Complex Adaptive Responses of Organizational Teams

AJB (Barry) Myburgh

Joburg Centre for Software Engineering, School of Electrical & Information Engineering, University of the Witwatersrand Johannesburg, South Africa

Email: barrym@jcse.org.za

Abstract

Forces of Management and Production Governance can direct a self-organizing team to achieve an appropriate state of dynamic equilibrium to enhance the probability of success. CASM (Complex Adaptive Situational Model) promotes an understanding of what it takes to establish those conditions and applies to any software engineering team, irrespective of the chosen way of working: agile or plan-driven. After reviewing some aspects of Complexity Science, CASM is presented as a model of complexity with four states of dynamic equilibrium in the zone between chaos and order. The four states are: Crafted Quality (agile), Controlled Quality (plan-driven), Managed Costs (WetAgile) and Self-Directed Quality. A band of software engineering feasibility is also described as the area in which teams are most likely to deliver successful projects. The journey across the band of feasibility is further described by introducing SEMAT, with Crafted Quality amounting to applying native SEMAT Essence, and Controlled Quality being achieved by introducing additional practices which satisfy more stringent governance requirements. CASM in its four states allowed introduction of the idea of describing an enterprise as a collection of CAS’s (the Enterprise Amoeba), thereby setting the scene for further research into the complexities of human-driven complex adaptive systems. CASM’s model of complexity could also be seen as making a contribution to the general theory of software engineering.

Keywords


© 2015 by Orb Academic Publisher. All rights reserved.

1. Introduction

For many practitioners, agile software development seems the best way to develop software. But old-style management often presents the biggest obstacle to successful adoption of agile approaches. The primary model described in this paper (CASM) was first published by Myburgh in 2014 [1] and it promotes understanding of what it takes to establish conditions which enable software engineering success, not only with agile approaches, but also traditional, plan-driven software engineering.

Humans easily relate to causal determinism – a thesis based on experience that future events (combined with the laws of nature) are sometimes the result of past and present events. Because of this we can catch a ball by predicting in which direction it is going. We can predict the return of Halley’s Comet in 2061. Causality also enables software developers to design, plan, and predict what software will do. Immanuel Kant (1724-1804) promoted universal causal determinism.

But causality is not enough. We can’t accurately predict the weather. We can’t predict the full combination of features, qualities, time and resources of a software project. As explained by Jurgen Appelo: Predictability has a devious sister called complexity [2, p. 2].

2. Complexity

Our attempts at understanding complexity involve: Dynamical systems theory; Chaos theory; Network theory; Game theory and other branches of science that are collectively known as the Complexity Sciences. Causality ruled the sciences from the 17th century. Complexity is a product of the 20th century. Complexity theory offers a new way of understanding the problems of...
managing organizations and producing software - even though our minds prefer causality over complexity.

Simplicity usually relates to the burden which a thing puts on someone trying to explain or understand it. Something which is easy to understand or explain is simple. In contrast to something complicated. We need to know the difference between complex and complicated.

The first dimension to consider is about the **structure** of a system and how well we understand it: simple implies easily understandable and complicated implies very hard to understand.

The second dimension is about the **behavior** of the system and how well we can predict it: ordered implies fully predictable and complex means somewhat predictable (but with many surprises). Chaotic is very unpredictable.

Complicated refers to a system’s construction being too intricate to understand. Complex and chaotic refer to a system’s behavior.

The human brain is wired to find purpose and causality in everything and we favour “linear thinking” to “nonlinear thinking”. So we easily reason that the global financial crisis was caused by bankers. The loss of jobs is caused by immigrants. Bad atmosphere at work is caused by the manager. The team didn’t make a deadline because of someone’s mistake.

The mental addiction to causal determinism has led people to use control to ensure desired outcomes. Engineers and other people with technical minds are particularly susceptible to the concept of control. Engineers developed scientific management - the command-and-control style of management. Engineers devised the kind of control systems we still find today which causes that would produce the outcomes exactly as they need them: through careful up-front design, with meticulous top-down planning. Appelo explains that agile management derives when hierarchical management embraces complexity and non-linear thinking and is a logical companion to agile software development [2, p. 11].

### 3. Challenges of Software Engineering

The software development industry started in an ad hoc way with the term “software engineering” first appearing in the 1968 NATO Software Engineering Conference where attention was given to the perceived “software crisis” of the time. In essence, it referred (and still refers) to the difficulty of writing correct, understandable, and verifiable computer programs. The roots of the software crisis have been recognized as being complicatedness, expectations, and change. All too often formal approaches introduced bureaucracy and delivered software much more slowly than the rate at which requirements were changing. At the same time, some teams of passionate and disciplined programmers, with ad hoc processes and flexible requirements, delivered products of higher quality at a fraction of the cost and in a fraction of the time.

The dilemma created by the constantly high rate of software project failure in the midst of a multitude of alternative ways of working, recently triggered the search for general theories of software engineering that could achieve recognition equivalent to that of, for example, Maxwell’s equations in the electrical engineering community. But where Maxwell’s equations deal with translating natural phenomena into usable practice, software engineering is all about people translating their ideas into operational solutions by applying process and technology. This translation is enabled by design which, according to John Gero and quoted by Kruchten [3], is a goal-oriented, constrained, decision-making, exploration, and learning activity which operates within a context which depends on the designer’s perception of the context. In the same article Kruchten explains that he had to extend the boundary of “software design” to include much more than software practitioners’ traditional activities as defined in the Software Engineering Body of Knowledge (www.swebok.org). In SWEBOK, software design covers only a narrow set of processes and artifacts [4]. But if we accept that design is making choices that will shape the final product, we must include some requirements activities and all coding and testing activities.

The significance of this statement is that, contrary to most other engineering disciplines, software engineering requires that the design process remains active throughout – up until the very moment that source software is translated into executable machine language. And people drive the design process. As is described later in this article, people are the active agents in a complex adaptive system (CAS). And CAS agents respond to constraints while applying rules. Thus a general theory of software engineering could be found in describing the constraints, rules and emergent behaviour of a CAS that is made up of a team of software engineers.

An early case study that deals with the tension between approaches is described in Dee Hock’s fascinating book “Birth of the Chaordic Age” (1999) [5]. He describes how, in the 1960’s, a management team responded to their concerns when the traditional approach to system delivery was failing. The team took ownership of the challenge and we shut ourselves in a room and didn’t come out until we had an approach to which we were totally committed. [5, p. 205] He also confirms that: *out of initial failure grew a magnificent success.* [5, p. 207]

In 2001 a gathering held in Utah resulted in formulation of the Agile Manifesto [6] which was, on the one hand, a reaction against the bureaucracy of the formal approaches, while on the other hand, also taking a stand against the chaotic processes and low quality products of undisciplined programmers. It gave substance to the search for a middle road between structure and non-structure, between order and chaos.

Evidence demonstrates that agile software development, when done well, shows a tremendous return on investment. But if agile methods have such positive effects, why doesn’t everyone use them? And why are so many software projects across the world still failing? Appelo refers to a “State of Agile Development Survey 2009” [2, p. 28] which identified the
The following factors as contributing to failed agile approaches:

- Management opposed to change
- Loss of management control
- Lack of engineering discipline
- Team opposed to change
- Quality of engineering talent
- Organizational need for planning, predictability and documentation

This seems to suggest that management preferences are the biggest obstacles to agile software development. A suggestion that is strengthened when Keskinen, Aaltonen and Mitleton-Kelly state that some of the most profound assumptions behind contemporary management theories must be re-evaluated [7, p.16]. CASM described in this article sheds further light on this thinking.

In 2009 a group of leading international software engineering personalities started collaborating on an initiative to “re-found” software engineering. Ivar Jacobson (Use Cases, UML, RUP), Bertrand Meyer (Design-by-Contract and the OO Language Eiffel) and Richard Soley (CEO of the Object Management Group (OMG)) established the SEMAT Initiative - Software Engineering Method and Theory. Supporters of the initiative signed a declaration somewhat reminiscent of the Agile Manifesto and since then a great deal of work has been carried out aimed at defining the kernel of widely-agreed elements - often referred to as “Essence”, meaning the essence of software engineering.

Figure 1 highlights two important aspects of the SEMAT kernel:

- The Areas of Concern (Customer, Solution and Endeavour) and
- The Alphas (Opportunity, Stakeholder, Requirements, Software System, Team, Work and Way of Working)

Areas of concern are addressed in terms of Activity Spaces which involve the actions taken to achieve objectives. Alphas represent essential aspects of software engineering and each progresses through a number of states (Alpha States) as the team conducts work.

As described in the submission to the OMG [8] and in the published book “The Essence of Software Engineering: Applying the SEMAT Kernel” (2013) [9], the SEMAT initiative promises to have a fundamental effect on the discipline of software engineering.
4. Complex Adaptive Systems

A view shared by many software development experts and Agile/Lean evangelists is that software projects are complex adaptive systems (CAS’s). CAS’s are composed of agents - as described by M. Mitchell Waldrop [20]. CAS agents can be molecules, neurons, web servers, fish, starlings, and people - always forming new emergent structures with new emergent behaviours. Keskinen, Aaltonen and Mitleton-Kelly explain that a CAS consists of agents, each of which behaves according to its own principles of local interaction, local logic and local rationality. They all have their own history, present and future [7, p26]. Software projects involve people who are constantly organizing and reorganizing into larger structures: Project teams; Social groups; Task forces; Committees; etc.

CAS’s are able to adapt to their environments: an infant learning to walk; a strain of bacteria becoming resistant to an antibiotic; car drivers evading a traffic jam; an ant colony learning about the location of spilled honey; a software team adapting to what their customer really wants. Moving to the sweet spot between chaos and order, they learn and adapt and navigate their way with Chaordic [5, p. 3] processes that are neither fully ordered nor truly chaotic.

In agile SW development, we often hear reference to scientific terms such as self-organization and emergence. The concepts of emergence and the factors leading to emergent results lie at the heart of CAS theory’s relevance to software development.

Examples of self-organizing systems include an ant colony, the brain, the immune system, a Scrum team, a CMMI-Dev team.

Scrum and CMMI-Dev are not methodologies with defined processes or sets of procedures - they are development frameworks. And the frameworks provide rules and constraints on behaviour that cause a CAS to self-organize into an intelligent state of dynamic equilibrium.

When applying complex systems theory to software development and management, we are treating the organization as a system. System dynamics - not to be confused with dynamical systems theory - is a technique from the 1950’s to help managers understand and improve their industrial processes. System dynamics recognized that structure is often a more important contributor to an organization’s behaviour than individual parts themselves. Systems Thinking was developed in the 1980’s and popularized by Peter Senge’s book “The Fifth Discipline” (1990) [21]. It’s about understanding how things influence each other in the whole, a problem-solving mind-set that views problems as parts of an overall system. In some ways similar to System Dynamics, but more subjective. Social complexity is the study of complexity in social systems and to manage social complexity, we need to understand how things grow - not how they are built. This is an extension of ideas promoted by Fred Brooks in 1987 when he explained that the very essence of software engineering lies in complexity, conformity, changeability and invisibility [22]. Appelo’s “Management 3.0” [2] applies complexity thinking. It assumes managers cannot construct and steer a self-organizing team. The team must be grown and nurtured.

Productive organizations are not managed with models and plans; they must emerge through the power of self-organization and evolution. Appelo suggests that complexity thinking is like the light that feeds all that grows [2, p. 50].

Appelo goes on to explain that at the project level, new emergent structures form and new emergent behaviours are displayed [2, p. 51]. Like any other CAS with interconnected agents (people) interacting with each other to form an integrated whole. Even though software projects have many elements, only people are the real agents - the active elements. Teams themselves are agents on the next higher level.

Items that are not agents include: Requirements; Features; Artefacts; Deliverables; Tools; Technologies; Processes; Practices. They cannot actively organize and reorganize themselves. They cannot initiate interaction with any of the other elements in the project.

People are the only ones capable of controlling software projects because only people have the level of complexity required to manage complex systems. And any complex system, if it is to produce useful results, needs some level of control. Not documented processes, or code generators, or project management tools, or the most exquisite upfront designs can ever hope to have the amount of complexity that any ordinary software project possesses. Processes, tools, designs cannot outperform their masters - without people they are useless. If some level of control is needed in a project, we’d be well advised to select people as the control mechanism – they’re the only ones complex enough to handle it.

Appelo emphasizes that the primary focus of any manager should be to energize people - to make sure that they actually want to do what’s required of them. Like a gardener looking after plants in a garden, a manager looks after the employees on the team/s. Motivation is the activation or energization of goal-oriented behaviour [2, p. 58]. It is therefore crucial for managers to activate or energize the people in the complex systems that we call software teams.

For centuries mathematicians have preferred to work with linear (ordered) systems and considered nonlinear (complex) systems to be a special group. But nonlinear systems are the norm and abundant throughout the universe, whereas linear systems are a rare and special breed. From the beginning of the universe, everything in it was shaped by self-organization. Self-organization is the process where a structure or pattern appears in a system without any central authority or external element imposing it through planning. Self-organization is the norm. It is the default behaviour of dynamic systems, whether these systems consist of atoms, molecules, viruses, species, businesses or software developers. Appelo emphasizes that self-organization is not a “best practice” - it is “default practice” [2, p. 100].

No matter how a team is managed, there will be self-organization. People will discuss and agree on lunch meetings, folder struc-
tures, workplace territories, birthday parties. Everything that management does not constrain - and much that it attempts to - will self-organize. Humans have behaved that way for 200 000 years.

But is what happens also happening in the “right direction”? Though every self-organizing system can have its own direction, the possible directions are limited by its environment. No self-organizing system exists without context. And the context constrains and directs the organization of the system.

Environmental constraints affect the direction taken by a self-organizing system. This is illustrated by considering the Game of Life - a simple zero-player game invented in 1970 by the British mathematician John Conway. It is “played” on a grid of cells, where each cell has eight neighbours, one in each direction, including the diagonals. The cells can be born and stay alive or die as determined by the application of rules. The Game of Life is an example of a cellular automaton - a mathematical system in which cells are influenced by other cells according to some set of predefined rules. It is particularly interesting because it is a fine example of a system with a small set of simple rules, having complex behaviour and ordering itself. The game also shows us that, whatever the initial situation is, the system will eventually always stabilize.

There is, however, one catch: the set of rules has to be chosen carefully. We therefore observe that rules must be tuned for a system to be both stabilizing and lively. A different set of rules leads to a different system with different behaviour. As described by Appelo [2, p. 149] Stephen Wolfram proposed a classification scheme for cellular automata - named universality classes.

- **Class I**: These are the systems with “doomsday rules”. No matter what pattern of living and dead cells at the start, everything dies within a few generations.

- **Class II**: These systems are a bit livelier, but not much. Each initial pattern quickly collapses to a set of very boring static configurations.

- **Class III**: These systems are at the opposite extreme: They are too lively. Each initial pattern in the system results in total chaos with no configuration stabilizing and nothing being predictable.

- **Class IV**: These are the systems with a set of rules not leading to dead, static, or chaotic configurations. Emerging patterns in this category are lively, creative, often surprising, but also stabilizing.

In dynamical systems, Classes I and II correspond to order. Class III corresponds to chaos. Class IV (of which the Game of Life is a famous example) corresponds to complexity. Given that complexity is usually explained as the region between order and chaos, this means that class IV finds itself between II and III.

Complex adaptive systems are systems that can find their own way toward that sweet spot of complexity, between order and chaos, where life blooms and creativity thrives. Scientists call it the edge of chaos, but they also could have called it the edge of order. This sweet spot represents a state of dynamic equilibrium between the constraints, parameters and rules that influence emergent behaviour of the CAS.

Every organization is a complex adaptive system. It’s like a game in which the rules are changed on-the-fly and where the job of designing the game is delegated to the participants themselves. The manager does not focus on creating the right amount of rules in the organization, but rather to make sure that the people can together create their own rules. And it’s their collaborative effort that allows the system to find its own way to the edge of chaos.

5. Complex Adaptive Situational Model

Humans, with the introduction of consciousness, invented morality, laws and authority. We defined preferred directions for self-organizing systems because some results are seen as valuable and others as harmful. We value human lives therefore consider malaria parasites and HIV viruses an undesirable result of self-organization. Appelo points out that we value many irrational and unnatural things too, like non-discrimination, peace, monogamy [2, p. 101]. Self-organization makes no distinction between good and bad, between virtues or vices, between valuable and harmful. Systems simply do whatever the environment allows them to do. Whatever they can get away with. And so, humans embraced the concept of command-and-control which enables attempts to steer self-organizing systems (businesses, teams, countries) in the direction that stakeholders considered to be valuable. That’s how managers got their positions and how governments try to run countries. They care about results. They want to make sure that self-organizing systems either produce valuable things (products and services), or refrain from harming valuable things (human lives, economic growth, natural resources). Managers want software teams to create valuable software that makes money or enables good service delivery.

Key constraints affecting the emergent behaviour of a team of software engineers as a CAS are broadly identified by this author as:

- Management Governance and
- Production Governance.

Management governance is a method or system of management practices that range from formal, high ceremony practices on the one hand, to informal, low ceremony practices on the other. Management governance affects the way in which an endeavor is managed.

The formal approach to management provides work products that could lead to high levels of visibility – producing project plan/s and progress reports, risk management plan/s and reports, quality management plan/s and reports, configuration management plan/s and status accounting reports, meeting agendas and
minutes, etc. On the other hand, the informal, low ceremony approach depends less on detailed, written communication, hence leaving less visible evidence trails.

Production Governance is a method or system of production practices that range from engineering, plan-driven practices on the one hand, to organic, agile practices on the other. Production governance affects the way in which output (software) is produced.

A key engineering practice is to work according to the sequential stages of the life cycle, e.g.: Requirements Analysis; Design; Implementation and Unit Test; Integration and System Test; Qualification Test. Visible artefacts are then produced, including software requirement specification, software architecture document, software design document, programming standard/s and test records. At the organic extreme we experience a situation where there is little emphasis on the life cycle stages and associated documentation, and high focus on the technical practices of software development.

This leads to CASM – the complex adaptive situational model. An agent-based model. As illustrated in Figure 2, this author’s hypothesis is that different combinations of governance constraints influence emergent behaviour, resulting in four possible states of dynamic equilibrium:

- Crafted Quality (agile)
- Controlled Quality (plan-driven)
- Managed Costs (WetAgile™)
- Self-Directed Quality

The terms “plan-driven” and “agile” have been used by Boehm and Turner [25] to essentially describe what this author
calls the Controlled Quality and Crafted Quality approaches respectively.

Choosing to use the terms “Crafted Quality” and “Controlled Quality,” allows us to take a fresh look at familiar approaches and in so doing, CASM stands apart from any fads of the day.

While CASM identifies four domains, it is important to realize that for a specific team during a particular endeavor, the team’s state of dynamic equilibrium will primarily apply to only one of the four domains as determined by particular, situational circumstances.

CASM in no way implies that the essence of software engineering is any different in the plan-driven, Controlled Quality and agile, Crafted Quality domains, but life cycle models will be different as later explained in this article.

The following CASM-inspired definition of Situational Software Engineering is proposed:

**Activities aimed at the development and/or maintenance of software that are conducted under the situational influence of management and production governance – sometimes in a Crafted Quality way and at other times as Controlled Quality.**

Today’s “Complex Adaptive Situational Model” (CASM) illustrated in Figure 2 started life as the model of “Situational Software Engineering” as described by Myburgh in 1992 [23]. Continuous application and research gave rise in 2005 to the second generation of the model, viz. the “Situational Process Model” (SPM), illustrating interaction between production processes and management & control processes [24]. CASM represents the third, published generation of the model and it identifies four behavioural domains that represent states of dynamic equilibrium involving people, process and technology responding to the constraints of environmental governance. A socio-technical system.

**A. Crafted Quality (The Curved Arrow)**

This domain suits the information age organization where management formality is relaxed (low ceremony management governance) and production processes are accelerated by doing things in parallel (organic production governance). An agile approach can be seen as an instantiation of Crafted Quality. A key benefit of the Crafted Quality approach is faster delivery – exactly what is required in the competitive environment of the information age organization. According to Bider, agile methodology is based on a simple motto: *Develop and introduce in practice as little as possible as soon as possible, and build upon it in the following iterations* [26].

Crafted Quality results when Agile Management meets Agile Development. Key features of this domain are the increased level of process parallelism and accompanying decrease in management visibility with greater dependence on tacit knowledge [26].

The metaphor chosen for this domain is a curved arrow. It emphasizes the adaptive nature of an agile team that can rapidly respond to change.

But Crafted Quality does not only have benefits. Product rapidly brought to market is often not nearly complete or defect-free, and potentially expensive re-work has often to be undertaken to address high levels of technical debt. Later in this article the author describes the propensity that different CASM domains have for incurring technical debt and Crafted Quality is shown to have a HIGH propensity for technical debt. This outcome is well demonstrated by software product with early releases that are plagued by defects, eradicated only after implementation of a number of upgrades to the product. This is tantamount to a lack of software assurance, the current state of which is described by O’Neill as dire, and that stems from a combination of neglect and unmet need [27].

**B. Controlled Quality (The Cube)**

In this domain the emergent behavior derives from constraints of engineering-style production governance and formal, high ceremony management governance. A well executed plan-driven approach to software engineering exemplifies Controlled Quality. One of the key benefits of the Controlled Quality approach is that quality requirements are formally addressed at each stage of the life cycle – both in terms of initially specifying the requirements and subsequently verifying fulfilment thereof. This offers the opportunity of fulfilling the need for higher assurance levels as described by Boehm: *At the same time that systems engineering and development need to become more agile, the growing interdependence of systems and people requires systems to have higher assurance levels* [28].

Bider identifies the following advantages of traditional systems development: *Four distinct phases of knowledge transformation that allows distributing the work between experts; explicit requirement specifications, and design specifications that facilitate using existing knowledge on design principles and programming; explicit requirement specifications that allow entering a contract agreement with fixed obligations on the side of the development team.* Bider also makes reference to the emergence of systems engineering as a discipline to handle system complexity (meaning, I suspect complicatedness) and also points out that objective complexity (complicatedness) increases relentlessly as systems evolve [26].

The Controlled Quality approach is well suited to handling increased system complicatedness and ensuring high levels of product assurance.

Other key features of this domain are the increased levels of process clarity and management formality.

The metaphor chosen for this domain is the cube. It emphasizes the disciplined nature of a team operating under conditions of defined constraints.

But Controlled Quality does not only have benefits. A number of situational characteristics must apply before it is even feasible for Controlled Quality to deliver value. These include the ability to drive out and specify requirements and having the time and skilled resources to analyze, specify and design the solution before development starts. Bider adds to the list: *traditional development lacks or has insufficient feedback loops for correcting errors accidentally introduced in any of the phases;*
using requirements as a basis for design and a contract agreement presumes that all such requirements could be understood and formulated; it may take too much time to go through one big cycle [26]. And attempts to drive out comprehensive requirements, architecture and design can easily lead to a state of “analysis paralysis” which is similar to what CASM calls “debilitating bureaucracy”.

Later in this article the author describes the propensity that different CASM domains have for incurring technical debt and Controlled Quality is shown to have a MEDIUM propensity for technical debt.

C. Band of Software Engineering Feasibility
It is not by accident that CASM is represented as a diamond-shaped model. This layout places the emphasis on what is called “the band of software engineering feasibility” which stretches from agile, Crafted Quality at the one end, to plan-driven, Controlled Quality at the other. Depending on circumstances, this implies that effective states of dynamic equilibrium of a software engineering team can exist anywhere along the band while still producing value-adding results.

An implied characteristic of the software engineering band of feasibility is that it is supported by a “management-by-measurement” culture, meaning that, no matter whether the way of working is agile or plan-driven, management will be enabled by recording, analyzing and responding to relevant measurements. For example, an endeavor where the Software Engineering Institute’s Team Software Process (TSP) is applied is expected to naturally lie solidly in the band of feasibility.

SEMAT’s Essence Kernel enables practices to define lifecycles, whether agile or plan-driven, by sequencing a number of patterns, one for each phase and/or milestone in the lifecycle. The lifecycles are illustrated using the template defined in the Object Management Group standard [8] and shown in Figure 3. The so-called Alpha Abacus. Each Kernel Alpha and its states are shown in a vertical column with their creation at the top and their destruction at the bottom. Milestones are shown as vertical bars across the grid starting with an inverted triangle to represent each milestone and continuing with a white line over which are shown the states to be achieved to successfully pass the milestone. Where achieving a state is either recommended or optional, the state is shown with a dashed outline and italicized text.

Using the template illustrated in Figure 3, a sub-clause in the submission to the OMG provides illustrations of a few typical software engineering lifecycles, including an exploratory lifecycle (Crafted Quality) and a waterfall lifecycle (Controlled...
This suggests that, as illustrated in Figure [5], various instantiations of the SEMAT Life Cycle Model could be placed at various points across the software engineering band of feasibility.

According to Boehm, the Incremental Commitment Spiral Model (ICSM) is a life cycle process model generator that has been successfully applied on projects from small, non-critical projects to very large, critical projects as a flexible but robust framework for system development [28]. This suggests that ICSM could be applied across the band of feasibility – small, non-critical projects on the left and very large, critical projects on the right.

Thus far we have considered Crafted Quality (agile) and Controlled Quality (plan-driven). But what of the other two domains that are not in the band of feasibility?

**D. Managed Costs (MaC) (The Explosion)**

This condition emerges when high ceremony management governance is applied, to a situation where technical work is performed with the freedom of organic production governance. Management thus expects evidence of Controlled Quality while simultaneously giving developers the organic freedom of agile production. This is a somewhat dysfunctional expectation as described by Applegate, McFarlan and McKenney [29, p. 184] and the “Managed Costs” name emphasizes that management will focus on cost and budget control while being quite disconnected from the day-to-day, technical activities of the team. The disconnect will also mean that attempts to emphasize schedule performance are likely to be frustrated. The author has experienced many situations where the disconnect has made it almost impossible for the Project Management Office (PMO) to produce meaningful progress reports and valiant attempts to generate reports have resulted in the PMO becoming known as the “propaganda management office”. Sheard seems to support this experience, writing: Many projects that did intensive planning and tracking did it for show, and did not make use of it to run the project well [30].

Could this explain why Appelo’s survey [2, p. 28] identified the following factors as contributing to failed agile approaches?

- Management opposed to change – clinging to high ceremony management governance.
- Loss of management control – that is perceived to be the result of moving to a low ceremony approach.
- Lack of engineering discipline – due to organic production approach.
- Organizational need for planning, predictability and documentation – associated with high ceremony management governance.

McMahon refers to statements by Scott Ambler, based on a Dr. Dobbs State of the IT Union survey conducted in November, 2009, that only 11% of respondents indicated that their existing governance strategy works well with agile teams. Ambler said that this is an indication that their organization is likely to apply traditional governance strategies (e.g. high ceremony management governance) and that this strategy will not work with agile teams [31].

Steve Pieczko [32] suggested that Managed Costs might be a hybrid condition experienced by a team that is migrating from Controlled to Crafted Quality and, while still “dripping from the waterfall,” they’re trying to be agile. Hence the alternative name “WetAgile.” This might sometimes be a valid explanation, but this author has experienced a number of situations where the somewhat dysfunctional, Managed Costs state seems to be permanent and a breeding ground for “management-by-politics” supported by the “propaganda management office”.

Later in this article the author describes the propensity that different CASM domains have for incurring technical debt and Managed Costs is shown to have a HIGH propensity for technical debt.

The metaphor chosen for this domain is the explosion which emphasizes the often crisis-driven reality of the domain.

Sheard observes that humans are good at pattern recognition, creativity, and rapidly figuring out reasonable ways to handle emergencies, but they are bad at paying continuous attention to things that don’t change [30]. This observation is thought-provoking: might the excitement of the crisis-driven Managed Costs domain mean that it represents the preferred situation for most people? And does it also help explain why so many people gladly assume that the well controlled, crisis-free Controlled Quality approach is just a bad memory?
E. Self-Directed Quality (SDQ) (The Sphere)
When conditions of low ceremony management governance interact with engineering production governance, the resulting state is Self-Directed Quality (SDQ) (The Sphere). A somewhat surprising situation. Why would practitioners elect to be constrained by engineering production governance when management governance expects no more than low ceremony? Supporters of Controlled Quality would see this as an unexpected bonus, while Crafted Quality agilists might think of it as madness. This author suggests two possible explanations.

The first might be because tools being used enforce typical engineering production governance. It was for this reason that first generation CASM actually called this domain “Automatic Quality” [23, p. 94].

A second, speculative explanation is that small, (one person?) software development initiatives aimed at producing open source software are executed by individual/s who must follow the required engineering production governance in order to have their software accepted into the open source libraries.

Later in this article the author describes the propensity that different CASM domains have for incurring technical debt and Self-Directed Quality is shown to have a MEDIUM propensity for technical debt.

The metaphor chosen for this domain is the sphere, emphasizing (from an engineer’s point of view), the utopian situation where effective engineering is performed without the need for high ceremony management governance.

F. CASM Domain Propensity to Technical Debt
O’Neill has described key aspects of technical debt [33]. In particular, a number of conditioning triggers have been identified. If a conditioning trigger is set, it means that the particular factor does contribute to technical debt. The conditioning triggers are grouped into categories: Management, Engineering and Process.

O’Neill’s approach has been slightly adapted by this author to investigate the propensity for technical debt in each of the CASM domains. This was done by estimating the likelihood of each factor contributing to technical debt, scoring the likelihood levels and using the results to determine HIGH, MEDIUM or LOW Propensity to Technical Debt.

Tables 1, 2 and 3 (adapted from O’Neill) show the likelihood of the factors contributing to technical debt in each of the four domains.

In the interests of quantifying the relative likelihood of technical debt being experienced in each CASM domain, the following scores were assigned:

- Usually YES : Score 3 points (Maximum 63 points)
- YES or NO : Score 2 points (Maximum 42 points)
- Usually NO : Score 1 point (Maximum 21 points)
- NO : Score 0

Based on the maximum scores, the levels are defined as follows:

- HIGH : More than 42 points
- MEDIUM : Between 22 and 42 points
- LOW : 21 points or less.

The results of scoring the tables are shown below – from highest to lowest:

Table 4. CASM Domain Levels of Propensity to Technical Debt.

<table>
<thead>
<tr>
<th>CASM Domain</th>
<th>Points Scored</th>
<th>Level of Propensity to Technical Debt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crafted (Agile) Quality</td>
<td>51</td>
<td>HIGH</td>
</tr>
<tr>
<td>Managed Costs (WetAgile) Quality</td>
<td>49</td>
<td>HIGH</td>
</tr>
<tr>
<td>Self-Directed Quality (Plan-driven)</td>
<td>39</td>
<td>MEDIUM</td>
</tr>
<tr>
<td>Controlled Quality (Plan-driven)</td>
<td>34</td>
<td>MEDIUM</td>
</tr>
</tbody>
</table>

G. When things go wrong
Stephen Wolfram’s proposed classification scheme was introduced earlier where Class IV corresponds to complexity, Classes I and II correspond to order and Class III to chaos. As illustrated in Figure 7, CASM’s four domains are suggested to correspond to Class IV - Complexity, with Classes I and II lying to the right of the band, and Class III to the left.

When the freedom of Crafted Quality is abused, the situation typically degenerates into a state of chaos (Class III).

Inappropriate responses to governance that desires a Controlled Quality outcome can easily result in creation of Class I or II situations with excessive order –experienced as debilitating bureaucracy.
<table>
<thead>
<tr>
<th>Source</th>
<th>Trigger</th>
<th>Condition</th>
<th>Conditioning Likely to be Set?</th>
<th>Trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LEGEND: CrQ – Crafted Quality : CoQ – Controlled Quality : MaC – Managed Costs : SdQ – Self-Directed Quality 3 – Usually YES : 2 – YES or NO : 1 – Usually NO : 0 - NO</td>
<td>CrQ</td>
<td>CoQ</td>
<td>MaC</td>
</tr>
<tr>
<td></td>
<td>Management</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>M1. Prioritized goals</td>
<td>Where schedule or cost is accorded priority over defect free delivery</td>
<td>3 1 3 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>M2. Organization levels</td>
<td>Where the software function is separated from program management by two or more levels</td>
<td>1 2 3 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>M3. Schedule</td>
<td>Where the number of months planned is less than the estimated month at completion</td>
<td>1 2 3 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>M4. Cost</td>
<td>Where the budget at completion is less than the estimate at completion</td>
<td>1 2 3 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>M5. Milestone completion</td>
<td>Where the completion schedule for any milestone completion planned date is replaced with a replanned date</td>
<td>1 2 3 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>M6. Headcount and effort</td>
<td>Where overtime, off the clock time, and personnel turnover rate is trending upwards</td>
<td>3 2 3 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>M7. Frequency of release</td>
<td>Where the frequency of release is daily or weekly</td>
<td>3 1 2 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Engineering</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>E1. Deep domain expertise</td>
<td>Where deep domain expertise is not widespread on the project</td>
<td>3 2 2 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E2. Software architecture</td>
<td>Where software architecture is not tightly coupled with middleware, operating system, and network services</td>
<td>3 1 2 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E3. Requirements known</td>
<td>Where requirements are not fully known</td>
<td>3 2 2 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E4. Technical risk</td>
<td>Where the source of technical uncertainty in function, form, or fit is high</td>
<td>2 2 2 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E5. Product size</td>
<td>Where product size estimates at completion exceed product size estimates planned</td>
<td>3 2 2 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E6. Complexity (Complicatedness)</td>
<td>Where cyclomatic or essential complicatedness trend upward from one product release to another</td>
<td>3 2 2 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E7. Scope of Delivery</td>
<td>Where full scope of the ultimate solution is not worked on.</td>
<td>3 2 2 2</td>
<td></td>
</tr>
</tbody>
</table>
Table 3. Technical Debt: Process, Trigger, Condition and Likelihood.

<table>
<thead>
<tr>
<th>Source</th>
<th>Trigger</th>
<th>Condition</th>
<th>Conditioning Likely to be Set?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>P1. Software Project Management</td>
<td>Where the software project management mode is low</td>
<td>CrQ  3</td>
</tr>
<tr>
<td></td>
<td>P2. Software Product Engineering</td>
<td>Where the software product engineering mode is ad hoc</td>
<td>CrQ  3</td>
</tr>
<tr>
<td></td>
<td>P3. Iterative development</td>
<td>Where incremental or iterative development of design levels and delivery stages is not used</td>
<td>CrQ  0</td>
</tr>
<tr>
<td></td>
<td>P4. Best practices</td>
<td>Where the use of best practices is rated low</td>
<td>CrQ  3</td>
</tr>
<tr>
<td></td>
<td>P5. Metrics</td>
<td>Where metrics are not used</td>
<td>CrQ  3</td>
</tr>
<tr>
<td></td>
<td>P6. Quality Assurance</td>
<td>Where quality assurance is not in place and functioning</td>
<td>CrQ  3</td>
</tr>
<tr>
<td></td>
<td>P7. Defect rate</td>
<td>Where the actual defect rate including both defect detection and defect correction exceed the expected</td>
<td>CrQ  3</td>
</tr>
</tbody>
</table>

**H. CASM Characteristics**

CASM has been introduced as a model of team behaviour and in broad terms, any software engineering team could, in response to the governance constraints imposed, be in any one of the four states of dynamic equilibrium. Each state has a set of defining characteristics. Table 5 describes characteristics of the domains. The table derives from published work (Myburgh [24], Boehm and Turner [25]) as well as from experience gained through practical application of the model.

Assumed characteristics of the Managed Costs and Self-Directed Quality domains require further, substantive research.

**7. Bringing CASM to Life – Options for the Enterprise**

The above analysis of situational factors demonstrates that different ways of working apply to different situations. A team of software engineers who are working on a focused initiative can be expected to adjust their way of working to be appropriate to the situation. However, in larger organizations where many teams are tackling many initiatives, one could expect different teams to be in different states of dynamic equilibrium at the same time. In Keskinen, Aaltonen and Mitton-Kelly we read:  
*An organization is not necessarily entirely tight or entirely loose. It is an imperfect system, a mixture of tightness and looseness, continuously created through communication and action [7, p16].*

To better understand this, we can consider the idea of a hierarchy of people-based complex adaptive systems – a system of complex adaptive systems. Working from the bottom up, we first find an individual person. (Remembering that a single person is already a CAS). If a few people collaborate towards achieving the same goal/s, we discover the next level CAS, viz. a team. Teams could also be contributing to achievement of common goal/s and hence a collection of teams could define the next higher level. For the purpose of this discussion, the highest level CAS will be the enterprise itself. Now, by employing the various metaphors associated with each state of dynamic equilibrium, one could see the following diagram as a representation of an enterprise as a collection of complex adaptive systems. The diagram might make one think of an amoeba – and it would...
Table 5. CASM Domain Characteristics.

<table>
<thead>
<tr>
<th>MANAGEMENT</th>
<th>CRAFTED QUALITY (CrQ) (AGILE)</th>
<th>CONTROLLED QUALITY (CoQ) (WATERFALL)</th>
<th>MANAGED COSTS (MaC) (WE-TAGILE)</th>
<th>SELF-DIRECTED QUALITY (SDQ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer Relations</td>
<td>Dedicated on-site customers</td>
<td>As-needed, formal customer interaction</td>
<td>As-needed, formal customer interaction</td>
<td>As-needed, informal customer interaction</td>
</tr>
<tr>
<td></td>
<td>Focused on prioritised iterations.</td>
<td>Focus on formal contract provisions that could include identification of prioritised increments.</td>
<td>Focus on formal contract provisions.</td>
<td>Focused on self-prioritised increments.</td>
</tr>
<tr>
<td>Best Business Practice tends to dominate (Risk-taking).</td>
<td>Best Engineering Practice tends to dominate (Risk-avoiding).</td>
<td>Best Solution Delivery Practice tends to dominate (Risk-avoiding).</td>
<td>Best Engineering Practice tends to dominate (Risk-avoiding).</td>
<td></td>
</tr>
<tr>
<td>Internalised plans (low visibility).</td>
<td>Internalised plans (low visibility).</td>
<td>Internalised plans (low visibility).</td>
<td>Internalised plans (low visibility).</td>
<td></td>
</tr>
<tr>
<td>Evolutionary, iterative delivery.</td>
<td>Incremental or full-scope delivery.</td>
<td>Incremental or full-scope delivery.</td>
<td>Incremental or full-scope delivery.</td>
<td></td>
</tr>
<tr>
<td>Qualitative control.</td>
<td>Qualitative control.</td>
<td>Quantitative control – not always aligned to work being performed, leading to crisis situations.</td>
<td>Quantitative control – not always aligned to work being performed, leading to crisis situations.</td>
<td></td>
</tr>
<tr>
<td>Planning &amp; Control</td>
<td>Classic PMBOK practices less feasible (parallel approach).</td>
<td>Classic PMBOK practices more feasible (sequential approach).</td>
<td>Classic PMBOK practices more feasible (sequential approach) – not always aligned to work being performed, leading to crisis situations.</td>
<td>Classic PMBOK practices more feasible (sequential approach) – not always aligned to work being performed, leading to crisis situations.</td>
</tr>
<tr>
<td>Risk contained by time-box.</td>
<td>Risk contained with Management Reserve.</td>
<td>Risk contained with Management Reserve, but probably not catered for in the initial budget.</td>
<td>Risk absorbed by individual.</td>
<td></td>
</tr>
<tr>
<td>High propensity to technical debt.</td>
<td>Medium propensity to technical debt.</td>
<td>High Propensity to technical debt.</td>
<td>Medium propensity to technical debt.</td>
<td></td>
</tr>
<tr>
<td>Communication</td>
<td>Tacit interpersonal knowledge (low visibility).</td>
<td>Formal, documented architecture &amp; knowledge.</td>
<td>Formal, documented architecture &amp; knowledge – not always aligned to work being performed, leading to crisis situations.</td>
<td>Formal, documented architecture &amp; knowledge as required for product acceptance.</td>
</tr>
<tr>
<td>More often unique initiatives.</td>
<td>More often repeatable processes and continuous improvement.</td>
<td>More often repeatable processes and continuous improvement – not always aligned to work being performed, leading to crisis situations.</td>
<td>More often unique initiatives.</td>
<td></td>
</tr>
<tr>
<td>Repeated initiatives remain challenging.</td>
<td>Repeated projects can become jobs.</td>
<td>Repeated projects could become jobs – if execution is aligned to work being performed.</td>
<td>Repeated projects can become jobs.</td>
<td></td>
</tr>
<tr>
<td>TECHNICAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirements</td>
<td>Could become chaotic.</td>
<td>Usually well organised.</td>
<td>Could appear to be well organised, but not always aligned to work being performed, leading to crisis situations.</td>
<td>Usually well organised, but could become chaotic.</td>
</tr>
<tr>
<td>Undergoing unforeseeable change.</td>
<td>Formalised project capability, interface, and quality. Architecture aligned to work being performed.</td>
<td>Formalised project capability, interface, and quality. Architecture not always aligned to work being performed, leading to crisis situations.</td>
<td>Formalised project capability, interface, and quality. Architecture not always aligned to work being performed, leading to crisis situations.</td>
<td></td>
</tr>
<tr>
<td>Development</td>
<td>Evolving architecture.</td>
<td>Guided by full-scope architecture.</td>
<td>Guided by full-scope architecture – not always aligned to work being performed, leading to crisis situations.</td>
<td>Evolving or full-scope architecture.</td>
</tr>
<tr>
<td>Simple design.</td>
<td>Extensive design.</td>
<td>Extensive design – not always aligned to work being performed, leading to crisis situations.</td>
<td>Extensive design.</td>
<td></td>
</tr>
<tr>
<td>Short increments.</td>
<td>Longer increments.</td>
<td>Longer increments.</td>
<td>Short increments.</td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td>Executable test cases define requirements.</td>
<td>Documented test plans and procedures.</td>
<td>Documented test plans and procedures.</td>
<td>Formal test plans and procedures to satisfy requirements for software acceptance.</td>
</tr>
<tr>
<td>PERSONNEL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Customers</td>
<td>Dedicated, collocated CRACK performers.</td>
<td>CRACK performers, not always co-located.</td>
<td>CRACK performers needed, but not always available.</td>
<td>Dedicated, collocated CRACK performer/s.</td>
</tr>
<tr>
<td>Learn largely by doing.</td>
<td>Learn largely by reading.</td>
<td>Learn largely by reading – documentation not always aligned to work being performed, leading to crisis situations.</td>
<td>Learn largely by reading.</td>
<td></td>
</tr>
<tr>
<td>Culture</td>
<td>Many degrees of freedom.</td>
<td>Framework of policies and procedures.</td>
<td>Framework of policies and procedures – not always aligned to work being performed, leading to crisis situations.</td>
<td>Self-limited degrees of freedom.</td>
</tr>
<tr>
<td>Thriving on chaos.</td>
<td>Thriving on order.</td>
<td>Appears to thrive on order, but lack of alignment to work being performed, leads to crisis situations.</td>
<td>Thriving on order.</td>
<td></td>
</tr>
<tr>
<td>APPLICATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Primary goals</td>
<td>Rapid value.</td>
<td>Predictability, Stability.</td>
<td>Rapid value which requires organic way of working.</td>
<td>Predictability, Stability</td>
</tr>
<tr>
<td>Responding to change.</td>
<td>High assurance.</td>
<td>Responding to change which requires organic way of working.</td>
<td>High assurance to meet upload requirements.</td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td>Smaller teams and projects.</td>
<td>Larger teams and projects.</td>
<td>Larger teams and projects.</td>
<td>Smaller teams and projects.</td>
</tr>
<tr>
<td>Environment</td>
<td>Turbulent.</td>
<td>Stable.</td>
<td>Appears to be stable, but lack of alignment to work being performed, leads to crisis situations.</td>
<td>Turbulent or stable</td>
</tr>
<tr>
<td>High change.</td>
<td>Low change.</td>
<td>Low change preferred, but high change often allowed, leading to crisis situations.</td>
<td>High or low change.</td>
<td></td>
</tr>
<tr>
<td>Project-focused.</td>
<td>Project/organisation focused.</td>
<td>Project/organisation focused.</td>
<td>Project-focused.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 6. Appropriate Software Engineering Responses to Situational Characteristics.

<table>
<thead>
<tr>
<th>SITUATIONAL CHARACTERISTIC</th>
<th>YES</th>
<th>NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Are requirements readily definable?</td>
<td>A CoQ, plan-driven way of working could be adopted on condition that delivery time-scales permit.</td>
<td>The CrQ, agile way of working is required.</td>
</tr>
<tr>
<td>Is there a comprehensive architectural description for the solution?</td>
<td>A CoQ, plan-driven way of working could be adopted. If the scope of delivery is large, an incremental approach will mitigate risk by delivering regular, pre-planned increments.</td>
<td>The CrQ, agile way of working is required.</td>
</tr>
<tr>
<td>Is there pressure to rapidly produce results?</td>
<td>The CrQ, agile way of working is required.</td>
<td>A CoQ, plan-driven way of working could be adopted on condition that requirements are definable.</td>
</tr>
<tr>
<td>Is there pressure to produce accurate schedules, budgets &amp; estimates?</td>
<td>If the accuracy is to be based on schedules, budgets and estimates that are derived from a detailed action plan for the initiative, then a CoQ, plan-driven approach is required. If the accuracy is to be based on the cost per time-box, then a CrQ, agile approach is indicated.</td>
<td>Schedules, budgets and estimates can be based on the cost per time-box and a CrQ, agile way of working is suggested.</td>
</tr>
<tr>
<td>Does the size of the initiative introduce significant risk?</td>
<td>A CoQ, plan-driven approach is suitable for mitigating this risk – on condition that other characteristics required for CoQ also pertain.</td>
<td>The CrQ, agile way of working is suggested so that the overhead associated with CoQ, plan-driven can be avoided.</td>
</tr>
<tr>
<td>Will implementation of the solution introduce significant change?</td>
<td>A CrQ, agile way of working allows for resistance to change being mitigated by limiting the extent of change associated with each iteration. An incremental, CoQ, plan-driven approach could also be used to limit the extent of change introduced during each increment.</td>
<td>Either CoQ, plan-driven or CrQ, agile approaches could be viable. Other situational characteristics will influence the decision.</td>
</tr>
<tr>
<td>Is there significant risk due to technology? (This suggests that unproven, state of the art technology is to be implemented).</td>
<td>A CrQ, agile approach should be followed by a team that is mandated to experiment with and get to know the new technology.</td>
<td>Either CoQ, plan-driven or CrQ, agile approaches could be viable. Other situational characteristics will influence the decision.</td>
</tr>
<tr>
<td>Does cost-of-failure represent a source of significant risk?</td>
<td>A CoQ, plan-driven way of working should be adopted to allow for product assurance. If the scope of delivery is large, an incremental approach will further mitigate risk by delivering regular, pre-planned increments that could be separately assured.</td>
<td>Either CoQ, plan-driven or CrQ, agile approaches could be viable. Other situational characteristics will influence the decision.</td>
</tr>
<tr>
<td>Does the software engineering team collectively have a high level of competence?</td>
<td>The team should be able to adapt to whatever way of working is situationally appropriate.</td>
<td>This situation represents a significant source of risk, and attempts to adopt a CoQ way of working could easily result in “debilitating bureaucracy”, whereas CrQ approaches are likely to evolve into “freelance chaos”.</td>
</tr>
</tbody>
</table>
be quite appropriate to use this as a metaphor for the Enterprise. In Keskinen, Aaltonen and Mitleton-Kelly we read: As the biological metaphor replaces the mechanical metaphor to think about the evolution and co-evolution of firms and goods and services, agent based models, complexity theory, and ideas from biology are making their way into the practical world of business. “Adaptive organizations” is now the buzzword [7, p6]. The Enterprise Amoeba in Figure 8 depicts an adaptive organization.

Self-organization is fundamental to every complex system. But in a human social system, self-organization alone is not enough. Appelo explains how Glen Alleman described the need for management by pointing out that there is a difference between self-organizing and self-directing and this is the role of management [2, p. 153]. This is not “directing” in the Command and Control sense. It is directing in the “required business value” sense. If self-organizing teams serve their customers, who “manages” the customer, when the customer is not prepared to behave in a “well-mannered” way? If there is more than one self-organizing team working on the same project, who coordinates the activities between these teams? When there are conflicts in resources, funding and requirements, who coordinates resolution of these conflicts? At least a little management is needed to steer self-organization in a direction that is of value to everyone in the system. This level of management is therefore required to steer the endeavors of the enterprise.

Appelo points out that Sanjiv Augustine calls it “light-touch leadership”. Appelo calls it alignment of constraints [2, p. 153]. (He refers to aligning constraints, and not aligning people, because it is only the constraints that we control. And the people, we can only hope, will heed the constraints).

There’s always plenty of opportunity for self-organization to take place, and quite often at least something will emerge from that. But a manager, having defined the system in the first place, and governing the system to protect it, must take the opportunity to ensure that what emerges has value. Because complexity science doesn’t suggest to simply wait for the right solutions to emerge. The way managers define boundaries and constraints strongly influences what emerges from a self-organizing team [2, p. 154]. It’s not about managing the people, rather managing the system – the enterprise amoeba. Directed self-organization in organizations is a matter of manipulating the constraints so that a group of people produces results valuable to the goals of the project.

8. Conclusion
CASM, the Complex Adaptive Situational Model described in this paper promotes understanding of what it takes to establish conditions that enable software engineering success. Not only with agile approaches, but also traditional, plan-driven software engineering.

Influenced by complexity science, CASM explains aspects of the state of dynamic equilibrium that is achieved by a software engineering team under the constraining influence of management and production governance. The team is a complex adaptive system with people as the active agents.

Four states of dynamic equilibrium are defined: Crafted Quality (agile), Controlled Quality (plan-driven), Managed Costs (WetAgile) and Self-Directed Quality. A band of software engineering feasibility is also described and it is suggested that successful team-based software engineering initiatives require teams to operate in that band which stretches from Crafted Quality to Controlled Quality. Management’s challenge is to appropriately apply constraints that enable the required state of dynamic equilibrium. Effective agile behaviour emerges when low ceremony management governance is combined with organic production governance. And effective plan-driven initiatives require high ceremony management governance to be applied together with engineering production governance.

The journey across the band of feasibility is further described by introducing SEMAT, with Crafted Quality amounting to applying native SEMAT Essence, and Controlled Quality being achieved by introducing additional practices which satisfy the more stringent governance requirements.

CASM in its four states allowed introduction of the idea of describing an enterprise as a collection of complex adaptive systems, thereby setting the scene for further research into the complexities of human-driven complex adaptive systems.

It has also been suggested that, by describing some aspects of the complex adaptive behaviour of a team of software engineers, CASM can be seen to make a contribution to the general theory of software engineering.

References


PIECZKO, S. E-mail correspondence, 2010.
