Project Change, a Way of Life

Version 1

Jurgen Appelo
jurgen@noop.nl
www.noop.nl
April 13, 2008

Introduction

In this article I try to link complexity science with agile software development. I attempt to show why there is no such thing as a best software development method, why managing scope is a too simplistic interpretation of the principle of "embracing change", why corporate standards for processes are a bad thing, and why you will never get things exactly right.

The article includes comparisons to biology and other types of complex systems, several little nuggets of wisdom, and some personal experiences involving my car.

Change Is the Only Constant

Environmental change is a hot topic nowadays. And though the possible causes for global warming are still being disputed – I'm quite sure that it's not *my* fault – everyone understands that people must learn to adapt to a changing environment. Trying to fight change is like me trying to prevent traffic jams. There's no point in being silly, other than offering some form of entertainment to those who know better. The ubiquity of change is nothing new, of course. The global climate has never stopped changing. The oceans and the sun have their moods too, ice ages come and go (see Figure 1). I deal with these changes by buying nice clothes, or taking them off. My car has air-conditioning, and someday I might swap it for a yacht, or an icebreaker.

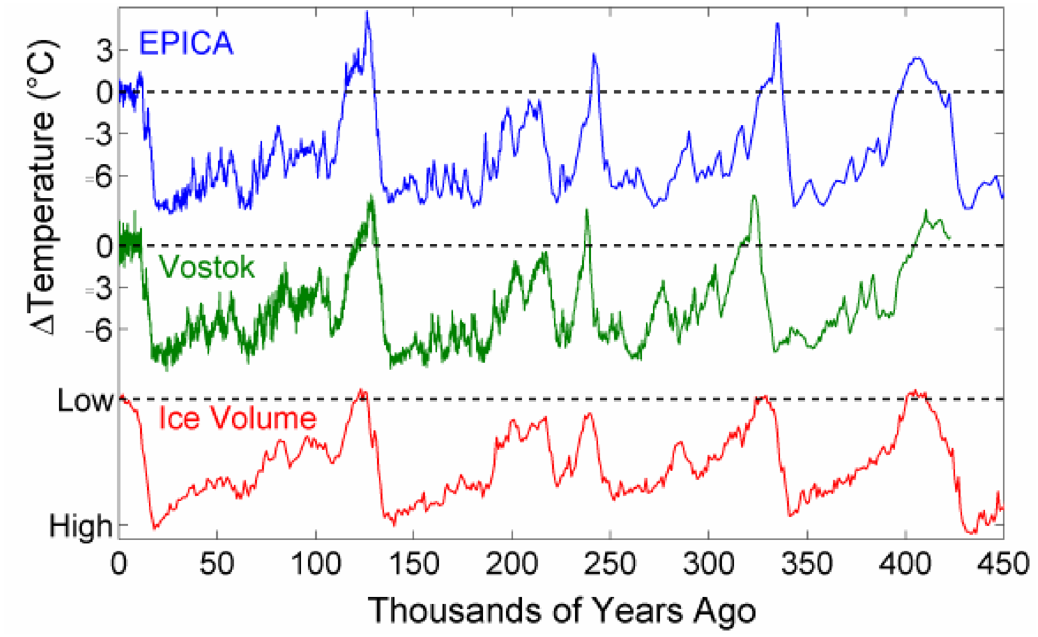

Figure 1: Ice ages and global temperatures

(Image created by Robert A. Rohde.)

The quote "change is the only constant" is attributed to Greek philosopher Heraclitus, and in many environments only those who "embrace change" – which is the subtitle of Kent Beck's bestselling Extreme Programming book – are able to survive. In biology, the mechanism that enables continuous change is *natural selection*. One of the best known examples of natural selection is the story of the peppered moth (*Biston betularia*), made famous by H.B.D. Kettlewell. During the industrial revolution, the black form of the peppered moth became much more common than the typical pale form (see Figure 2). The moths, which rest with open wings on tree bark, adapted their wing color when the trees in polluted areas of Britain became dark and sooty. (Air pollution in those days was a hundred times worse than it is now.) Predator birds had an easy time picking out the pale moths, while the dark ones became harder to find. The species simply returned to its more typical pale wing color when the air cleared in later decennia.

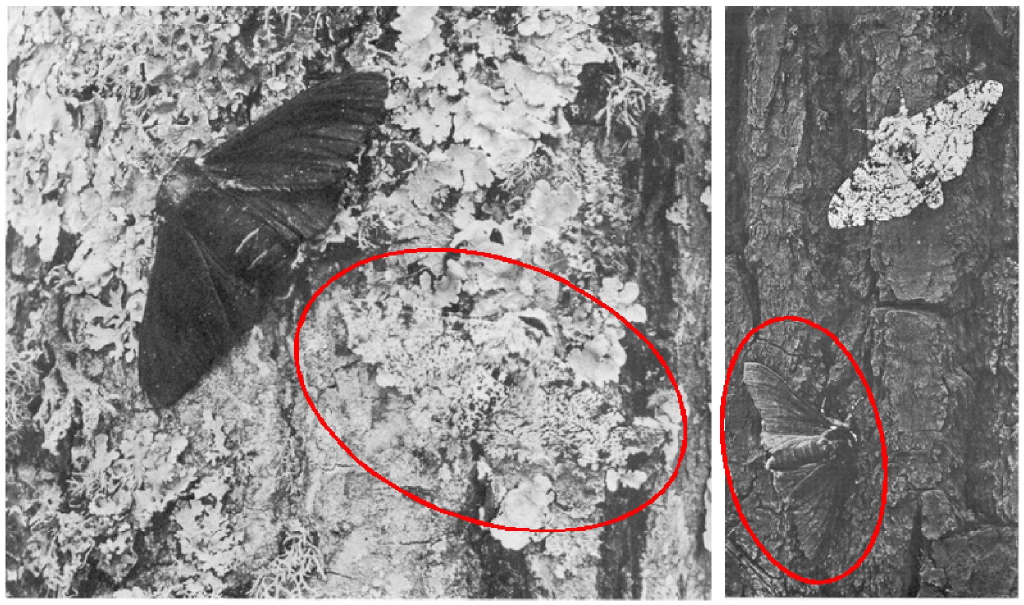

Figure 2: Two forms of the peppered moth

(Both illustrations from H.B.D. Kettlewell's 1959 article, "Darwin's Missing Evidence." In Evolution and the Fossil Record. San Francisco: W.H. Freeman and Company, 1978, pp. 28-33.)

These days it is an established fact that software products must often be adapted to environmental changes, and not just by changing the color of the packaging. The introduction of the *euro* as the new official currency in Europe in 2002 required businesses throughout the continent to spend many millions of French francs, German marks, Italian liras, Spanish pesetas, Austrian schillings, Portuguese escudos and Dutch guldens to be spent on software changes. Well-respected authors like Robert L. Glass and Frederick P. Brooks have described that successful software products often require *more* maintenance than the unsuccessful ones. The reason being that people like to try their favorite software in new unanticipated situations, and because successful software tends to outlive the hardware and business processes that were expected and considered during its initial creation. For example, many software products were never expected to outlive the 20th century and required a fix because of the *Year 2000 problem* (often incorrectly called the *millennium bug*).

A changing environment leads to software change requests. The *Manifesto for Agile Software Development* names "Responding to change" as one of its core principles. This is in stark contrast with the more traditional view that assumes that projects can be handled in stable environments. And while continental Europe has never convinced the British to switch to the euro, the British have been quite successful in convincing continental Europe to switch to the *PRINCE2* project management method. This method – an acronym for "Projects in *Controlled* Environments" – is an example of a traditional view on project management. It assumes that outcome, time and resources in a project can be pre-defined, and that the environment is controlled. But if environments could be controlled the peppered moth might have found it easier to change the color of the tree bark, instead of its own wing color. (And I would have bought myself an open car instead of a car with a glass roof.)

Three Ways to Change a Project

It might be worth pointing out that a changing environment does not necessarily translate to a changing project *scope* (new features or revised quality). The two other sides of the *Iron Triangle* (see Figure 3) are *time* and *resources*, and they are also subject to change (as in a changed timeline or team velocity, or changes in people and tools). In this respect it is interesting to note that agile methods usually describe only processes for handling scope change. I know of no methods explicitly defining processes for handling variable time or variable resources. In fact, the most common argument is that, of the three variables time, scope and resources, only time and resources can be "fixed", or specified as being "constant", while scope remains the one that is allowed to vary. (Apparently, this is what *time boxing* is all about.) But in a real environment *everything* is a variable. Scope is simply the easiest of the three to use for adaptive strategies in situations where *any* of the three variables has changed due to external pressure. This is why agile methods focus on managing variable scope, but it doesn't mean that time and resources need no management. They do. Practices for *resource management* (like recruitment and tools selection) and *time management* (like productivity management and individual efficiency) are essential in any organization and any project. They are just not normally covered by standard methodologies, and you will have to discover the best practices elsewhere.

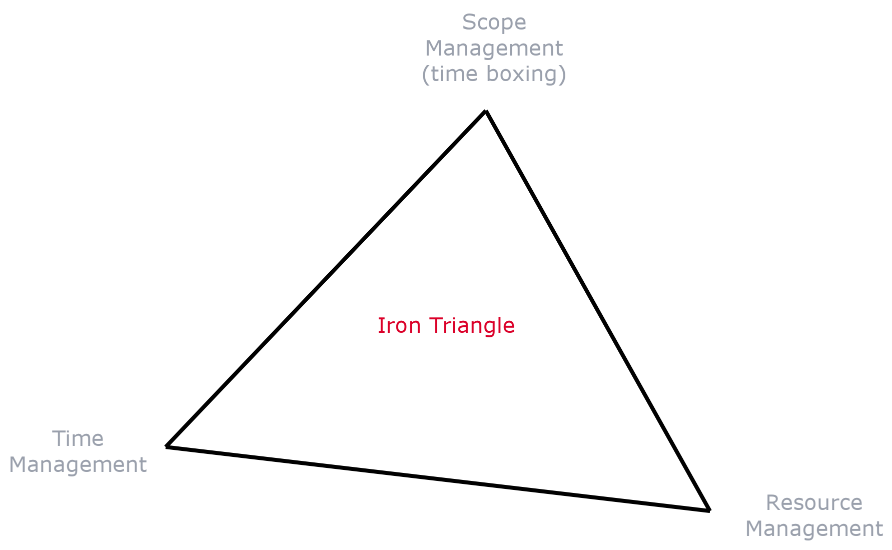

Figure 3: The Iron Triangle: Managing scope, time and resources

When I'm driving in my car to attend an important meeting, I often try to take care of all three variables scope, time and resources. Traffic jams and road works are a nuisance in my country, which means that I have to take into account the possibility of increased scope (alternative and often longer routes that get me where I need to be). My appointments (expected time and duration) sometimes change suddenly, always beyond my control, which is why I keep my mobile phone with me wherever I go. And my beloved car might fail me someday – only theoretically, of course – as my primary resource, which is why I carry a 24-hour road assistance card with me. And I always keep a bath towel in the trunk. You just never know when global warming is going to hit us.

Change the Projects to Change the Products

When talking about changing environments, it is important to understand that there's a big difference between products and projects. For many years people have been trying to compare projects, created by different organizations, in different situations. Comparisons are found in many reports and polls about best practices, project size and project success. *But what is a project?* Is all the effort that we put into version 1.0 of a system a complete project? Or do we distinguish different projects when multiple teams have been working on different subsystems? Does the same project include version 1.1 shortly released after the first version? Or do we treat that version as a new project?

Can we go to the extreme and aggregate all versions, released over a time span of several years, as one big ongoing project? Or are they necessarily different projects, with project boundaries defined by either major or minor releases?

This is deeply confusing. Isn't it strange that, after working in this business for 15 years, I still have no clear picture of what constitutes a project? I have come to the conclusion that I actually don't like the word. It is completely interchangeable with "work", "foo", "stuff" and "things". I don't understand anyone who comes up to me and starts talking about some project or other. Sometimes I want to grab people by their shoulders, shake them and ask them "Please… Make sense! Talk English! What do you mean??"

The standard view in other disciplines, like mechanical engineering, civil engineering and electrical engineering, is that a project ends when a product goes into production, and the maintenance phase takes over. This arrangement works fine for motor engines, bridges and devices for erotic electro stimulation. But in software engineering our products are often used long before the projects are considered to be finished. Besides, it is well known that the maintenance phase of a software product often swallows up the bulk of a customer's budget. This is because, contrary to motor engines, bridges and interesting electrical devices, most software products are never finished. So, what constitutes a project? Here are some standard definitions:

- *A project is a collaborative enterprise, frequently involving research or design, that is carefully planned to achieve a particular aim. (Oxford English Dictionary)*

- *A project is a finite endeavor – having specific start and completion dates – undertaken to create a unique product or service which brings about beneficial change or added value. (PMI/PMBOK)*

- *A project is a temporary organization that is needed to produce a unique and pre-defined outcome or result at a pre-specified time using pre-determined resources. (PRINCE2)*

The textbook definitions do not seem to agree. A project can be either temporary (finite) or endless; it can be either collaborative or individual; it may have complete plans or just some start and completion dates; and it can have a pre-defined aim or it allows for any beneficial change. I'm afraid that this doesn't answer any of my previous questions. It only raises new ones!

Tool builders have muddled the waters even further by giving the term "project" a technical and context-dependent meaning. A "project" in our development environment does not match with the "project" as it is created in our source control system, because both are (necessarily) organized in a different way. Neither of them matches the "project" as we have defined it in our issue tracking system, because issue management covers a wider area than just code. And the "project" folder on the network drive is different in its own right, because its scope extends to any non-coding initiatives and activities that are in some way related to the system that is under construction. With so many views on our projects it seems like a wonder we are even able to get some working products out the door.

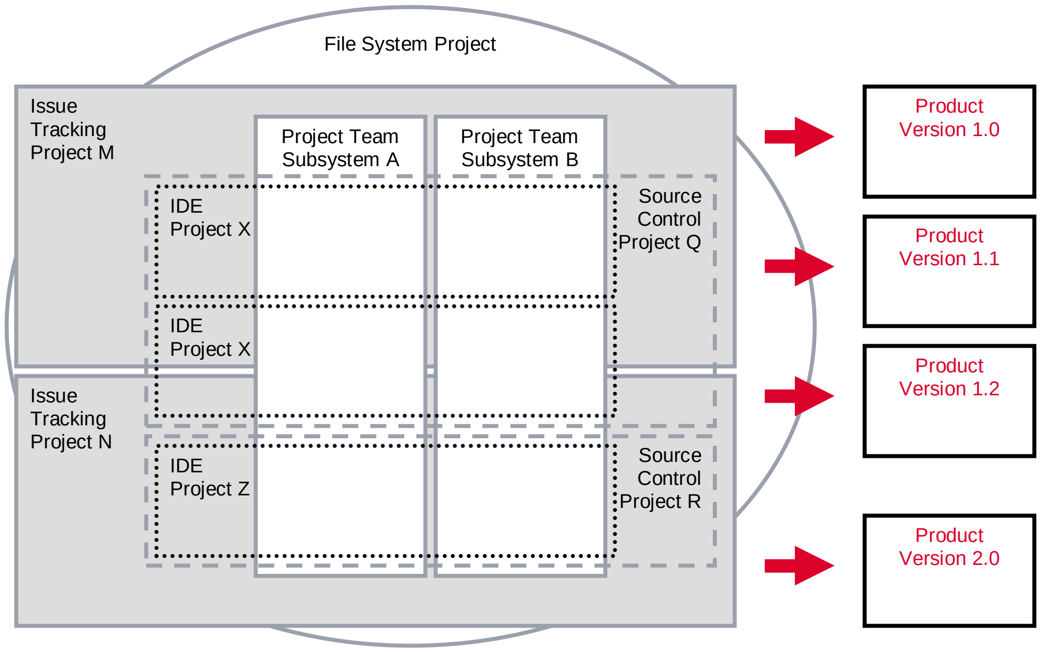

Figure 4: A plethora of projects results in a sequence of products

The term "project" is context-dependent, and some arrangement of projects (teams, subsystems, tools, etc.) is needed to produce a successful product. Environments evaluate products, not projects, because the products are tangible, contrary to the multitude of projects that were defined in order to have them built and delivered. Therefore, when discussing success, I prefer to talk of products. Figure 4 illustrates this idea. Some configuration of *projects* (teams, subsystems, tools) ultimately results in a sequence of *products*. We evaluate *products* to see how successful they are in their environment, and whether they need to change. And then we look at the *projects* to see how they need to adapt to the required changes.

Many people will tell you that you have to have the right processes in place in order to cope with scope changes. But this view is insufficient. In my opinion, *any part* of the multitude of projects (people, tools, processes *and* project configurations) is a candidate for change. In order to respond to environmental changes, you may have to change your people, or your tools, or your processes. You may even have to change yourself.

Every Product Is Successful... Until It Fails

When is a software product successful? We all know industry reports (particularly the infamous CHAOS report of the Standish Group) are always saying that only a small number of software products are "successful". But what does that mean? People have been struggling to find a proper definition for years and they are still not in agreement. One traditional view has it that a product is successful when it is *delivered on time, within budget, and according to specifications*. Others say that a product is successful when *it matches a customer's expectations*, paying back its investment in the form of business value created, as laid down in a properly defined business case. Another view is that a product is successful *when the stakeholders are happy*, whatever this may signify at the time of delivery. But I think they are all wrong.

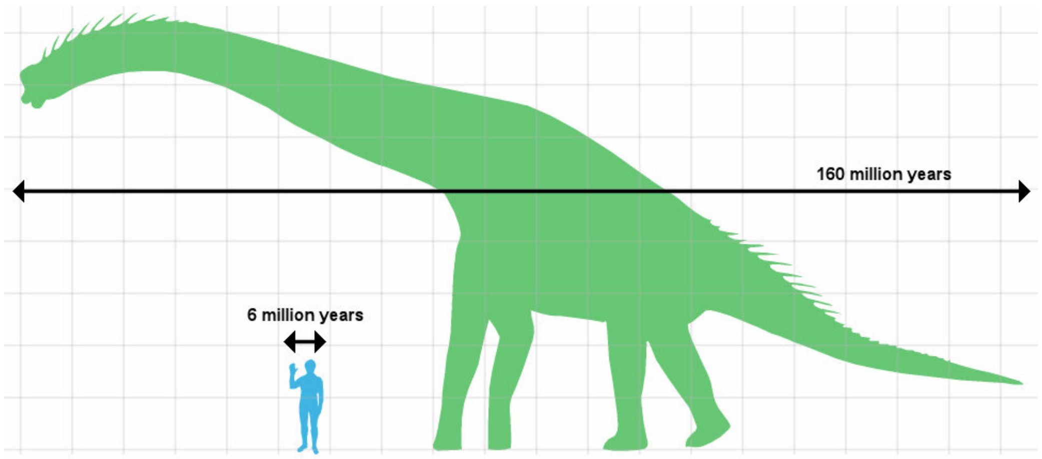

Figure 5: Dinosaurs versus Great apes (incl. humans)

(Image created by Matt Martyniuk.)

Do you think dinosaurs were successful? And do you think humans are successful? I suspect that many people answer 'no' to the first question and 'yes' to the second. However, dinosaurs have ruled the earth for about 150 millions years, while the family of *hominidae* (all species of great apes) now exists for six million years -- with humans wreaking havoc on the planet's surface for less than 200,000 years. It appears that humans still have plenty of time to prove that they are more successful than dinosaurs (see Figure 5). And do you think horses are successful? My daughter probably does, but she wouldn't have found the late and great paleontologist Stephen Jay Gould on her side. Gould pointed out that almost all species of wild horses (of the *Equus ferus* family tree) have vanished from the earth. Only *Equus ferus caballus* (the domesticated horses) can be considered successful in the sense that they have adapted and allowed *Homo sapiens* to sit on them, which is likely to have prevented their extinction. I think it is apt to say that every species is a success until it fails and goes extinct. Given the fact that 99.9% of all species are now extinct, failure appears to be in abundance.

I believe that every software product is a success, until it fails.

Some products that I have contributed to were a success for only a very short time, until the customers cancelled them because they finally figured out what they really wanted, which was something completely different. Even though these products never made it to their first release dates, team members and customers had been working happily together, but the business cases changed and they ran out of budget. I have known other products that were on time, within budget and according to specifications when, at the time of their first release, it appeared that they could not live up to our customers' expectations. Did they fail? Not really, because we found ways to recover from our errors, adapting to the new feedback, and delivering versions that won back our customers' trust. I also know products that are still being funded, several years after their first release date, despite the fact that they never returned their investment. It seems they are able to postpone their failure by retaining some stakeholders' support, for whatever reasons that may be. Maybe some people see value in these products simply because it gives them something we never had anticipated. Maybe they just enjoy sitting on them.

Last year I bought myself a new car, and I consider it a big success (see Figure 6). It looks fast, fancy and furious; it has a big sound system, and it has blue lights shining on the pedals. (I really like the blue lights.) However, those are not the main reasons for my contentment. I love my car because I love driving. To me, a car is a success as long as it takes me where I want to go, in a comfortable way, and without giving me any trouble. Basically, anything on four wheels that goes faster than I can walk is a successful vehicle to me. This includes quads and golf carts, and skelters with non-climate-neutral propulsion. But I know my car will only be a success as long as it lasts, because someday in the future I will have bought myself another one.

Figure 6: Me and my car

Success is the continued absence of failure. In my opinion, other definitions seem to be insufficient. Products can be of some value to someone, even though they are not on time and within budget; even though they never returned their investment; and even though they may not satisfy all stakeholders. Species are successful until they go extinct. My car is successful until the day it fails to please me. Products are successful until the day they have lost all users. The media player *Winamp3* was not as big a success as *Winamp 2* was. Due to many problems with version 3 it lost many users and people were reverting to the older stable version. Nullsoft, the creators of Winamp, responded appropriately by adapting and merging the best parts of both versions into *Winamp 5*. (Microsoft faces a similar situation with *Windows Vista* versus *Windows XP*, which makes everyone wonder how Microsoft is going to adapt.) The principles of embracing change and continuous adaptation are intended to postpone the inevitable moment of losing the last user. But all software products *will* fail someday. I'm 99.9% sure of that.

Everything Is Relative, Fitness Too

The success of a product is always relative to its environment. I consider my car to be quite a success. The blue lights shining on the pedals, and the sound system pounding on my eardrums, have contributed significantly to this perception. But I'm sure some other cars would have been an even bigger success, possibly with even prettier lights and heavier sound, if only the size of my purse had matched the size of their price tags. I also know other people would never care for my car. They have other criteria to measure their favorite vehicles against. Some feel happiest when driving a second-hand pink mini-bus, preferably without air-conditioning. Some don't even care for blue lights on the pedals.

When discussing the success of species, biologists prefer to talk of *fitness*. Like success, fitness is relative. There is no absolute fitness in nature because there is no common scale to measure it against. Fitness depends on the niche a species is filling, the environmental conditions that it has to cope with, and any other species that happen to exist in that same environment. It is said that species *coevolve*. They often have a hard time reaching and maintaining their fitness levels because they keep adapting to each other. Natural selection makes sure that species change to keep up with changes in their environments.

The only useful measure of success (or fitness) of a software product is people's continued investments in it. The fitness of a species is determined by its ability to consume energy and transform it into offspring. The fitness of a product (including all its copies) is determined by its ability to consume people's time and money, transforming it into business value. Selection pressure in software development is the pressure that products are under not to lose their users. As with species, selection acts on the *phenotype* of software products – the set of all properties as perceived by the environment (see Figure 7). This includes functions, qualities, pricing, packaging, etc. Products can lose their users because functionalities do not meet with the customers' (changed) expectations, or because performance and security, or some other quality criteria, do not conform to the latest standards. Products can also lose their users because a competing product has entered the market, or because the need for it has evaporated, or simply because all copies of the software broke down on the same date. There can be many reasons for the loss of users, but the end result is always the same: Fitness drops to zero. The product dies.

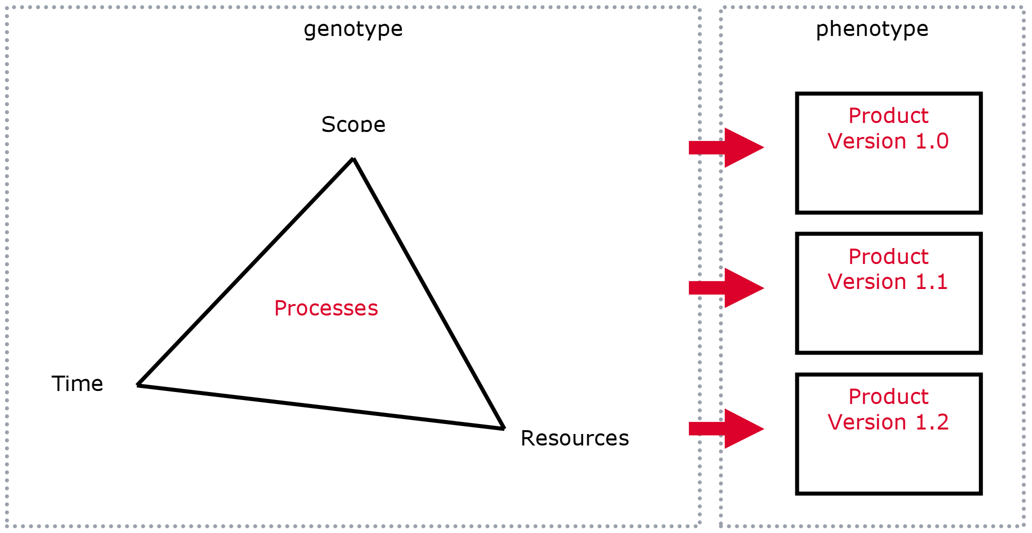

Figure 7: The project (genotype) and the product (phenotype)

Adaptation of products takes place at the level of the *genotype*. For a species the genotype is its DNA, which "programs" the individual organisms with strategies for survival and reproduction. For software products the genotype is the set of all practices applied during their construction. It is the sum of scope, resources, time and processes in a software project that determines the fitness of a product, its chance of survival and the business value it delivers. In software projects we try to find out whether customers are going to like the product, and whether quality criteria will be up to the latest standards. In software projects we have to keep an eye on any competing alternatives that customers might choose from, and we have to make sure that the business case for a product's existence remains valid.

The fitness of anything, whether it is a car, a species or a software product, is always relative to its environment. It is evaluated on the basis of its external (phenotypic) properties, and anticipated by internal (genotypic) programming. If you understand this simple mechanism, you will understand that survival is a never-ending struggle to improve resources and processes in software projects. Continuous improvement is – quite literally – a way of life.

What's Driving Our Improvements

The first couple of years after getting my driver's license, I was a really bad driver. Taking a seat behind the wheel (if I could find the correct one) really freaked me out. I vividly remember several cases of horror and despair when the engine of a car failed on me right before a traffic light went green. But after two flat tires (simultaneously) in the Nevada desert, one flat tire in the Interior of Brazil, a broken gear box in the Interior of South Africa, and ending up in a ditch alongside the Loch Ness Lake in Scotland, I eventually learned to handle all kinds of circumstances.

A *performance system* is the name for the collection of rules in a complex system that determines how it behaves under the input that it receives from its environment (see Figure 8). A rich performance system is one with lots of rules for many different situations. My driving style is a performance system which I have been tuning to near perfection over a period of eighteen years. For example: I now know that I should never drive in reverse over a strip of road spikes, that while driving in the night in underdeveloped countries I should watch out for pot holes, that I must treat the gear box of cheap rental cars with gentle care, and that the shoulder of a road may not be the best place to turn a car around in a pitch dark night. They never told me these things during my driving lessons. I had to experience them, and I had to update my performance system accordingly by adding and updating the rules that I keep in my head. In complexity theory this is called *rule discovery*.

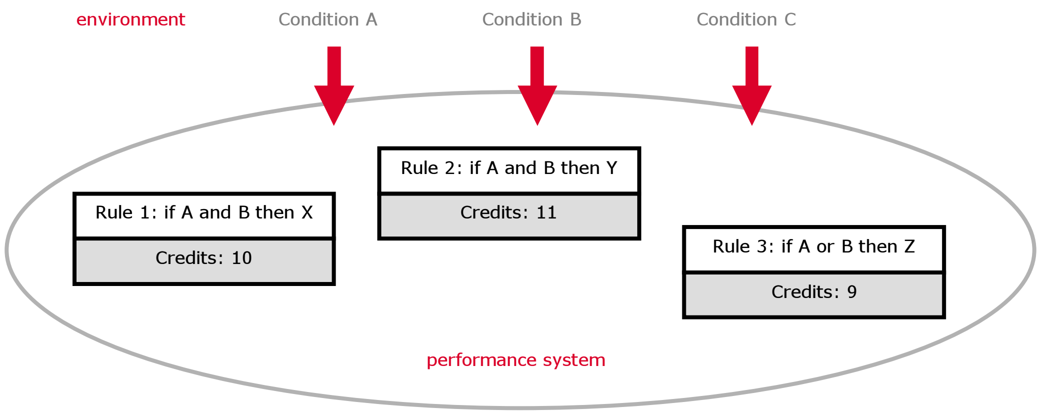

Figure 8: A performance system of three rules

Another part of the learning process in a complex system is called *credit assignment*. Every time I like the results of having applied a rule, I assign one more credit to it. The applicability of the rule, given the context of a situation, is confirmed and its importance is increased for that situation. Likewise, every time a rule did not give me the result I liked, I deduct one credit, and its relative importance for the current situation is decreased. Of course, this credit assignment is something I rarely do consciously. For example: I never change to a lane on the left while I'm passing a car on the right. After having once been hit on the side by a truck that I was overtaking, I always want to keep an eye on every vehicle that I'm passing by. After eighteen years of driving I have a very complex set of rules, most of them applied on a subconscious level, with some rules being very important but only in specific situations (rules 1 and 2), while other rules are less important but applicable to a wider variety of circumstances (rule 3).

Competition of Rules

In a performance system there are usually contradictory rules with different priorities, as with rules 1 and 2 in Figure 8. The system will select either of these two rules, when applicable, with the chance of selecting rule 2 being slightly higher, given the higher credits that it has earned. However, in a changing environment the credits that are being assigned to the rules might change. When rule 2 ceases to lead to satisfying results rule 1 may swing back, receiving more credits than rule 1, and consequently being selected more and more often. The performance system built into the DNA of the peppered moth is responsible for having its wing color swung back and forth between black and white. And the performance system inside my head is responsible for alternating between several routes for driving between my home and the office, where I let my selected route depend on weather conditions, the time of day, road works and whether or not I'm in the mood for my favorite (and expensive) Italian caterer.

Organizations maintain their own performance systems. Some do this explicitly, in the form of documented processes, but most maintain rules in the minds of employees and team members. Rule discovery is the principle of learning new practices and new ways of doing things. Credit assignment is the principle of finding out, by experience, which of these rules work best in which situations. Project evaluations, introspections, reflection workshops, daily standup meetings and several other best practices have been proposed by experts to assist organizations in their rule discovery and credit assignment processes. But one should not forget that these best practices are themselves rules in the performance system, and should likewise be subjected to the credit assignment principle. Furthermore, people always discover and prioritize rules, even without workshops and meetings. Rules about handling customer emails, rules about high-priority maintenance issues, rules about handling changes to the planning, rules about vacation and sick days, rules about file names and storage, rules about beta and live deployments, and many, many more. Any formal software process improvement initiatives in an organization contribute to the performance system, but most rule discovery and credit assignment is done in people's minds. They don't read manuals while driving their car, and they don't read manuals while responding to input from the environment. There's nothing wrong with that. It's just the way the world works.

What we can learn from all this is that we should be prepared to let best practices compete with each other. It's OK to occasionally try out a new way of carrying out system tests, or a new way of documenting requirements. And even when there are company-wide standard procedures for source control or daily standup meetings, every now and then you should try some new (or old) alternatives, and carry out a credit assignment. It is essential for any organization that needs to adapt quickly to a dynamic environment.

There are performance systems in biology and many other disciplines. They enable us to learn how complex systems manage themselves. My experiences while driving taught me how to free myself from the road spikes, pot holes and ditches that I got myself into. Unfortunately, hazards on the road are always changing. I haven't seen any road spikes in ten years, but nowadays it is important to watch out for people wearing iPods. Who knows what the dangers are in ten years?

The Race to Avoid Failure

Despite all our efforts to adapt and improve, it sometimes seems to have no effect whatsoever. Developers are never completely happy with the tools they are using. Users are never fully content with the software we build for them. And team members are never quite satisfied with the processes in their software projects. Why is that? The answer can be found in an old children's book from the 19th century.

Species do not evolve with the aim of becoming better at what they do. They evolve to suppress the risk of extinction. Success is the postponement of failure. Scientists have found that the ability of families of species to survive does not improve over geological time. From the fact that the risk of extinction in ecosystems has never dropped, it follows that species have never succeeded in becoming any better at avoiding it. This means that the goal of evolution is not to lower the chance of failure. It is to prevent the risk of failure from increasing. There are examples of species, including crocodiles, pandas, sharks, sturgeons and horseshoe crabs, often called *living fossils*, that haven't changed an eyelash in a million years. Apparently, their environments didn't require them to change. And when environments don't change, species don't bother with the effort either.

When species change, it is usually not just because of changed weather conditions. Species don't lead isolated lives. They are linked inextricably and they often need to adapt to each other's changes. For example, plants might evolve tougher surfaces and chemical repellents to fend off hungry insects, while at the same time the insects evolve stronger jaws and chemical resistance mechanisms. Species change to remain in the game. It is like an evolutionary arm's race, which has been given its own colorful name: *The Red Queen Race*. The term is taken from Louis Carroll's "Through the Looking-Glass", where the Red Queen said to Alice:

"It takes all the running you can do, to keep in the same place."

The Red Queen Race is an evolutionary hypothesis describing that a complex system needs continuous adaptation in order to maintain its current fitness, relative to the systems it is co-evolving with. Some argue that the Red Queen Race, or the principle of co-evolving species, is an even more important driver of evolution than any other kind of environmental changes.

Figure 9: Alice and the Red Queen

(Image used with permission from CartoonStock.com.)

The Red Queen Race explains why most users are never completely satisfied with the software products they are using. After all, even though the products get better with each release, the users keep adding new requirements. Software products do not evolve to become better at what they do. They evolve to postpone the (inevitable) moment that they will be discarded. Success is the postponement of failure. And when environments don't change, software vendors don't bother changing their products either. And why should they? Lack of strong competition is why Microsoft did not release any new versions of *Internet Explorer*, after version 6, for more than five years. One might even argue that the threat of being pushed back by competing products is an even more important driver of software evolution than the new requirements of existing users. A vendor may be able to ignore its users, but it cannot ignore its competition.

My current car cost me twice as much as my first one, and it has ten times the number of features. But has it made me any happier? Only for a short while, I'm afraid. The fact that it has just one parking sensor in the rear, and not on any of the other five sides, is starting to get on my nerves. And the heating in the seats takes too long to climb to a comfortable temperature. And the brightness of the blue lights on the pedals cannot be adjusted. Day by day, ever so slowly, my car is falling behind in the Red Queen Race.

The Rise and Fall of Systems

Every complex system (whether DNA, a brain, a business company, a teenage street gang or a software project) is constructed from a large number of elements, connections and rules. One configuration is just one version out of the many different possible combinations of the parts that comprise the system. Now I will challenge your imagination by asking you to visualize that all these different configurations of a system are points on a one-dimensional scale. Two configurations are said to occupy two adjacent points on this scale when they differ in only one element, rule or connection. For example, you may compare two snapshots of a software project that are similar in every respect, but with one resource or best practice changed into some alternative. (I understand that this may stretch your imagination to the limit, but please, bear with me.) Likewise, two configurations are said to occupy two points far away from each other on the scale when the two versions of the system are completely unlike each other. For example, you can compare two snapshots of a software project, with all resources and practices changed into alternatives. Of course, properly drawing up all combinations of a real system in a graph would actually require thousands, millions or even billions of dimensions, but I'm afraid that this would be a little hard to turn into an ordinary chart on two-dimensional paper. Therefore I will settle for a more abstract visualization, using just one dimension.

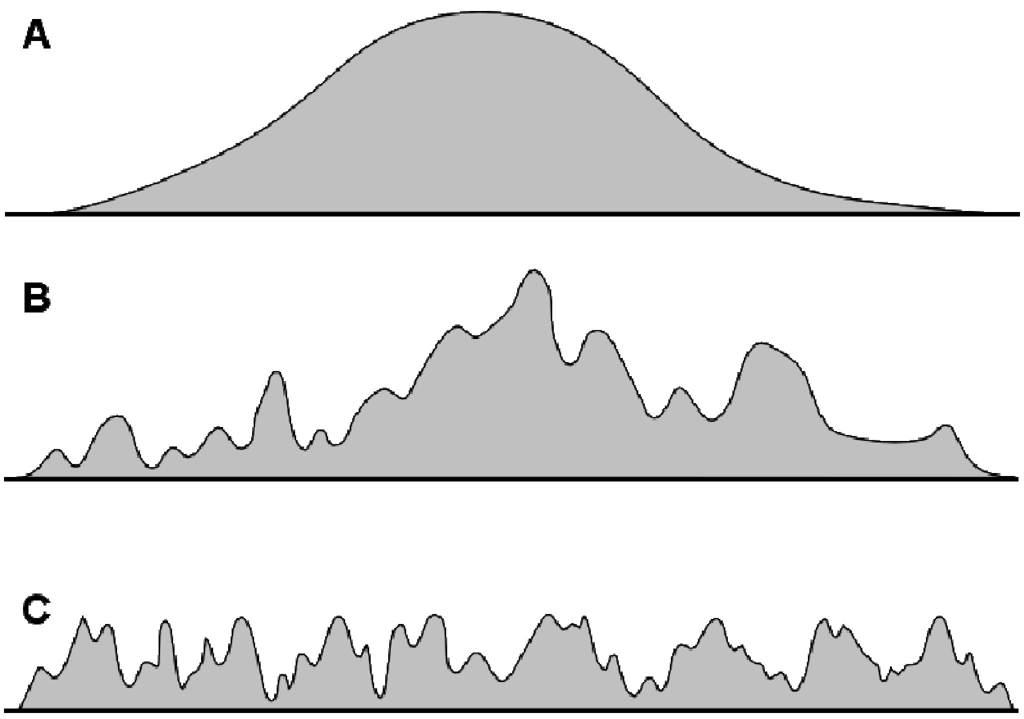

Figure 10: Fitness landscapes, easy (A), rugged (B) and random (C)

Given a specific environment we can imagine that (theoretically) one is able to measure the fitness (or chance of survival) of all configurations of the system. When plotted in the second dimension this gives us what system theorists call a *fitness landscape* (Figure 10). It plots the distribution of fitness of a system (the phenotype) based on a scale that represents all possible combinations, or states, within the system (the genotype). This fitness landscape is static as long as the environment is static. But we know that environments are never static, and changes in the environment usually require changes to our software projects. This is because a different environment results in a different fitness landscape for the same system.

When we change one part of a system (one gene, one employee, one teenage gang member, one best practice) into something else, it follows that the system moves either to the left or to the right on the fitness landscape, probably making it either more or less fit. *Complex adaptive systems* are able to make these changes to themselves, and those that are able to find the highest peaks on the fitness landscape are the ones most able to survive. Systems that have the ability to tune themselves in such a way are said to be doing an *adaptive walk* across the fitness landscape (Figure 11). An adaptive walk is the process by which a system changes from one configuration to another, often by gradual steps, in order to stay fit. The system 'walks' across the fitness landscape, and each step may lead to an improvement in the performance of the system against the (changed) criteria imposed on it by the environment. Software projects do their adaptive walks by always adding and replacing features, qualities, people, tools and processes.

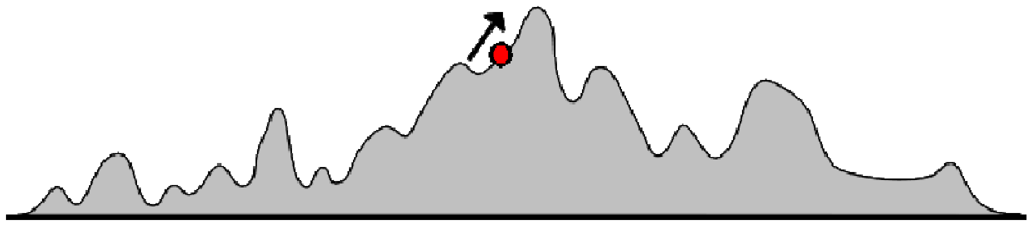
Figure 11: An adaptive walk across the fitness landscape

For biological systems the search across the fitness landscape is not an intelligent one. DNA is mutated in random ways, and species do their adaptive walks in all directions, including every wrong one. But natural selection comes to the rescue by making sure that the individual organisms that happen to have landed (blindly) on a higher position on the fitness landscape are the ones most successful in reproducing themselves. Human-made systems apply a different strategy. We cannot afford to simply try out every combination of features, resources and processes. In our case not natural selection but *conscious selection* comes to the rescue. Humans have the intellectual capacity to make an educated guess on where the higher peaks are, even though the fitness landscape is an abstract thing. We balance features against qualities, we fire and hire employees, we discard and select tools, and we add and rework the processes in our software projects, hoping (and often expecting) to walk in the right direction, improving the fitness of our systems along the way.

Shaping the Landscape

The shape of the fitness landscape is directly related to the interconnectedness of a system. This is easy to understand. Suppose that all elements in a software project have no influence on each other. In that case replacing one resource or process with another will have no effect on any other part of the project. Each individual element has its own isolated effect on the adaptive walk across the fitness landscape, which is positive, negative or neutral. It then follows that there is one and only one best configuration for the entire software project, namely the one in which each individual part has a positive (or neutral) effect on the system's fitness. This configuration corresponds to the single highest peak in landscape A of Figure 10.

You will understand that such a situation is unrealistic. In most complex systems there is a level of interdependence between the individual elements. Genes for the growth of feathers and genes for the growth of wings are related in such a way that they have a combined effect on an animal's fitness. The same applies to genes for fins and genes for gills. But an animal born with an arbitrary mixture of these, like a combination of feathers and fins, is unlikely to be fit enough to survive. There is a mathematical principle underlying the interdependencies of elements and the form of the fitness landscape. Figure 10a applies to systems with no dependencies among their elements. Figure 10b applies to systems where elements have moderate dependencies. Figure 10c applies to systems with many interdependencies.

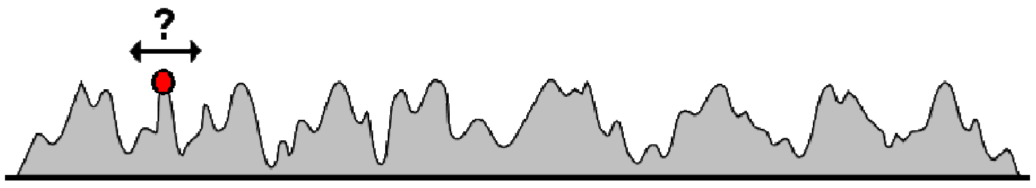

Figure 12: An impossible walk across a chaotic landscape

It appears that, with a large number of interdependencies in a system, the height of the accessible peaks falls. It results in a fitness landscape with a random collection of peaks, not one of them clearly being the highest. This is called the *complexity catastrophe* and it limits the potential of a system to achieve an optimal performance (Figure 12). Therefore, the ruggedness of a fitness landscape is a crucial property. Systems should never have too many interdependencies, or their efforts in adaptation become chaotic. It is the reason why there should be only a moderate interdependence between features, qualities, people, tools and processes in a software project. Changing any one of these in a project must never lead to a chaotic walk over the fitness landscape, or else any improvement effort will yield completely unpredictable results. The *high cohesion, low coupling* principle is well known among software architects. It means that classes must be well-formed (high cohesion) and that there shouldn't be too many interdependencies (low coupling). Something similar applies to processes. It is often claimed – even by Kent Beck himself – that Extreme Programming is a method with many interdependencies. This might mean that you cannot replace a couple of processes of XP or your improvement initiatives will run into chaos. Other methods consist of more loosely coupled practices, which makes them easier to adapt in a changing environment.

Never Forget How to Run

Changing environments and the Red Queen Race have dramatic implications for fitness landscapes. They make it seem as if they are made of rubber. It is as if the peaks and valleys are always on the move, and forever rising and falling. A system with a configuration that was fit yesterday may be unsuitable for the environment that it must live in tomorrow. Today's best practices may be tomorrow's worst practices. Species, business managers, teenage gang members and project managers have to keep changing, because it takes all the running they can do just to stay on top of a moving peak. They have to change to stay in the game, and if a peak drops and turns into a valley, they quickly have to find themselves another peak.

In relatively stable environments the fitness landscape doesn't change much. Once an organization has found a high peak, it can comfortably stay there. It can switch from adaptation to optimization, making sure that it makes use of its situation in the most efficient and effective ways possible. But with changing environments adaptation is more important than optimization. In stable environments, systems tend to lose the ability to change. People forget how to change when the environment they live in has always seemed the same. The danger is that they may not notice it when their comfortable peak has been dropping slowly and turned into a valley. I believe that contentment with the success of your software projects may be your worst enemy. Your once brilliant colleagues may suddenly turn out to be way behind the times. The tools you have been using may not be giving you the best results anymore. And your favorite development method, which was once an asset, may have slowly turned into a liability.

This is what it means to be agile.

The Agile Manifesto never said you should stick to XP or Scrum or any other method. It says you must understand and embrace change. This is why improvement of resources and processes must never stop. It must be your way of life. Don't ever be content. Keep running! Keep changing features, qualities, people, tools and processes. Take a short break every now and then, review the landscape to see what the peaks are doing, and then resume the race. (It might help if you have a nice car.)

Conclusion

In this article I have argued that, because change is the only constant, there is no such thing as a "controlled" environment. There is only a changing environment, which constantly evaluates your products (the phenotype) and you have to respond to environmental changes by improving your projects (the genotype). Your improvements are not limited to changing features. It also requires changing qualities, people, tools, processes, and project configurations. These changes are necessary to postpone failure, for as long as possible, which equals the loss of all users of your product.

You are not alone in your improvement efforts. Your users are changing too, and so are your competitors. This is called the Red Queen Race, because all coevolving parties have to keep improving just to stay in the game. Your organization is a performance system that needs to be adaptive by allowing the competition of best practices. Strict enforcement of standards limits your organization's ability to respond to environmental changes. Another thing to watch out for in your software projects is the interconnectivity of all things, which should not be too high, because it leads to improvement efforts being chaotic. There is also the risk of contentment with success in a relatively stable environment, because your organization might have forgotten how to adapt when the time to change is near.

Note: This article is to be part of a book I'm writing about complexity theory and software development. You may follow my efforts, and silently watch me struggling, on www.noop.nl.

Sources and References

"Manifesto for Agile Software Development" <http://www.agilemanifesto.org/>

"Dinosaur" <http://en.wikipedia.org/wiki/Dinosaur>

"Euro" <http://en.wikipedia.org/wiki/Euro>

"Great ape" <http://en.wikipedia.org/wiki/Great_ape>

"Heraclitus" <http://en.wikiquote.org/wiki/Heraclitus>

"Ice age" <http://en.wikipedia.org/wiki/Ice_age>

"Living fossil" <http://en.wikipedia.org/wiki/Living_fossil>

"Project" <http://en.wikipedia.org/wiki/Project>

"Project management" <http://en.wikipedia.org/wiki/Project_management>

"Red Queen" <http://en.wikipedia.org/wiki/Red_Queen>

"Wild horse" <http://en.wikipedia.org/wiki/Wild_horse>

"Year 2000 problem <http://en.wikipedia.org/wiki/Y2K>"

"Winamp" <http://en.wikipedia.org/wiki/Winamp>

Beck, Kent (2005) *Extreme Programming Explained: Embrace Change*. Upper Saddle River: Addison Wesley

Brooks Jr., Frederick P. (1995) *The Mythical Man-Month: Anniversary Edition*. Addison Wesley

DeMarco, Tom and Lister, Timothy (1999) Peopleware: *Productive Projects and Teams (Second Edition)*. New York: Dorset House

Gell-Man, Murray (1994) *The Quark and the Jaguar: Adventures in the Simple and the Complex*. New York: Owl Books

Glass, Robert L. (2003) *Facts and Fallacies of Software Engineering*. Boston: Addison-Wesley

Gould, Stephen Jay (2002) *The Structure of Evolutionary Theory*. Cambridge: Belknap Harvard

Highsmith III, James A. (2000) *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. New York: Dorset House Publishing.

Holland, John H. (1995) *Hidden Order: How Adaptation Builds Complexity*. New York: Basic Books.

Lewin, Roger (1992) *Complexity: Life at the Edge of Chaos*. Chicago: University of Chicago Press

Mallet, Jim (2003) "The Peppered Moth: A Black and White Story After All" <http://www.talkreason.org/articles/mallet.cfm>

Miller, John H. and Page, Scott E. (2007) *Complex Adaptive Systems: An Introduction to Computational Models of Social Life*. Princeton: Princeton University Press

Office of Government Commerce (2002) *Managing Successful Projects with PRINCE2 (Third Edition)*. Stationary Office

Solé, Richard and Goodwin, Brian (2000) *Signs of Life: How Complexity Pervades Biology*. New York: Basic Books.

Standish Group (1995) *The Standish Group Report: CHAOS*. <http://www.educause.edu/ir/library/pdf/NCP08083B.pdf>

Waldrop, M. Mitchell (1992) *Complexity: The Emerging Science at the Edge of Order and Chaos*. New York: Simon & Schuster

Profile

Jurgen Appelo is Chief Information Officer at ISM eCompany (www.ism.nl), recently rated as the #1 fastest growing technology company in The Netherlands. He leads a horde of 50 software developers, development managers, project managers, consultants, quality assurance managers, service managers and kangaroos, some of which he hired accidentally.

Jurgen is primarily interested in software engineering, quality improvement and complexity theory, from a manager's perspective. He is trying to write a book about this, and he keeps track of it on his blog (www.noop.nl). However, sometimes he puts it all aside to do some intensive programming himself, or to spend some time on his ever-growing collection of science fiction and fantasy literature.

Jurgen lives in Rotterdam (The Netherlands) -- and sometimes in Brussels (Belgium) -- with his partner Raoul. He has two kids, and an imaginary hamster called George.