Applied

On Understanding Software Agility— A Social Complexity Point Of View

Joseph Pelrine
MetaProg GmbH, CHE

Over the last decade, the field of so-called Agile software development has grown to be a major force in the socio-economic arena of delivering quality software on time, on budget, and on spec. The acceleration in changing needs brought on by the rise in popularity of the Internet has helped push Agile practices far beyond their original boundaries, and possibly into domains where their application is not the optimal solution to the problems at hand. The question of where Agile software development practices and techniques make sense, and where are they out of place, is a valid one. It can be addressed by looking at software development as a complex endeavour, and using tools and techniques from the Cynefin method and other models of social complexity.

Introduction

Over the course of the last decade, a soft revolution has taken place in the field of software development. Experiences with projects delivering late and over budget have led people to question some of the basic tenets of software project development and management. Starting with a few provocative theses in Kent Beck's 1999 book *eXtreme Programming*, the field of so-called Agile software development has grown to be a major force in the socio-economic arena of delivering quality software to people on time, on budget, and on spec. The acceleration in changing needs brought on by the rise in popularity of the Internet has helped push Agile practices far beyond their original boundaries, and possibly into domains where their application is not the optimal solution to the problems at hand.

So, where do Agile software development practices and techniques make sense, and where are they out of place? To answer that question, it is necessary to first understand how, and more importantly why, Agile practices work. In the mid 1990s, it was maintained that practices such as *eXtreme Programming* could not possibly work, although even then dozens of projects successfully completed had proved otherwise. Later, after sufficient empirical evidence had accumulated to irrefutably prove that Agile practices do work, the question of why they worked still remained.

As a consequence of the increasing complexity and unpredictability of the world around us, Agile practice is increasingly seen as the solution. Agile represents a new paradigm in the truest sense, a complete abandonment of old methods that cannot be done in half measures. At its core, Agile addresses complexity in a manageable fashion, attuned to the needs of the human psyche. The Agile approach, though, constitutes a revolution in our modes of thinking, working and interacting. Agile processes have grown and developed as the body of knowledge acquires new ideas from its practitioners. Expansion is not always a positive force, however. The discipline that originally allowed Agile to explode linear, mechanistic development practices has often vanished, replaced by a cargo-cult "by-the-rules" interpretation of Agile based on checklists. It seems today that some 'Agile' teams are practicing nothing more than 'air guitar and attitude' (to quote Alan Kay).

Ultimately, models are only as good as the people applying them. Too many teams have come to regard Agile as something like a cookery recipe—follow this set of instructions and procedures for a tasty result. But in software development, as in cooking, what you get out is not simply the sum of what you put in. We need to develop an understanding of what makes Agile work, and indeed what makes it fail. This understanding is a prerequisite for sustaining and scaling Agile efforts. Acknowledging that Agile is not working in a particular situation is an inherent part of Agile practice, but it's one that's often ignored.

The purpose of this paper is to explore the following questions:

- Is software development (in whole or in part) a socially complex endeavour?
- What can be gained by treating it as complex, and using tools and techniques from social complexity science?
- Why do Agile practices work so well?

Why is Agile the best method around for meeting challenges in software development? As projects become more complex, and customer requirements ever more ambitious but ever less clearly defined, how does Agile help developers produce usable software on time and on budget? Despite the recent quantum shift in the field towards the adoption of Agile, many organizations have not yet made the conceptual adjustments necessary to apply it successfully. No method is without its detractors, and those who have yet to be convinced by Agile can point to the lack of hard evidence, of rigorous analysis, of a theoretical, scientific basis, in the literature. For Agile fans, rigorous does not mean rigid, and it is not 'anti-Agile' to question the assumptions on which we base our processes, quite the opposite is true!

The core realization inherent in Agile is that people build software, that team dynamics are fundamental to it, and that teams of people are complex and unpredictable. The need to factor psychological and cognitive concepts into project design and implementation is novel in a world where mechanization and unifor-

mity are frequently encountered, and often admired, within corporate cultures. A more realistic model of corporate information sharing began with Knowledge Management and its applications, but Agile goes much farther.

Modern software development is performed by teams of motivated individuals. The prevailing attitude for much of the field's history, though, has been to treat software development as a predictable 'factory' process, where adding a given amount of money, time, programmers and managers will produce the desired result. Within this context, the development process is broken down into a sequential pathway, with deliverable outcomes predicted at set points. This 'traditional' approach is exemplified by the Waterfall model. It can work—if the requirements are known, in detail, right from the start, the product is straightforward, and nothing goes horribly wrong. But who has ever worked on a project like that? Successful Waterfall projects do exist, but they are few and far between, and on closer analysis may not be adopting 'pure' linear management models—some flexibility, the beginnings of Agility, has crept in!

Trying to establish computing as an engineering discipline led people to believe that managing projects in computing is also an engineering discipline. Engineering is for the most part based on Newtonian mechanics and physics, especially in terms of causality. Events can be predicted, responses can be known in advance and planning and optimize for certain attributes is possible from the outset. Effectively, this reductionist approach assumes that the whole is the sum of the parts, and that parts can be replaced wherever and whenever necessary to address problems. This machine paradigm necessitates planning everything in advance because the machine does not think. This approach is fundamentally incapable of dealing with the complexity and change that actually happens in projects.

Traditional	**Agile**
Sequential	Iterative
Defined	Empirical
Plan-driven	Result-driven
Big-bang	Incremental
Specialised teams	Cross-functional teams
Test at end	Test-first

Figure 1 *Traditional vs. Agile.*

Consider what happens if you manage a project like a production line. Developers are assigned tasks, code is pumped through, and the finished product rolls off at the end. There are two problems with this paradigm. Firstly, the production line approach is more suited to generating multiple repetitive units, something that is rarely entailed in software development. Secondly, as soon as the product stops working at the end, the entire production line must be analyzed and fixed to solve the fault. Usability is a good example—often, aspects of product usabil-

ity are left until the end stages of a project, with the expectation that it can be fine-tuned as needed. Frequently, there are flaws deep within the software that are not trivial to fix. The whole process must be reworked, but the team has already given it massive investment in terms of resources, time and effort. For the author, adopting Agile becomes the task of increasing awareness of, and finding the best process for answering the question, 'when do you want to know you have a problem?' (That assumes, naturally, that you do want to know you have a problem, an equally important point!)

> *One of the most highly developed skills in contemporary Western civilization is dissection: the split-up of problems into their smallest possible components. We are good at it. So good, we often forget to put the pieces together again* (Toffler, 1984)

The reductionist approach described by Toffler has served us well in the past, but we need to move beyond it. Many people still regard building software as a complicated undertaking, but in fact it is a defining example of a complex or a 'wicked' problem. The concept of wicked problems was originally proposed by Horst Rittel and Marcus Webber (Rittel & Webber, 1973). Wicked problems have incomplete, contradictory, and changing requirements; and solutions to them are often difficult to recognize as such, because of complex interdependencies. Rittel and Webber stated that while attempting to solve a wicked problem, the solution of one of its aspects may reveal or create other, even more complex problems. Rittel expounded on the nature of ill-defined design and planning problems, which he termed 'wicked' (that is, messy, circular, aggressive) to contrast against the relatively 'tame' problems of mathematics, chess, or puzzle solving. In the author's experience, certain ground rules of Agile software development have emerged that address the limitations of the Waterfall and other linear development paradigms in tackling such problems.

Communication and team dynamics represent the other area where Agile differs fundamentally older development paradigms. The functioning of the team, and the contributions and roles of individuals within the team, are fundamental to productivity. Team roles are no longer fixed, but members are allowed to self-organise. Management takes on the role of facilitating and coaching the team, rather than issuing orders. Scrum (Schwaber & Beedle, 2001) sees self-organizing teams as a fundamental aspect of the process. In applying Scrum, there is an emphasis on skills, not knowledge, and there are few rules. The author has distilled three 'rules of thumb/rules of Scrum' from experience in practice: the first is 'we don't make mistakes, we learn,' i.e., set up a safe-fail work environment where it is OK to learn and to correct behavior, estimates etc., on the basis of that learning. Secondly 'whoever has the risk, makes the decision.' Increase awareness of roles, rights, and responsibilities of the various partners in the development process. And last 'if you're not having fun, we're doing something wrong.' Keeping people happy and motivated isn't easy over a long project, but there are techniques that can be used from the outset to promote good team practice.

Getting Comfortable With Complexity—Sense Making The Agile Way

What is a complex system? Complexity theory can be considered one of the most revolutionary products of 20th century thought. Theories of chaos, complexity and emergence have shattered the conceptual frameworks of science, technology and economics, and provide unifying themes across previously distant disciplines. Scientists, sociologists, economists and engineers are finding common ground that transcends the terms of reference of each particular field. We have gone from the assumption that everything can be modelled given enough time, intelligence or processing power, to the realization that not everything we experience can be drilled into predictable patterns that we can recognise and understand. The human mind does not readily grasp complexity. It is counterintuitive; we prefer to recognise patterns in mechanistic systems.

How can a complex system be defined? Ask ten or twenty people working on complexity and emergence to describe such a system and you will get as many answers. One of the best sets of criteria for complexity is provided by professor George Rzevski:

1. INTERACTION—A complex system consists of a large number of diverse components (Agents) engaged in rich interaction;
2. AUTONOMY—Agents are largely autonomous but subject to certain laws, rules or norms; there is no central control but agent behavior is not random;
3. EMERGENCE—Global behavior of a complex system "emerges" from the interaction of agents and is therefore unpredictable;
4. FAR FROM EQUILIBRIUM—Complex systems are "far from equilibrium" because frequent occurrences of disruptive events do not allow the system to return to the equilibrium;
5. NONLINEARITY—Nonlinearity occasionally causes an insignificant input to be amplified into an extreme event (butterfly effect);
6. SELF-ORGANIZATION—Complex systems are capable of self-organization in response to disruptive events, and;
7. CO-EVOLUTION—Complex systems irreversibly coevolve with their environments.

Is software development a complex domain, and if so, why? This is the key question. At one level, the software development process seems to fulfil all of Rzevski's criteria, but on another level there seem to be exceptions and questions. This question may not be able to be answered definitely, but as we will see, interesting things happen when we TREAT software development as complex. We might also question which other domains may benefit from this treatment.

Many customers and developers alike regard building software as a complicated undertaking, but in fact it is a prime example of a complex problem. In adopting Agile processes, the field is beginning to address this and to become more comfortable with complexity. Unfortunately, the typical Agilist perception of complexity is not quite aligned with any of the main scientific definitions of the term. Agile literature abounds with romanticized, subjective interpretations of terms such as complexity, self-organization, emergence, which can only be understood by remembering that 'a little knowledge is often a dangerous thing'.

If we even succeed in establishing that developing software is a complex endeavour, a wicked problem, how then do we address it effectively? Complexity is counterintuitive to many. This is one of the reasons that a mechanistic, Newtonian view of projects has persisted in management thought.

> *Even as it was toppled from its unassailable position in science, Newtonian mechanics remained firmly lodged as the mental model of management, from the first stirrings of the industrial revolution right through the advent of modern-day MBA studies* (Petzinger, 1999).

Complexity theory represents software development more realistically than the engineering model. Understanding the theory is only the first step. How can complex problems be tackled practically on a daily basis? How can one differentiate easily between the complex and, e.g., complicated aspects of a complex domain? The art of management and leadership is having an array of approaches and being aware of when to use which approach.

Thinking About Complex Problems

The challenges of a Wicked problem are manifold. At the outset, goals may be unclear, yet expectations are high. We are tempted to set out a grand plan, mapping the project from start to finish, with meticulous allocation of time and resources. Yet the chances of such a plan being followed are remote—even if the initial stages appear to be going well, reality will rapidly cause divergence from the pathway. New information, changing variables and requirements, external factors such as competitor activity, cannot be factored in to a plan made months before. However, how many times do managers insist on struggling forward with a battered, modified version of the original plan?

This tendency to cling to our initial assumptions and plotted course is down to the way our minds deal with new situations. The process of first-fit pattern matching evolved to make us capable of fast decisions in danger, based on previous experience. Once that 'fit' has been made it's hard for us to let go and consider alternatives within a complex problem. It also makes humans bad at cutting their losses and changing tack mid-project. Research shows we value things we already have more highly than things of equal or greater value that we don't possess, for example. We're also good at seeing patterns where none exist, and

imputing causality in random chains of events. A classic example is cumulative winnings or losses from betting on heads or tails in a coin toss. These purely random outputs can be modelled by a first-order Markov chain, which as is well known, readily exhibits pseudo cycles and pseudo trends, with stationary mean and non-stationary variance.

The problem of how to figure out a solution to a complex problem goes further. It is another part of complexity science known as multi-ontological sense making. The sense making process says that there is not one fit solution. Sense making is looking at things pre-hypothetically, that is crossing the line between unknown and known. As G. Spencer Brown said in his book *The Laws of Form* (Spencer-Brown, 1979), the first thing to do is to draw a distinction, which is exactly drawing a line between unknown and known. What we can know is cause and effect, which is the basic observation we make, i.e. phenomenology. We see something happen along the temporal axis and we often input causality: the first caused the second. Because our level of resolution of perception allows us to perceive them as to separate events, we interpret a causal connection between them. I push that light switch and a light goes on, I do it again and again and the light always changes. So I assume that there has to be some repeatable cause of connection. In this way we can predict the future.

Dave Snowden says, "sense making is the way that humans choose between multiple possible explanations of sensory and other input as they seek to conform the phenomenological with the real in order to act in such a way as to determine or respond to the world around them."

A basic premise of sense making is that we need to understand our thought processes when we analyze things. Our opinion, our evaluation of something says as much about us as it does about the thing we are looking at. This is called cognitive bias, and it influences our interpretation of everything around us, for example, what we consider to be complex.

Agile As A Technique For Addressing Complexity

The basic science necessary to understand complex systems was just starting to be established when the first Agile literature was published. At that time, the works of Stacey, Nonaka, and others sufficed to provide ideas and impulses for some Agile pioneers, but lacked the full breadth and rigour necessary for providing a foundation for understanding the Agile process as a whole. Only with the publication of Dave Snowden's papers on the Cynefin model did a system emerge that finally allowed researchers and practitioners to understand social complexity science, and its position as the theoretical basis of software Agility.

This paper will discuss one of many aspects of social complexity science, the Cynefin approach, and one of its practical applications to Agile software development. Many aspects of software development fall into the complex domain.

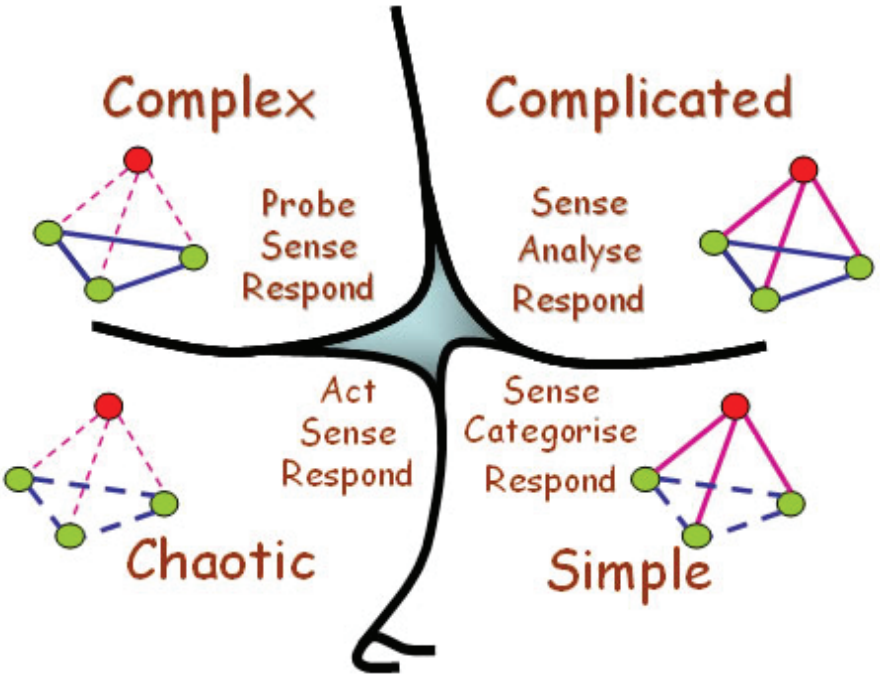

Figure 2 *The Cynefin Framework: Common Summary*

The Cynefin sense-making model has been described in a number of papers (Kurtz & Snowden, 2001; Snowden, 2005), and will not be covered in detail here. In addition to the sense-making model, though, the Cynefin method contains a number of techniques and exercises, which can be used to help groups make sense of their domain, helping them understand which methods and techniques can then be best applied.

In a study conducted over a number of years, the author has run the Cynefin 'butterfly stamping' exercise (see Cognitive Edge methods in the references) with over 300 people involved in Agile software development, with the goal of sensitizing them to scientific definitions of complexity and related concepts, of interpreting their cognitive biases related to software development, and of understanding whether software development as a whole could be considered a complex domain. During an introductory session, the participants are asked to brainstorm and collect topics they deal with and activities they engage in as part of their work. Later, after explaining the Cynefin model, definitions of the different domains, and the sense-making process, they do the exercise by assigning to the different Cynefin domains a set of situations, themes, and subjects provided by Dave Snowden, and which the participants were agnostic about. After "warming up" with these situations, themes, and subjects, and getting an active awareness for the meaning of the different domains, the participants then make sense of the activities and topics they identified and collected themselves.

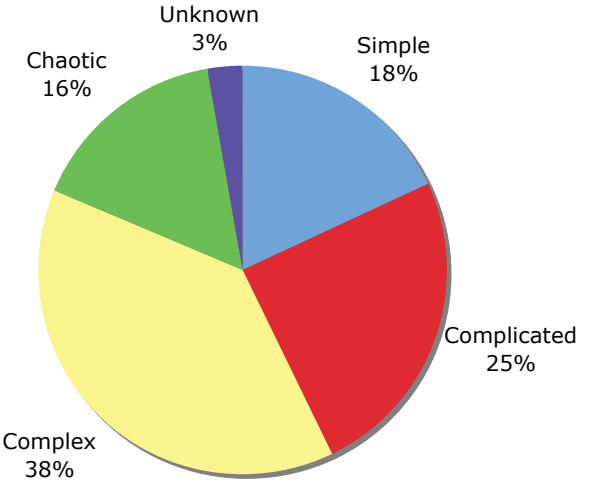

Figure 3 *Breakdown Of Typical Activities In Software Development*

Table 1 offers a sample of typical activities provided by participants, together with their sense-making results:

Simple	Complicated	Complex	Chaotic	Unordered
Knowing when a task is done	Ambitious (political) time-line	Changing requirements	Arguing about coding standards	No release deadline
Monitoring actual time spent	Fixing the build	Countering a belief in magic	Retrospectives without consequence	Resource shortage
Featuritis	Finding who to talk to	Task Estimation	Project volume too big	Lack of trust

Table 1 *Some Typical Software Development Activities*

Interpreting the results of the exercises led to the following realizations:

- Software development is a rich domain, with aspects and activities in all the different domains. The interactions between these aspects and activities are themselves often of a complex nature.
- Software development is a multi-level domain with self-similar characteristics, i.e., activities often tend to consist of sub-activities, each of which may be located in a different domain to the basic activity.
- The activities tend to be weighted more to the complicated and complex domains, with activities related to the coding aspect of software development landing in the complicated (or sometimes simple) domain, and activities associated with project management landing in the complex (sometimes chaotic) domain. Tasks dealing with interaction with a computer tended to be in the ordered domains, tasks dealing with interaction with other humans tended to be in the un-ordered, i.e., complex and chaotic, domains.

- The highest percentage of tasks and activities were in the complex domain. Although this is not sufficient to argue that software development as a whole is complex, it does suggest that many parts of it are amenable to analysis and treatment using complexity-based tools and techniques.

Success In Software Development Is Only Retrospectively Coherent

One thing that makes complex systems complex is their causality. As Dave Snowden says. 'If the system is chaotic/random then agent behavior is deterministic, which means I can use statistical instruments. If it's constrained, then the constraint structure allows predictability/repeatability. Strong constraints produce linear causality; weaker constraints provide repeatability that may be non-linear. However the moment you get the phase shift into a coevolutionary relationship between agents and system then there is no repeatability except by accident. In that context there is no meaningful causality, and any causality is only retrospectively coherent.'

In an ordered system, if you do something, you expect a specific result. Do it again, expect the same result. It's that simple. In a complex system, causality emerges as the system itself emerges, so that at the end, you can say how you got to where you are, but you can't guarantee that by doing exactly the same things, you'll get to the same place again—and you probably won't. In complex systems, we say the causality is retrospectively coherent. A classic example of retrospective coherence is task estimation. Before you do a task, it's almost impossible to estimate how long it will take. Afterwards, though, you can say exactly how long it took, and why it took that long. The same goes for projects. As a project goes on, the reasons for its success become established, not before. After it's over, you can say that the project was a success, and that certain things took place during the project—but you cannot say that the project was a success because these certain things took place!

Now shift the focus to a project. We tend to make the mistakes above when it comes to project planning. What worked last time? Why? Well, it must have been the people, the methodology, the meeting schedule—let's do it the exact same way again. Is it any surprise, then, that a project planned on this basis is likely to be a flop?

Contrary to Einstein's definition, in a socially complex system, insanity is doing the same thing over and over again and expecting the same result!

Complex Activities Require A Probe-Sense-Respond Model Of Action

Now that we have a reasonable basis for asserting that software development (as a whole) is a complex endeavour, or rather treating is as such, let's turn back to the Cynefin model. Since causality in the complex domain is retrospectively coherent, we'll always only know afterwards whether our efforts were crowned with success. To maximise flexibility in the face of this un-

certainty, the Cynefin method suggests a probe-sense-respond technique. Set boundaries for the system to emerge. Employ numerous probes, which will provide feedback on what works and what doesn't. Apply sense-making to the results to the feedback. Then respond by continuing or intensifying the things that work, correcting or changing those that don't. Tighten this into a small iterative loop, observing emergent patterns, amplifying good ones, and disrupting bad ones, until you end up successful at your endeavour. In fact, it's often the case that by applying this process, you discover value at points along the way. Your final product can end up looking very different to the original plan, and being very much better. You could not have defined these benefits at the outset and aimed for them; these are an example of emergent properties of the complex system.

The 'apply-inspect-adapt' model of agile development is a probe-sense-respond model. The Scrum project management framework utilises an iterative, incremental model of development, with work divided into iterations (called 'Sprints'), and a review and reflection step at the end of each iteration. This technique, called 'inspect and adapt', should properly be called apply-inspect-adapt, otherwise the focus is not on actually doing anything. If we think about this, it becomes clear that the apply-inspect-adapt loop is nothing else then the probe-sense-response cycle used in the Cynefin method for dealing with issues in complex domains.

This insight brings us full circle. We used a social complexity method to gain understanding of the cognitive bias of Agilists towards the field of software development, and ended by noticing that the Scrum framework implements the exact method called for by the Cynefin method for managing work in the complex domain. This leads us to the following conclusions:

- The theoretical underpinnings of Agile methods need to be understood for such methods to become truly scalable and sustainable. Insights, and answers to many of the questions, can be found in social complexity methods such as Cynefin.
- Agile methods such as Scrum provide a lightweight, proven framework for managing work in the complex domain, be it software development or something else.

Software development is a rich domain, containing many aspects, a large percentage of which can be classified as complex. The interaction between these aspects is also complex. Just as we have benefited from treating software development as complex, and taking advantage of the toolbox of social complexity, namely the Cynefin method, so the field (as well as many other fields of human endeavour) would benefit from a multi-ontological approach, taking the best techniques for the various domains, and combining them in an appropriate and flexible manner. More work is needed to reach a deeper understanding of the

inter-workings of agility and complexity, and it is the author's hope that the first (and following) workshops on complexity and real-world applications will not only provide insights, but also motivate other researchers to look into these fascinating fields.

Acknowledgements

Kent Beck, for being a strong source of motivation and inspiration; Dave Snowden and George Rzevski, for their input and insights into social complexity; Mark McKergow, for his excellent review and comments; Evelyn Harvey, for research and editorial assistance

References

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*, ISBN 9780201616415.

Cognitive Edge methods: Butterfly Stamping, http://www.cognitive-edge.com/method.php?mid=45.

Kurtz, C. and Snowden, D. (2003). "The new dynamics of strategy: Sense-making in a complex and complicated world," *IBM Systems Journal*, ISSN 0018-8670, 42(3): 462-483, http://www.research.ibm.com/journal/sj/423/kurtz.html.

Petzinger, T. (1999). *The New Pioneers: Men and Women Who are Transforming the Workplace*, ISBN 9780684846361.

Rittel, H. and Webber, M. (1973). "Dilemmas in a general theory of planning," *Policy Sciences*, ISSN 0032-2687, 4: 155-169.

Schwaber, K. and Beedle, M. (2001). *Agile Software Development with Scrum*, ISBN 9780130676344.

Snowden, D. (2005). "Multi-ontological sense-making: A new simplicity in decision making," http://www.cognitive-edge.com/ceresources/articles/40_Multi-ontology_sense_makingv2_May05.pdf.

Snowden, D. and Boone, M. (2007). "A leader's framework for decision making," *Harvard Business Review*, ISSN 0017-8012, 85(11): 69-76.

Spencer-Brown, G. (1979). The Laws of Form, ISBN 9780525475446.

Joseph Pelrine is C*O of MetaProg, a company devoted to increasing the quality of software and its development process, and is one of Europe's leading experts on Agile software development. After studying philosophy, psychology, and music in Vienna, his interests led him to work in the field of artificial intelligence and software development. He worked as an assistant to Kent Beck in developing eXtreme Programming, and is Europe's first certified ScrumMaster Practitioner and Trainer. Joseph Pelrine is an accredited practitioner for the Cognitive Edge Network, and his work focus is on the field of social complexity science and its application to Agile processes.