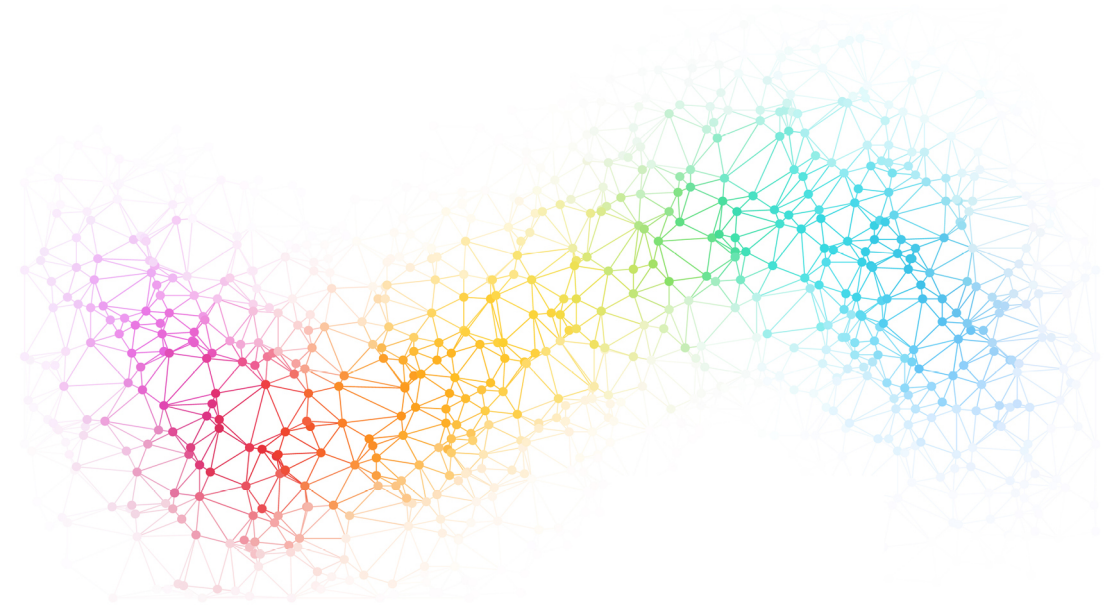
Antifragile Software

Building Adaptable Software with Microservices



Russ Miles

Grant Tarrant-Fisher
Sylvain Hellegouarch

Antifragile Software

Building Adaptable Software with Microservices

Russ Miles

This book is for sale at http://leanpub.com/antifragilesoftware

This version was published on 2016-02-08

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Russ Miles by spreading the word about this book on Twitter!

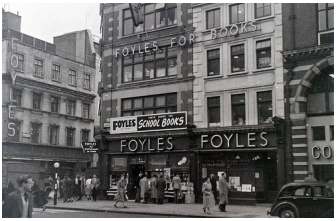
The suggested hashtag for this book is #antifragilesoftware.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#antifragilesoftware

Contents

Introduction



Foyles Bookshop, my second favourite bookshop and home of many books that are not at all boring, but many in IT that are.

> *"We've spent over a decade now becoming more and more agile and adaptable in our ways of working. Unfortunately our software is now struggling to keep up with the pace of innovation that is increasingly being demanded by modern businesses. It's time to sort that out."* - Russ Miles

There are enough boring books out there on software development, enough probably to fill Foyles[1] many times over.

This is ***not*** one of those books.

[1] http://www.foyles.co.uk

What this book is NOT about

Rather than waxing lyrical on highfaluting thoughts, concepts and ideas in software development or getting tangled up in the exhausting and ultimately mostly meaningless discussions (see: wars) around languages and frameworks, this book is going to dive into the ***practical tools I've found that really help with building software*** and meeting the challenges of modern software development.

That means the ***following topics are rendered essentially meaningless and so have been completely steered clear of***.

Brand X Process

Processes are merely ways of organising work with the aim of trying to ensure that something valuable happens.

This book will be avoiding discussions of the form "Scrum is better than Kanban because… <insert biased and inane argument here>" and instead concentrate on practical tools that you can use to tailor your own process.

As a wise man once said, do what works for you. I aim to help you to figure out what works for you, not sell you on Brand X of anything in particular.

Brand X methodologies

Methodologies are colloquially used to mean processes in software development. Methodology is a misnomer typically used by people who wish to make something sound much more

important and complicated than it actually is, often in order to drive up consultancy day-rates or justify cash-cow Certification regimes.

Treat ***any methodology as suspect and with intense skepticism***. In fact, ***treat all doctrines, ideas and rules that way*** and you'll do well in software.

Doctrine & Dogma (sort of)

I have a sneaking suspicion, but ironically I can't necessarily empirically prove, that ***the majority of what is broadcast in software development in terms of self-evident practices and processes is often just simply someone's own beliefs with no empirical basis whatsoever***. Nothing that would get past a half-decent scientific journal's panel anyway.

On their own, and when clearly stated as ideas and beliefs for your own consideration, ***doctrine from experts in our field is not necessarily damaging or unimportant***. If we held out for clear proofs of every tool, process or practice we came across then I'm fairly sure we'd never get any work done. ***The problem is that in the majority of cases that doctrine quickly becomes dogma***; the ***unvarnished and unequivocal truth to be defended to the death*** by its adopters.

I've lost count of the number of times I've heard the argument "you're not doing it right, to be really Agile you need to be doing X". I pick on Agile here because it is an area of thought in software development that, due to its broadness and even in the face of clearly useful foundational principles, it still attracts its fair number of ***coaches, consultants and bigots*** (I often like the term zealot) even some 10 years since its inception.

The problem with these individuals and organisations is they do not have ***skin in the game***. They are ***offering unempirical advice and belief as if it was certainty with no direct threat from the action being taken on that advice***. This is dogma prevailing.

When Dogma prevails, valuable outcomes are lost in the noise of opinion-driven accuracy to some authority. This has no place in this book.

Here I shall be talking about exactly ***how I design systems, with real skin in the game every time I do***.

Philosophical Underpinnings

One means of avoiding dogma is to display your reasoning ready for it to be (hopefully) fruitfully critiqued and mined by your happy or unhappy readership. The other is to only promote those things that you have done yourself, thereby having real skin in the game. We'll apply both here.

Every book that aspires to push things truly forward has a set of philosophical underpinnings that inform the thought and ideas that it proposes and this book is no different, except that I'd like to make my philosophical underpinnings more explicit than most by listing them out here.

Axioms

> *"...is a premise or starting point of reasoning."*, Wikipedia

Axioms are the undisputed claims upon which I can then build up everything else I put forward in the rest of this book.

I could quite easily simply dump a collection of tools, practices and processes in this book on you my understanding reader without some sort of shared understanding underpinning the whole exercise. This is what many books do, but more often than not I'd then spend the rest of my career explaining the ideas underlying those techniques, and in fact many do spend a lucrative career doing so.

My aim is a different one here and so, in order to get a few questions out of the way, here are my a-priori axioms that I'd like you to consider as the basis for our thinking throughout the rest of the book:

- *Your software's first role is to be useful*
- *The best software is that which is not needed at all*
- *Human comprehension is King*
- *Mechanical Sympathy is Queen*
- *Software is a process of research & development*
- *Software Development is an extremely challenging Intellectual Pursuit*

Axiom 1: Software's first role is to be useful

The effect and outcome of software, it's contribution to desirable value that was not present without the software, is the primary responsibility of good software.

While the process and crafting techniques used to create software is certainly important, software at a minimum must at least exhibit some usefulness to people as an outcome whether they be a business or an individual.

Axiom 2: The best software is that which is not needed at all

This axiom is more controversial, especially given the silo that the majority of contemporary software developers work within.

A software developer should be most primarily concerned with enabling *valuable change*[2] rather than simply focussed on shipping software.

If this axiom is accepted, valuable change becomes a maxim of a software developers thinking as they approach a problem domain where software *might* be applicable.

If this broadness of options is recognised, software is one possible answer to the problem domain but should be placed in a context of other recognised options for meeting the challenges of the domain.

As other axioms here state, developing software is a challenging intellectual pursuit and, even with the enabling factor of adaptability, can result in a complex solution where an alternate solution was possible.

Simply stated, this axiom puts to you that software is one option but by no means the ***only option when enabling valuable change***, which is in our opinion, ironically perhaps given the name, the role of the modern software developer.

[2] http://www.infoq.com/presentations/patterns-software-delivery

Axiom 3: Human Comprehension is *King*

> *"I don't want 'beautiful code' that I can marvel at in wonder of the smartness of the all-powerful creator! Give me instead 'Cartoon Code', something as clean, clear and comprehendible as reading the funnies in a newspaper; but perhaps not quite as funny..."* - Russ Miles

Software is communication primarily between yourself, the original author of your code, and others, the people who will need to be able to change the code.

If your architecture, design and code is not clearly communicated with the aim of maximising a reader's comprehension of your software how can a reader be expected to understand and update your software?

Code comprehension is one of the major forces that can enable, or hinder, a person's ability to confidently adapt your software.

Whole software products have been abandoned, even though at the time they were functional, on the justification that the teams of people involved in working that code ***no longer understand nor are confident enough to change and adapt the existing codebase.***

It is important to choose to ***optimise your code in the first instance in order to maximise the comprehension of others.***

This extends to all aspects of your code, even to your test code. ***Test code is crucially important as its aim is to clearly communicate the intentions behind your code.***

If readers can understand your intention, they will have greater confidence when changing the code and tests as intentions for the software change.

To this end, ***simpler architecture, design and code should focus on maximising human comprehension by effectively documenting intent and minimising cognitive overhead*** in the person struggling to comprehend your software.

Prefer clarity of intent in your code and avoid anything that introduces confusion, such as surprise!

Axiom 4: Mechanical Sympathy is *Queen*

> ***"to get the best out of any car, you have to have a sympathy for how it actually works and then you can work in harmony with it", Jackie Stewart***

Mechanical Sympathy may have its roots in Formula 1 car racing, that high speed processional sport that we brits love, but Martin Thompson has importantly re-introduced this facet of thinking into the code we design and write.

Mechanical Sympathy is a philosophical approach to designing and writing software whereby ***the consumer of your software, in this case the machine, is primarily considered***. The nuances of the underling machine are clearly understood and have a large effect on the code you write as you strive to make the most of what the machine is trying to accomplish. This approach is particular important ***where low latency is crucial*** to the success of a piece of software.

Mechanical sympathy and maximising for human comprehension can seem to be at odds with one another. However to place the two as distinct competing forces can result in a false dichotomy.

As long as human comprehension has been thought about and optimised for then applying the tenets of machine sympathy to compromise some aspect of comprehension where applicable is a useful and appropriate secondary goal.

Axiom 5: Software is a process of *Research & Development*

> ***"We don't know what we're doing…", "You are not a software developer, you make change happen and software is just one tool…"***

We don't know what we're doing and that's ok! Perhaps it might be more accurate to state that we don't know *exactly* what we're aiming for when developing software, especially when embracing the natural change that occurs.

When embarking on building a software solution, arguably even before we decide that any software is needed at all, we embark on a ***journey of research and discovery***. Our research will span everything from the concepts and language used in the problem space we're addressing right through to the best tools, techniques and languages we can employ to produce the change needed to address those problems.

We are researchers and developers and this has wide-ranging impacts on how we manage our work. Recognising the importance of research and discovery in software solution development shifts our thinking from the factory floor (i.e. let's just churn out some more widgets!) to the status of change agents and problem solvers. Software is often our answer, but we are also responsible for exploring, understanding and researching the problem space because, more frequently than not, this is poorly understood even by those who believe they know what is needed.

Axiom 6: Software Development is an extremely challenging Intellectual Pursuit

This might come as a surprise to some but ***software development really doesn't happen when someone is hammering enthusiastically, or not, on a keyboard***. That, as those of us in the trade would say, is just the 'output'. ***The hard part has already been done***.

Software is ***designed in the mind, collaboratively in conversation and on whiteboards*** before it goes anywhere near turning it into characters of text in a program. ***The majority of the effort in software development is in understanding what might be needed, and then turning that into a design that can deal with the test of time***.

It is that test of time, the ultimate stressor on a design, that this book aims to help with.

When developer's are thinking, they are working. When they are typing, they are turning their hard work into something that can be used. Unfortunately thinking is very hard and, when it comes to software and the variability of understanding what people might want, this ***places software development firmly in the camp of the most vital and intellectually challenging jobs of the 21st century***.

What are we really dealing with, then?

This book is not anti-or pro anything other than helping you succeed in designing and deploying better software and doing more with your life. I believe, and have personally first- and second-hand witnessed, benefits from using the techniques discussed in this book where groups of people are tasked with the confusing and ephemeral task of 'building software'.

Too much time is wasted building the wrong thing, or building something that takes herculean levels of effort to keep it stumbling forward like a drunk sumo wrestler up a hill. The key challenges of software development can be distilled into two areas:

- *How do I avoid Over Production?* We're creating too much software, and what we do create is often not valuable.
- ***How do I create and maintain software that adapts as fast as the needs placed upon it?*** We need to create software that meets the needs of ubiquitous and accelerating change.

For the more twitter-friendly audiences, these two challenges can be simplified to:

- Challenge 1: Building the right thing (or not building anything at all!)
- ***Challenge 2: Building the right thing, right***

This book ***focusses on Challenge 2*** and can be thought of as a set of weapons for defeating that particular bully.

Camilla's Bookshop, Eastbourne. My favourite bookshop in the world, featuring the amazing owner Camilla in this photo. A bookshop that is as exciting as the books it holds. Never boring.

A Nod Towards *Over-Production*

Before we move on to the main challenge of this book, there's a little context-setting I'd like to do on the subject of "Building the right thing" as it relates to where the various destinations that the rest of this book intends to take you on a journey to.

Over Production is a Lean Waste[3], and it is on the increase in Software Development across the industry. For our purposes, Over Production can be defined as *building the wrong thing*.

This includes the cases of building the right thing at the wrong time, and of course the wrong thing at the right time. The temporal characteristic of rendering a product the wrong thing is included in the general definition of *building the wrong thing*

As software developers learn to organise and complete their work more efficiently using work management techniques such as Agile processes, the potential for greater productivity is unlocked. At least greater productivity potential is unlocked while the challenges of reacting quickly to change through adaption can be kept under control.

The flip side of this productivity is that it is much more likely to produce the wrong thing. Increased efficiency does not lead to increased effectiveness of what is produced.

[3] http://www.isixsigma.com/dictionary/8-wastes-of-lean/

You wish your software development engine was even half this sexy, or effective.

If viewed as an engine of software production, it could be argued that current Agile and Craftsmanship techniques in the software development industry are having huge impacts in helping that engine fire on all cylinders. However taking the analogy a step further, the engine may be firing on all cylinders but the direction the engine is heading in, and its ability to change course, is an entirely different matter.

Even if you apply the patterns and techniques from this book to help you build the right thing, right, i.e. build adaptable software, that is only an important *enabler* that supports the possibility of building the right thing. It does not guarantee that useful software will be an outcome. For more on techniques that work towards overcoming that problem, take a look in the **Further Reading** chapter at the end of this book.

That said, ***architecting and building software that enables you to build the right thing continuously is no mean feat***, so we begin our journey with the ***biggest barrier to a piece of software being adaptable…

Architecture and the way we think about software.

Book I - The Philosophy of Change in Software

The Elephant in the Standup -> Philosophy & Architecture -> Stressors & Antifragility - Simplicity Principles - Microservices - Natural Selection & Innovation - The Payback

Software Architecture *is* Philosophy

> *What is philosophy in this context - The illusion of perfection - The delusions of architects - (Re)introducing the unknown unknowns and how to deal with them*

What is Software *Architecture* and *Design*?

There are a lot of definitions of what software architecture and design actually *is*; almost as many definitions as there are for what a software architect or designer actually does.

For this book I want to be a lot more concrete on what architecture is and how it affects the software you build, and so here are my working definitions:

- Software Design is Thinking in Context
- Software ***Architecture is Philosophy***...

Software design happens whenever you are thinking, conversing or sketching out a specific solution.

Software architecture is a little more esoteric, and so needs a little more explaining…

What is Philosophy?

There's nothing like using a broad and misunderstood term in order to explain another broad and misunderstood term, but that's what "Software Architecture is Philosophy" can look like at first glance.

My working definition of philosophy is:

- The love of wisdom, technically as per the definition of the word. This is nice but hardly more helpful.
- A set of ***thinking tools, ideas, concepts, experience and biases*** that *you* bring to bear on your solutions. ***Much more helpful to our discussion here.***

When initially considering a piece of software to construct, or a solution to build, or a system to bring to life[4] you bring your own philosophy to bear on the situation. ***You bring your own past, and sometimes painful, experiences and biases.***

You bring your preference for the JVM, because it's familiar. You bring the decisions that ***"worked ok before"***, as well as the ***pain of what didn't for you.*** You bring the ***desire to use the "safe" tools*** that you saw work before, plus the guilty desire to use new tools in order to make the project interesting to you.

You bring your ***friendship with vendors and other professionals*** because you feel safe with them and would like to work with them on this. You bring your ***beliefs, and occasionally evidence, that certain practices work well*** for organising

[4] All of which are essentially the same thing but vary in an order of magnitude in complexity.

people and work, and you're ready to argue and defend them because alternatives are unknown and scary *to you*.

It's all about you. When considering a system of software you're bringing all of this to the problem at once; much of it subconsciously.

You may also bring your honest desire to ***"do better this time"***.

Oh, and you also bring your deep-seated belief in what you know of reality known as Epistemology to the academics. This belief is the strongest because ***you believe you know how this project is going to work***. How it's all going to go well.

Unfortunately it is this belief and underling philosophical perspective that is often fundamentally ***in conflict with the reality of researching and developing software***.

It's this belief that I'm going to attempt to kick around now, hopefully challenging it and realigning it a touch so we're ready for the main theme of this book.

Software Architecture Abides

The philosophy of the modern software architect

A wise man said "Software Architecture is the Big decisions", and promptly left the room for misinterpretation on exactly what those decisions are. My take is that that just doesn't go far enough.

For me, software architecture is the collation of the thoughts, ideas and biases of the individuals involved in decisions that have a huge impact on how the software evolves. It's like setting the rules to a game, but with the danger that those rules are often informed by our own implicit biases. It's those biases I'd like to take a look at here.

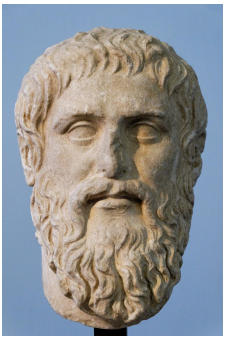
The biases and perspectives that are collectively brought to bear on the 'big decisions' are the philosophy of the individuals coming to those decisions. That may sound a little odd, but if you consider philosophy to be the collective ideas, fears, reasonings and perspectives of those people involved, then philosophy becomes a toolbox for making decisions (rationally or irrationally).

Lots of people these days point the finger squarely at the prob-

lems of big-bang, waterfall development projects of the past.

Often those projects and that approach are a fair and easy target for criticism as they often fail (over budget, not fit for purpose; the list of valid criticisms is well-known in the industry). In the next breath you'll often hear from the agilistas that 'waterfall thinking' is still a problem in organisations that are attempting to be more agile, but what is this problematic 'waterfall thinking'?

Plato and the *Waterfall*



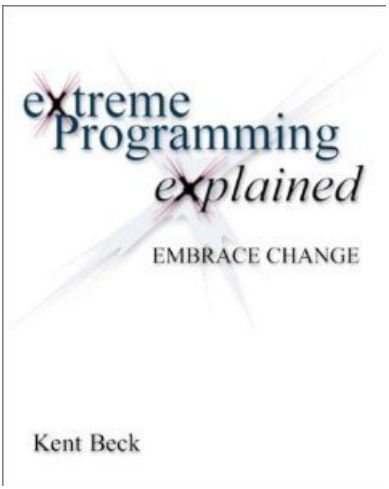
Simply put, the root of the issue is a belief in the perfect form and so essentially I'm blaming Plato[5] for this one. Plato's early philosophy, those works attributed to him whether Socrates is a central character or not, focussed on a philosopher striving towards understanding the perfect form through excellence, or arete[6].

At first glance this seems fairly innocuous until you frame the next two thousand years according to the statement by Alfred North Whitehead that philosophy has been during that time "a series of footnotes to Plato". Plato's thinking permeates western culture with his ideas of perfection, even though he came to regard these same thoughts as a little dubious himself later on in his life.

[5] Plato on Wikipedia.
[6] Arete on Wikipedia.

What does Platonic form and perfection have to do with software and architecture, much less agility? The problem is perfection, and the implication of perfect knowledge.

Waterfall Thinking was based on the belief that perfection could be achieved. The perfect solution to the problem. This would work out fine, but for the nature of our reality. That nature includes two forces against perfect knowledge, time and change.

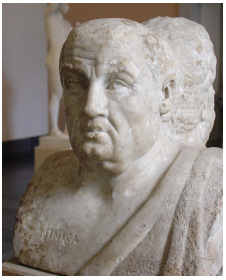


This early Agile software development book nailed it with its subtitle

The problem itself often changes with time, enter stage-left the ideas of Agile software development. In the early days of agility the emphasis was on embracing change. That might have got lost in the confusion of certification, coaching and other nonsense, but in essence agile software development was a big step forward as it dropped the idea of perfection, and more importantly the idea of perfect knowledge.

Agile software development killed Waterfall and Platonic perfection at the same time, and good riddance I say! Agile software development came from an entirely different school of thinking, a school that knew that adapation was the key because we really don't know what is going to happen next. We are involved in research and development, and that requires a completely different philosophy, the philosophy best characterised by the school of Stoicism.

Agility and *Fortuna*



One school bucked against this idea of perfection, embracing reality at its core. That school of philosophy of the Stoics. The stoics looked very differently at the purpose of philosophy, the principle question being "given that we don't know why is going to happen next, how best should we live our life".

No summoning of perfection there. In fact Seneca[7], a prominent and very successful Roman stoic, summoned a goddess to communicate the fickleness and unpredictability of reality. That goddess was ***Fortuna***.

Instead of a dream of perfection with enough planning in Stoicism instead we have an ultimately pragmatic view of how to approach life and the big decisions with the knowledge that we can't really predict much at all. Seneca has earned fame for writing:

[7] Stoicisim through Seneca on Wikipedia.

> *"The next day is not promised you ... nay, the next hour is not promised you!"* - Seneca the Younger

In other words:

> "We can't predict the future."

Or to give things a software delivery emphasis:

> *"We don't now what we're doing and we don't know what we want, but that's ok and normal!"*

Admittedly this can seem rather depressing on the surface of things. At every step, Fortuna is there to either reward us with the cornucopia or hit us with the rudder; either way, we need to deal with it, there's no use complaining!

In Stoicism, and from Eastern philosophy to some degree in Buddhism, we learn to accept and adapt to inevitable change and it's this view that is at the root of agile software development. Perhaps a better term for agile software development would have been *adaptable* software development but unfortunately that's for the history books now.

In agile/adaptive software development we do a lot of work to embrace change in our processes, attempting to deliver while being prepared for the unknown because that is the nature of reality and the intellectual pursuit that is software development.

Fortuna should be the greek god of software development to remind us that we really don't know what's going to happen next. That leads us neatly back to software architecture and those big decisions; what can Fortuna and stoicism offer us there?

Architecture needs to *embrace Fortuna*

Fortuna, greek goddess of adaptable software (research &) development

If change is inevitable, and we're involved in research and development of software solutions, and we're trying to adapt as successfully as possible to that change, and we can't predict the change, then we need to be stoic and do out best to *prepare for change*. That change could come at design-time or runtime for our software, neither is more important than the other.

In the past, architectural decisions had huge weight *because* they were seen as unchanging. Architects were given hallowed prides of place in an organisational structure because of the brittleness of those 'big decisions'. This is waterfall thinking, and needs to stop.

Instead we need to accept stoic thinking into our architectural decisions, prioritising our architectural decisions for change. It's time for our software's architecture and design to catch up the agility that we're beginning to see in our processes and organisations. It's time to drop the possibility of the platonic perfection, and adopt a whole new set of thinking tools and design and implementation approaches that accept that we don't know much when we're asked to create some software.

Software development is a voyage of discovery, Fortuna is there at every step, where first- (we don't know) and second-order (we don't know what we don't know) ignorance is the norm. In this world of imperfection we need to reason differently about our software design and architecture by first considering a new

concept: stressors.