight limit), and so on

This is a simple and traditional business structure and probably works well with small customers. Now assume that business grows and customers become larger. Big customers, supermarkets for example, order direct, either by phone or more likely by computer and electronic data transfer. They pay by bank transfer. Orders to be delivered into just one of their distribution depots may consist of several truckloads. We have thus added some new business processes:

- direct ordering
- payment by bank transfer

But we have also fundamentally altered the role of customer services, and the sales force's role has become one of business development. Customer services are now responsible directly to the customer for the fulfilment of each order. The sales-force's role and objectives have changed; and customer services' 'customer' is now the real customer. This change may seriously upset the effective working of both customer services and logistics, and reduce the number of on-time accurate deliveries until both departments reorganize to accommodate new processes and new responsibilities.

So how can we handle business process changes like this in such a way that the teams (and external contacts) with which customer services, logistics and sales force communicate are insulated from the change?

CHAPTER 5

BUFFERING

Introduction

In the preceding chapter we looked at how we could structure an organization such that when business processes change, or perhaps when a team changes its structure as a result of losing or gaining an individual with some key skill, the teams (and external contacts) with whom each team communicates are insulated from the change. If teams were completely independent, this would not be a problem. But teams are linked by both computer systems and personal contact with other teams. We saw that if we structured the organization correctly by creating teams which are as autonomous as possible in the sense that any other way to divide up staff into teams would result in more overall communication between teams and less within teams, then the knock-on effects of change within a team on other teams is minimized. But 'minimized' here means minimized with respect to any other way to cut the organization. There is, however, a way to reduce the impact on other teams further if we are allowed to create some artificial organization 'constructs'. Exactly which construct we use depends upon what we want to achieve.

One way to reduce the impact is to erect some sort of organizational *veneer* which makes a team's contacts – its visibility to others – look the same irrespective of changes internally. A hypothetical pattern for this might look something like:

Veneer pattern

Name: "Team veneer"

Problem: Need to provide an unchanging interface between teams even when the internal organization of the team or the business processes it supports change.

Context: The team is subject to frequent changes of staff or staff responsibility or business processes, or the business processes are not well defined and there is considerable checking, expediting and decision making needed by individuals, or both. Note that it is impossible to foresee when radical changes to business processes will be needed, since these may be driven by competition, the economy, the stock market or other difficult-to-predict forces

Success criteria: A team which, to those who work with it, appears unchanging and predictable to work with.

Solution: Create formalised interfaces to the team – as seen from other teams and from the outside (real customers, for example). These formalised interfaces might be something as simple as a customer services 'ordering point', whose function is to accept orders from internal customers (e.g. the sales-force) or real external customers

in the same way; behind the scenes (i.e. within the team), these orders may be treated differently but this difference should not be visible to internal or external customers. The team operates on the 'black-box' principle as described in 'From genes to business' in Chapter 4 of *The Coevolving Organization*

Rationale: The loss in efficiency caused by creating such black-box interfaces is marginal when compared with the much larger gain in stability to the business as a whole. Part of the business – whether one team or some larger organizational entity – can be reorganized with no visible loss of performance to other groups in the business which depend on it.

This is a very high-level pattern. In reality, it is the template for some more specific patterns for particular business processes. We might have patterns for:

'*team veneer – order acceptance*'
'*team veneer – despatch*' (e.g. liaison with logistics)
'*team veneer – future stock availability*' (for example, liaison with manufacturing for work in progress and with production planning for querying or adjusting next week's production)

In these examples, the salient point is the engineering of the person-to-person interface such that if internal manufacturing were replaced by co-manufacture (by a third party) or logistics were turned on its head by the outsourcing of depot operations, each such area appears to other areas to be functioning exactly as before. The same would apply to a pattern for logistics:

'team veneer – logistics truck management'

where the design of the logistics team was such that the links between each sub-team:

- dispatch planning – the amount of stock to be shipped and when
- the allocation of stock to trucks
- stock picking
- truck loading

were 'veneered' such that any change to one was invisible to its internal customers and suppliers. The sub-teams managing stock picking and truck loading operations are suppliers (of dispatch services) to the stock allocation sub-team, who in turn are a supplier of stock management *and dispatch services* to the dispatch planners, who are, in turn, suppliers of overall dispatch services to customer services. Note that a business's products move one way (from manufacture to customer services to logistics to customer) while the internal customer/suppliers 'contracts' usually work the other way.

This example has been elaborated to demonstrate two points:

- customer services, for example, should have no knowledge of – should actually be unable to find out (!) – how the orders they send to dispatch planning are allocated to stock, are loaded and subsequently sent to the customer. If they can find out, they may start making assumptions (with the best of intentions…) which will throw deliveries awry when a business process or organization change occurs somewhere downstream in logistics.

- teams (objects) can be contained within others, like a nest of Russian dolls. And so teams can be built up of sub-teams whose interfaces can also be veneered. There is, of course, a point of diminishing returns when the sub-team is so small, perhaps one individual, that it ceases to be sensible or economic to do so or is too small to make decisions autonomously.

Unfortunately, although this veneer pattern gives some ideas on how to buffer one area from another, it is too high level and unspecific to be of use. To remedy this we need to use the object-oriented pattern ideas introduced from page 16 onwards.

Following are the five patterns which are the foundation for buffering and for solving other related organization or process problems caused by over-tight coupling of teams or business processes. Each pattern is useful in a specific situation.

- **Adapter** (decouples two areas by transforming one interface to another; this is the fundamental 'veneer' pattern)
- **Façade** (loosely, a variation of Adaptor for an area with many interfaces)
- **Mediator** (converts a mesh-like organization or business process structure into a star)
- **Chain of responsibility** (decouples requestor from responder when it cannot be predicted which team or process will handle a request)
- **Bridge** (decouples variations in definitions – policies, process definitions and the like – from their implementation)

These names are the ones used by IT system designers, and the IT versions of these patterns are described by the Gang of Four. They will each be specified in the format of a pattern using the object-orientated concepts previously introduced and described using examples from real business organization or processes. The term 'requester' is used to denote anyone from another team or from outside the business needing to communicate with someone in the team; this communication could be a phone call, email or business-to-business (i.e. system-to-system) electronic transaction. For each pattern, a description of the pattern in object-orientated design language is included. For those unfamiliar with object-orientated design conventions, the two main types of 'arrow diagram' which will be used are as follows:

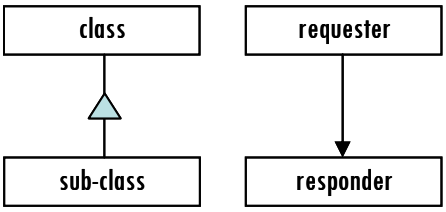


Figure 2 - class and object diagrams

The shaded upward arrow displayed midway between two classes indicates that the lower class ('Electric locomotive') is a subclass of the upper class ('Locomotive'). The solid black arrow displayed usually at the end of a line connecting two boxes indicates that the item (class or object) at the arrowhead end is called by the other item. This calling will normally create an instance of an object of the called-item class.

Although these patterns are likely to prove the most useful ones in practice, they do not form a complete pattern language peculiar to certain types of organization or business process problem. Much less do they form a comprehensive pattern pool of all possible organizational patterns. They are intended to provide a foundation on which users can build further patterns peculiar to specific organization or business process circumstances. *And, as with the edge of chaos, self-organization and highly-optimized tolerance concepts and the NKCS mechanism, they provide a framework – a language – with which to analyze and discuss organization and business process issues.*

Adapter

Problem:
There is a need to change the structure of a team while letting requesters continue to call in an established way – perhaps because there are so many of them.

Context:
The way in which requesters call cannot be changed, but we need to change the structure of the team they call.

Success criteria:
Requesters call in the same way and do not realise that the structure of the team they are calling has changed

Solution:
Create an interface which, to the requester, looks just like the established way to call. The interface then maps the call to the new team structure, i.e. it converts the external view of the team to the new internal structure

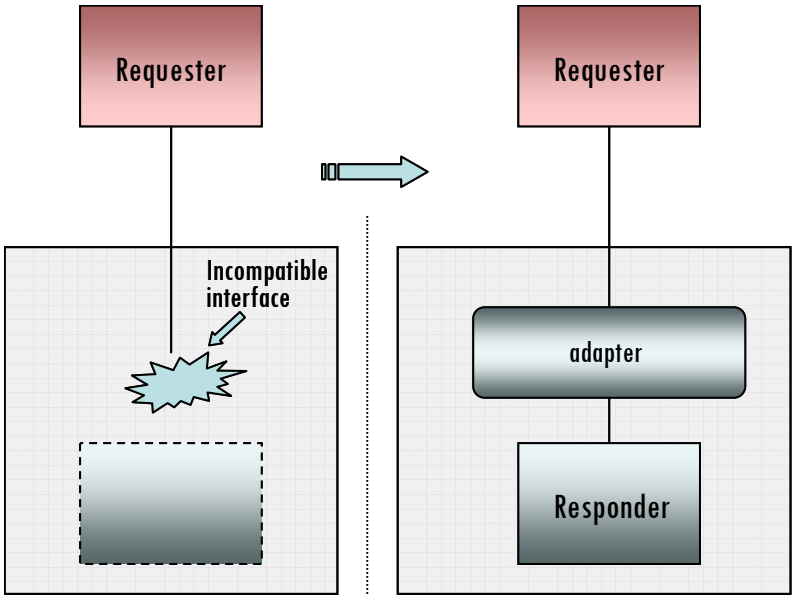


Figure 3 - Adaptor pattern diagram

Object-orientated design notes
The diagram below shows two ways to use Adapter.
The first uses classes in which subclass 'adapter' inherits from two parent classes 'virtual requestor' and 'responder'. As a result of this inheritance, Adapter has definitions for both interfaces and can convert one to the other *and* perform the role of responder (since it inherits responder's operations as well as its interface).
The second way uses objects: subclass 'adapter' does not inherit the responder's function but instead simply calls responder using the correct interface.

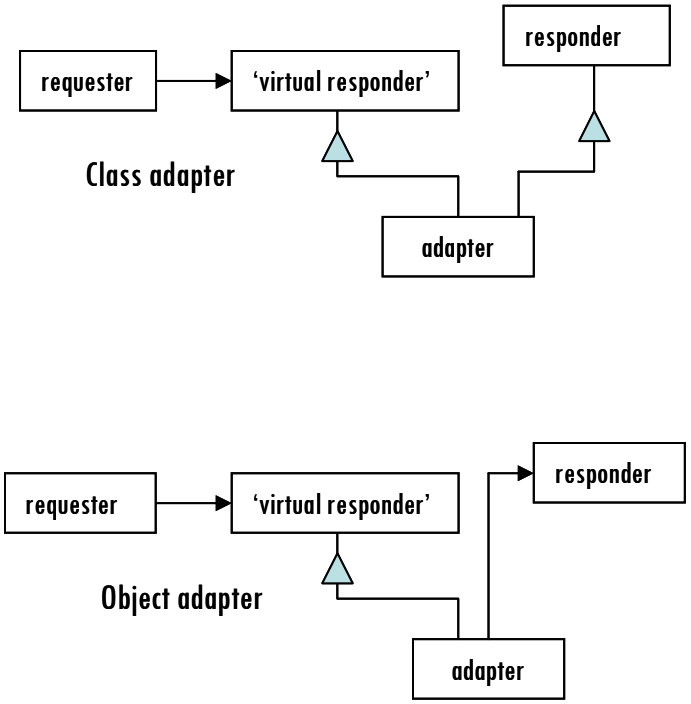


Figure 4 - Adaptor pattern OMT

OMT is Object Modelling Technique – see page 65

Façade

Problem:
Requesters are finding it difficult to get in touch with the appropriate responder in the team

Context:
The team has many different contact points for internal and external requesters. Most requesters have a standard request and relatively few have specialised requests.

Success criteria:
Low level of redirected calls

Solution:
Create a standard interface for 'normal' calls. The sub-teams behind this interface are not regrouped into a new team but remain in their own sub-teams because this is otherwise the most autonomous way to split the team.

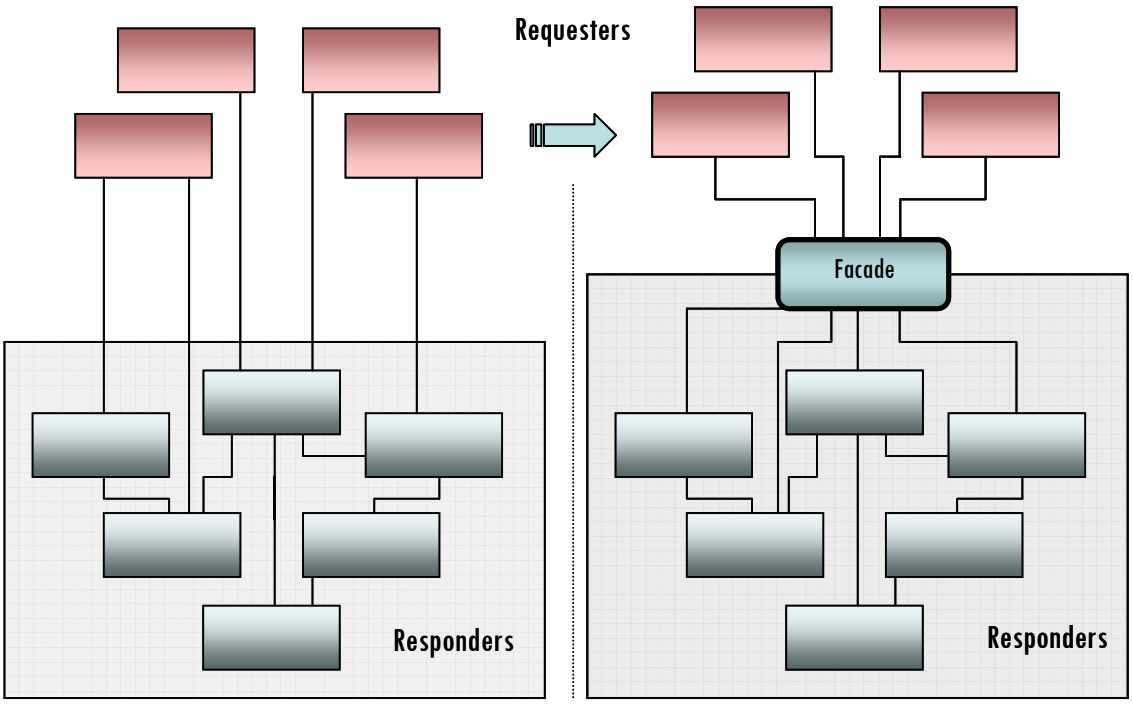


Figure 5 - Facade pattern diagram

Object-orientated design notes
The diagram below shows how to use Façade.

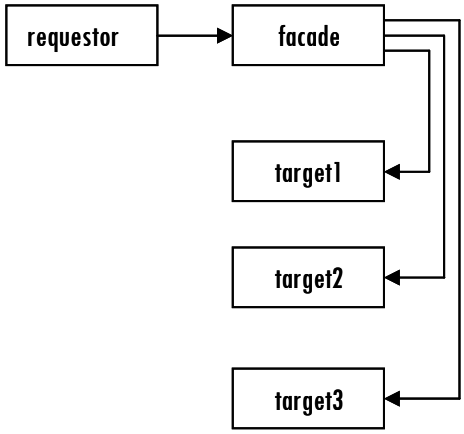


Figure 6 - Facade pattern OMT

Façade is implemented with classes (note that the targets are not subclasses of Façade). For simplicity, only three of the six targets are shown in the diagram.

Mediator

Problem:
Team communication is over-complex even though individual teams communicate with others in a simple way

Context:
Teams in all or part of the business communicate with each other in a simple and logical way (i.e. the team groups are the most autonomous possible), but the overall network is complex, i.e. is a mesh rather then a hierarchy or sequence of the type A=>B; B=>C.

Success criteria:
Neither communications nor requests for decisions go round in circles.

Solution:
Create a central point (sub-team or electronic equivalent) through which all communications between these teams are directed. Communications circles can be detected and prevented. This converts a mesh into a 'star'.

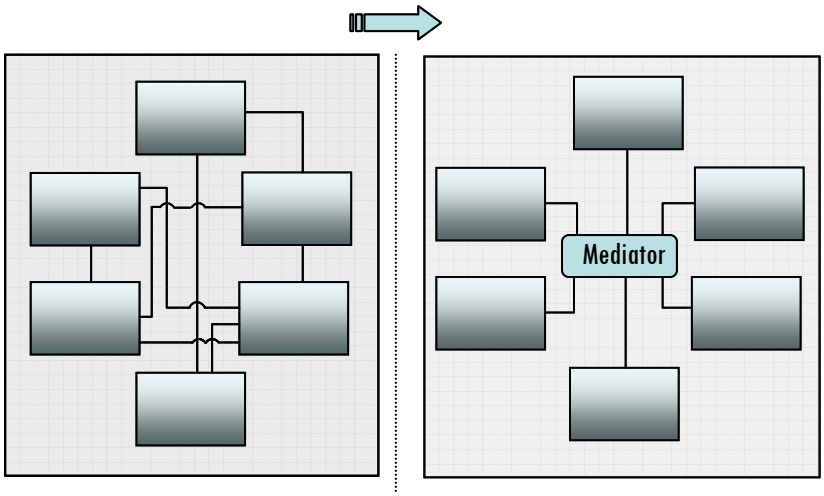


Figure 7 - Mediator pattern diagram

Object-orientated design notes
The diagram below shows how to use Mediator. As before, only some of the targets are shown.

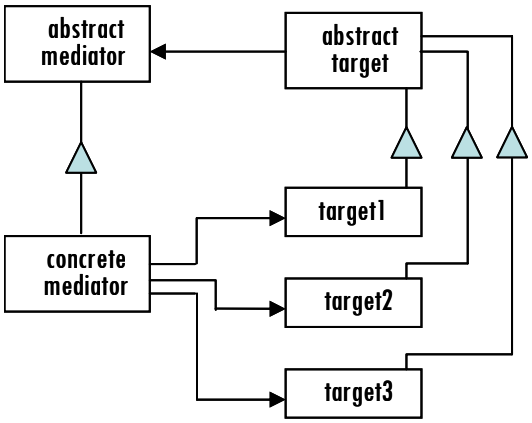


Figure 8 - Mediator pattern OMT

Chain of responsibility

Problem:
If the requester's request is arcane and the number of specialities handled by the team is large, it may be difficult for a central point to decide where the request should be handled.

Context:
Large teams with many specialities where requesters generally do not know who to contact. A façade (above) can handle common calls but lets those needing specialist support communicate with the specialists directly. This, however, assumes that the requester knows which specialist will handle the request.
Requesters are emails and business transactions rather than human requesters.

Success criteria:
The requester is unaware that the call is being passed from specialist to less specialist sub-teams.

Solution:
Requesters are passed initially to a specialist sub-team which might be able to resolve the call. If they cannot, the call is passed to a less specialist sub-team, and so on until a general 'catch-all' sub-team fields the call.
In the example below, a requester makes a request without knowing who would handle it. If team Responder 1 is unable to handle it, the request is passed to Responder 2 and so on – *without reference to the requester* who has no idea (and cannot find out) who will handle the request.

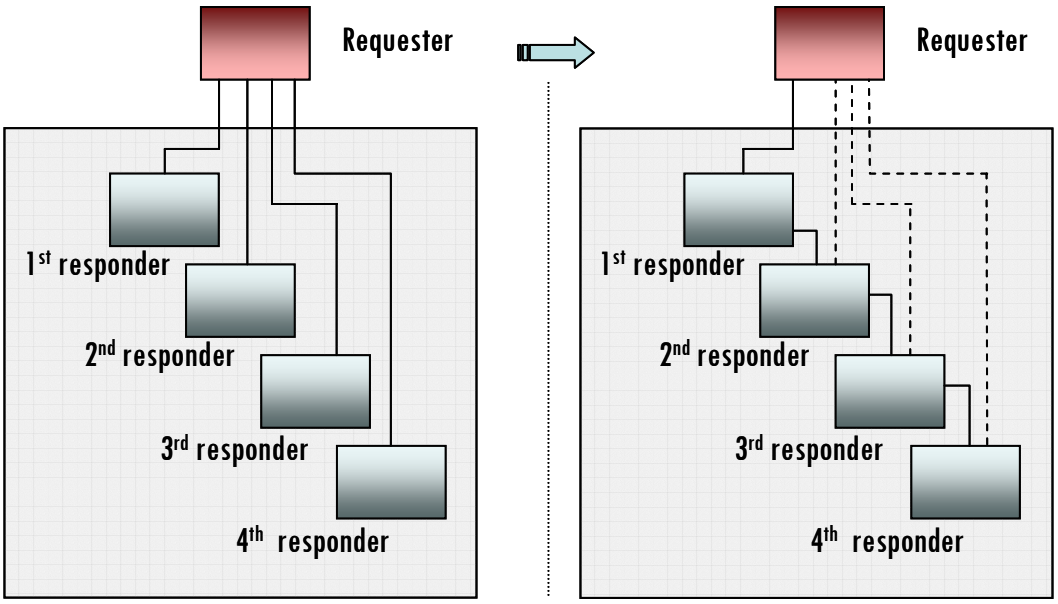


Figure 9 - Chain of responsibility diagram

Typically (not shown) there would be an additional ‘request handler’ operation which enabled a request to be passed on to the next responder in the chain.

Object-orientated design notes
The diagram below shows how to use Chain-of-responsibility. As before, not all the targets are shown.

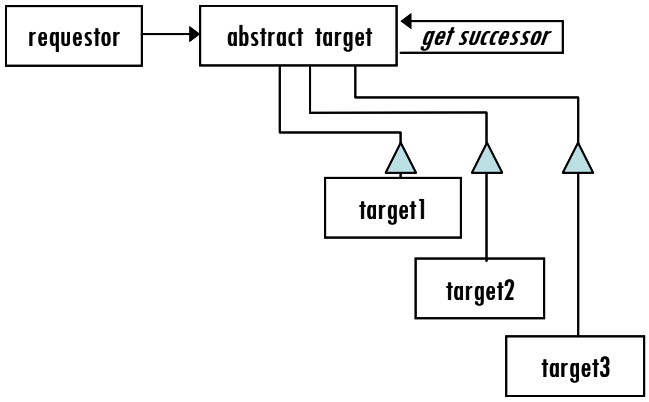


Figure 10 - Chain of responsibility OMT

The various targets – the classes which, for example, undertake progressively less specialised 'help-desk' functions – are all subclasses of 'abstract target'. The 'get successor' internal request allows any target to request that its successor is invoked.

Bridge

Problem:
Adding a new business process results in an explosion of country-specific implementations.

Context:
Corporate manuals exist on how each department must be structured and which processes it must follow. Departments structured along these lines exist in each country in which the business trades. Additions and (occasionally) deletions to the corporate manual occur regularly.

Success criteria:
Additions and deletions to the processes within the corporate manual can be implemented in each country without a 'combinatorial explosion' of variations.

Solution:
Instead of each country-specific team having manuals derived from the main corporate manual detailing each process as it applies in that country, the corporate manual and country-specific implementations are decoupled as in the example below. The first diagram shows what happens when the definitions (classes) are not decoupled from the country-specific implementations. The descriptions of approved training methods, company personnel grading principles and – to be newly added – company career planning guidelines are intermixed with the country-specific implementations of those policies. When HR develops a new speciality, succession planning for example, or moves into a new country, the number of implementations explodes; for example, for a (conservative) five policy areas to be implemented in twelve countries, there are sixty implementations. The fault is that we have failed to distinguish between the policy definitions (which are not country-specific) and the implementations (which are).

The second diagram shows the simplification which results from separating the two. It is worth clarifying why this separation is so successful. What we have actually done is to separate and ring-fence the two types of variation: additional policies and additional countries are not related.

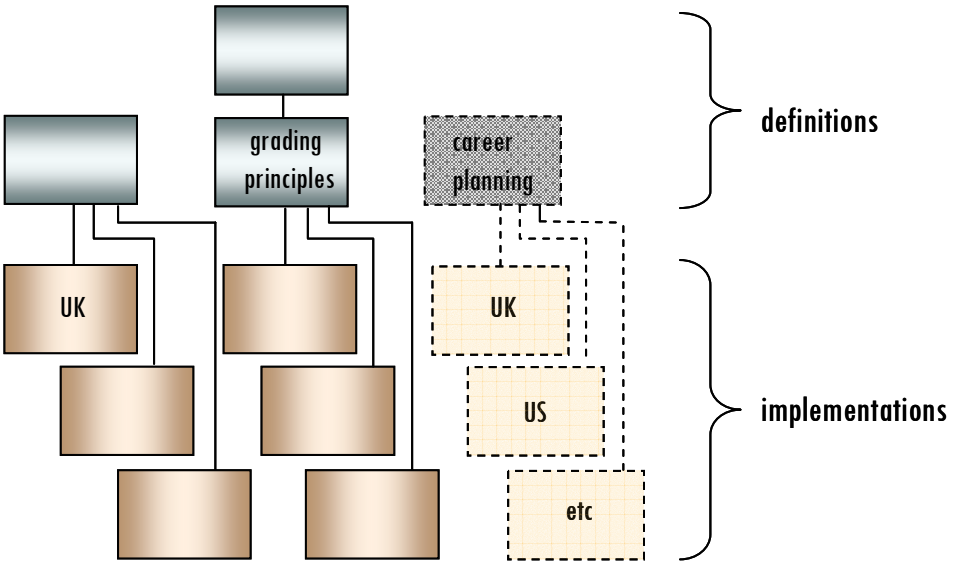


Figure 11 - Bridge pattern 'before' diagram

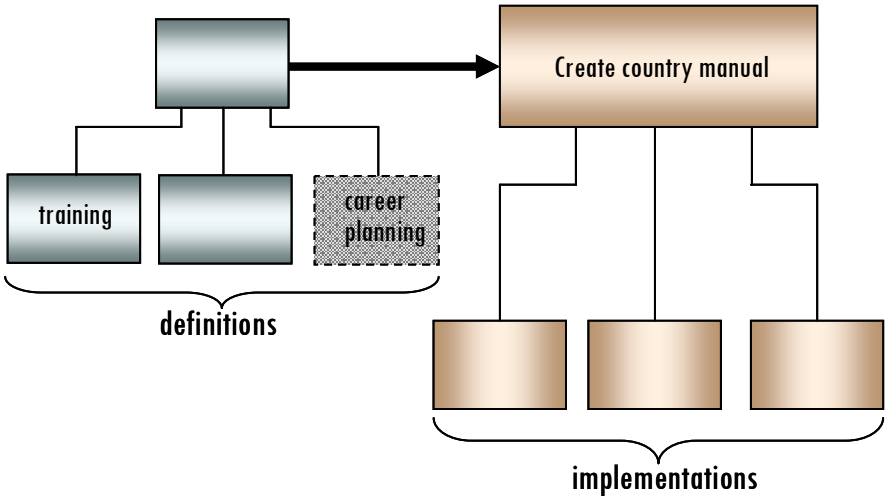


Figure 12 - Bridge pattern 'after' diagram

Object-orientated design notes
The diagram below shows how to use Bridge.

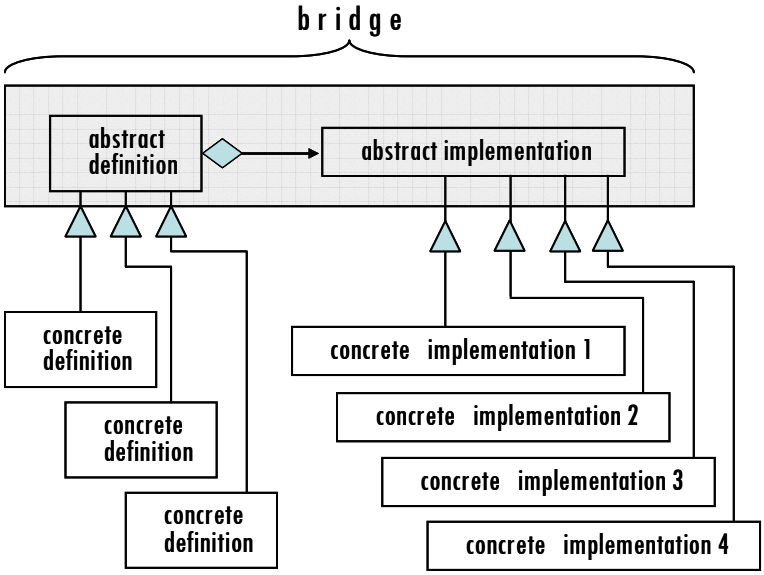
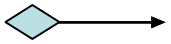


Figure 13 - Bridge pattern OMT

The  symbol is described on page 65.

Deploying buffers

We said earlier that these five patterns are not the only ones which can be used for describing organization structures and not even the only ones which might be employed as buffer patterns, but they are the most useful ones. So exactly where do we deploy them? We could conceivably buffer *every* business process and its supporting organisation. But buffering has a cost:

- business processes would need additional bridge processes (buffers) between them instead of one process feeding seamlessly to the next

- it may need more staff. A team 'fine-tuned' to operate one process or a set of processes may need extra staff to handle the buffer itself. For example, a customer services team which was set up to handle orders only from the sales force may need disproportionately more people if it is to handle orders from retail customers or wholesalers or via electronic data interchange as well *in a transparent way* and maintain the same quality of service. In other words, setting up the organization and processes to handle

any source of order *may* cost more than creating dedicated teams to handle each type or order.

Since buffers are only of value if the processes or organization change, it sounds sensible to use them to ring-fence processes or teams which are more likely to change and to leave other more static areas alone. This, to readers of *The Robust Organization* at least, should look suspiciously like Highly Optimized Tolerance…

CHAPTER 5

BUFFER PLACEMENT

Introduction

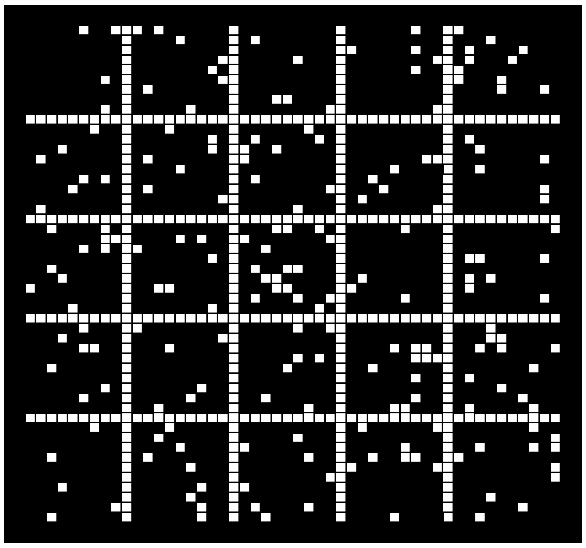
The previous chapter described the most common types of buffer pattern. It concluded by noting that inserting buffers between processes and between organizational groups such as small teams had a cost: the buffers were themselves additional (but small) processes which may introduce some inefficiency, and the resulting structure may need more staff. We thus need some rules to determine where it is cost effective to insert buffers and where it is not. More precisely, we want a way to specify where buffers should be placed based on an analysis of risk – what the likelihood is of a process or team being affected by any change which would result in its interfaces to other processes or teams altering significantly. This is exactly the type of problem Highly Optimized Tolerance addresses.

Highly Optimized Tolerance (HOT)

HOT is described at length in *The Robust Organization*. What follows is a brief summary which uses the same forest fire example.

Most forests which are left in their wild state – not managed in any way – will occasionally experience forest fires. These fires burn until either a natural firebreak is encountered (perhaps an area left fallow by a previous fire) or the forest is totally gutted. Trees re-grow more or less at random through self-seeding from the remaining trees. Other things being equal, a forest which is densely wooded is more likely to experience a large fire, one covering a wide area, than a forest which is sparsely wooded because the fire in the dense forest can jump easily from tree to tree with no gaps to hinder it. There is thus a balance between the tree density and impact of a spark: the more trees in any one area, the more likely it is that a spark will have a widespread impact.

Forests used for commercial lumbering on the other hand have firebreaks deliberately constructed. Firebreaks have a cost, not just of initially felling trees and subsequently keeping the firebreak clear but in lost revenue: each firebreak means fewer trees to harvest. The forest manager thus needs to balance the commercial *yield* from the forest – the cost of creating and maintaining the firebreak plus the lost revenue from keeping areas fallow when they could contain valuable trees – with the revenue *loss* resulting from a fire if one took place. If sparks were equally likely to occur in any area of the forest and this likelihood were known, the positioning of firebreaks is relatively easy to calculate. A square forest would have a rectangular grid of firebreaks looking something like that shown in the diagram below. The light areas are parts where there are no trees, either because there is a firebreak or because a tree has yet to grow there (perhaps it was burned down in a previous fire and its site has not yet been reseeded).



This diagram shows a forest where sparks are equally likely to happen anywhere. There is no guarantee that if a spark occurs, a fire will inexorably follow; the spark may hit a vacant site or even a firebreak.

However, if sparks are concentrated in particular areas of the forest (i.e. the distribution of sparks is not random), then it is clearly better value for money to place firebreaks closer together in those areas where fires are more likely to start and to space them widely elsewhere. For example, assume that there is a picnic site at the centre of the forest and that sparks from careless picnickers are thus more likely in the neighbourhood of the centre than elsewhere. The optimum spacing of straight-line firebreaks would then look something like that shown below, although there are other ways to construct firebreaks which are not straight lines. In this diagram, the centre of the forest is closely ring-fenced by firebreaks. A fire breaking out there cannot spread very far. The corners of the forest, on the other hand, have been assumed to be areas where sparks are relatively unlikely to occur. Creating firebreaks in this way maximizes the yield for a particular distribution (likelihood pattern) of sparks.

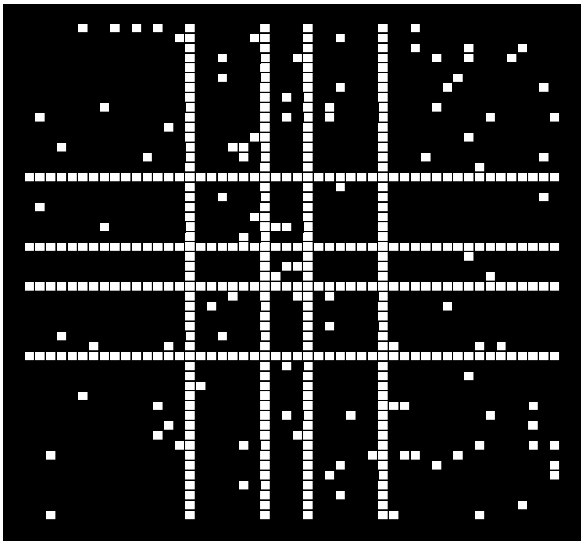


Figure 14 - HOT with a. equal and b. centred probability of sparks

However, if a spark hits one of the corner areas – which is possible but much less likely than one hitting the central area, the damage is much greater since there is more forest to burn between the wider-spaced firebreaks than in the centre.

More generally, HOT has three characteristics:

- design is used to apply a resource (firebreak) such that the *overall* yield is maximized (which is normally the same as minimising losses). The resource is either limited or has a cost associated with it which offsets the value of the yield: applying too much resource can reduce the yield

- the resource reduces the total losses sustained as a result of some external event (spark). These losses may be caused by a chain reaction of the initial event (an external spark ignites a tree) causing other events (fire spreading to neighbours)

- the external events happen with some known probability distribution (some areas of the forest may be more likely to receive an external spark than others)

One consequence is that the greater yield (average tree density) renders the forest more vulnerable to *unanticipated* (rare) external events. But the HOT forest is also the most robust for the particular amount of resource deployed. *And 'robustness' here is simply a measure of how stable the yield is in the face of anticipated risks.*

Buffer placement

This robustness is exactly what we are seeking for deployment of process or organizational buffers. Simplistically, we can:

- identify the major areas within the business which have historically been most subject to change, or which, with knowledge of the business's own strategy and what is happening to competitors, will be most likely to change

- within each such area, rank the business processes or organizational groups in order of likelihood of change

- define suitable buffer patterns for each business process or organizational group

- evaluate the cost of implementing and operating each buffer and estimate the cost of disruption if the typical changes actually occur

- implement buffers for those business process or organization groups for which they are cost effective

Anyone familiar with HOT may detect two subtle differences between HOT's formulation and what is proposed here. HOT uses the likelihood of an external event such as a spark occurring (which *may or may no*t have consequences such a fire) whereas we have ignored any root cause of change and simply estimated the likelihood of the change happening. In addition, HOT tries to position barriers such that the *overall* yield is maximized, whereas in this example we are notching up benefits area by area. In our context, fortunately, these differences are irrelevant.

CHAPTER 6

FROM IT TO ORGANIZATION

Introduction

The use of patterns and decomposition in object-orientated design and programming has been plumbed in depth since 1995. The converse – the use of ideas developed for systems architecture for designing organisations – is, however, an almost virgin field. In answer to the unspoken 'why bother', it is worth noting that computer operating systems such as Windows XP and their related network technology are arguably the most complex artefacts ever designed. Reproduction and natural selection together have certainly created more complex living forms, but computers and networks are *designed*. Most of the problems faced by those who are redesigning the structures of their businesses have already been faced, generally successfully, by IT practitioners.

IT practitioners also learned one lesson many years ago: to avoid monolithic (all in one piece) systems, and this applies to business application systems as well as computer operating systems. Since the message of this book and its predecessors is decentralization, or at least the avoidance of over-centralization, it is worth looking at what was wrong with the original monolithic systems.

There were four fundamental issues:

- size
- multifaceted nature
- impact of failure
- complexity

And to make life more difficult, these were found to be interrelated.

Size on its own is not inherently a problem. Designing large things just takes longer or needs more designers than small things. But the science, or rather art, of estimating how long a new operating system would take to build and test is embryonic. IBM faced this on a grand scale when it tried to design from scratch an operating system for a complete range of computers suitable for anything from a tiny office to the largest corporation or science research establishment. The initial result, OS/360, eventually worked and derivatives are still in use today, but the delays were severely embarrassing to the world's then largest computer manufacturer, the cost overruns were frightening, and the product was highly unreliable at the outset.

It was found that there were simply not enough technical and project management people available anywhere with the right level of experience. Designing and writing a computer operating system is not like designing and building the steelwork shell of a skyscraper, where one floor is very much like the one below and design and construction are largely sequential and repetitive. Once engineers and

construction staff have designed or built one floor, they simply do the same thing one floor up. In other areas, working in parallel to speed things up is possible. Railways, for example, are built this way, but perhaps the best and most relevant example is the creation of mathematical tables before computers were invented. The world of tables has largely disappeared, but at one time they were indispensable for tradesmen, builders, designers, actuaries, bankers and, most notably, navigators to whom accurate astronomical tables were essential. A mathematician would devise a formula and break the evaluation of it into simple discrete steps. He or she (almost always a 'he') would then calculate some 'pivotal' values – the formula evaluated at well-spaced intervals ('every 100', say). Filling in the gaps would be farmed out to people known as 'computers' who would undertake the very large number of simple and tediously repetitive calculations necessary either by hand or using a simple mechanical calculator. Calculations would normally be done in duplicate by different people and the results cross-checked. Until the final printing, therefore, when results were collated, it was possible to calculate the values needed for large tables quite quickly using lots of human 'computers' working in parallel. Writing the programs which comprise a computer operating system like OS/360 is a totally different process. In general, each piece is different in nature from each other part; very little is repetitive. It is *multifaceted*, and this makes design and writing take a lot longer as there are no economies of scale.

Complexity

OS/360 was, for its time, large and multifaceted, but it was also *complex*. Much of it was one large chunk of programming. This was customised on first installation to suit the computer and devices connected to it, but the result ran as one piece. This meant that failure in any one line of programming could bring down the entire system rather than just abort the function being undertaken. For example, a fault in the part of the system which dealt with sending lines of print to a printer could abort not just printing but everything else as well. It was only much later (with MVS – loosely a grandchild generation of OS/360) that each major part of the system was isolated such that any failure there would be dealt with by failure management programs written specifically to cope with failures in that area. As far as is known, the additional lessons from the development of Highly Optimized Tolerance to ring-fence areas during design to a degree proportionate to the *likelihood* of a failure has not yet been incorporated into any computer operating system, although Microsoft are aware of it. The source of the complexity was only realised later: although the very many programs which comprise OS/360 were designed to link to each other (where necessary) via formally-documented interfaces which specified what information would be passed from the caller to the program being called, little or no effort had been made to design things such that the caller was prohibited from finding out what the program being called actually did; it could and often did peek into the called program's private information or make assumptions about how it worked or both. This was bad practice at design time but often fatal when changes were made to the called program. These unofficial 'cross-connections' between programs could lead to knock-on effects when the called program then called yet another one. These side-effects are a hallmark of *complexity*: instead of a simple controllable hierarchy where

program A calls program B to do something on its behalf without knowing – *without being able to know* – how it does it, we have a skein of cross-connections whose results are unpredictable.

Large multifaceted systems, particularly computer operating systems and networks, use precisely-specified interfaces between their thousands of constituent parts. Furthermore, these interfaces are 'layered' in the sense that program A links to program C via program B and has no idea how to talk directly to program C or how program B does so. Neither does A know how B or C work. Networking and especially router technology was touched on in *The Coevolving Organization*. It is a fertile source of the best examples of layering (the OSI seven-layer model, for example) but also contains something more subtle which as far as is known has not been covered elsewhere before: the dynamic (time-based) nature of interaction between objects when they are constrained. For example, if objects W, X and Y are each coupled to object Z and are interacting with it, Z may be unable to respond to Y because it is too busy responding to W and X which either got in first or are of higher priority. This has close parallels with how traffic is managed over constrained communications links where data packets are expedited, re-prioritised, delayed and sometimes deliberately dropped.

Network routing

Data traffic from one site to another is sent and received using items of equipment called 'routers'. Routers can if necessary pass data packets from point to point over many individual links until they reach their eventual destination. They handle transient errors and reroute traffic if a link fails. Routers need to exchange information on how to get from A to B when several links are involved (for example, A to X; X to Y; Y to Z and finally Z to B). If an individual link fails, routers directly connected to it pass the word on to other routers ("avoid link X to Y – it is faulty; try another way around"). Since this exchange of information between routers is itself data traffic and may take some time to percolate around a large network, it is possible that the failing link may right itself again before the information about its failure had arrived at the farthest reaches of the network. There will then be contradictory messages ("link X to Y is faulty" and "link X to Y is OK") circulating at the same time which, in a mesh (any to any) network can cause a storm of conflicting information to fly between routers.

The Coevolving Organization described a fundamental problem faced by all network designers: whether to split a network into autonomous chunks so that such 'broadcast storms' can be contained within their chunk of the network (which then makes the network of limited use to those who want worldwide communication) or to stay with a single network and risk such disasters which have a high impact but are relatively rare. It also described the usual compromise: to create freestanding areas and then link them together at one or two points on the boundaries that separate them. The routers in area A would then contain a map of the links in area A alone. Any links in another area B would be invisible from within A. All that a router in A needs to know is that any packet of data addressed to a destination somewhere in B has to be forwarded to a special router on area A's boundary. This boundary router would

then take responsibility for sending it to its opposite number in B that would be fully up to date with what routes in B led where.

Some communication of network information across the areas has to occur. If not, a router in A would not know which destinations lay in B. But information about what *links* lead where in B and which ones were currently operational stays confined to B. Routers in area A will discuss link availability with each other. Routers in B will do likewise. But this will not happen between a router in A and a router in B. A big failure in one area will have limited impact on another area. Both data and the information about link availability can flow uninterrupted around A even when B is struggling.

Since this looks like a good solution, it raises the question of whether we should create more areas like the creation of the progressively smaller and more numerous cells used by mobile phones in urban areas where the density of phones is high. This, however, introduces problems of its own. The fewer the points of interconnection between A and B the greater the dependence on the availability of the boundary routers (and the links between them) that look after all communication between A and B. What we have gained in resilience *within* each area we have lost in the connections *between* areas. In coevolution terms, the areas are objects. The links between boundary routers give the C-coupling between areas. The (average) number of links between routers in any one area gives K. The effects of a temporary technical problem – perhaps information about a link failure – which occurs in a high-K area reverberates around the whole area in an unpredictable way. If the routers in an area are connected in a hierarchy or in the extreme case a simple low-K star with each link connected directly to the boundary router, this impact of network failures is confined. But now the system has become more vulnerable to a failure at the centre of the star. Managing a star network is easier than managing a mesh. Such a network is very resilient to failure outside the centre but a failure at the centre itself can have a catastrophic impact.

The Internet

Throughout the 1990s, the Internet appeared to be an archetypal example of a system which had evolved ‘naturally’ like a biological system in response to user demand rather than having been formally designed. Voluminous data on its physical structure and performance are available and these data show the ‘power-law’ signatures of self-organization (see *The Coevolving Organization*). But although the Internet has no central control and the traffic patterns may appear to adapt automatically to congestion or failure of a link without intervention by the user or even by the communications link supplier, it now appears likely that this power-law behaviour is a consequence of the vast amount of design for both performance and resilience which has gone into the Internet’s TCP and IP communications protocols and their physical implementation in routers and is not a natural consequence of the self-evolution of the Internet. In other words, the Internet’s apparent self-organized behaviour is a consequence instead of network designers attempting to optimize link usage while minimising congestion and minimising the impact of failures on the Internet as a whole. Inevitably, these designers tried to ensure that the impacts of

outages at the most likely points of failure were contained. So instead of being a self-organized system, the Internet looks like an example of HOT.

Private communications networks and the Internet are thus both examples of designed systems rather than ones which 'just growed like Topsy'. As noted above, the structure of both private networks and the Internet will have areas where the routers at each site know of the existence of each other site and how to contact them directly but outside which communication is only possible via intermediary 'boundary' routers. And if the design is done well, the sizing and positioning of these 'autonomous networks' and the way in which they are coupled using boundary routers would have been done only after careful evaluation of the likelihood of failure at different points in the network and the impact of such as failure on the entire network. The designer would attempt to minimize the network-wide effect of likely failures subject to the constraint that having too many small autonomous networks can reduce the reliability of the network. This is a result of traffic *between* areas travelling via a few critical boundary routers and their associated links. Furthermore, resilience is reduced because there are fewer ways for traffic between areas to be rerouted.

Business processes

This same principle can be applied when structuring business processes and their associated organizational groups. Breaking the processes into many discrete areas which are buffered using one of the buffer patterns described earlier can make transaction flow between processes highly dependent on the availability and performance of the buffers themselves. Too many buffers can thus unintentionally create artificial points of congestion and failure. Too few – particularly at the points where change is most likely – subjects the organization to the internal chaos which buffering was intended to obviate.

The Coevolving Organization described what happens in a real organization – a collection of general practitioners' (family doctors') practices – when the normally independent practices combined their power to buy services from a particular hospital. If, when the practices were separate, practice A pushed hospital X to drop its costs for a particular surgical procedure and practice B did the same *but not at the same time*, the hospital may find different ways to make the economies demanded by each practice. It has time to react to the first demand before responding to the second. Its link (C-coupling) back to practice A may result, for example, in an increase in costs for practice A elsewhere in its budget, like the boxer riding a punch and coming forward again. But when practices combine their C-couplings, the result is similar to the effect on a boxer being hit by several punches *at the same time and in the same place*. Merely adding C-couplings together may well understate the resulting impact on the recipient because the couplings now act in a coordinated way and make the same demands, volume discount for example, at the same time. This co-ordination comes via the C-couplings *between* the practices. So the net impact of links between areas can be more complex than is at first apparent. The impact of a C-coupling 'push' from two or more 'attacking' objects to a target object depends on the time lapse between the respective pushes. It is greatest when impacts coordinated by C-

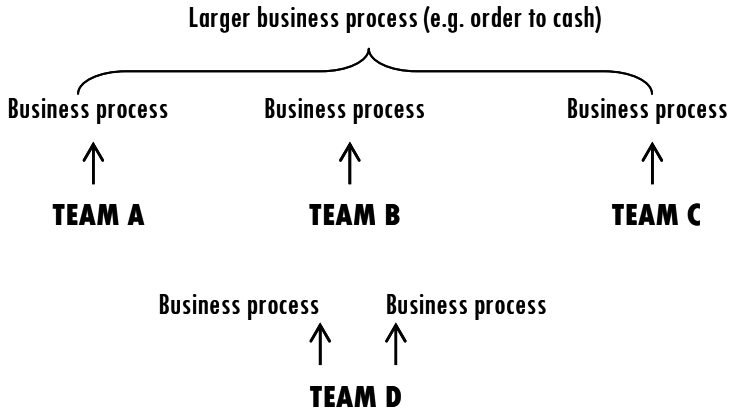
couplings *between* the attacking objects enable pressure to be applied to the target object *at the same time*.

But what of the reactions of the target object – the hospital in the preceding example? The simplistic assumption is that it will react to simultaneous impacts from C-coupled 'attackers' additively (just add up the individual impacts). But real target objects are not that simple. The hospital will have limited capability to respond if fifty local general practitioner groups all ask for different priorities or service discounts at the same time. If for no other reason, the hospital's accountants and service delivery managers will be unable to respond to all the requests at once because they themselves form a bottleneck. Communications network designers are familiar with this very problem – data packets arriving internally at a site's router for delivery to another site do not normally arrive at a predicable steady rate. Instead, they arrive in bursts which contain data from different users working independently. There is fortunately no person-to-person C-coupling, or the impact if everyone conspired to send large quantities of data at the same time would be a solid traffic jam. Nevertheless, the traffic is targeted at a device (the router) which is the gateway to a communications link with a restricted capacity. In such circumstances, the router's job is to prioritise, delay and sometimes even drop data packets such that the link capacity is used to best effect.

Programs and teams

The Coevolving Organization called each organization entity, a department for example, an *object*, although the reason may not have been apparent at the time. Let us equate each such organization object with a computer program which is part of, say, a computer operating system. If an object (customer services, say) makes assumptions about how another object (logistics, say) which is its 'internal service supplier' (the supplier of warehousing and delivery services to customer services) will fulfil its 'supplier' contract, then any change in the logistics organization can have a knock-on effect on customer services, irrespective of the formal business processes they both adhere to. The same is true if logistics makes some assumptions about orders sent to it by customer services for delivery. Perhaps customer services had been in the (laudable) habit of checking that manufacturing had sufficient work in progress which will result in enough manufactured stock being available for a delivery next week. If customer services cease doing this, perhaps because the individual concerned moves to another role or because the team is reorganized, logistics will suddenly find they have stock shortfalls for no apparent reason.

If computer programs can be equated to organizational entities – objects, what is the equivalent of the business processes that the organization (i.e. the supporting teams) tries to correspond to? The short answer is that programs also correspond to business processes. (Note that we are *not* necessarily talking about the programs, perhaps part of applications systems such as SAP AG's R/3, which are used to automate the business processes.) This imprecision arises from the fact that a high-level business process is built up from smaller processes, and that supporting staff may be organized into teams which cover sub-processes within the high level process, or alternatively more than one process – as illustrated in the diagram which follows:



And how do we define where the boundaries of either the business processes or the supporting teams should be? By:

- identifying what are the smallest units which are most autonomous, i.e. most independent of their peers. They are only connected to their parent in the hierarchy which is either an organization parent (a site asset management accounting team's being part of the country Finance organization) or a business process parent.

- identifying where likely changes will occur, either in business process or in organization (there may be tentative plans to outsource IT Service Delivery for example)

When we know the boundaries, we use HOT principles to insert buffers where they are most cost effective.

CHAPTER 8

REFERENCE MATERIAL

Patterns and wholeness

Chris Alexander (references **4, 5** and **6**) was the first to give an analytical exposition of why buildings and collections of buildings "don't work" – why they often do not function as intended and why they are unpleasant to inhabit. His starting point was to analyse how abstract 'things' – which may be supporting or conflicting – interact, and how misfits between these 'things' and their environment can be minimized. Alexander's work spawned considerable interest from other areas, notably object-orientated software design (see *The Coevolving Organization* Annex – Information Technology). Appendix 2 of reference **4** contains the proof of a highly relevant theorem: "given a system of binary stochastic variables, some of them pair-wise dependent, which satisfy certain conditions, how should this system be decomposed into a set of subsystems such that the information transfer between the subsystems is a minimum". The significance of this to designing an organization should be readily apparent to readers of *The Coevolving Organization* (see Chapter 4 – How big should an object be?): one design criterion for selecting coevolving objects is that they *naturally* communicate between themselves as little as possible (i.e. communication needed by business processes is primarily *within* objects). If this is not true, the carving up of the business into objects has been done wrongly and there is a better way to do so which concentrates communication *within* objects and reduces it *between* objects. One can (loosely…) apply the formulation of HOT PLR (see *The Robust Organization*): if we have a fixed maximum number of barriers between business areas, we want to place the barriers such that the communication between areas (i.e. across the barriers) is minimized relative to any other way of placing barriers. Alexander introduced the idea of 'patterns' (in reference **5a**) which can be used at a local (decentralized) level to create structures – which in our case are the internal processes of organization units – each of which has the most appropriate fit for its purpose.

Alexander's best-known work (reference **5b**) describes 253 patterns which could be used to create building and spaces which are 'alive' – meaning that they fulfil their function but more importantly that the inhabitants 'feel at home' in them, something difficult to quantify but very real to the inhabitants themselves. This book is one of a three-part series. The first (**5a**) describes the origins of patterns, pattern languages and pattern pools and is the best place to start – particularly for those who aren't architects but are fascinated by Alexander's ideas. The third book in the series (reference **5c**) covers in great detail the implementation of Alexander's ideas in a large-scale design process for the University of Oregon. Alexander's later series of four books (references **6a** though **6d**) takes things much further. The first (**6a**) revisits the need for a successful building to be 'alive'. It characterises this 'life' as the way in which certain features of buildings have an innate connection to human feelings. Alexander proposes that this life is the result of using up to fifteen basic geometrical

forms to create the 'wholeness' of a structure. This, in turn, engenders the subjective feeling that these structures are 'right'. In other words, what makes good architecture – architecture which people feel 'easy' with – *is amenable to analysis*. In Alexander's words (page 236), "Systems…which have these fifteen properties to a strong degree will be alive, and the more these properties are present, the more the systems which contain them will be alive". The second book (**6b**) builds on the first and demonstrates how simple evolutionary processes resembling natural growth – 'structure-preserving transformations' – can be applied to these forms to create new structures or to flesh out and enhance existing structures. These transformations are, in fact, 'active' versions of the geometric forms themselves. In other words, each geometric form is used bootstrap fashion to grow itself and to assist the growth of other forms. The bootstrapping process is applied across the embryonic structure in a ten-step iterative sequence which enables the burgeoning forms to evolve with their neighbours in a coherent way such that the 'wholeness' of the structure, and hence its effect on the feelings of its inhabitants, is preserved and enhanced. The third volume (**6c**), which has not yet been published, describes a large number of 'living' buildings and spaces designed by Alexander and others. The final book (**6d**) is a deep and often mystic reflection on the more fundamental issues of consciousness, the nature of self and, above all else, wholeness – the indivisibility of self from the outside world. Alexander summarised the relationship between his Nature of Order and current complexity theory in reference **8**.

Object orientated design

The Gang of Four's 'bible' (reference **1**) is the standard textbook on patterns for object-orientated design. Like Alexander's *Notes*, it started life as joint-author Erich Gamma's PhD thesis. It contains 23 patterns grouped into 5 creational patterns, 7 structural patterns and 11 behavioural patterns. A few (such as Adaptor) apply mainly to classes but most apply to objects. The difference between the two is roughly the difference between a design handbook or blueprint (which, after design is complete, are fixed) and real-life operation where objects can invoke the services of other objects in a dynamic and unpredictable fashion.

	Class-type pattern	**Object-type pattern**
Creational	Create objects using subclasses	Create objects by using the services of other objects (none of the five buffer patterns are in this category)
Structural	Compose classes using inheritance (Adaptor[1] is an example)	Define ways to assemble objects (Adaptor, Bridge and Façade are examples)
Behavioural	Define flow of control or a process using inheritance	Describe how several objects work together to perform a task which no single object can perform (Chain of Responsibility and Mediator are examples)

[1] Adaptor appears twice in this table as it can be used as a class pattern and as an object pattern – as illustrated on page 34

Those without an IT background or unfamiliar with object-orientated programming and the box-and-line diagrams of Object Modelling Technique (OMT)[2] used earlier to illustrate the class and object relationships for the five buffer patters may find reference **1** hard going. If so, reference **2** provides a slower-paced introduction which explains how using patterns can solve some of the problems (such as huge inheritance trees) caused by using object-orientated design slavishly.

[2] or its successor Unified Modelling Language (UML). We have used OMT for consistency with the Gang of Four's "Design Patterns"

CHAPTER 9

QUESTIONS AND ANSWERS

Q: You showed earlier that there were two steps to creating good organization groups. The first simply minimized interaction between each team and other teams (following Alexander). The second permitted some artificial organization design constructs – façades and the like (following the Gang of Four) – whose aim was to minimize further the knock-on effects of changes within teams. This sounds somewhat familiar…

A: It should not have escaped readers of *The Coevolving Organization* and T*he Robust Organization* that there is a strong analogy between:

- 'edge of chaos' – the optimal point to which to decentralize if we are restricted to using simple more-or-less random changes within an organization, and
- Alexander-like simple minimization of interactions between teams

and also between:

- 'highly optimized tolerance' which allows the edge of chaos point to move further in the direction of chaos (and thus be more optimal) if we are allowed the freedom to impose artificial designs on the organization, and
- the object-orientated artificial organization constructs

In other words, if we know roughly how an organization reacts (via its business processes) to changes, whether external (attacks from a competitor, for example) or internal, and in the light of this knowledge apply deliberate *design* to how processes and the teams running them interact, the more successfully it can operate its linked series of business processes without major disruption when a foreseeable change occurs to the business processes. We would identify areas of likely variability in advance and create façades and bridges to buffer processes from each other. This does, of course, leave the business exposed to unlikely changes. The buffers are equivalent to the HOT firebreaks which are placed to isolate areas most likely to be hit by a spark at the expense of other areas where sparks are much less likely. In business process terms, stable areas – ones less likely to suffer radical process change – are left unbuffered. This makes the effect of an unanticipated change greater because the business processes remain tightly coupled and the knock on effect of a change is more far reaching.

Q: I'm an architect and I don't fully buy your argument about splitting things into pieces which are as autonomous as possible. This is how urban planning worked twenty years ago – and to some extent still does – creating isolated groups of houses and shops connected by major roads. Superficially fine and 'clean' on a design plan, except that people don't live in this artificially segregated way.

A: Correct. And this is also true of how people actually work in organizations, where the patterns of communication and, in larger offices and campuses, the patterns of people movement are a complex set of overlapped semi-autonomous groups. Some groups are, indeed, driven by business processes and the organization structure supporting them (i.e. the 'official' family tree). Other overlapping groups emerge from cross-area task forces, matrix management and social ties. Instead of a tree structure, the result is a 'semi-lattice' – a tree in which each leaf can be attached to more than one twig, and each twig to more than one branch and so on. Alexander highlighted this in a paper (reference **7**) which was shunned by the 'keep it clean and simple' urban planners who felt it spoiled their elegant but unworldly designs.

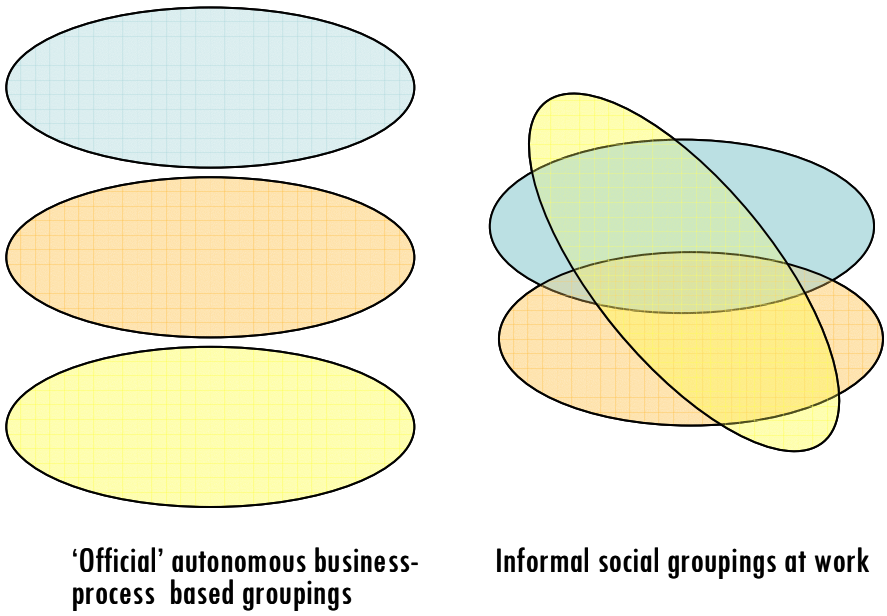


Figure 15 - Trees and semi-lattices

Q: You said that different objects can have identical interfaces but are entitled to act differently in response to identical requests. But you also highlighted the similarity between the collection of requests which can be presented to an object and a Pattern Language. Does this mean that different patterns mean different things to different people?

A: We said earlier that the collection of all valid requests to an object is called its interface. The different formats of requests are called 'signatures', so an interface is a collection of signatures. Signatures may naturally group into subsets. To use the example of drill-instruction, "Quick march", "Squad halt", "Left turn" "Right dress" "Change direction right – right wheel" and so on are a collection of marching-related drill tasks. Let us call this group of tasks 'March-type'. There may be another which is only relevant to the armed infantry called 'Arms-type' (such as "Present arms"; "Slope arms"). A squad of infantry will respond to both March-type and Arms-type commands (its 'interface' will consist of 'signatures' of the March-type and of the Arms-type.). On the other hand, British cavalry, who are the most reactionary element of the British army and in 1914 were still using horses and lances[3], would respond to commands of "Gallop", "Quit and cross stirrups" and the like, commands meaningless to any other group of soldiers. Each group of related signatures (related commands) is called a **type**. The same type can be used by different objects, and each different object is entitled to respond in own way.

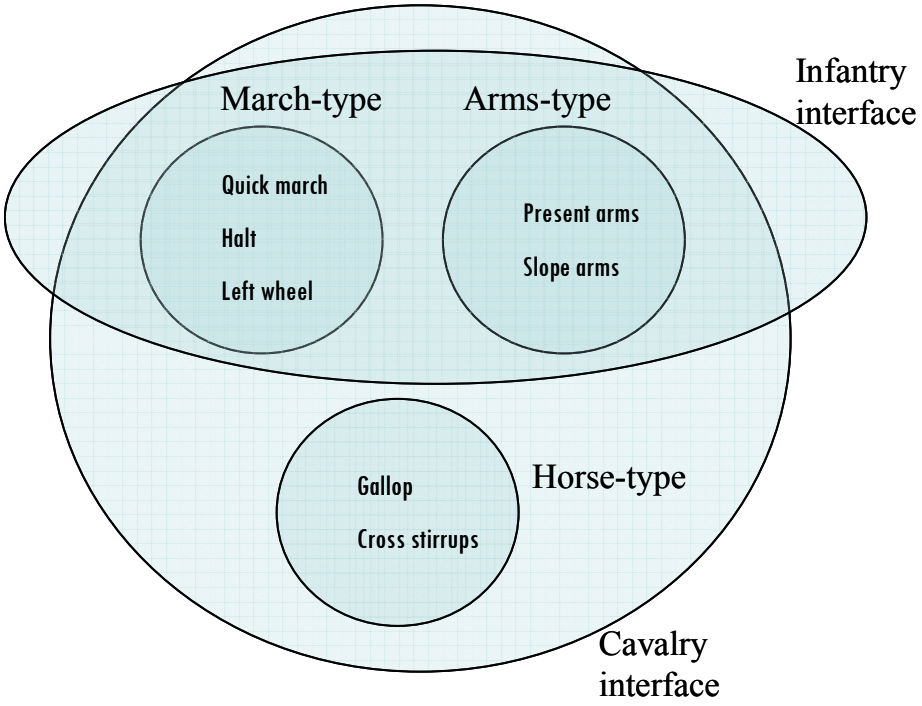


Figure 16 - Military commands form a language

A more precise comparison with Alexander's pattern language concepts is that the collection of all commands for all armies is similar to a *pattern pool*. Each command

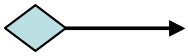
[3] the last British lance-versus-lance attack occurred on the 7th September 1914 when Lieut. Col. David Campbell charged with two troops of "B" Squadron of the 9th Queen's Royal Lancers and overthrew a Squadron of the German 1st Guard Dragoons. The 9th, who were founded in 1751, did not give up horses in favour of light tanks until 1936…

is similar to a pattern. Each command may be responded to somewhat differently by different troops depending on the context (nationality; position of other troops and buildings and so on) but it will always be responded to sensibly and in a recognisably similar way. The collection of all commands which are responded to by a particular interface is similar to a *pattern language*. If the cavalry *and* infantry of a national army both respond to the commands relevant to them in the same way (they both 'Quick march' in the same way, for example), one could instead regard a national army as having a pattern language, with the minor variations between units (for example, the speed at which they 'quick march') being regarded as variations due to their context.

Note, however, that an object on its own is not a pattern: we need to specify (as a minimum) its context – which almost certainly will include other objects, the forces which are resolved when we use it (i.e. our success criteria), and the outcome of using it. This would (or should…) have been documented in the manual for troop training which should be the drill-instructor's bible. Historically, the infantry soldier would be given none of this extraneous information, and his response to words of command would have been to obey without question; he would have behaved like an object (!) whereas the drill movement itself was akin to a pattern. Drill movements are linked together into larger movements: the spectacle of Trooping the Colour which is beloved of visitors to London and held on HM The Queen's official birthday in June is a complex drill pattern composed of numerous individual smaller drill patterns which have been adjusted to fit within the geographic confines (context) of Horse Guards Parade in Whitehall, Central London.

Q: I've just completed an object-orientated design course and the box-and-line diagrams you used to illustrate the class and object structure of the buffer patterns are wrong! The subclass-to-class lines shown as  are OK, but the class-to-class lines which instantiate an object are misleading. You show them as solid lines like  but shouldn't they be shown as dashed lines like  ?

And what about that strange shaded diamond one used in the Bridge pattern?



A: Ah…an unsuccessful attempt to simplify OMT diagrams. Lines with solid black arrows at one end have been used as a general indication that one class instructs another class to 'do something' – usually 'create an object'. (In OMT, one object calling another is indicated by a dashed line.) A solid line with an arrow at one end indicates that the calling class keeps (maintains within itself) a reference to another class. The shaded diamond at the far end of an arrow in the Bridge pattern indicates that the object at the 'diamond' end is an *aggregation*[4] of objects at the other end (for example, a car is an aggregation of one or more wheels). In the Bridge pattern, aggregation means that the 'abstract definition' does not merely know about the

[4] beware: the Gang of Four use the terms 'composition' and 'aggregation' in exactly the *opposite* way around to that defined in the more recent Unified Modelling Language (UML)

existence of the 'abstract implementation' but contains it and is responsible for it, in the way our electric locomotive is composed of (among other things) a large electric motor. Neither locomotive nor motor has an independent existence. This is an example of *object composition*: a way to avoid having very deep class hierarchies by splitting the hierarchies into separate groups of classes and then letting one class reference the other.

Q: My object-orientated design course made great play of clustering design elements which were basically alike into common families. You took the HOT approach. Why can't commonality analysis be used to group processes together

A: HOT decides how much resource (firebreak; buffer) to apply and where to apply it using the probability of external events (sparks; organizational or business process change) happening. IT system designers have a similar problem: how to structure systems such that the impact of subsequent change is minimal or at least contained. This usually implies that the side-effects of a change are minimal and well-understood. Jim Coplien in his PhD thesis (reference **3**) described one way to achieve this:

- decompose systems into families of items which have commonality (i.e. which naturally cluster together because they have common elements, but are not identical), then…
- within each family, identify what makes each item different (i.e. identify variability)

Each family then forms a class hierarchy with variation becoming more pronounced as we move down the hierarchy towards the final (concrete) class. Commonality/variability analysis can, in principle, be applied to any system but is most suited to software design.

.

BIBLIOGRAPHY

Books

1. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. "Design Patterns - Elements of Reusable Object-Oriented Software" (Addison-Wesley 1994)

2. Shalloway A. and Trott, J.R "*Patterns Explained*" (Addison-Wesley 2002)

3. Coplien, J.O. *"Multi-paradigm design"* (PhD thesis for Free University of Brussels - 2000)

4. Alexander C. "*Notes on the synthesis of form*" (Harvard University Press 1964)

5. Alexander C. "*Centre for Environmental Structure Series*" (Oxford University Press)
   5a. "*The timeless way of building*" (1979)
   5b. "*A pattern language*"[5] (1977)
   5c. "*The Oregon experiment*" (1988)

6. Alexander C. "*Nature of order*" four-volume series (Centre for Environmental Structure 2003)
   6a. "*The phenomenon of life*"
   6b. "*The process of creating life*"
   6c. "*A vision of the living world*" (yet to be published – as at October 2004)
   6d. "*The luminous ground*"

7. Alexander C. *"A city is not a tree"* (in two parts: part 1 in Architectural Forum Vol 122 No 1 April 1965 and part 2 in Vol 122 No 2 May 1965)

8. Alexander C. *"New concepts in complexity theory"* (www.katarxis3.com - May 2003)

9a. Stewart M. "*The coevolving organization*" (Decomplexity Associates 2001)

9b. Stewart M. *"The robust organization"* (Decomplexity Associates 2003)

9c. Stewart M. *"The emergent organization"* (Decomplexity Associates – to be published)

[5] with Sara Ishikawa and Murray Silverstein

INDEX

A

A Pattern Language, **13**
Alexander, Chris, ii, 1, 59

B

Balanced Scorecard, ii
barriers
  positioning to maximize yield, 49
buffering, **13**, **29**

C

C-coupling, 54, 55
  co-ordination of, 55
  strengths of, 1
classes, 22
coevolution, 54
commonality/variability analysis, 66
communication, 54
  data (routing of), 53
computers
  human (for calculating tables), 52
configuration (of a patttern), 4
context (of a pattern), 3, 7
contexts
  hierarchical, 8
Coplien, Jim, 66
criteria
  interdependence, 9
cross-connections, 52

D

design, i, 49, 54, 59
  object-orientated, 16
  of IT systems and networks, 51
design patterns, 2

E

economy, 55
edge of chaos, i, 62
ensemble (of a pattern), **7**
EOC, i

F

firebreak, 2, 48
  in forest, 47
forces (of a pattern), 4
form (of a pattern), 7

G

Gang of Four, iii, **60**

H

hierarchy, 54
high-K, 54
Highly Optimized Tolerance, i, 45, **47**, 62
HOT. *See* Highly Optimized Tolerance

I

interface
  to an object, 21
Internet, 54

L

landscape
  (deformation of), 55
  rugged, 10
languages
  simulation, **18**

M

misfit
  in a design problem, 11
multifaceted
  characteristic of systems, 52

N

NKCS (landscape modelling), 1
Notes on the Synthesis of Form, 7

O

Object Modelling Technique (OMT), 34
objects
  coevolving, 59
OS/360 (operating system), 51

P

pattern
  Adaptor, 31
  Adaptor (main definition), **33**
  Bridge, 31
  Bridge (main definition), **42**
  Chain of Responsibility, 31

Chain of Responsibility (main definition), **39**
class and object types, 60
definition by Alexander, **5**
Facade, 31
Façade (main definition), **35**
Mediator, 31
Mediator (main definition), **37**
veneer (prototype pattern), 29
pattern language
definition, **15**
Pattern Language concept, 1
pattern pool
definition, **15**
picnic site, 48
power law, 54
process
business, 59
protocols
communications, 54

R

resilience (of a network), 54
robustness
definition, **49**
router
boundary, 53

S

self-organized criticality, i
semi-lattice (vs trees), 63
signatures (of an object's interface), 64

T

The Coevolving Organization, iv, i, 9, 16, 17, 23, 25, 26, 30, 53, 55, 56
The Emergent Organization, ii
The Robust Organization, 45, 47
theorem (binary system decomposition), 59

U

UML, 31, 61

Y

yield, 49
of a commercial forest, 47

The Pattern Organization

ISBN 0-9540062-8-3