Generative Patterns of Software

Qualifying Dissertation

Tim Hoverd

July 2008

Abstract

The adoption of design patterns as part of software development has had a profound effect. The patterns in use, though, are essentially static in nature showing different designs that might work differently in a particular context. The originator of the notion of patterns and that of a pattern language, Christopher Alexander, has also proposed that a set of *generative* patterns, which influence the development of buildings might also have analogues that are appropriate for a generative approach to software development.

This document describes both Alexander's generative patterns and the general notions of design and architecture that exist in the development of both conventional software systems and complex systems. In order to understand Alexander's patterns sufficiently, so that the hypothesis relating those patterns to software development can be properly investigated, an approach to a "field theory" of Alexander's patterns is described. A preliminary implementation of some aspects of this field concept, based on an adaptation of the Ising model of magnetic domain formation, is shown. Initial results from this implementation are sufficiently encouraging to point the way to further work in this area, leading to an improved understanding both of Alexander's generative patterns and of how such an approach might work in the context of software development, in particular for that of complex systems.

Acknowledgements

This research is supported by Microsoft Research.

I have been hugely assisted by both Susan Stepney of the University of York and Christopher Alexander of the University of California, Berkeley. My thanks to them both.

Contents

1 Introduction

The current research is part of the EPSRC[1] and Microsoft Research funded "Complex Systems Modelling and Simulation Infrastructure" (CoSMoS) project which is aiming to *"build capacity in generic modelling tools and simulation techniques for complex systems, to support the modelling, analysis and prediction of complex systems, and to help design and validate complex systems"* (CoSMoS, 2007).

The part of this project under discussion is specifically concerned with the architecture of complex systems and how that architecture could be *generated* within the context of an extant complex system. The starting point for this generation is initially taken to be Christopher Alexander's *Generative Patterns* (Alexander 2002). Alexander is an architect, in the sense of building, and as made many contributions to architecture one of which in particular, that of pattern languages, has had a major influence on computer science.

In more detail, the research:

> *"will investigate fully generative pattern languages, how their use affects software architectures, and how they can benefit CoSMoS. Alexander sees Pattern Languages as a foundation for generative methods: how to use pattern languages to generate and transform configurations, and the way that those configurations unfold into systems. He hypothesises that such unfolding configurations, when being generated and transformed in a principled and deeply structured manner, possess qualities beneficial to software architectures, including being robust, maintainable, adaptable, and configurable. We will investigate this hypothesis in the context of complex systems."*
> (CoSMoS, 2007)

The work described in this document has been carried out with Alexander's assistance.

The rest of this document examines what is meant by the term *software architecture* and how such architecture influences the development and behaviour of both classic and complex systems. A *complex system* is defined to be a system which is composed of a group of communicating components that as a whole exhibit properties that are not obvious from the properties of the individual parts; such properties being referred to as *emergent properties*. The term *classic system* is used to refer to conventional, non-complex, software systems. Broadly, such systems are those where the desired outcomes of the system's execution are expressly designed into the system, rather than emerging from the system's execution. The behaviour of a classic system may well be *complicated* but it might well not be a *complex* system.

Further, the roles of various sorts of *patterns* are examined, in particular how *generative patterns* might have a role in complex, and classic, systems development. The results that have been achieved so far are examined, as is the way those results are intended to be extended in the next phase of this research.

The main focus of this work is therefore intended to be on *generative* approaches to software design and development.

[1] EPSRC Reference: EP/E053505/1.

1.1 Terminology

One of the difficulties with writing documents of this sort is that the word "architecture" can be used to refer to two different things: the architecture of software systems and that of buildings and building sites. In order to avoid ambiguity when the word *architecture* appears unqualified then it is referring to *software architecture.*

2 Background

There are three distinct aspects to the background to the current research, all of which are explored in this section. These aspects are:

1. *software architecture* and the various definitions and techniques that are associated with it.
2. *design patterns* and the way that they have been taken up by the software development community.
3. *generative patterns* and how they have been defined in the context of buildings. Also, what they might mean in the future of software development and in particular in the creation of useful complex systems.

2.1 Software architecture

2.1.1 Definitions of software architecture

This research is ultimately concerned with software architecture and in particular whether generative patterns, as defined in (Alexander 2002) have benefits for software architectures.

Although the term "software architecture" is commonly used it is worthy of some preliminary definition. There are many extant definitions which actually show a range of meanings. The most straightforward definition is typified by:

> *"The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them."* (Bass, Clements & Kazman, 1998, p6)

> *"... software architecture is a set of architectural (or, if you will, design) elements that have a particular form. We distinguish three different classes of architectural element: processing elements; data elements; and connecting elements."* (Perry, Wolf 1992)

These definitions, and others like them, are concerned primarily with the *structure* of a software system. As such it is not clear how *architecture* differs from *design* which would commonly concern itself with structure.

A further definition, and one which is more relevant to the actual practice of software architecture, at least in classic systems, is:

> *"Architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution."*
>
> (IEEE Computer Society, 2000)

This definition is more useful for this current research as it specifically encompasses the notion that the organisation of a system might evolve through its lifetime. In this context “evolve” just means the process of developing, growing and changing. It does not refer to Darwinian evolution. This inclusion of evolution in this definition is relevant to the current research in that it is to be expected that the architecture of a complex system will change as that system executes and its behaviour, desired or otherwise, emerges.

2.1.2 Structures in space and time

In general the extant definitions of software architecture address both spatial structure—how a software architecture is represented as collection of connected components—and temporal structure—how those components are used as the system executes and interacts with itself. For example, a software architecture might include a transaction manager component which is connected to many parts of the system. The architecture includes both that component with its inter-connectivity and also the patterns of behaviour that create, commit and roll-back transactions during the system’s execution.

In general a software system is constructed of a number of components, usually with well defined interfaces between them. It has a particular behaviour which is usually a direct consequence of the way the components are composed to make the whole system. All of these aspects of a system change over time, both as the system is being developed and when it is executing.

That is, the architecture of a system is concerned with how the system’s components are composed and how those components behave in time. Design techniques have been developed over the history of software engineering all of which in some way are concerned with the definition and development of various abstract views of the system. It is an interesting observation that these techniques, for example those included by the UML graphical languages (Fowler, 2004) are typically good at representing abstractions of structure but rather poor at representing abstractions of process. UML, for example, allows the user to represent abstract components as superclasses and interfaces but includes no equivalent ways of representing abstractions of process; either of execution or development.

2.1.3 Components and ADLs

Many techniques exist for describing languages for defining architectures and their usage. Perhaps the most successful of these is the *4+1 model* of software architecture, first proposed by Kruchten (1995). Although originally defined using the Booch notation (Booch, 1993) this technique has since been refined to use the various UML notations for a rather different purpose from that for which they were originally intended. In particular the “4” views of software architecture allow the expression of:

- a *logical view* of the system, which describes the structures that are part of the system,
- a *process view*, which shows the processes that execute in the final system, using the components described in the logical view,
- a *development view*, which describes the parts of the logical view that are developed together and, finally,
- a *physical view* which shows how the processes in the *process view* and the products of the *development view* are supported by the physical hardware.

Finally, the "+1" aspect of the model reflects the scenarios (that is, *use cases* in UML) that show how the other four aspects of the architecture interact to produce the desired final result.

In addition to the 4+1 model there are many other "Architecture Description Languages" (ADLs) in the literature all of which give some way of describing an architecture. Extant ADLs include:

- *Koala* (Ommering, 2000) which is specifically targeted at software configuration in consumer electronic products, Koala is intended to ease the complexity of software configuration in family of such products all with subtly different requirements for the control software
- *Rapide* (Luckham & Vera, 1995). This ADL is different because it is *event based*.
- *Wright* (Allen, 1997) is an attempt at a mathematically formal ADL being based on CSP (Hoare, 1985).
- *Darwin*(Magee et al, 1995) has the unique characteristic that spatial connections are not defined in the ADL but are implied by a language that defines behaviour, allowing particular sorts of formal reasoning.

There are also several attempts at classifying and comparing ADLs such as (Medvidovic and Taylor, 2000).

However, all of these reflect a static view of architecture, not the generative aspect that this research is concentrating on even though that aspect of architecture is to some extent apparent in the IEEE definition of software architecture given above.

2.1.4 Requirements and emergence

Classic software systems are always constructed in the presence of system requirements, even if those requirements are not explicitly recorded. Further, those requirements are usually separated into *functional* requirements and *non-functional* requirements.

These terms are not easy to define precisely, in particular because the satisfaction of non-functional requirements tends to create new requirements which are functional in their nature. Nevertheless:

- A *functional requirement* defines a function that a piece of software must be capable of performing. That is, these requirements define the behaviour of the system as it consumes its inputs and produces its outputs.
- A *non-functional requirement* defines the criteria by which the operation of a system can be judged. They are the *qualities* of the system encompassing things like security, safety, availability, maintainability and performance.

This distinction leads to a useful way of distinguishing *design* from *architecture* in software development. It can usefully be said that:

- *Design* is that aspect of software development that addresses the satisfaction of the system's functional requirements and
- *Architecture* is that aspect of software development that addresses the satisfaction of the system's non-functional requirements.

It is in these terms that software architecture and software design is discussed in this document. Note that this particular definition of architecture does not wholly concur with those given earlier in section 2.1.1. However, this particular definition does have some utility because *architecture* and *design* are distinguished which was not the case for the earlier definitions.

In general, the process of software development is one of the definition of a number of useful abstractions and the subsequent development towards an implementation. Design is therefore the development of those abstractions that address the basic functionality of the system. For example in the design of an air traffic control system such abstractions would represent aircraft, airspace sectors and routes. These abstractions are then *refined* ultimately into the code that executes in the final system.

Architecture, in contrast, is the development of abstractions that address the non-functional requirements of the system and in the example these would be models of performance, security or usability. Such models are not usually capable of *refinement* into code. Rather, other designs must be constructed that *satisfy* the architectural models. These latter designs are also subsequently refined towards the implementation.

Notwithstanding these different starting points all aspects of architecture and design will, of course, end up being defined using the simple mechanisms available to the programmer: classes, threads, processes, data structures and so on.

The distinction between these activities is significant for this research because a useful approach to generative patterns that address the general issue of software must address both design and architecture.

This model, though, only really applies to classic systems. In the world of complex systems there is no such simple picture, for the reason that there is no clear relationship between the requirements for a system and the abstractions used in the development of the system. This is because the real results of the complex system, which would be expected to be part of the requirements if they actually existed, emerge from the execution of the system rather than being explicitly coded into the system.

For example, the Game of Life (Gardner, 1970) is a complex system where each cell in a two dimensional space is either *alive* or *dead*. Each cell is square and has 8 immediate neighbours. The game proceeds in a number of steps and at each one each separate cell decides whether it will be alive or dead at the next step according to some simple rules that are expressed purely in terms of whether the neighbours of that cell are alive or dead. Specifically, the rules are:

- Any live cell with fewer than two live neighbours dies, as if by loneliness.
- Any live cell with more than three live neighbours dies, as if by overcrowding.
- Any live cell with two or three live neighbours lives, unchanged, to the next generation.
- Any dead cell with exactly three live neighbours comes to life.

These apparently simple rules are expressed just at the level of the cells but they have remarkably complicated consequences which appear as a number of emergent properties. One of the most obvious is that certain patterns of life and death appear to move across the game space. Such *gliders* are a significant consequence of the rules of the system but there is nothing in the code for the Game of Life that is a refinement of a glider abstraction that was conceived of early in the design of the system. That is, the gliders have not appeared by any sort of conventional design process which has produced artefacts that been refined into an implementation. Remarkably, the Game of Life and its gliders provide a Turing-complete computation system (Berlekamp et al, 1982), something that was complete unsuspected at the time of the original development of the game.

Complex systems are, therefore, rather different from classic systems as regards requirements. In general, emergent properties are not the consequence of specific requirements and are not abstracted and refined in the manner familiar from classic systems. This is reminiscent of (Polack & Stepney, 2005) which examines the distinction between functional requirements which are "preserved by refinement" and non-functional requirements which do not refine in this manner. The satisfaction of non-functional requirements is, according to the definitions here, a process of *architecture.* But, the notion of emergent properties, as in the gliders example, is also one that does not refine. As such the gliders, and emergent properties in general, could be considered to be some sort of architectural construct.

As discussed earlier, the notion of architecture is usually targeted at non-functional behaviour such as that of performance. If a high-performance variant of the Game of Life was required then it is likely that having an explicit representation of things like gliders would be a useful strategy, as that would allow various performance improvements in the code. That is, rather than laboriously calculating what each of the cells do individually the code would be allowed to say "Oooh look, there's a glider. I know what they do without having to look at all the underlying details.". The Hashlife algorithm (Gosper, 1984) is in many ways an example of this sort of approach, although rather more generalised in that it recognises many possible equivalences in the Game of Life space rather than just gliders.

In this case, then, the architectural requirement for improved performance is satisfied by constructing an artefact that is a concrete representation of an emergent property. The emergent property is in some way reified as an explicit part of the system..

This discussion leads to intriguing observation that in some way a complex system's emergent properties are the same sorts of things as the properties that are the satisfaction of a classic system's non-functional requirements. This observation is still to be explored in detail.

2.2 Patterns

The appearance of design patterns in the 1990s has had a profound effect on the world of software development. The design patterns movement is originally based on Alexander's "A Pattern Language" (Alexander et al, 1977) which shows a series of 253 "patterns" that are applicable in the domain of building and planning. Each pattern "describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" (Alexander et al, 1977, page *x*). In essence, then, Alexander's patterns are elements of reusable building design.

The notion of patterns was taken up by Gamma et al (1994). They proposes a small collection of 23 software design patterns each of which has similar characteristics to Alexander's patterns and are indeed described by reference to the same Alexander quote as used in the previous paragraph (Gamma et al 1994, p2). Each pattern is described in standardised manner with (Gamma et al, 1994, p3) four essential elements:

1. A *name* which is a succinct description of a design problem and its solution.

2. The *problem* that the pattern is applied to.
3. The *solution* to that problem.
4. The *consequences* of using the pattern.

In many ways it is the names that are the best thing about the (Gamma et al, 1994) patterns as they have become part of the vernacular of software development. So, designers will routinely talk about using "Visitor", "Observer" or "Composite", sure in the knowledge that this will communicate just the required amount of information. Finding these names is difficult; Gamma *et al* comment that *"Finding good names has been one of the hardest parts of developing our catalog."* (Gamma et al, 1994, p3).

The use of design patterns in software development now extends to both the original patterns in the book, the many other patterns that have appeared in the literature and on the internet and usually project specific patterns that document how a common problem is to be solved for a specific development. For example, the same approach has been applied to:

- patterns for requirements analysis (Fowler 1997),
- testing patterns (Young et al 2005),
- coding (Beck 1996).

The notion of patterns has even been applied to software development itself in the concept of *Process Patterns* (Ambler, 1998).

All of these patterns, and in particular the ones in both (Alexander *et al*, 1977) and (Gamma *et al*, 1994) are more than just isolated examples of good design. Each pattern is explicitly part of a larger body of work (hence Alexander et al's "Pattern *Language*"). Each pattern makes reference to other patterns and describes when other patterns might be more appropriate; one pattern is often expressed using the language provided by other patterns. For example, Gamma et al's *Façade* pattern references the *Abstract Factory* pattern as a way of creating objects in a system independent manner and also says that *Façade* objects are often also instances of the *Singleton* pattern.

Similarly in Alexander *et al* (1977) the *Small Work Groups* pattern, which relates to the structure of the space people work in, is related to the *Self Governing Workshops and Offices* and *Flexible Office Space* patterns. Almost every single pattern in these references is related to others. Gamma et al (1994) even includes a diagram showing how all the patterns are inter-related.

The notion of patterns has also been applied to software architecture. For example (Fowler 2003) examines application architecture[2] and includes patterns called *Optimistic Offline Lock* and *Pessimistic Offline Lock* which neatly encapsulate a common architectural decision, whether to use optimistic or pessimistic locking, as a pair of design patterns.

Pessimistic locking refers to a locking strategy where one user locks some shared component so that another user is prevented from accessing the shared component until the prior lock is

[2] *Application Architecture* is a term in common use to indicate the structure of a system from the point of view of a running application. As such, it tends to concentrate on things like the application's required transactional structure. It is usually contrasted with *technical architecture* which is the set of hardware and software products needed to support a particular application architecture.

released. *Optimistic* locking makes the assumption that an attempt to access a shared component simultaneously is actually very unlikely and merely detects that such an access has been made when the second accessor tries to modify the shared resource.

The issue of locking choice therefore concerns, as discussed earlier, the system's non-functional requirements; in this case something about how concurrent access by multiple users to the same data is to be handled. That is, the architectural patterns affect the way the system satisfies its non-functional requirements.

2.3 Generative patterns

The existing usage of patterns in software development applies to largely "static" patterns. That is, the patterns describe stable configurations that are usefully used in a design but which retain that form throughout the life of the system. That is, they are not associated with any generative aspects, the core focus of the current research.

Software development as a whole, though, is moving away from a static approach to design and development. Traditional design and development has always emphasised an approach typified by the timely production of detailed designs and architectures. Once produced these artefacts are intended to be mostly static, only being changed as the requirements for the system change. The best approaches to such design—for example that referred to by (Booch, 1995) as "architecture-centric", which emphasises the production of a core system that is then perturbed by the inevitably changing requirements—allow certain variability as the design develops but the essentially static structure remains.

However, this approach is now being challenged. In particular, the agile movement (Agile Manifesto) has shown a way to build systems where it is accepted that the design will be continually "refactored" throughout the development as described, for example, in (Fowler 1999). In this environment, new structures are only introduced when the currently executing part of the system actively *needs* that new structure. That is, rather than slavishly following the original design the specifics of the design are allowed to change during development as a direct consequence of the partially constructed system executing in the real system context.

Most importantly, if the need does not arise, which is commonly the case as the requirements change continuously, then that new structure is not required and is not implemented. This contrasts with traditional approaches where the structure would have been implemented from the outset and would doubtless have continued cluttering up the system for all time.

This more "dynamic" approach to system development is becoming more common in classic systems. It is also an essential aspect of complex systems. When such systems are executing then new structures appear, they are said to emerge, as a consequence of the small scale structure of the parts of the system. For example, in the Game of Life (Gardner, 1970) then an emergent property of the rules that are in every cell is that the structures known as gliders will, if they emerge, appear to move across the space.

The architectural issue about things like the Game of Life is that there is no part of the underlying program that directly corresponds to a glider. However, it is probably the case that the overall performance of the system would be improved if there were such a program component, that could efficiently represent the movement of the glider should one appear.

However, explicitly coding for such a construct would be a consequence of knowing in advance that this behaviour was going to emerge.

One of the things that this current research is looking for, then, is a mechanism that "spots" things like the emerging gliders and provides a special component that supports them efficiently. This mechanism would allow the complex system (or indeed the classic system, there seems to be no large difference from this point of view) to run until something could be observed about the system. The generative mechanism could then *generate* some new artefact in the software architecture that supported the new observation.

Presently, this is very ambitious, but it does seem likely that something of this form will be needed in the future of systems, especially complex systems, development. Ultimately this is because a group of humans, of a size capable of handling a development, will not be capable of directly designing all parts of a system. That is, if computer systems continue to get more complex then mechanisms of this form will become necessary.

The intent, therefore, is to look to be able to generate parts of the system that are a consequence of the system's structure and how it executes, even at a very early stage, in a given environment. This sort of execution is familiar from (Thompson, 1961) which looks at biological development as a consequence of growth within a particular environment. The environment is important for the biological systems and it should be equally important for software systems. This should mean that the exact same system, executing in a different environment, should *unfold* (Alexander, 2009) differently.

There is some sort of continuum of software system development, with at least the following sorts of system feasible:

1. A *static* system where the system is always exactly the same structure as the way it was first coded. This is the sort of system typified by a *stepwise-refinement* approach to development as in (Wirth, 1976).
2. A *dynamic* system where new instances of pre-defined structure are created at run time. This is the sort of system typified by most object oriented designs as in (Meyer, 2000).
3. A *generative* system is defined as one where new structures are created at run time which may then be instantiated or used directly in the already running system.

Conventional system design and implementation is about type 1 and 2 systems. This research is looking at ways to develop a type 3 system.

2.4 The nature of order

It is these sorts of issues that are addressed in Alexander's *Nature of Order* (Alexander, 2002). This is a four volume publication which addresses the general notion of rules which generate components of a building structure. These rules are ones that Alexander (2002) maintains produces result that have a particular property which he refers to as *Life.*

2.4.1 Centres

The Nature of Order describes how the built environment can be described in terms of what Alexander calls *centres*. These are not point like structures but rather refer to generalised regions of interest in the context of development of the building site. He defines centres, in (Alexander 2007a) as *"A center is a zone of coherence that occurs in space. That is all one can say.*

There is no more elementary substance from which centers are manufactured. Centers can only be manufactured from other centers." The important part of this is the *zone of coherence*. That is, it is just a region which can somehow be seen as being separate from other regions.

A simple example of Alexander's centres is in the context of a single sheet of A4 paper. When this is empty it is devoid of interest, although perhaps it is to be regarded as a single centre. However, the addition of a single dot makes clear the presence of a number of structures, each of which might be an interesting part of the design that encompasses this dot. For example, the drawings in Figure 1 appear in (Alexander, 2002).

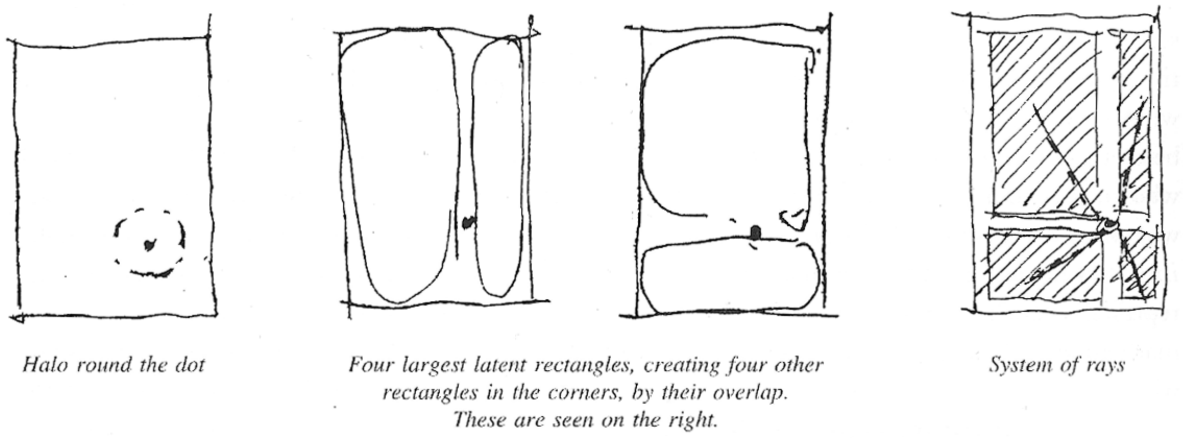

Halo round the dot
Four largest latent rectangles, creating four other rectangles in the corners, by their overlap. These are seen on the right.
System of rays

Figure 1: Centres due to a single dot (Alexander, 2002, v1, p82)

In fact, Alexander estimates that there are in the vicinity of 20 centres that appear on this diagram just by the addition of this single dot, as follows:

1. The sheet itself.
2. The dot.
3. The halo around the dot.
4. Bottom rectangle trapped by dot.
5. Left hand rectangle trapped by dot.
6. Right hand rectangle trapped by dot.
7. Top rectangle trapped by dot.
8. Top left corner.
9. Top right corner.
10. Bottom left corner.
11. Bottom right corner.
12. The ray going up from the dot.
13. Ray going down from the dot.
14. Ray going left from the dot.
15. Ray going right from the dot.
16. The white cross formed by these four rays.
17. Diagonal ray from dot to nearest corner.
18. Diagonal ray from dot to next corner
19. Ray from dot to third corner
20. Ray from dot to furthest corner.

Alexander says that:

> *"these stronger zones or entities* [that is, the 20 regions above]*, together, define the structure which we recognize as the wholeness of the sheet of paper with the dot. I refer to this structure as the wholeness..."*

This is unfamiliar language to scientists. It is also an unfamiliar style of thinking which is difficult to formalise in the usual scientific manner. However, the essence of what Alexander means is clear in that these structures are definitely perceived by observers of the paper with and without dot.

2.4.2 Properties

In addition to the notion of centres themselves, Alexander's *Nature of Order* proposes a set of 15 properties that describe aspects of a system of centres that are seen by him as having "life". He also defines these properties as "operators", for example in (Alexander, 2007a). These operators are the *generative* aspect of the properties. That is, they are used to enhance a system of centres from a possibly primitive starting point, to a complete design. In Alexander's examples, though, this *unfolding* takes place with the direction of a skilled designer. For example, Alexander (2007b) gives an example of a sequence of application of the properties, as operators, that leads to the final design of a window in a particular opening in a house.

The properties are listed in the following paragraphs. In each case the property is defined twice: as a property from (Alexander 2002, pp 239—241) and as an operator from (Alexander 2007a). Some of the properties can be seen as applicable to software architecture as well as building and, where appropriate, some notes are included to this effect. Similarly there are noticeable parallels to other fields of endeavour, in particular musical structures which have temporal properties that are relevant to Alexander's properties. Again, some notes are included where relevant.

Levels of Scale

> Property: *"how a centre is made stronger (more coherent) by the smaller strong centres within it and the larger strong centres that surround it*
>
> Operator: *"modifies the given center, by embellishing it with smaller centers. These smaller centers are typically one half to one third the diameter of the original center, but sometimes smaller. They may be created within the original center, or in the space adjacent to it*

At first sight this property could easily be seen as relevant to software design or in fact most engineered artefacts. That is, there is usually some gradation in the value of almost any variable across a system. However, Alexander's property does not refer to simple gradation. It is more quantised than that in that Alexander sees a change in magnitude of between one half and one third between adjacent levels. The engineering analogue of this is not easily apparent. It is also worthy of note that the property definition relates both smaller and larger centres. The operator representation, though, merely foresees the application of the operator as a way of creating *smaller* centres.

Strong Centre

Property: *"Strong Centres defines the way that a strong centre requires a special field-like effect, created by other centres, as the primary source of its strength."*

Operator: *"This is a generic operator which simply makes the coherence of a center stronger, by making it more "center-like". It does so by calling any of the following operators: The Thick boundary, the Levels of scale operator, the Gradient operator, and others."*

A software analogue of this property is not immediately apparent. However, there are two things of note here. Firstly, Alexander is making explicit reference to some sort of *field* in the definition of this property, and he frequently returns to this concept elsewhere. This concept has been further examined and is further discussed in section 5 of this document.

Secondly, Alexander clearly sees some sort of calculus of operators in the notion of the *Strong Centre* operator "calling" other operators.

Thick Boundary

Property: *"the way in which the field-like effect of a centre is strengthened by the creation of a ring-like centre, made of smaller centres which surround and intensify the first. [It] also unites the centre with the centres beyond it, thus strengthening it further."*

Operator: *"This operator places a thick boundary around or partly around the zone occupied by a weak center, thus making the center more coherent."*

In the terms of software design this could be seen as a Façade or Adapter, where some software structure is made more isolated (and hence generally useful) by being accessed using some specific pattern. It should be noted that the *Thick Boundary* is not just an extension of an existing centre but is, as is apparent from the property definition above, a centre in its own right. This fits with the proposed software analogue where patterns like Façade and Adapter are fundamentally separate from the accessed structure.

Alternating Repetition

Property: *"the way in which centres are strengthened when they repeat, by the insertion of other centres between the repeating ones".*

Operator: *"This operator repeats centers to form an array. This may happen in one, or two, or three dimensions. The key effect of the operator is that it then creates a second system of centers between the loose packing of the first centers, in such a way that the first centers and the second centers are made strongly distinct, by shape or material or color, and become more coherent, by virtue of the alternation. In the course of the operation, the operator often changes the shape of the first centers, to make the in-between, second centers well shaped."*

In terms of software design this property is rather more difficult to interpret. There is no obvious area where a repetition of the form *abababa* occurs. However simple *repetition* is frequently observed, most obviously temporally as an iteration.

The operator definition is phrased specifically in terms that are for buildings. For example it refers to a three dimensional structure and specific shape attributes such as colour and material. Such attributes would not be appropriate to the components of a software system. However, other attributes could well be and such a property as *Alternating Repetition* might well apply to those components.

The same notions are seen in many other areas. For example, alternating repetition is a common feature of musical structures, most clearly in a simple verse/chorus/verse/chorus arrangement.

Positive Space

Property: *"the way that a given centre must draw its strength, in part, from the strength of other centres immediately adjacent to it in space."*

Operator: *"This operator is one of the most important, but hardest to define. It is to be applied to any center, and helps to shape the so called 'empty' space in the center. The positiveness of space comes from a combination of good shape, local symmetries, boundedness and above all from the appropriateness of the space for human purposes. This operator is applied most typically to the latent centers formed in the space between other centers, to give this space form.`"*

Good Shape

Property: *"the way that the strength of a given centre depends on its actual shape and the way this effect requires that even the shape, its boundary, and the space around it are made up on strong centres."*

Operator: *"This operator directly influences shape. If a rough outline of a shape has been generated, this operator examines the overall convex pieces of the shape, and tries, as far as possible, to strengthen or emphasize these pieces, within the segments of the curved boundary, in such a way that makes the overall shape more distinct."*

In software it is not clear what *shape* means. However, there are lots of *goodness* properties. For example, the absence of the "smells" discussed in (Fowler, 1999) is a primary criterion of *goodness*. In general, finding a suitable analogy for Alexander's notion of *shape* in software could be the key to seeing how to apply his operators, or transformations of that general sort, to a software system.

Local Symmetry

Property: *"the way that the intensity of a given centre is increased by the extent to which other smaller centres that it contains are themselves arranged in locally symmetrical groups"*

Operator: *"This operator strengthens a given center by introducing one or more local symmetries – most often a bilateral symmetry. If the center already has a natural axis of orientation, the symmetry is made to coincide with it. Otherwise, it orients the symmetry to make it as congruent as possible, with the field induced by other nearby centers (i.e. where it seems natural). It is best to put the symmetry on a center that is already nearly symmetrical."*

It's not clear how this property could be seen to apply the components of a software system of any sort. However, it could be seen as applying, again in a temporal manner, to things like the well-nesting of transactions in a distributed transactional system, and indeed in a piece of nested pseudo-code.

Such symmetries do appear elsewhere. For example, in the structures that appear in music, and especially in the more formal arrangements such as canons and fugues. Again, though, the symmetries are often apparent in the temporal domain.

Contrast

> Property: *"the way that a centre is strengthened by the sharpness of the distinction between its character and the character of surrounding centres".*
>
> Operator: *"The coherence of this proto-center, is enhanced by contrast, whether of color, or material, or gradient, or density. The contrast operator increases the contrast between the inside and the outside of the center, to make the center stronger."*

Many aspects of software design could be seen as being *contrasts* of one form or another. For example, *information hiding* is essentially a contrast between the parts of the system that are hiding information and the parts that are denied access to that information. Many of the aspects of "good" software design could be looked at as establishing the right sort of contrasts. Furthermore, such "good" design is often represented by things like the Gamma et al (1994) patterns such as the Façade and Adapter mentioned in the context of the *Thick Boundary* property.

Roughness

> Property: *"the way that the field effect of a given centre draws its strength, necessarily, from irregularities in the sizes, shapes and arrangements of other nearby centres."*
>
> Operator: *"In the course of unfolding, as the operators push and shove, to make various things happen, as required by the operators, it happens, very often, that something does not quite fit neatly. Instead of creating a perfect, or pristine shape, it is then necessary -- absolutely necessary -- to relax certain conditions, in order to make the configuration work successfully."*

Again, this property has no clear software design analogy. That is, software designs are usually characterised by exact repetition, not by many minor variations. However, each of those repetitions in a software design is in fact different due to the different context.

Such small variations also appear in other forms. For example, the musical ornaments of appoggiatura and acciaccatura[3] essentially "roughen" the strict structure of the music, making it more pleasing to the ear.

Gradient

> Property: *"the way in which a centre is strengthened by a global series of different-sized centres which then* ***point*** *to the new centre and intensify its field effect"*

[3] These are the "grace notes" of musical notation: small notes that may not have any defined time value but which are "crammed in" to the basic time signatured structure.

Operator: *"This operator creates gradients that point towards or away from a given center. The common gradients are gradients of size, gradients of contrast, gradients of spacing, gradients of orientation."*

From the point of view of software design many such gradients can be seen. For example in the hierarchy of responsibilities that is inherent in a multi-tier enterprise architecture.

Deep Interlock and Ambiguity

Property: *"the way in which the intensity of a given centre can be increased when it is attached to nearby strong centres, through a third set of strong centres that ambiguously belong to both[4]."*

Operator: *"This operator is used at an interface between two adjacent centers. Its purpose is to create a zone, usually an ambiguous zone, forming a third center between the two original centers. It is made ambiguous, in the sense that there are ties from one side, and ties from the other, with the result that there is a visible ambiguity about which of the two outer centers this new center belongs to."*

The notion of a intermediate layer which looks different from the point of view of two other adjacent layers is actually common in software design, in particular in the structure of stubs and proxies that are created in distributed systems; for example when using the Java RMI (Grosso, 2001) mechanism.

Echoes

Property: *"the way that the strength of a given centre depends on similarities of angle and orientation and systems of centres forming characteristic angles thus forming larger centres, among the centres it contains".*

Operator: *"This operator has mainly to do with angles and curves and ratios. As the collection of centers grows, there will be a certain predominant angles, or curves, or ratios or proportions in the shapes that have been created. This operator, then uses the statistics of the angles that so far dominate the configuration, and introduces these angles (or curves or ratios) as a default in the drawing of later centers that are created, thus slowly giving the whole system of centers a family resemblance shared by many of them."*

Alexander's definitions are all in the terms of angles and curves which are clearly important when designing buildings. However, if looked on a more generally about family resemblences then it could be seen as applicable to software, in particular in things like repetitive patterns of processing where a particular system handles all possible inputs in the same general pattern. Note that this is again a temporal analogy.

Simplicity and Inner Calm

Property: *"the way the strength of a centre depends on its simplicity - on the process of reducing the* ***number*** *of different centres which exist in it, while increasing the* ***strength*** *of these centres to make them weigh more."*

[4] This essentially non-hierarchical structure is also familiar from (Alexander, 1965).

> Operator: *"This operator is a clean-up tool working along the lines of Occam's razor. It simplifies a configuration. It removes, as far as possible, all superfluous structure."*

There are two obvious software analogues to this property. The first is the calm that returns, temporally, to a system when a prior transaction is committed. The second is an appeal to the maintenance of simplicity "Perfection in design is attained not when there is nothing more to add, but when nothing remains to be taken away". (St. Exupéry, 1939)

In either case the notion of a calm simplicity to systems is an appealing one, and in many ways the antithesis of the "bad smells" that appear in (Fowler 1999).

The Void

> Property: *"the way that the intensity of every centre depends on the existence of a still place - an empty centre - somewhere in its field".*
>
> Operator: *"This is a pervasive operator, working at many levels of scale. The basic idea of the operator, is that at the core of a center, there is always some undisturbed and perfectly peaceful area which lacks busy-ness or excessive structure. It is very important that each serious center, has, within its boundary, some area like this. Often this area is large in extent, compared with all the other elements that have a great deal of structure. This operator can be expressed arithmetically, as a statistic on the whole configuration."*

Not Separateness

> Property: *"the way the life and strength of a centre depends on the extent to which that centre is merged smoothly - sometimes even indistinguishably - with the centres that form its surroundings."*
>
> Operator: *"This operator comes into play after the majority of centers have been established. The purpose is to overcome any separation that is caused between the configuration and its environment, or between any individual center, and its immediate environment. To mobilize this operator, wherever a boundary is too sharp, bridges should be formed, by chains of centers, which cross that boundary, thus creating a softer and more permeable edge."*

Alexander's *centres* are the essential embodiment of his ideas about generative patterns. Although his definition of centres has it that *"A center is a zone of coherence that occurs in space. That is all one can say."* it is clear that he does see more attributes of centres in that the definitions include the possibility of a centre having *strength*, a *core*, *shape*, *colour* and *material*. Recent discussions with Alexander have also included the notions of centres having an *origin* and a *footprint* and an *extent*. That is, centres do appear to have more attributes than just being *a zone of coherence in space*.

2.4.3 Process

Behind the notion of centres and properties, Alexander sees a process that guides the overall application of these generative patterns. The idea is essentially an iterative one in that the 15 properties are repeatedly applied to the space that contains the centres. Alexander (2002, vols 2, 3) shows how the overall process is applied to the centuries long generation of St. Mark's Square in Venice.

As can be seen, each of Alexander's properties is essentially defined in two ways: a property of the space and as an operator that creates new centres so as to optimise one or more properties.

That is, each of the properties is a metric for some part of the space, saying whether it has "life" (as Alexander puts it). So, for example, the *Levels of Scale* property might say that the centres in a particular space did indeed have a suitable gradation of scale.

Secondly, the operators are used to construct new artefacts in the space that give rise to new centres. These new centres expose the given property in a suitable way. So, if a particular space has a large structure, and nothing else, then the *Levels of Scale* property can be seen as an operator that constructs new structures, giving rise to new centres, that more closely fit the metric defined by the *Levels of Scale* property.

That is, the overall cycle of observe/generate is iterated for the space, and recursively descends the structures of centres that exist in the space.

One aspect of this that is as yet unclear is what parts of this process lead to the removal of structures in the space. The properties as they stand are essentially constructional but it is clear from looking at building sites as they evolve in time such as St. Mark's Square in Venice, used as an example by Alexander (2009), that structures are removed. It would seem reasonable that the removal of some structures would enhance the metrics of some of the properties, most obviously of *The Void*. The cycle therefore is to be one of iteratively applying "observe/generate-or-destroy".

3 Generative patterns of software

So, as has been discussed, there is a case for generative patterns of *software* and that is the main thrust of this research although an essential prerequisite is to understand and formalise Alexander's generative approach to building. This applies to classic systems but most particularly to the complex systems that are the focus of the overall CoSMoS project and this research. This generation would be seen as modifying the implementation of the system as it executed in its environment.

In a classic system, such generative changes could be the introduction of a cache of frequently accessed data or the introduction of optimistic locking. Such changes fit with the notion of software architecture addressing the non-functional requirements. That is, with the addition of such a cache the system has the same functional behaviour, but its performance is potentially improved.

In a complex system it could be introduction of specific processes to represent properties that had, previously, merely been emergent properties. For example, one of the complex systems examined by the CoSMoS project is the *boids* system (Reynolds, 1987). This system simulates the behaviour of a collection of birds flying in space using a small number of surprisingly simple rules. As the system executes "flocks" emerge in its behaviour. That is, the "boids" tend to start flying around as a number of groups rather than as individuals. One consequence of an implementation of the generative patterns of software would spot the flocks, in some way, and create architectural structures to support them.

Similar mechanisms would ideally exist to extend the implementation of the complex system, not just to optimise the performance and expose existing emergent properties, but to allow new properties to emerge as a consequence of the way the system executes in a specific environment. Alexander's hypothesis in (Alexander, 2009) is essentially that the implementation of software analogues of his 15 properties (and more particularly the related operators) would allow such beneficial new behaviours to appear, to *unfold* as he describes it. The current research was originally proposed specifically as a means of exploring and expanding on this hypothesis.

This generative approach needs to be, in principle at least, autonomous. That is, once a complex system starts executing then the ideal embodiment of the generative patterns discussed in this research would be one that created new structures in accordance with a set of generative patterns.

The process is required to be autonomous because, in a complex system, the system's behaviour is not explicitly designed in. Rather it is emerging. Hence, new constructs must also emerge from the system's execution. Perhaps it is right to say that these new constructs must emerge from the system's *patterns* of execution.

A suitable approach to this sort of problem will inevitably be patterns based. That is, a set of architectural patterns, and an environment where systems are generated, that in some way is related to the sorts of things Alexander discusses in the Nature of Order; it seems unlikely that the exact same patterns as Alexander's will be useful for software systems, in the same way that the patterns in (Gamma et al, 1994) are very different from those in (Alexander et al, 1977). Whilst it was relatively easy to generalise the approach in (Alexander et al, 1977) to software as in (Gamma et al, 1994) the task of generalising the generative patterns is much harder.

These patterns would allow for spotting some particular behaviour in a running system, perhaps the concentration of dots around a boids flock or a regularly moving pattern of glider cells, then some property would be available to synthesise a new "centre" that supported the execution of this detected characteristic. Further, the patterns would allow completely new behaviours to emerge, or "unfold" as Alexander puts it, as a consequence of the current system and its environment.

There is something quite different about this sort of pattern from the ones that are discussed in (Gamma et al 1994). All of the patterns in that book represent various abstractions of the software design. For example:

- A *façade* is an abstraction of a particular sort of interface to a another abstraction
- An *adapter* is an abstraction of a different sort of interface, as is a proxy.
- A *composite* is an abstraction of a structuring concept.

That is, all these patterns are abstractions of various products of the development process. The sorts of patterns that are needed for the these new generative patterns are abstractions of parts of the development process itself, a rather different structure.

4 Research direction

The direction of this research, then, is towards a set of generative patterns that could feasibly be implemented by some autonomous agent and “applied” to a running complex system, or indeed classic system. These patterns should be of the general form of the properties and operators defined by Alexander in (Alexander, 2002). In particular, they must represent in some way aspects of the development process itself, rather than abstractions in the final system.

This does, though, leave us with three significant issues which are essentially the current topics of this research:

1. Exactly how do Alexander’s generative patterns work?
2. How are the ideas in those patterns applicable, if at all, to the software development process?
3. How can a running software system, classic or complex, be made subject to development following patterns of this form?

Clearly, these questions follow on from each other, although in an agile approach to this work, there seems no reason to adopt a purely waterfall approach to these questions. That is, it is not necessary to wait for an answer to question 1 before thinking about question 2.

The rest of this document examines each of these questions and describes current and future work in each area.

5 Patterns and Fields

The first question, then, is how do Alexander's Generative Patterns work? It is necessary to understand this in a sufficiently mechanistic manner to be able to implement some sort of agent at some point. There is a hidden assumption here in that it seems likely that generative patterns that were of use in software development would not be the exact same patterns as Alexander uses. However, the assumption is that they would be sufficiently similar, at least in their mode of operation, to make further study of Alexander’s patterns useful. Initially, at least, that seems a supportable assumption.

5.1 Generative patterns meta model

In order to discuss how the patterns might work, some sort of notion of the context in which they work is needed. An early approach to determining this was the definition of a simple meta model of the actions of the patterns. This was developed in UML and is shown in Figure 2.

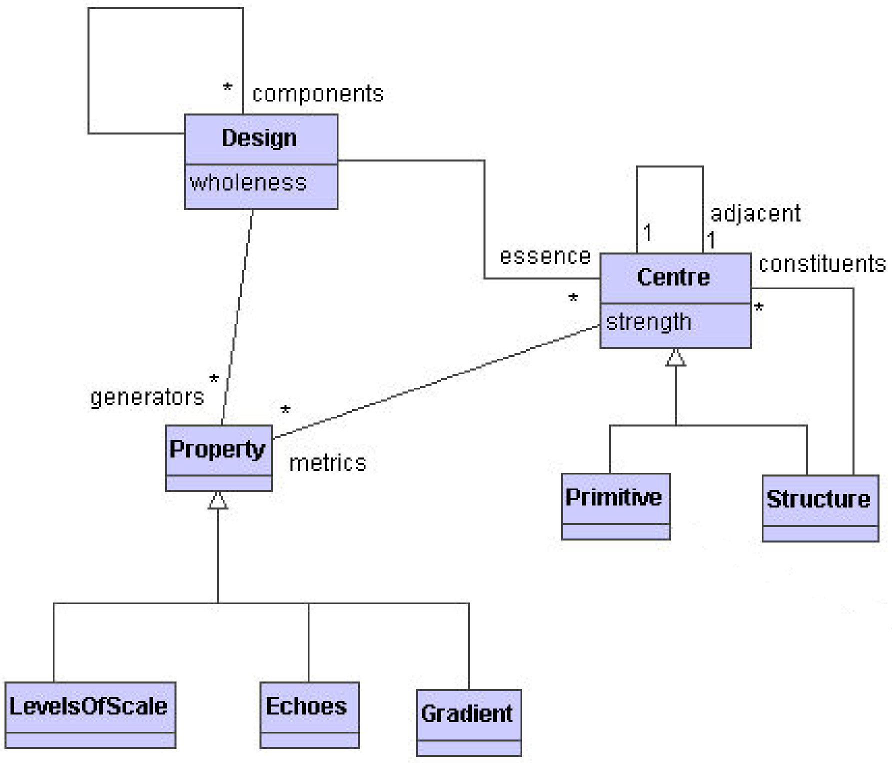


Figure 2: Generative patterns meta-model

The model is rather simple, but seems to adequately summarise the essence of the property formulation of Alexander's Generative Patterns. In this model, a design is represented as a hierarchically structured collection of designs (that is, each design may have a bunch of components which are other designs. Further, the design is distinguished by an attribute that Alexander calls "wholeness" which is some sense of overall "life" of the design. (Much of the Nature of Order is concerned with the issue of "life" and way that it can be detected in built constructs.)

A design is said to have an "essence" (another of Alexander's terms) which is the collection of centres that populate the space of that design. Centres abut one another and also have a hierarchical structure. This structure is represented here, in pleasing bit of self-reference, using the *composite* design pattern which allows the explicit distinction between primitive centres, which have no contained structure, and centres which do have such structure. Each centre is said to have a "strength", a further Alexander terms.

Finally, the set of properties, as discussed earlier, is present. For any one design or component of a design (and it should be noted that in this meta model all designs at all stages of generation are seen as co-existing) then there are a set of properties that express metrics relating to the centres that are the essence of that design. Any one property can also be seen as responsible for generating a new part of the design, presumably as a response to the values of the associated metrics not being within some range. Each property can actually be one of the set delineated by Alexander. (The diagram only shows three, to avoid clutter.)

As a meta model that is suitable. But, the obvious missing issue is how a particular property can be said to determine that a particular metric has an unsuitable value or what component of a design should be constructed so as to resolve that issue. That is, there is no *mechanism* underneath this model, it is just a static representation of the results of a dynamic process.

5.2 Fields hypothesis

An obvious early observation of Alexander's Generative Patterns was that their operation was in way reminiscent of the field theory that is common in the study of things like electromagnetism. Alexander himself often talks about fields, as in the definitions of the *Strong Centre* and *Thick Boundary* properties in section 2.4.2 of this document. One aspect of this that seemed originally appealing, and still does, is that the centres might in some way be seen as a consequence of some suitable derivative of the field, for example the *gradient* of a scalar field or the *curl* of a vector field. These concepts are described in any undergraduate textbook covering vector calculus such as (Ramo et al 1965).

This fields hypothesis was confirmed as being worthy of examination at an early stage by Alexander who showed some early work (Alexander & Cowan, 2008) on some sort of "fields theory of patterns". The attraction of such a representation is that it offers a mechanism for explaining how the behaviour of the various operators could be derived from some underlying form. What is more, that underlying form is amenable to automated analysis meaning that the notion of some autonomous agent for progressing the development of some complex system might be feasible.

However, just saying "field theory" is not particularly useful. The precise direction for this part of the research was refined in various discussions with Alexander. The results of these discussions were really that there were at least two sorts of fields. One of these is a sort of "electro-static"-like field which is due to things like the edge of a space and the artefacts in the space. Initial discussions indicated that it was probably a useful starting point to think of the field as being normal to these artefacts.

For example, Figure 3 shows how the field would be initialised i an empty square space.

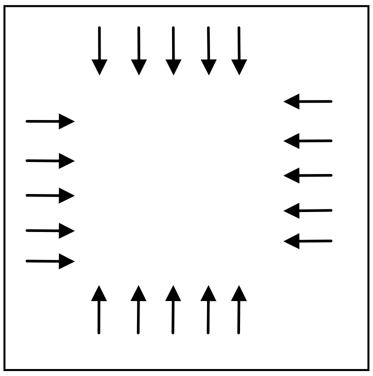


Figure 3: Field normal to edge

The current work on fields, which is described later, is all based around this general sort of field.

5.2.1 Fields in the Alhambra

The notions of fields were refined in some further analysis that Alexander did with various plans of the Alhambra in Spain. For example, Figure 4 shows some parts of the field that he sees in that complex structure (Alexander, 2008):

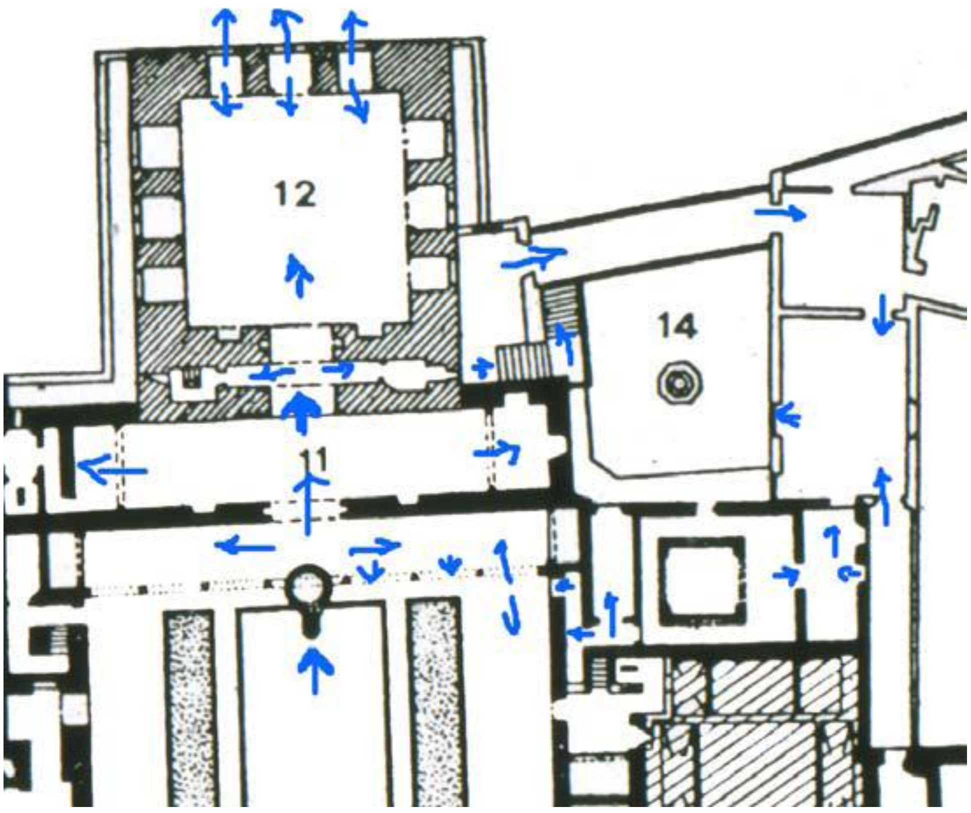


Figure 4: Fields in the Alhambra (From Alexander, 2008)

In this case the blue arrows are the field that Alexander sees as "pointing" from the smaller centres to the larger centres. He describes this field as

> *"...shows arrows that typically run from the smaller to the larger centers. The arrow means the smaller center "hangs off from" the larger one, or "helps the life of the larger one." It can also mean that the arrow points in the direction which means "is the parent of."*

In more detail Alexander sees the field being related in a complex manner to the shape of the originating artefact. For example, Figure 5 shows how he sees this field around a particularly complex pillar in the Alhambra.

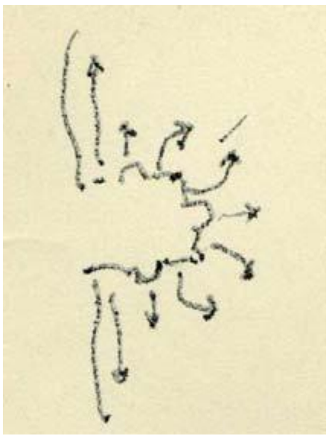


Figure 5: Field due to Alhambra pillar. (From Alexander, 2008)

A further type of field came from these discussions with Alexander where he talked about the notion of centres "pointing at each other". This "flow-like" field also looks worthy of further investigation but as yet this has not occurred.

The notion of multiple fields is probably going to be necessary as the non-linear interaction of more than one field could give rise to a range of complex behaviours.

5.3 The Ising model

One obvious possibility is that the centres that are a consequence of the fields might well be the result of some sort of symmetry breaking process. That is, a field could well indicate that a centre should exist in one of a number of possible positions. Just like a pencil standing on its point collapses to one of many stable but equivalent positions as the symmetry is broken it seems likely that the same thing would apply to the centre-generating field in that a centre could be created in one of a number of equivalent position. That is, the centre-generating field would allow for a number of "potential centre" positions, one of which would be chosen as the symmetry was broken.

If this analogy with symmetry breaking is useful then the field should allow for some sort of "energy function" meaning that the process of finding centres could be seen as the field collapsing into a lower energy configuration. This is, of course, just an analogy but models like this are useful in all forms of engineering.

Given these issues one possible model that presented itself was the Ising model (Ising model) of magnetic domain creation. This model—due to Ernst Ising, a German physicist—is a simple model of the creation of magnetic domains in ferromagnetic materials. It models the material as a number of cells in a plane where each cell has a "spin", either up or down, that expresses its magnetic moment. The model applies statistical mechanics and shows how below a certain critical temperature, the cells in the space start aligning with their neighbours so as to lower the overall energy of the configuration. The energy of a configuration is calculated by a particular energy function which, for each cell in the configuration, essentially multiplies the energy of a cell with those of its immediate neighbours. The neighbourhood used for this calculation is the simplest possible "Von Neumann neighbourhood" consisting of the cells to the north, south, east and west of each cell. The neighbourhood is shown in Figure 5.

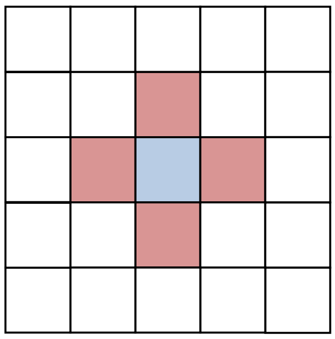


Figure 6: Von Neumann neighbourhood

Core to the model is the notion of the current temperature, as the probabilities that are attached to the sites changing spin value are derived from the Boltzmann probability distribution function which defines the probability that the overall system will be in some particular state at a particular temperature.

The spin S_i of each cell is represented as either +1 or -1 representing a spin direction either above or below the plane of the model. The energy function for a particular cell is then defined to be:

$$E_i = -J \sum_{j=1..4} S_i S_j + BS_i$$

Equation 1: Ising model energy function

where:

- J is the "coupling energy" defining the strength of the spin-spin interaction,
- S_i is the spin of a particular site,
- S_j is the spin of each of the four adjacent sites and
- B represents the coupling energy related to the external magnetic field.

I implemented a simplified version of the Ising model using Java, initially to establish that the model did behave as it was claimed. The main simplifications were to set $J = 1$ and to remove the effect of the external magnetic field by setting $B = 0$.

Figure 7 shows an example of the execution of this model. It shows a typical starting position, where each cell in a 200x200 array has a randomly allocated a spin which is rendered as either a black square or a white square depending on whether it is $+1$ or -1:

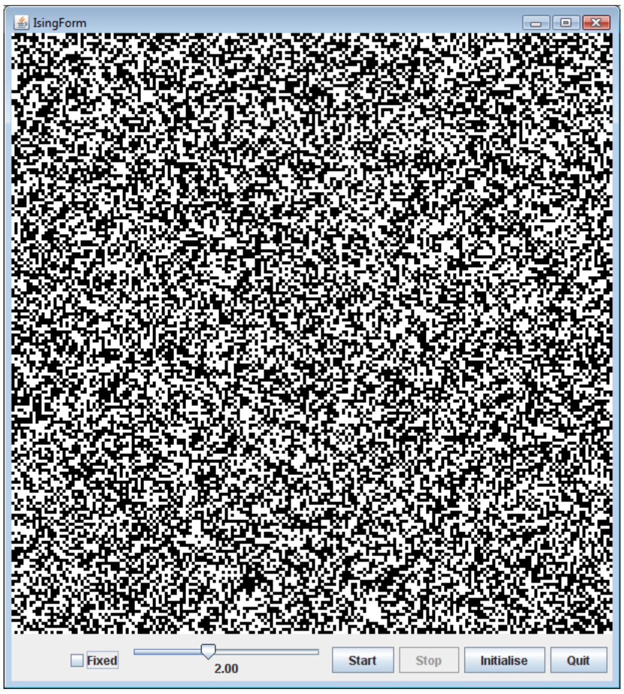


Figure 7: Initial state of Ising model

It should be noted that in this implementation that simulation is taking place on the surface of a toroid. That is, the left of the space abuts the right, and the top the bottom.

The execution of the Ising model follows a Monte Carlo style approach, as in this pseudo code:

```
do
    select random site in model
    calculate current energy of site
    calculate energy of site should the site change spin value
    calculate probability that site will change state according to Boltzmann
        probability distribution function at the current temperature
    change the chosen site in accordance with the calculated probability
while true
```

The probability of the state change is defined, for an energy difference greater than zero, by:

$$p = e^{-\Delta/kT}$$

Equation 2: State change probability

where:

- p is the probability of the state change occurring,
- Δ is the energy difference of the site should the state change occur,
- kT describes the value of the thermal energy due to an assumed "heat bath" environment. Usually k is Boltzmann's constant and T is the temperature in Kelvin. However, by setting $J = 1$ a different (and arbitrary) set of energy units have been defined here.

The particular value of the temperature is critical to the execution of the model. If the temperature is sufficiently high the sites just "jiggle" around and no pattern emerges. Essentially, the thermal energy is swamping any potential reductions in the value of the energy at any particular site..

However, if the temperature is lowered sufficiently, then "magnetic domains" start to form as is shown in Figure 8.

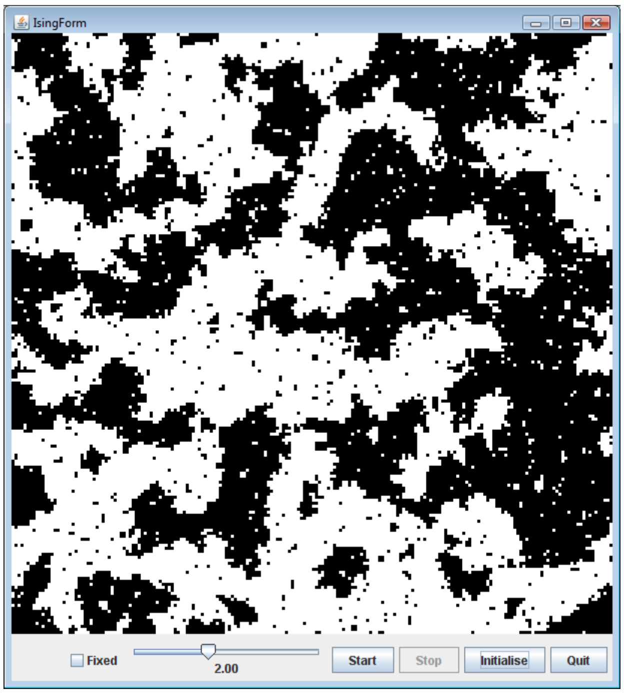


Figure 8: Domain formation at kT = 2

In this particular case the temperature (or, more accurately, the value of kT) is 2.0 as can been seen in the slider at the bottom of the window.

As can be seen from the "noisy" form of this display there is a still a significant probability that a cell that is completely surrounded with "black" neighbours will nonetheless become "white". However, such noise disappears quickly. If the temperature is lowered still further then the probability of such a change is reduced and the simulation is less "noisy". For example, Figure 9 shows the display when kT is set to 0.07.

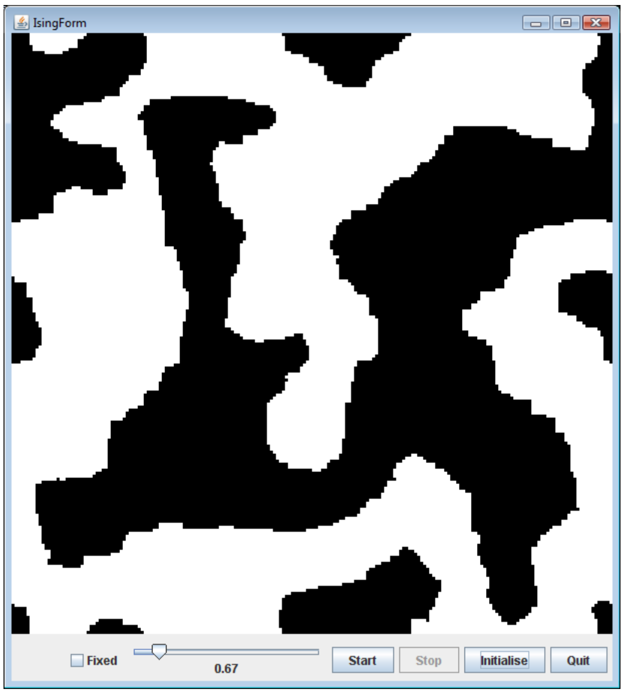


Figure 9: Ising model with kT near zero

If this model were accurate then if were run long enough at a low enough temperature then it would eventually collapse to a configuration that is close to the lowest possible energy configuration, which would have all cells the same spin and hence colour. With a non-zero temperature there is always a probability that any particular cell will change state. However, the probability that this will happen at a low temperature is small and it is likely to soon revert to the original, lower energy, configuration.

However, this simple state is not what is observed with the implementation as the size of the neighbourhoods is very small, just the four nearest cells to a chosen cell. The consequence of this is that the model often reduces to a configuration where the display has a small number of stripes of one colour or another. For example, Figure 10 shows a single large stripe that formed after the model was left executing for about an hour.

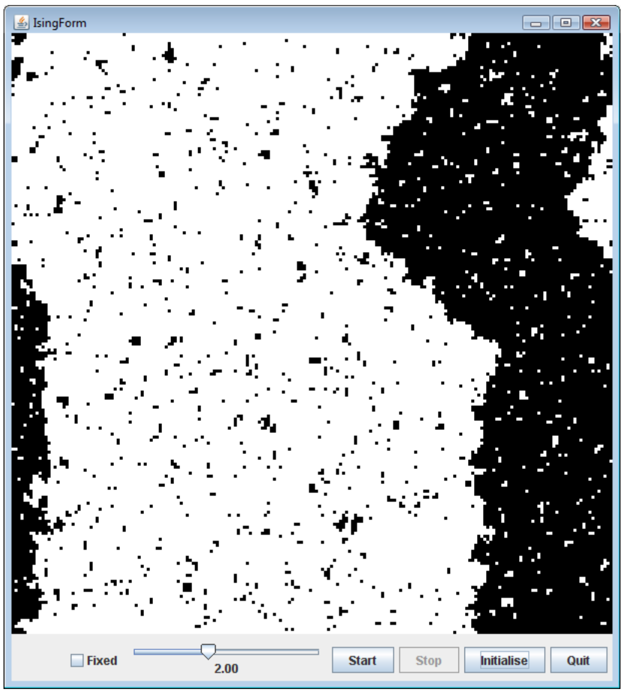


Figure 10: Stripe formation as a consequence of Von Neumann neighbourhoods

The stripes appear because there is no mechanism for evaluating the energy function across the entire space (which would include both black and white parts of this example). The only place where both the black and white portions enter into the energy calculation is at the edge of the stripe. Given the shape of the neighbourhoods then there is no reason for the edge of the stripe to move in any direction and so it tends to stay in position.

It would be possible to make the neighbourhoods larger, which also require using some coefficients to represent the effects of distance on the energy function. However, this has not been done as it is likely that the computational complexity of the implementation would overwhelm the computing power that is currently available.

5.3.1 Adapting the Ising model

The Ising model, then, is a suitable test bed for some of the fields ideas. In particular, it is computational environment where a simple energy function is used to determine how individual parts of a model behave in the presence of an overall field. However, in order to examine the fields that could be useful for Alexander's Generative Patterns more detail is needed in the model. In particular the putative patterns field has a direction that is more detailed than just “up” and “down”.

As a test of the basic idea the Ising implementation has been modified to allow the field vector in each cell to have one of 12 values, representing 30°, 60°, 90° and so on where these values are in the plane of the model, rather than “up” and “down” as for the Ising spin values.

Rather than just using black and white these directions were rendered as one of 12 colours, derived from the colours around the edge of a “colour wheel”. Further, the energy function was modified to use the dot product of each pair of vectors in the site being examined:

$$E = -J\sum\nolimits_{j=1..4} S_i \cdot S_j = -J\sum\nolimits_{j=1..4} S_i S_j \cos\theta$$

Equation 3: Adapted Ising model energy function

where $\theta$is the angle between the two vectors.

That is, the energy function is modified so that if the angles of the vectors in two adjacent cells coincide then the value of the function is at a minimum whereas if the angles are 180° apart then the value of the energy function is at a maximum.

Note that the vectors all have the same magnitude; it is only the direction that is changing.

In order to represent the notion that the field is normal to the edge of the space and of artefacts in that space the Java code was further enhanced to detect such edges and to calculate the normal direction of the field. Further modifications allow for the user to draw such artefacts as are required directly onto the space using the pointing device. In this implementation the notion of the model executing on the surface of a toroid was abandoned, as the notion of an "edge" to the space was needed, and the space is now a simple bounded 2D rectangle. In all the figures shown in this document the rectangle is a 200x200 site square. However, it could easily be modified.

A final refinement is that the display was modified to not only show the coloured display of the cell vectors, but to also display the magnitude of the energy function. This is done in monochrome where, in the reversed manner used by astronomers, black represents the highest energy and white the lowest. Some additional controls were found to be necessary to adjust the black and white level in order to better see the detail on this display.

The end result is a display as is shown in Figure 11. In this case the model is again initialised to a purely random starting position apart from the normal edge vectors. Note that the coloured blobs around the display give an indication of the directions that relate to the cell colours. That is, the predominantly blue bar at the top of this diagram is the colour assigned to a cell that has a vector pointing vertically upward.

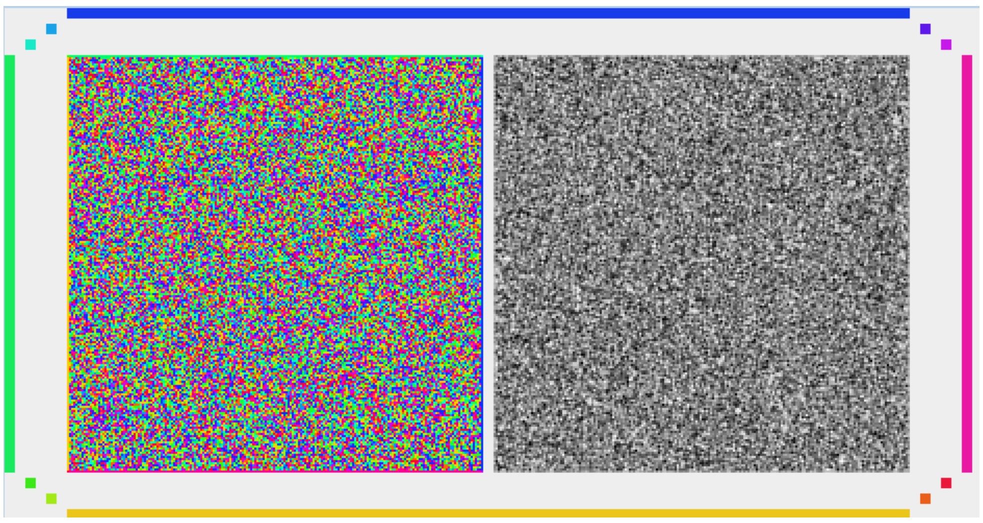
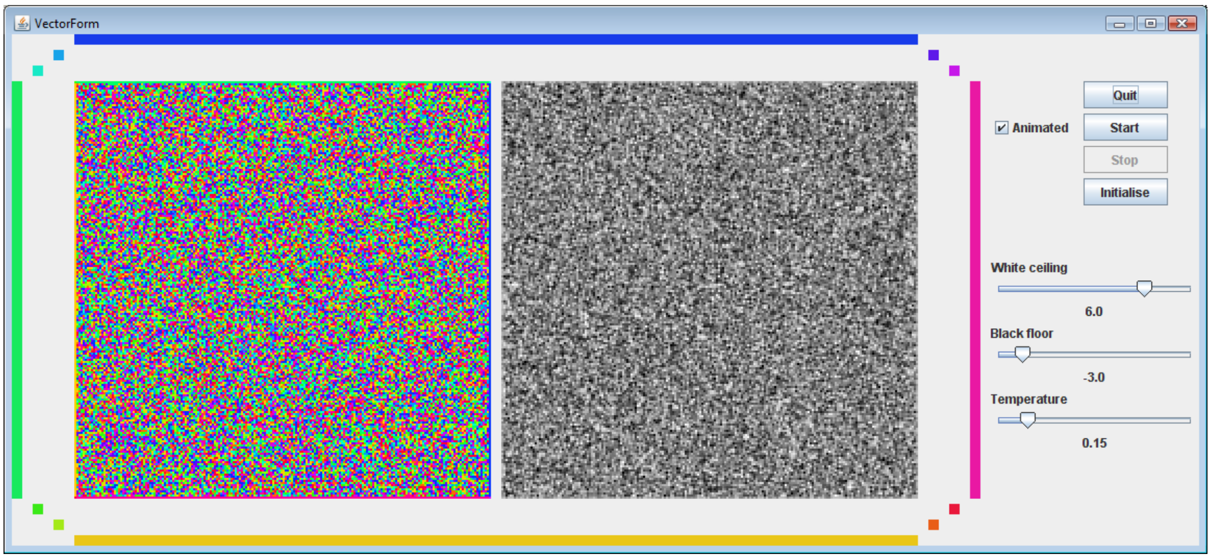


Figure 11: Adapted Ising model display with colour key

The complete adapted Ising display window is shown in Figure 12 where the sliders to control the value of kT and those for adjusting the black and white levels for the energy display can be seen:



White ceiling 6.0
Black floor -3.0
Temperature 0.15

Figure 12: Adapted Ising model display

It is not surprising to find that executing this new simulation is much more costly of CPU time than the simpler Ising model. In the first place the calculations being done are more complex, involving trigonometrical functions. More important though is that the essential process that is going on is one of waiting for the effects due to the fixed vectors at the edge to propagate across the entire space. As the cell neighbourhoods are still small the effects take a considerable amount of time, and computation, to emerge. Simplistically, the effect of a particular edge vectors can only propagate, at the maximum possible speed, by one cell for each calculation. That is, for the effects of the vectors' directions at the left of the space to propagate to the centre of the space, meeting the effects of the vectors propagating from the right of the space then each effect must traverse 100 sites. Even assuming that the effects of propagation are immediately determined this requires 100 cycles of evaluation of the model, typically each cycle would re-evaluate each of the 40,000 sites in the model meaning that 4,000,000 sites would have to be re-evaluated.

The effects actually propagate rather more slowly than this due to two effects. Firstly the effects of the thermal energy provided by the temperature slow down the overall calculation. In fact, the critical temperature for this model is actually rather lower than that for the simple Ising model, as the use of the vector dot products gives a wider range of results which are more easily perturbed by the thermal energy. Secondly, the effects of the edge vectors are more complex than indicated by a simple process where those effects propagate by one cell for each calculation. In reality realistic results take an hour or more of computation on a typical desktop PC.

As a consequence of this some effort was put into optimising the performance of the code. In particular the values of the energy function are memoised and accessed using a hash function based on the values of the vector directions in the adjacent cells to the chosen cell. Further, the screen updating has been removed meaning that the computation can proceed without needing to modify the display especially often. Even with these efforts the calculation is still slow.

It does, though, yield results that are immediately interesting. For example, after the simple state shown above—with the edge vectors held normal to the edges—is left running for about 15 minutes a display like that shown in Figure 13 appears.

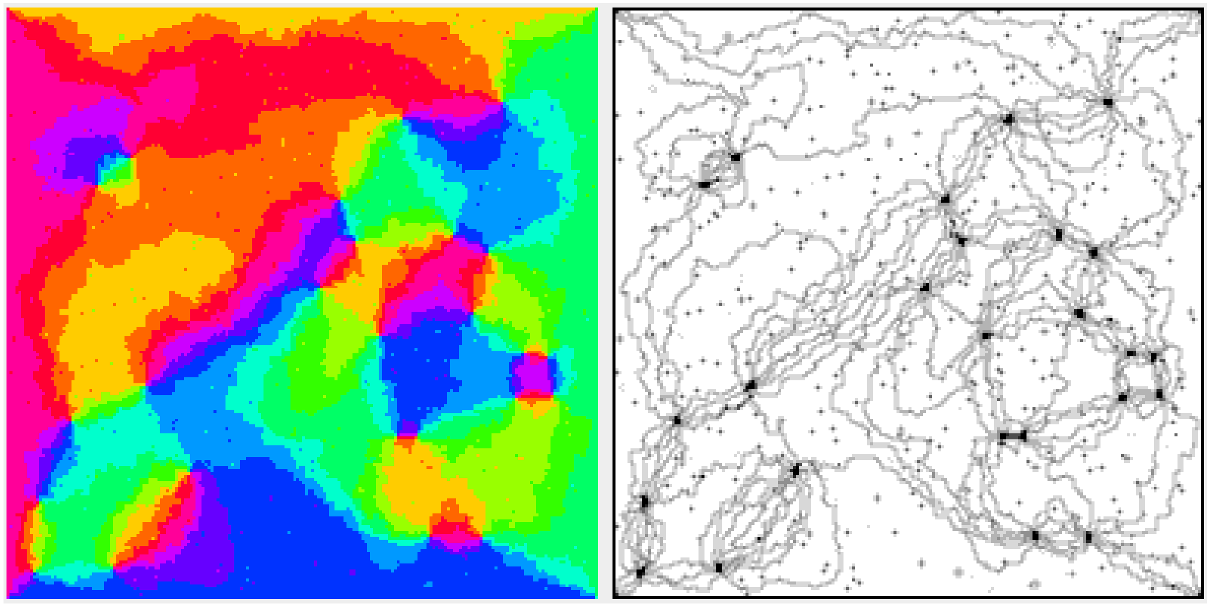


Figure 13: Clumping in the adapted model

This simulation, as for the others here for the adapted Ising model, was performed at a value of kT of 0.1.

As in the conventional Ising model the region has "clumped" together into domains where the angles of adjacent vectors are more or less aligned. However, as the edge vectors are fixed it is required that the complete field shows an overall rotation. That is, as the vectors at the bottom point "up" and the ones at the top point "down" then there must be some change throughout space so that the individual vector directions move from "down" to "up". This change cannot occur suddenly as that would yield points of very high energy and therefore this rotation happens gradually.

However, the very fact that the rotation must be present forces there to be a number of places where there are higher values of the energy function. These are seen clearly as the darker blobs on the right hand display but they can be related to the points in the coloured display where a sequence of colours can be seen around a particular point on the display.

These points are immediately interesting as the hope would be that they might be related to Alexander's centres, or at least the origins of centres, if the model were implementing something sufficiently closely related to Alexander's Generative Patterns. Whether this is actually the case remains for future work but it is nonetheless attractive.

These points are referred to as "poles" because some thought and examination of the colours reveals that they are places where the vector arrows are all pointing "inwards" (towards the pole) or "outwards" (away from the pole). Hence, the analogy with a magnetic field is appealing.

Further execution, in the region of an hour, of the same model makes the poles much clearer. In the subsequent map, shown here as Figure 14, there are five clearly visible poles.

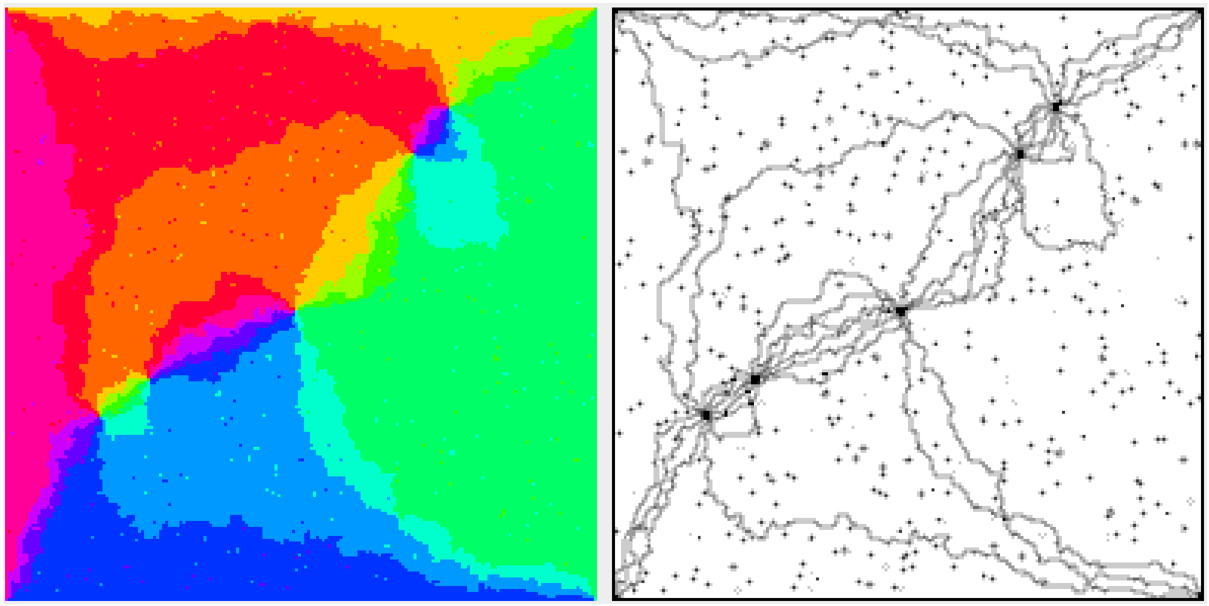


Figure 14: Pole formation in the adapted model

However, it is to be expected that further execution would eventually remove the two pairs of poles that are in the top right and bottom left leaving the lowest energy possible single central pole configuration.

Further work is needed to establish whether the initial appeal of the poles has any utility.

More complex models emerge from a space that contains more artefacts. For example, Figure 15 shows the initial field when set up with a number of objects in the space.

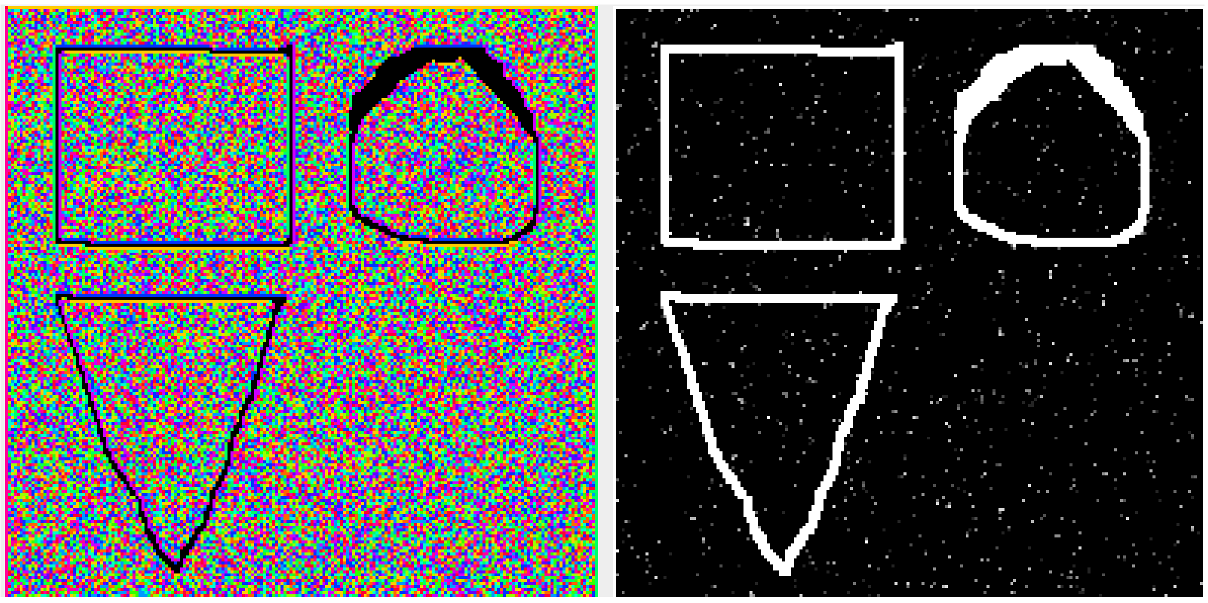


Figure 15: Complex adapted Ising space

Not surprisingly, a much more complex pattern appears in the field after some execution, as is shown in Figure 16.

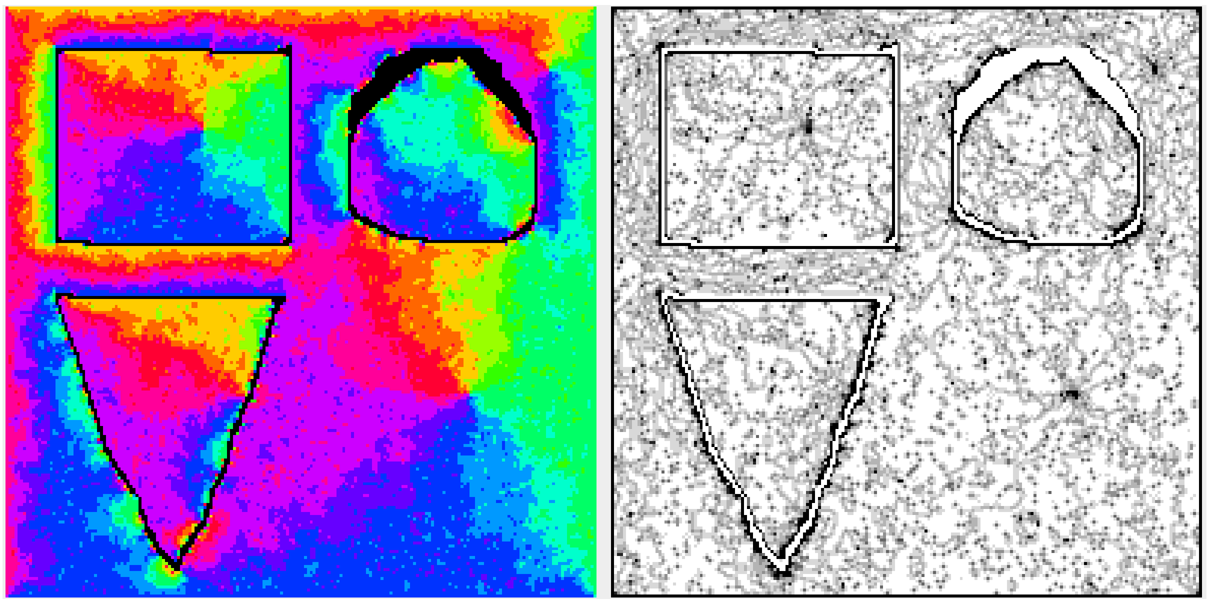


Figure 16: Field propagation in a complex adapted space

It is interesting to note that the space *inside* the square at the top left has a small version of the whole model inside it, complete with a single pole. Note that the space must include an odd number of poles in order to get the rotation required by the normal edge vectors.

It is also clear that the single pole in the main space has been forced to a position that is both a consequence of the overall edge vectors but also the normal vectors due to the artefacts in the space. To some extent, then, this field is qualitatively doing the general sorts of things that Alexander talks about.

A further example is the space shown in Figure 17 which has a single large (poorly drawn) circle in it.

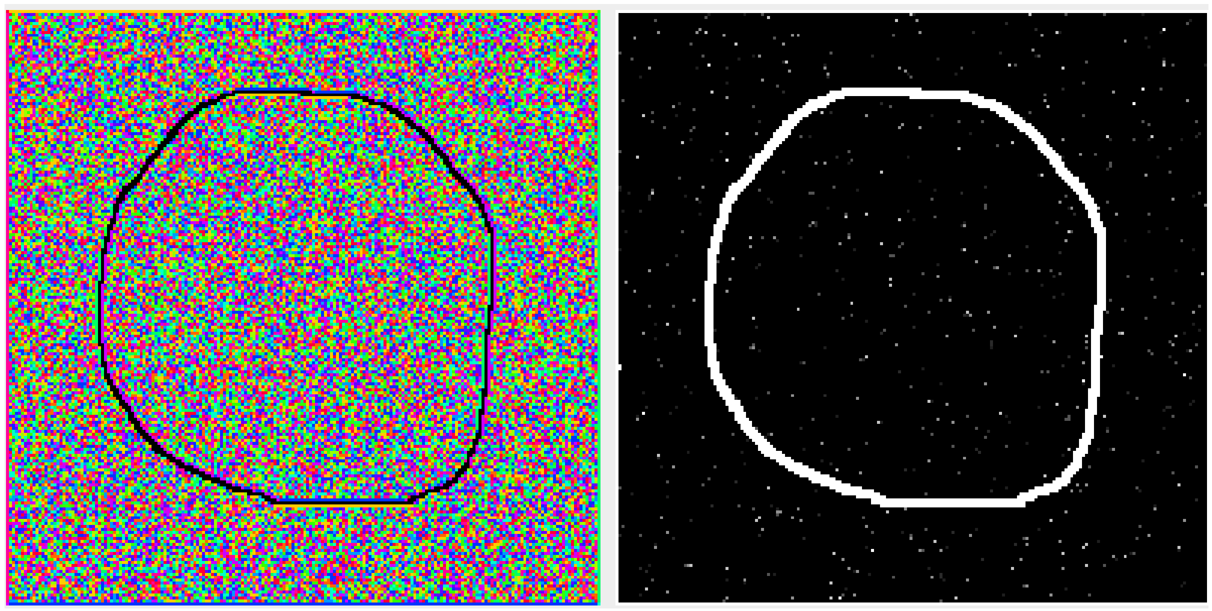


Figure 17: Adapted Ising model with large circle

When left to execute for about an hour then the pattern shown in Figure 18 emerged.

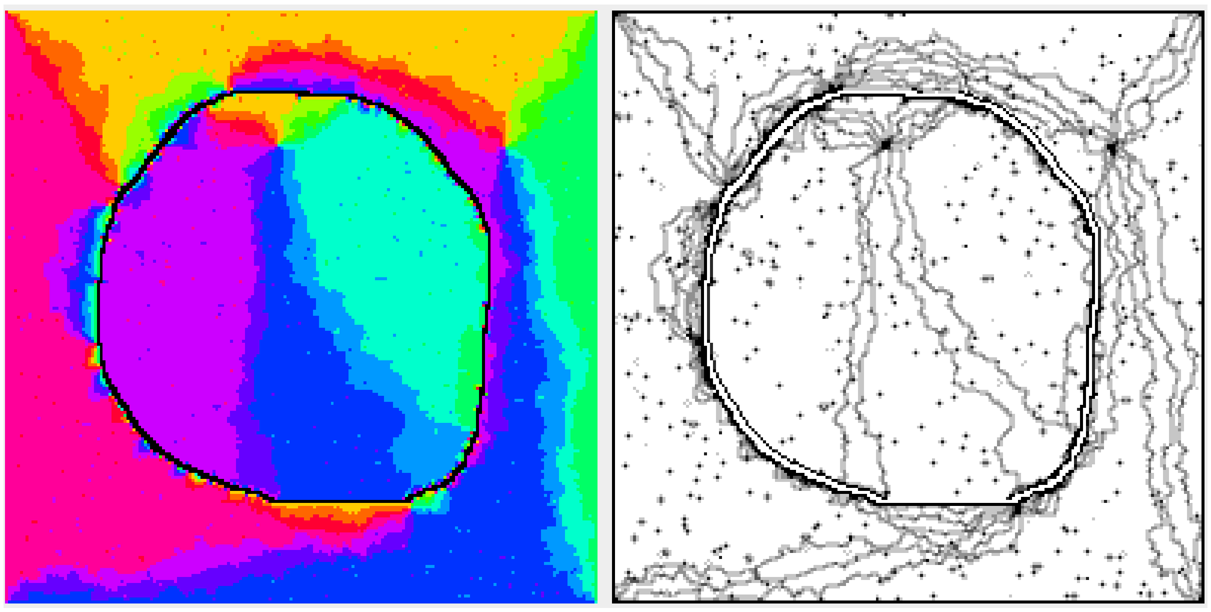


Figure 18: Post execution adapted Ising model with large circle

Again, there are poles in this result but the presence of the circle has forced the main pole to appear in the top right part of the display. Intuitively, its appearance here is equally likely to its appearance in one of the other three corners. There are also other poles around the edge of the circle, in addition to a predominant one in the space inside the circle. It is not yet clear whether these "edge poles" are phenomena worthy of investigation or whether they are a consequence of the poor quality of drawing which has made the boundary of the circle rough.

One of Alexander's generative properties is *Roughness* and this example shows what appears to be a result in a field as a direct consequence of that roughness. It appears as though the rough parts are being used as "nucleation points" for the creation of high energy parts of the field. Whether this is actually an example of Alexander's property, though, needs further work.

5.3.2 Initial conclusions on fields

This model as it stands is not capable of exposing the range of expression that Alexander's properties require. However, it does seem rich enough to be worthy of future work. There are a number of simple extensions that can be made to the software in order to allow further investigation. Amongst these the most immediately obvious, apart from further attempts to profile the code's execution with the intent of further improving the performance, are:

- Look into finer graining of the vector directions. Some of this has been done already and the indication is that the artefacts that are seen are not merely the result of quantisation issues. However, it is worthwhile checking slightly further.
- As has been discussed the essential notion in this model is to represent how a field that is due to the edges of a space, and the artefacts that it contains, spreads through that space. As such it is possible that the magnitude of the vectors is significant; it is clear, for example, that when Alexander makes drawings of his perceptions of the fields that he draws arrows of different lengths and the current model does not allow for that. If variations in the field magnitude are significant then it is likely that there would be some

sort of "decay" aspect to the field as a cell became further and further away from the edge vectors. The current model does not implement anything like that and some experimentation is sensible. In the current model the poles are those places where the vector arrows "collide". With the additional of variable magnitude they might be places where the magnitude approaches zero.

- Similarly, the small neighbourhoods in use in the code do not allow many "distance" effects. Although it seems likely that it could just lead to a massive combinatorial explosion, with the inevitably deleterious effects on performance, some attempts at getting such "distance" effects into the model by manipulation of the neighbourhoods is worthwhile.

In the longer term, other aspects should be investigated in particular:

- Adding additional types of field, in particular a divergence free "flow" field and looking into the effects of the field combining in various ways.
- The current fields are all static, equilibrium fields. As a generative process develops the fields will change as a result of the application of the generative operators. Whether the fields should be allows to stabilise, and what effects leads to that stabilisation, is currently an open question and worthy of investigation.

6 Generative patterns of software

The Ising, and adapted Ising, models show that some *implementation* of Alexander's Generative Patterns may be feasible in that it may be possible to detect centres that are the basis for applying Alexander's operators. The principal objective of this work, that is, the examination of Alexander (2009)'s hypothesis, relates to the application of these sorts of ideas to software systems, both complex and classic.

The patterns under discussion are concerned with generative effects that influence the development of a building or building site. Hence, it is to be expected that related patterns for software will influence the process of software development itself, rather than just the form of the final software. Although some of Alexander's examples, such as that of St. Mark's Square as in (Alexander, 2009) are shown from the point of view of the completed buildings he also documents the explicit application of the generative process, for example that of the Boston housing project (Alexander, 2009).

The equivalent stage to a completed building is a running piece of software. That is, in order to use generative patterns to influence the development of some piece of software, and therefore to modify its architecture, then that software must to some extent already be executing.

It is therefore inevitable that the use of generative patterns in software development would require a different approach to software development itself. In particular, it would require a more "experimental" approach to software development where the objective would be to run the software, or at least some software, as early in the development process as possible. In principal, the generative patterns would then be able to assist in the further development of the software until it met both its functional and non-functional requirements.

Such an approach to software development is already becoming more and more the norm for classic systems. The various "agile" (Agile Manifesto) approaches that are becoming common take as a starting point the notion that a project will construct running software from a very early point in the development. Indeed, a core tenet of XP, one of the original agile approaches (Beck et al, 2004), is to "build the simplest thing that could possibly work". Such a system, which must be able to run, is then monitored, tested and refactored into a final form that is seen as satisfactory for meeting the system's requirements.

This is a very different approach from the traditional one for software development (Royce, 1970) which is often characterised[5] as following a "waterfall" process where one task must be completed before starting the next. However, such a development process, in particular with "Big Design Up Front" (BDUF) is still a part of much contemporary software development.

It is interesting to observe that the process that Alexander follows, for example in (Alexander 2007b) is in effect an agile process. He marks things out in the physical world with tape and other temporary structures until such time as they *must* be committed to their final form as concrete, glass, brick, etc.

One of the main reasons why the agile approach is viable for software development is that the requirements for a software system are never static. The environment within which some software executes may change, often as a direct result of the software being executed, which has the effect of changing the requirements. Similarly, the users of some software may alter the way they use the system, implicitly changing the requirements. Or it may be the case that the original procurers of the system realise that some changes are required.

All of these aspects and many similar ones have the effect of continually changing the requirements and as a consequence software systems are rarely static. As such if some generative patterns were available for use during the development of the software then the same patterns would continue to be useful during the life of a system. Such use could even have the beneficial effect of continuing the development in the long term and avoiding the oft-noted effect where a piece of software essentially rusts away over time until it is no longer useful; usually because the software's environment has changed in several fundamental ways.

The general notion of the meta model described in section 5.1 should therefore be applicable to a software system. Although the thrust of this research is towards complex systems there seems to be no reason why that model does not apply. The essential abstractions in the model can be seen to refer to the software itself, some representation of the current behaviour of the software (the "centres" of a building site) and the properties that describe the value of the centres and generate new aspects of the software.

In order to use this general approach then, we need to:

a. produce software in a manner that means it is available for execution as soon as possible in the development process,
b. have some way, analogous to Alexander's centres, of describing aspects of the software's execution that are revealing of its behaviour and
c. have some properties, of that record of execution, that provide a set of suitable metrics of that execution and suggest ways of changing the software's architecture in order to improve the overall behaviour.

[5] This characterisation is unfair as the original Royce (1970) paper did not actually propose a waterfall approach.

All of this must apply to complex systems. On first sight, there seems no reason why it might not apply to classic systems as well, although it is likely that the specifics of b. and c. would be different.

At the current state of this work, very little has been done on these issues. However, as Alexander's patterns are solidly based on insights of the physical world a viable strategy might well be to express important aspects of software execution as some sort of spatial structure. Properties of that space might then be able to guide the generation of future parts of the software.

There are already some aspects of software development that are broadly reminiscent of this. For example, the Java hotspot optimiser (Sun, 2002) makes performance optimisations in a manner that is driven by how the running code uses the *execution space* of the program.

One technique that may offer a way of representing as a two dimensional space the execution of a software system is that of Self Organising Maps (Kohonen, 1995). These are essentially neural networks which learn from their input data in a way that physically summarises the variations in that data. However, little work has as yet been done on this possibility.

7 Proposal

In conclusion, the proposal for work to be done in the rest of this research has several facets:

- Work should continue to understand Alexander's Generative Patterns and to describe them and their effects precisely, most likely by some sort of fields model expressed using the adapted Ising model described here or something similar.
- Investigate mechanisms to represent the execution of a complex system in some sort of spatial manner.
- Using the architectural insights of the rest of the CoSMoS project, use those mechanisms along with generative patterns of the similar sort of form to Alexander's, to influence the structure of running system.

This is a very ambitious group of ideas and it is likely that even if it is successful, it can only be successful for some subset of the overall problem. For example, only some sorts of execution might be modelled, or only one system architectural change represented.

8 References

Agile Manifesto, 'Manifesto for Agile Software Development', [Online] Available at: http://agilemanifesto.org. Last accessed 1st July 2008.

Alexander, C. 1965. *A city is not tree*, Architectural Forum 122 April (1965): No. 1, pages 58—61 and No. 2, pages 58—62.

Alexander, C, Ishikawa, S & Silverstein, M with Jacobson, M, Fiksdahl-King, I & Angel, S. 1977. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press

Alexander, C, 2002. *The Nature of Order: An Essay on the Art of Building and The Nature of the Universe.* Four volumes: *The Phenomenon of Life, The Process of Creating Life, A Vision of a Living World* and *The Luminous Ground.* The Center for Environmental Structure, Berkeley, California.

Alexander, C. 2007a. *Coherence Operators (from Sustainability and Morphogenesis).* Unpublished.

Alexander, C. 2007b. *Morphogenesis of a Window In a Texas House (from Sustainability and Morphogenesis).* Unpublished.

Alexander, C & Cowan S. 2008. *The Field of Wholeness*. Unpublished.

Alexander, C. 2008. *Alhambra Vector Field Examples; Preliminary Memo*. Personal communication.

Alexander, C. 2009. *Harmony-Seeking Computations: a Science of Non-Classical Dynamics based on the Progressive Evolution of the Larger Whole*, to appear in The International Journal of Unconventional Computation

Allen, R. 1997. 'A Formal Approach to Software Architecture'. Ph.D. Thesis. Carnegie Mellon University. *Technical Report* CMU-CS-97-144, May, 1997.

Ambler, S, 1998, *Process Patterns, Building Large-Scale systems using Object Technology*, Cambridge University Press.

Bass, L, Clements, P & Kazman, R. 1998,*Software Architecture in Practice,* Addison Wesley Longman.

BDUF. , 'Big Design Up Front, [Online] Available at: http://en.wikipedia.org/wiki/Big_Design_Up_Front. Last accessed 1st July 2008.

Beck, K, 1996, *Smalltalk Best Practice Patterns*, Prentice Hall.

Beck, K, Andres, C. 2004. *Extreme Programming Explained: Embrace Change*. Addison Wesley.

Berlekamp, E. R, Conway, J. H & Guy, R. K. 1982. *Winning Ways for Your Mathematical Plays Volume 2: games in particular*. Academic Press.

Booch, G. 1993. *Object Oriented Analysis and Design with Applications*. 2nd edition. Addison Wesley.

Booch, G. 1995. *Object Solutions*. Addison Wesley.

CoSMoS, 2007. *CoSMoS: Complex Systems Modelling and Simulation Infrastructure: grant proposal*. Unpublished.

Fowler, M. 1997. *Analysis Patterns: reusable object models*. Addison Wesley Longman.

Fowler, M. 1999. *Refactoring: improving the design of existing code*. Addison Wesley Longman.

Fowler, M. 2003. *Patterns of Enterprise Application Architecture*. Pearson Education.

Fowler, M. 2004. *UML Distilled: a brief guide to the standard object modeling language*, Addison-Wesley.

Gamma, E, Helm, R, Johnson, R & Vlissides, J. 1994. *Design Patterns: elements of reusable object-oriented software,* Addison-Wesley.

Gardner, M. 1970. 'Mathematical Games: The fantastic combinations of John Conway's new solitaire game "Life"'. *Scientific American.* 223: pp 120–123, October 1970.

Gosper, B. 1984. 'Exploiting Regularities in Large Cellular Spaces'. *Physica D. Nonlinear Phenomena*. Volume 10, pp 75—80.

Grosso, W. 2001. *Java RMI*. O'Reilly Media.

Hoare, C.A.R. 1985. *Communicating Sequential Processes*. Prentice Hall.

IEEE Computer Society, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*: IEEE Std 1472000. 2000.

Ising Model, 'Ising Model', [Online]
Available at: http://en.wikipedia.org/wiki/Ising_model. Last accessed 1st July 2008.

Kohonen, T. 1995. *Self Organising Maps*. 3rd edition. Springer; London.

Kruchten, P. 1995. *IEEE Software.* 12 (6) November 1995. pp 42—50.

Luckham, D & Vera, J. 1995. 'An Event-Based Architecture Definition Language'. *IEEE Transactions on Software Engineering*, 21 (9), September 1995, pp 717—734.

Magee, J, Dulay, N, Eisenbach, S and Kramer, J, 1995. 'Specifying Distributed Software Architectures'. *Proceedings of 5th European Software Engineering Conference (ESEC '95)*. Sitges, September 1995, LNCS 989, (Springer Verlag), 1995, pp 137—153.

Meyer, B. 2000. *Object Oriented Software Construction.* Prentice Hall. 2nd edition.

Medvidovic, N & Taylor, R. 2000. 'A Classification and Comparison Framework for Software Architecture Description Languages'. *IEEE Transactions on Software Engineeering*. 26 (1) January 2000, pp 70—93.

Ommering, R. 2000. *The Koala Component Model for Consumer Electronics Software*. IEEE.Computer, **33**, 3, March 2000, pp 78—85.

Perry, D, Wolf, A. 1992. *Foundations for the Study of Software Architecture.* ACM SIGSOFT Software Engineering Notes, 17:4, October 1992.

Polack, F & Stepney, S. 2005. 'Emergent Properties Do Not Refine'. *Electronic Notes in Theoretical Computer Science* 137 (2005) pp 163—181.

Ramo, S, Whinnery, J & Van Duzer, T. 1965. *Fields and Waves in Communication Electronics*. John Wiley and Sons.

Reynolds, C. W. 1987. 'Flocks, Herds, and Schools: A Distributed Behavioral Model'. *Computer Graphics, 21(4) (SIGGRAPH '87 Conference Proceedings)* pp 25—34.

Royce, W. 1970. *Managing Development of Large Scale Software Systems*, Proceedings of IEEE WESCON, August 1970.

St. Exupery, A. 1939. *Terre des hommes.* ***III****: L'Avion.* [In French.]

Sun, 2002. *The Java HotSpot™ Virtual Machine*. White paper available online at http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/JHS_141_WP_d2a.pdf. Last accessed 1st July 2008.

Thomas, J, Young, M, Brown, K & Glover A. 2004. *Java Testing Patterns.* Wiley

Thompson, DW & Bonner, J (ed). 1961. *On Growth and Form; abridged edition.* Cambridge University Press.

Wirth, N. 1976. *Algorithms + Data Structures = Programs.* Prentice Hall.