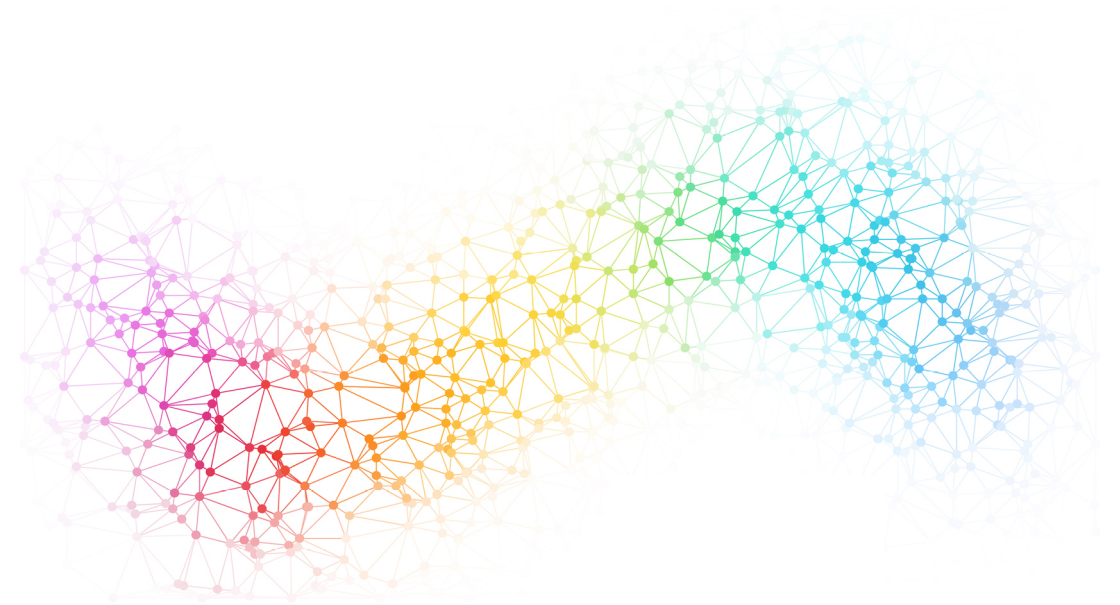
Antifragile Software

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

For Mali

For Ashton

For Siobhán

From Russ: ***I hope your learning journeys never end***

Preface

Grabbing the Code

In this book a very specific approach to providing code samples has been taken; one that makes it as easy as possible for you to try out the concepts being taught as well as give you lots of opportunities to dig deeper, take time to think, and challenge your understanding.

So instead of providing a generic or plain unrealistic sample application, we're tried to use a test-driven approach that is best used as you read the book. Think of it as a 'ready, try, think and learn' approach. Each step we call a 'Koan', but more on that in the first time you experience the Koans in Book 1 in the chapter on Simplicity.

The samples for this book are absolutely critical to your learning experience so please take the time to complete the following instructions to get the code set up on your own development machine.

There are also several ways you can work through the code for this book.

Option 1: Browse the code on GitHub

First and foremost you can simply browse the code online at GitHub[1]. There are a number of examples here that illustrate the lessons and journey in this book.

Each of the code samples can be navigated by clicking on the appropriate GitHub repository for the part of the book you are currently studying.

If you just want to browse the code then Option 1 is probably enough for you. If you want to actually work withe code locally while you read the book, but don't want to go to the hassle of using the Git[2] tool itself, then Option 2 could be for you.

Option 2: Download the Code as Zip Files

If you want the code samples locally then the easiest way is to simply download the ones you want. If you navigate on GitHub[3] you will then see a "Download ZIP" button.

Click on that button and a Zip file of the complete source code for that Koan will be downloaded to your desktop and then you can unzip it to a place of your choosing to work from as you work through the Koans.

[1] https://github.com/orgs/antifragilesoftware

[2] The Git Source Code Management Tool site is available at http://git-scm.com

[3] https://github.com/orgs/antifragilesoftware

About The Author

Russ Miles

Russ Miles

"An expert is someone who has succeeded in making decisions and judgements simpler; through knowing what to pay attention to and what to ignore" – Edward de Bono, "Simplicity", 1998

"Complexity is the silent killer of delivering the right software, or change, at the right time; it is singly responsibly for killing many good ideas and companies. A focus on simplicity is the answer, but simplicity is not easy. Through our techniques and practices, I help software delivery organisations and teams ensure their solutions are as simple as possible while not missing the mark by over-simplifying." – Russ Miles, 2013

Russ Miles is CEO & Founder of Russ Miles & Associates[4] where he works with his clients to help deliver simple and valuable software and change.

Russ' experience covers almost every facet of software delivery having worked across many different domains including Financial Services, Publishing, Defence, Insurance and Search.

Russ helps to change all facets of the software delivery process in order to remove unnecessary and costly complexity in everything from developer skills and practices, through applying the right processes for the job at hand, to ensuring that the right change is delivered, be it through software or otherwise.

[4] http://www.russmiles.com

Passionate about open source software, Russ worked with SpringSource prior to the company's acquisition by VMware, leading the Spring Extensions project and helping international clients to simplify their software by effectively applying the Spring portfolio of projects.

Russ is also the Geek on a Harley[5] and you can follow his travels online.

[5] http://www.geekonaharley.com

Acknowledgements

I'd like to thanks lots of people, I really would! The problem is that basically this was a singular work of a genius mind, constructed on a mountain-top without Internet access and after several decades of contemplative solitude… so who would I thank?

Ok, maybe that solitude was not quite as lonely…

First up a huge thanks to my fiancee and partner-in-crime, Siobhan Gadiot. You're not only my colleague in all things, you're also my muse.

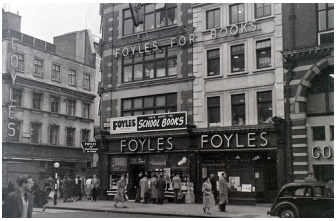
Thanks to my wonderful daughter Mali and son Ashton. I'm biased, I know, but you both really are wonderful[6].

To Mali & Ashton, I hope you never lose your love of books and your wonder with the world, and if you do then just remember how it felt because that's an important feeling to take forth on your journey.

Russ and Mali Miles

[6]…and cheeky, and naughty at times … and going to cause me totally undue happiness and heartache I'm sure! but anyway…

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[7] many times over.

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

[7] 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*[8] 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.

[8] 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[9], 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.

[9] 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

The Parable of The Elephant in the Standup

Fortuna - Change is Necessary - Common Lies in Software - The Enemy: The Elephant

It's the 10th sprint of your project; ***you're feeling good***.

You're surrounded by smiling colleagues who are happy with the software they've built. Happy that they're going to be building software of value. ***Life is good***.

It's at those moments when we find that life is ready at all times to throw us a curve ball or, as Seneca would point out, ***Fortuna*** brings the Rudder as well as the Cornucopia.

The Cornucopia has been the experience of your team for the past sprints, with ***success building on top of success*** as you work your way

through the product backlog. The Rudder symbolises the flip-side, when Fortuna ***rips away your comfortable rug with one un-anticipatable twist of the wrist***.

The rudder is when your product owner utters the one line you've been ***dreading***...

"I'd just like one small change..."

Grief, Anger, Disbelief…

You begin to go through several phases of ***grief***. The first phase of grief is ***anger and disbelief***.

"How dare you ask us to make one small change?!

How dare you think you know whether a change is small or not!?

That's it, I'm going to cancel the sprint…".

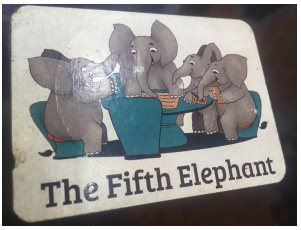
None of this you say out loud, but there's no escaping the fact that your first reaction is total outrage.

...Guilt...

You rapidly proceed to the next phase, which is dominated by the emotion of ***guilt***. Why guilt? Because deep down you know that ***the change being asked for in fact "should" be that small, but it's not***.

Time to meet the enemy...

The Elephant in the Standup



The Elephant in the standup is all assumptions and decisions that led to your code that is now in conflict with reality.

Carefully ***designed and architected for what you knew yesterday***, the elephant is a silent participant in your meeting that is now woefully in conflict with the new change.

And what do we do when we meet the Elephant..?

Everybody *Lies*

The most popular three ‘lies’ are:

“It’s a Big Change, no honestly…”,

“The Extended-Holiday of ‘Refactoring’” and

“Raising the question of ‘Technical Debt”…“

Lie 1: It's a Big Change really...

"Things aren't that simple...

... that one small change you're asking for is really a big change."

Sometimes, these pleas are very true. Not all changes that appear small are in fact small.

This lie is not about those times. It is when we know the change could in fact have been small. Should have been, if only we'd known the change was coming.

But the software was ***not built with change in mind; we thought we were getting it right!*** We couldn't have anticipated this incoming change, and now it is sitting there ***completely in conflict with the software we've put together***.

What should have been a small change could now scupper our entire project, and so we often head to the second and third lies...

Lie 2: The Extended-Holiday of 'Refactoring'

"Umm, we understand that you need that change but in order to accommodate it we're really going to need to do some refactoring, maybe for a sprint or two.

You've heard of the agile practice of refactoring right? Good.

We're going to be doing refactoring for a minimum of a couple of days, but more likely a couple of sprints just so we're ready for the change you've made us aware of.

What happens if you change your mind in the future?? Well yes, we might then have to refactor again, but hopefully only for a couple of days.

The other option, and I say this only as a suggestion, is to maybe make sure you know exactly what you want up front before we get into designing and coding, otherwise we might have to go round this again..."

Refactoring should be *Quick*, *Easy* and *Constant*

Refactoring is ***one small change that does not affect the behaviour of the system***.

Refactoring may affect the structure or some other qualitative factor of the code, ideally ***maximising for human comprehend-ability***.

When practiced regularly and constantly, refactoring is a powerful tool to keep your code clean. It is a ***quick and constant exercise***, not a drawn-out, productivity sapping interlude.

When you say you need to 'refactor' for an extended period of time you are simply not refactoring any more. Refactoring might be the small steps you apply, but it's all part of a bigger activity; an activity we really don't want to name.

You're frightened to mention its name because it sounds like you made a mistake. ***You are in fact doing re-design*** in order to evolve your software so it can accommodate the new change you've encountered that you couldn't anticipate in advance.

Refactoring used to cloak re-design in this way is bad, but nothing like the final lie we use…

Lie 3: Sorry, you're in Technical Debt…

"I'm sorry, have you heard the term ****Technical Debt***? Seems familiar? Well unfortunately that's what you've accrued and, yes, it's probably ***your fault for pushing us too hard to complete the work*** for the previous sprints.

We've just ***not had the time to properly consider all the ramifications*** of what we've built because we were too busy trying to make you happy, and so now it's time to ***suck it up and pay back*** some of that debt that you had no idea you were accruing.

A simple payment of a couple of weeks/months of effort with ***nothing new being delivered*** should do the trick.

No, we can't be sure you won't have some debt left and need to pay that back at an undisclosed point in the future…

What's that you say? It seems unfair? ***Why are you crying…"****

Technical Debt is for real

Technical debt is when you are *deliberately* cutting corners in design and your code to be faster today, aware that you will have to round those corners off in the future when there is more time.

If there is ever more time…

There is little doubt that ***technical debt does accrue as designs deviate under the pressure of project and product delivery***, but this is not really what is happening with the Elephant.

The Elephant in the Standup is *not* Technical Debt, *nor* Refactoring

The Elephant is not a result of cutting corners in your design and code, knowingly or otherwise, it is the ***inevitable result of a well-thought-out design for yesterday coming into contact with the reality of change today***.

Now you've met the lies, it's time to meet the real reason that you're having to use them.

Why do we have the Elephant in the Standup? And how can we beat it?!

In order to beat the Elephant you're going to ***need to think differently*** and that means examining ***how software architecture and design is commonly thought about and how that needs to change***.

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[10] 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

[10] 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.

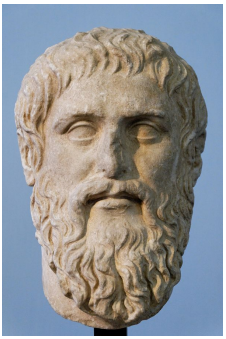
The biases and perspectives that are collectively brought to bear on the 'big decisions' are the philosophy of the individuals coming to those decisions. That may sound a little odd, but if you consider philosophy to be the collective ideas, fears, reasonings and perspectives of those people involved, 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*



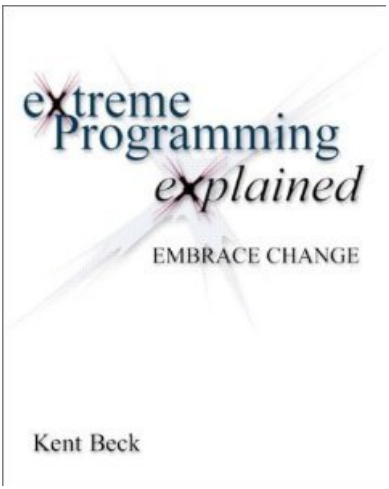
Simply put, the root of the issue is a belief in the perfect form and so essentially I'm blaming Plato[11] for this one. Plato's early philosophy, those works attributed to him whether Socrates is a central character or not, focussed on a philosopher striving towards understanding the perfect form through excellence, or arete[12].

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.

[11] Plato on Wikipedia.
[12] 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.

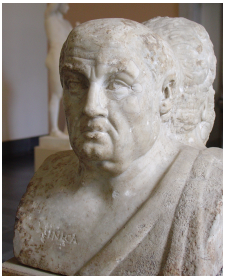


This early Agile software development book nailed it with its subtitle

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[13], 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:

[13] 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.

Stressors on Software

Stressors - Innovation - Wild Success & Wild Failure

A stressor can be thought of as ***any force that pushes your software's limits***, stressing the architecture and design in predicted and unpredictable ways.

The best metaphor here is the one given in Nassim Nicholas Taleb's "Antifragile" book[14], that a stressor is like a weight exercising a muscle. In our case, the muscle is the software system and the weight can come in many forms, including but not limited to changing needs over time.

In Taleb's analogy, a muscle will gain from encountering a certain degree of stress from a stressor as it's designed to improve with it. This is exactly what we'd like from our software: to ***not just embrace change as a stressor but to actually thrive and improve based upon it***.

Of course a muscle will break if the weight is too high, but the cost of having a muscle that can handle any weight is prohibitive on your body's resources. Once again, this is similar to software systems, where ***there will always be stressors that, if they go over a certain bound, may break your system***.

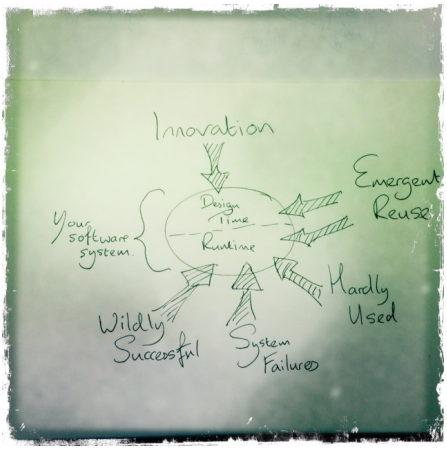
So a stressor is a force on your software system that, at a certain level, may improve the system or, if it goes over a certain level,

[14] http://www.amazon.co.uk/Antifragile-Things-that-Gain-Disorder/dp/0141038225

may break it. As an architect, it's your job to ***categorise the stressors you're designing for*** and start to reason about the limits you'd like to impose.

Change in requirements or functional/non-functional needs is clearly a stressor on your software system's design, but that's often too broad a definition to be particularly useful. So here I'm going to name some stressors that you might consider in you architecture and design work.

This is not an exhaustive list, but it's a reasonable starting point and when considering your own software architecture it's a good idea to explore these in much more detail. Stressors worthy of consideration in software systems arise on two occasions that largely run in parallel throughout the lifetime of a software system: *design-time* and *runtime*.



Some Categories of Stressors in Software

In summary:

- **Innovation** Desirable functional change, for whatever reason
- **System wildly successful!** System needs to be scaled up because it is being used a LOT. The essence of "success shouldn't kill you"
- **System hardly used** System needs to be scaled down as it's not being used a lot.
- **System failures** Nodes coming up or going down, hardware failure, transient network conditions.
- **The emergent need to reuse** Can be understood when you hear the phrase: "Wouldn't it be great if we could reuse functionality X from your system"

NOTE: I deliberately don't make a big deal out of *development* and *maintenance* here. I believe that there is very little difference in modern software development, where releases are expected constantly and frequently, between development and maintenance. It's all development, or more accurately research and development, and the only difference is the amount of legacy software and decisions involved ,and the aim of this book is to help you build a legacy that embraces, or even thrives, on change as much as possible.

Stressors are not inherently *bad* or *good*, they are just reality. That's not to say that a stressor taken to an extreme shouldn't have a damaging effect on your system, it most certainly will.

A classic example of this is when Netflix hit the problem of a whole Amazon cloud region going down. Netflix had done a great job of architecting their system for certain stressors, but that was the one that broke the muscle!

In a world of finite resources, and yes even in the cloud that's the case, it's important to consider stressors while architecting and designing your system so that you can achieve a trade-off between embracing those stressors and acknowledging that some stressors might just break things. Just having considered stressors is a good starting point!

You've seen what happens when a stressor is 'fought', this is in evidence when the category of stressors associated with change are resisted by your process or software design itself and was described in detail in the "Parable of the Elephant in the Standup".

What that parable teaches us is that we need to build an architecture that accepts stressors, possibly even gains from them, and doesn't fight them.

For that challenge we need to coin a new term in software development, although a mature term in economics and other fields: ***Antifragility***.

Antifragility in Software

Adaption - The Triad - It's in the Relationships

"All this shall pass", Heraclitus

"And..."

"All this shall fail..."

"All this shall be learned from..."

"All this shall be better!", Russ Miles

Software systems come under the force of stressors all the time, and the first key to this book's approach is to recognise that and accept it. The message is a little Zen-like in that respect.

Don't fight the stressors, use them to make the system better.

The problem with aiming for the 'perfect' system is that you won't achieve it, and given that stressors are variable there is no perfect system at any given point in time; the system is in flux and will be adjusting to what is best for a given situation.

The fallacy of the perfect system is unfortunately not just a passive philosophy, it's led us to believe that we need to build

software that is able to withstand change, or at best ignore it. These qualities are wrapped up in a type of software system that sounds like something you'd want. Something worth striving for.

"Robust/Resilient Systems, TM"".

How many times have you seen adverts for software frameworks or platforms that promise you robustness? Seemed a no-brainer to just go for it, after all who would *not* want a robust system? What's the alternative? A fragile, pain-in-the-neck system that requires an operations department larger than some districts of London to keep it running?

No, fragility is not what we're necessarily looking for, but then again Robustness is not the opposite of fragility; antifragility is.

Change, Adaption and Anti-fragility

From the previous chapter, change is one key stressors on a software system and so it represents both a problem and an opportunity. We've now got plenty of process techniques that are aiming to help us embrace change.

The new problem we have is that our software techniques are left way behind.

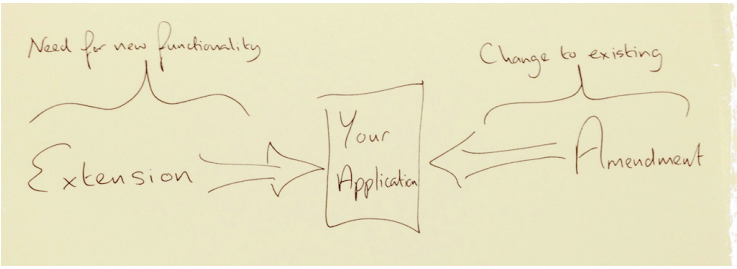
The Change Stressor and the Adaption Response

Change is a force upon your application's structure and, more importantly, on the assumptions you have made to

date that comprise the structure of your software. Change is a ubiquitous and accelerating force that is often sourced from a number of factors including:

- Discovery of the need for new functionality; we'll call this *Extension* of your application.
- A clearer understanding is achieved of the existing, possibly erroneous, functionality; we'll call this *Amendment* of your software.

These two forces on your software solution are shown in the following sketch:



Forces of change on your application

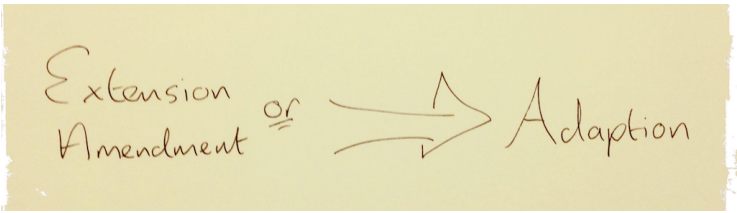
Either of these two types of change potentially has the power to cause significant pain when architecture, design and implementation is approached.

A common anecdote that effectively explains what we mean by change would be:

> ***You experience the true pain of change when someone simply asks "Can you make this small***

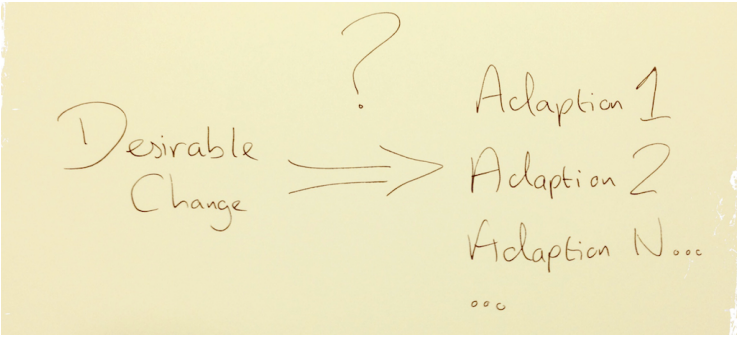
> ***change here..." and you come close to weeping as the answer that is forced out of you is "It's not that simple..."***

On the other side of the equation, *Adaption* is the reaction that is necessary from the thing, in our case your software, that is the subject of the change. There is a causal relationship between change and adaption; where change is the *cause*, adaption is the *effect*:



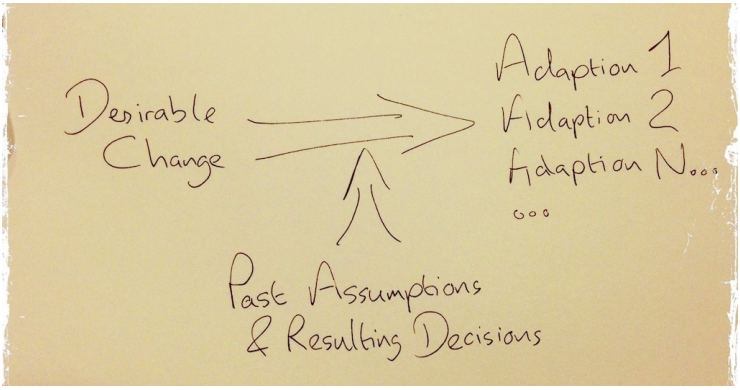
Adaption in Response to Change

There are often many effects that could meet the necessary needs of a cause. The following sketch shows the relationship between a change and various possible adaptions:



Various Types of Adaption may meet the needs of Desirable Change

There is a further force at play that is often forgotten when considering change and adaption in your software, and that is the force of all the assumptions that you've made in your software prior to encountering the new desired change shown on the following sketch:



Various Types of Adaption may meet the needs of Desirable Change

This restrictive force against change is then reinforced if your

code is confusing and incomprehensible, remembering ***Axiom 3: Human Comprehension is King***.

When your collective assumptions about future change are aligned with the incoming Change, and your code and it's intention are humanly comprehendible, the impact of Adaption can be small. When your assumptions are not aligned with future change, the impact of adaption is large and painful and also the range of adaption options can be reduced.

Assumptions on future Change are rarely aligned directly with the actual Adaption that is necessary. What's needed is a better understanding of exactly why and how your prior assumptions come into conflict with Change, and whether you can make more informed decisions with an awareness of inevitable Change that can minimise the impact of Adaption.

Minimising the pain of adaption while recognising that change is inevitable and that some assumptions need to make decisions such that progress can actually be made is what this book is all about. ***Collectively the tools and techniques for embracing adaption lead to Adaptable Software***.

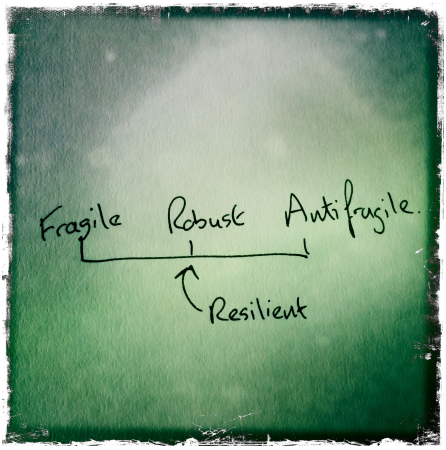
Antifragility

Adaption and Change are hardly restricted as concerns to our wonderful world of software development. Borrowing heavily from Nassim Nicholas Taleb's excellent book "AntiFragile: Things that gain from Disorder"[15], change and the need for software to adapt can be seen as an extremely positive force in software development. The very fact that your software is being

[15] http://www.amazon.co.uk/Antifragile-Things-that-Gain-Disorder/dp/0141038225

subjected to change means it is worth the effort in the first place. The tragedy is if the amount of effort to make that change happen is hugely disproportionate to the change itself.

One way of thinking of adaptable software is to think of it in terms of Taleb's Triad, namely:



The Triad

In Taleb's view, and paraphrased here, ***fragility is characterised by being "breakable" or to be "handle[d] with care"***. In software we can naturally identify the downsides of software built that is fragile to change, typically software without good unit, integration and functional test strategies will suffer from being fragile in the extreme and cause untold headaches to anyone who is attempting to change the code.

Robust is often mistakenly seen as the natural opposite of fragile, characterised as ***"resilient" or "solid"*** and part of Taleb's argument is that simply aiming for robust as the Aristotelean 'golden

middle' is to be missing a trick.

Robustness was valued for a long time in software development and was the lynchpin of the waterfall approach, as opposed to agile software development techniques. However even when applying common techniques such as Test-Driven Development it is still trivial to develop software that is at least robust to change as if it were a 'good thing'. At closer inspection, ***robust software is in fact the very definition of the Elephant in the Standup***.

Antifragility is Taleb's preferred condition for any system that ***needs to exist in a world of uncertainty*** (which is to say, almost every system we know of). ***Antifragility is characterised as truly *thriving*** **on change and other stressors***.

In Antifragile systems, and software is no different, change and other stressors can be seen as a positive factor giving the system an opportunity to improve.

Systems that are fragile will typically collapse in the face of stressors (change being one big group of stressors, especially on software). Robust systems at best ignore stressors, and at worst resist them. Antifragile systems not only embrace the stressors on them, but in fact improve and thrive in the face of those stressors.

What Taleb has very effectively coined as Antifragility we would identify as Adaptability although the terms are only similar. Adaptability implies embracing change, Antifragility goes further and aims to build systems that ***thrive on change and other stressors***.

I'll use Antifragile for the goal of the type of software that

the book aims to help you build; software that truly thrives on encountering stressors such as the inevitability of change.

Defining Antifragile Software Systems

Creating great software that embraces change to build the right thing, right, requires you to apply techniques that help you ***discover what is the best route towards the goal of useful software***, and to create structures in your software that ***do not unnecessarily block your routes to that goal*** by being able to adapt to the ubiquitous and accelerating pace of change.

The best metaphor I've found for understanding an antifragile system is given by Taleb in his book and it likens a system to a muscle. When you go to the gym, you apply weight stress to the muscle and effectively convince the muscle that it had better be prepared for a world where gravity is much stronger. The muscle responds by improving itself into a stronger muscle with possibly aesthetically pleasing side-effects.

This is often an extremely valuable effect to enable in our software systems. Imagine a system that didn't just embrace design-time change but actually thrived and improved on it. Where wild success was a stressor that actually improved the emergent runtime architecture? Where wild failure simply results in the system adjusting things down to a minimum, like a muscle rarely being used taking less and less resources from the rest of the system.

Building antifragile software is about applying techniques that help you manage the necessary change that will be applied to your software going forward, without over-designing your

software for change and reducing your productivity. Turning the challenge of change and stressors into an opportunity by embracing and even thriving on it in your software.

This book captures a selection of techniques to help you build and deploy antifragile software for yourself.

The Need for Antifragility in Software: Enabling Innovation

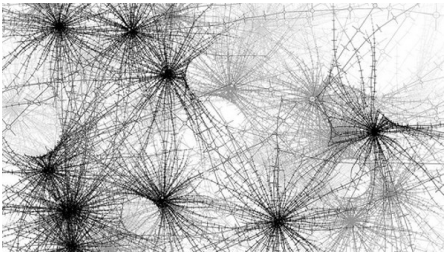
> "40% of companies from 2000 were no longer in the Fortune 500 a decade later in 2010" -CNN

As well as the intimate and personal embarrassment of finding the elephant in the standup places you in a difficult corner, the pace of change is increasing like an out-of-control steam train running down an infinitely high hill.

Contrary to this analogy, this is not a bad thing. It is merely the natural pickup of pace of innovation as IT moves many of its underlying specialised concerns into a commodity scenario. More time is being spent on shipping software that is better and faster, rather than just being able to ship.

Change is a natural and positive force if turned to the needs of real innovation, but only if your software can keep pace with the rapidly escalating delivery speedometer.

Emergence and Antifragility



Systems emerge out of necessity, they facilitate progress, and they're exploited until they become redundant.

In the face of change in particular, it is well-recognised that we need to allow architecture and designs to emerge as the solution-space is explored. Emergence is therefore seen as a key property of agile software development, but there is not a lot of concrete guidance of how to approach software systems in order to support, or even encourage, emergence.

An antifragile system doesn't just allow change to happen, it actively looks for stressors to be introduced as, given they are not beyond its limits, they represent opportunities to improve.

The same is true of emergent design. By considering how we build antifragile software systems we are laying the foundation for a great design to emerge and to continue to emerge in the face of time, change and other stressors.

Without considering antifragility, at best you'll have a system that will ignore the potential of emergence; at worst, you'll have a system that actively resists the possibility of anything emerging.

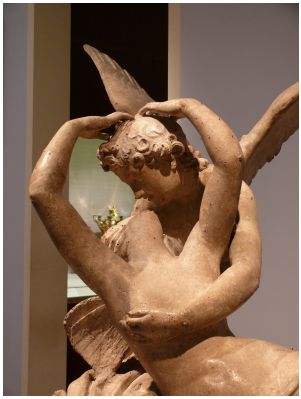
Antifragility is in the Relationships

Parts of a system[16] can be fragile, robust or antifragile depending on their needs and constraints. The important thing is to construct a system that collectively is antifragile and thrives on its stressors. The question then becomes: if the parts could be anywhere on the triad, where does antifragility come from?

The best way to explain where antifragility exists in a system is to look at one of the great mysteries of the human condition: ***Love***.

[16]Here I use the term 'part' to denote an individual component of the system. Something identifiably discrete. Soon I'll be re-characterising these parts as microservices, but for now I just want us to view our software as a system of parts that have relationships with one another as although antifragility is currently simplest implemented using a micro service-based approach, this may not always be the case in the future. We are an industry that loves new terms and new ideas. Antifragility supersedes those technical evolutions by focussing on qualities of a software system that are valuable in the face of change and discovery.

Love is Antifragile, an interlude



Love is Antifragile; Love is a Relationship; Antifragility is in the Relationships

Hang on, are we going to be talking about love in a technical book?! What are we thinking? Please stay with me, all will make more sense in a few more paragraphs…

Love is antifragile. When two people are in love, and that relationship comes under stress, then if it's really love then ***the relationship will not just survive, it will improve***.

That's not to say that *any* stressor can be coped with, ***there'll be situations that will break up even the most in-love couple***.

Love abides through its natural antifragility, even though the individuals involved may be robust, fragile or antifragile in themselves.

Relationships between Parts are the Key

The main focus of antifragility is in the relationship between the parts of the system and this is where we can come back to software. Antifragility is in the relationships between parts of your system. ***It's a property of the connections and what is exchanged between them***.

The parts matter less than the connections.

So as you'll see when we begin to evolve systems towards antifragility we'll be focussing on what parts should be like and ***how the relationships between those parts can be comprehended and codified to thrive based on the stressors the system may encounter***.

Before we begin that journey however there's one more key technology-agnostic property that we need to consider first.

A property that is foundational to antifragile software.

That property is ***Simplicity***.

Simplicity in Software, *Defined*

Objectivity - Koans - Count the Concerns - Continuum - Essential & Accidental

"Simplicity Matters", Barbara Liskov

The single biggest force against change in your software is lack human comprehension. You can't evolve software if you can't comprehend it, this is ***at the heart of the axiom "Human Comprehension is King"***. Key to understanding is clarity, and clarity is helped enormously by reducing cognitive overhead.

We all know what cognitive overhead feels like. It's there, buzzing around and clouding our thoughts when we're trying to read a piece of code that does lots of things, many of which we are only vaguely aware of.

Human beings can handle at most 1 area of focus at one time and whenever that limitation is abused our mind does its best to keep up, usually task-switching like crazy, making the whole effort that much harder and actually causing a feeling of disliking and blockage against working with the code and its surrounding system.

There is one very important tool that can help us with this overhead, that helps us limit the pain our code might cause in

those who have to read and comprehend it. That tool is simplicity and it is at the heart of everything from antifragile software systems to microservices.

Simplicity is *Objective*

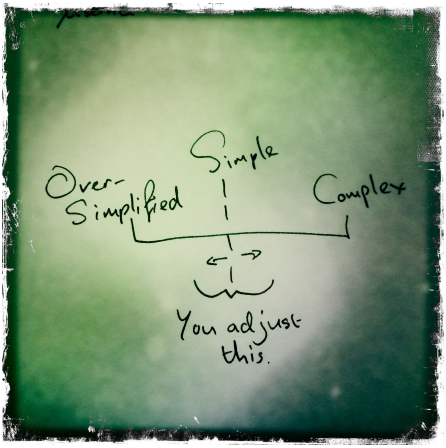
The first thing you need to know about simplicity is that it is objective. This means that we can all recognise when something is simple, and that its not usually possible to argue much about it.

At the point is is important to define what we mean by simplicity when looking at code and software systems. For our purposes I'm going to be using the definition of simplicity kindly codified by Rich Hickey, creator of the Clojure language.

Rich explains that simplicity is defined as the number of concerns *entangled* within a given context. Simplex would mean "one braid", and so one concern present. Complex, or complected as Rich would phrase it, is where one or more concerns are entangled together and need to be considered together, effectively blowing open the doors to cognitive overhead.

The Simplicity Continuum

Simplicity is expressed as a continuum, going from Over-simplified, through some point of optimum Simplicity, before moving into the region of Over-Complex.



The Simplicity Continuum.

The aim for the software developer who values simplicity is to continually move the bar along the continuum from complex towards over-simplified, stopping just short of over-simplification. Over-simplification is effectively described by Ed de Bono in his excellent book "Simplicity" as:

> **"Reduction to the point that any further reduction will remove important value"**

So at all times we'll be looking to simplify our code to the point where it can be made no simpler, i.e. minimize the concerns present and entangled, avoiding reducing so far that our code no longer does what it needs to do. This point will vary depending on the language, framework or even paradigm being applied, but remember that simplicity is objective and so simplicity does offer a measure for comparison regardless of choice of technology.

Essential & *Accidental* Complexity

Some problems are naturally complex. Either because the actual problem is complex and involved lots of factors, or because of constraints brought by choice of language, framework or paradigm, there will be a point at which no further organisation, reduction or hiding can be achieved. I call this the **point of essential complexity**.

- **Essential Complexity** is the complexity inherent in the nature of the problem being solved and has to be accepted. This is quite different from its evil cousin, **accidental complexity**.
- **Accidental Complexity** is everything we add to a given problem, that could be reduced if we gave the design more thought or if we valued simplicity higher.

Why do we introduce accidental complexity?

There are several theories as to why we as developers tend to increase complexity in the solutions that we build. Some focus on the ego of the developer, others focus on the need to for job protection or just knowing no better.

My favourite quote is one attributed to a little-known study by Michael A. Jackson, which is that we have a tendency to create complex solutions to simpler problems because "we are bored by the business problem".

We simply cannot help ourselves, faced with YABA (Yet Another Business Application) we subconsciously will do anything to make the situation interesting to us. This involved choosing complex tools and lanagges to solve problems that could be solved far more simply.

The interesting thing is that this is not a conscious process, and I for one have not encountered any software developer who is not subject to this challenge. This is why **simplicity must be valued and acted upon at all times** when designing and implementing software, because if we let our guard down even for a minute, accidental complexity will prevail.

Simple Architecture & Design?

Moving on from complexity in the code we write, the next level to consider things is the design and architecture of our software.

How do we build software that is not only simple at the component/service level, but also that is simple at the higher-level design and architecture? In addition, at the same time doesn't over-simplify and lose those power properties of embracing change that we sorely need.

The answer lies in a newly coined style of software architecture, ***microservices***.

Microservice-based Architectures for Antifragility & Simplicity

Thriving on Change - Peers & Pipelines - Lego

If only it was all like this...

As the joint forces of simplicity and antifragility descend on our philosophy of software development the options are varied for exactly how these should, or even could, be implemented. One architectural style that is getting a lot of attention, for good reasons, is a microservices-based approach to architecture, and

so it is that approach that we're going to explore fully in this book.

This is not to say that microservices are the only way to produce simple and antifragile software systems. Microservices have certain characteristics that make them ideally simple and at the same time flexible enough to handle the different demands of various stressors that we may choose to optimise for. At the same time, microservices encourages a level of fine granularity that supports evolving parts of a system in relatively isolation, giving a real opportunity to beat the elephant in the standup.

It is my belief that microservices is a transitionary term used to differentiate software architectures that do not optimise for simplicity and antifragility from those that do, and so it's a useful label for those qualities rather than implying that those qualities can *only* be achieved with this architectural approach.

Embracing Change with Microservices

So what are microservices? Like many terms in our crazy industry, it's a blurry statement and open to huge misrepresentation. I've worked now on what are commonly understood to be microservice-based software systems for around 4 years, long before the term moved into the common lexicon.

It is within the framework of stressors that we can begin to define microservices. The key to a system that can embrace change is granularity and modularity of design, code and deployment.

With that in mind, here are the properties that commonly come under the banner of being a microservice:

> Single-purpose, simple (i.e. disentangled), autonomous and separately deployed services organised according to the change stressors on the system and that share nothing, especially state!

That's it. No mention of Source Lines of Code (it's not the 80s after all), and no mention of whether a service is an object or a function. No mention of whether the service is invoked synchronously or asynchronously using HTTP or something else. Those are detailed design decisions and there will be highly context-dependent reasons for choosing one option over another.

A microservice-based architecture is simply a one that contains microservices, with the devil in the details of how those simple microservices interact with one another while still supporting the runtime and design-time stressors. Microservices provides the right level of granularity to support change, but it is in the links between microservices that change needs to the be accommodated.

It is in the links between services where antifragility can be applied.

Embracing or even Thriving on Change in the Links between Microservices

The various design and runtime stressors are prepared for if you build a system of microservices, but the challenge then shifts into the links between services.

For example, if you are going to deal with Wild Success, then microservices allow you to scale at the individual services that

are most challenged by the shape of that success. Depending on how you've designed the incoming and outgoing connections from those services, you may still find that the scaling can be a challenge.

Currently the majority of microservice-based architectures use a simple selection approach to the links between services. The first-line choice is often to use HTTP, possibly RESTful and possibly asynchronous, as HTTP is a wonderful protocol for supporting many of the stressors that can be applied to those links. Second-line choice is usually some sort of messaging approach with a broker involved. Third-line, usually reserved for low-latency options, is to use something more bespoke that might require more effort to achieve the level of de-coupling between microservices to support the stressors being applied.

Embracing change and its stressors is an excellent starting point, and is the aim of designing your system using microservices. Finally we have an architectural approach that looks to embrace change on its own terms.

You can go even further. You could design the links between services such that you can embrace change and for the majority of microservice-based systems that is good enough. It's certainly a lot better than where we were. However if you go beyond that you can end up with a system that not only embraces change but in fact thrives on those stressors. A system that doesn't just adjust but *improves* based on the stressors it encounters. This is where microservices provide the foundation of *antifragile* software systems.

Where this achieved a system provides a fertile landscape for innovation just not possible with a monolith. New languages

and frameworks can be experimented with and applied on a service-by-service basis, reducing the risk of these so-called 'big decisions' to small decisions that use the objective measure of simplicity as guidance.

So microservices are an excellent first step to embracing change, by looking at the links and the system as a whole you can even go as far as enabling antifragility and its powerful benefits as well.

Lies, Damned Lies, and *Lies about Microservices*

Contrary to popular belief, ***I hate the term "microservices"***.

The point has nothing to do with the size of the services at all, and the name itself tends to convey that to the interested adopter. Unfortunately that is not where the confusion stops either.

There are a lot of navel-gazing, authoritative statements being thrown around about the term *microservices*, let's dispell some of the more ridiculous ones.

Here are my personal favourites:

- Microservices have to be REST services - This myth is particularly tenacious as REST is a perfectly good architectural style in software development for when you want to integrate with many heterogenous systems. Microservices *might* employ REST, however it's far from the only way and in fact is quite limiting in what it allows.
- "Microservices are less than X lines of code…" - You pick a value for X here, regardless the idiotic use of SLOC as a

measure of anything important is evidence enough of the ridiculousness of this claim.

- "Microservices should be small" - Smaller than what? Please, we're supposed to be computer scientists…
- "Microservices are Cloud Native" - Cloud Native is a term coined by Netflix for how they wanted to establish their own systems to completely run in the AWS public cloud. It captures that use case perfectly, and microservices is an architectural style that underpins it. However microservices are not only useful in these environments.
- "Microservices can be re-written in a day!" - or hour, or minutes, or … whatever. Yes, the microservice is likely simple enough that it might be re-written quickly, but there's not hard and fast rule!

If there is so much misunderstanding out there, what are microservices all about then really? ***What's the payback?***

The Payback: Speed of Innovation

> *Innovation - Forces Against Innovation - Technology Innovation Stressors - Optionality - Natural Selection*

> ***"40% of Fortune 100 companies from 2000 were no longer there in 2010" or "Innovate or DIE!"*** - Russ Miles

Building systems of microservices can be a challenge, especially when attempting to make a system that exhibits the advantageous qualities of antifragility.

I wouldn't be even suggesting you consider valuing these things if there wasn't tangible and clear payback from that effort. In this case, the playback is *essential* to modern businesses, and comes in the form of the slightly blurry concept of ***innovation***.

What is *innovation*?

It is often easiest to describe innovation in contrast to another term that is often confused with it, i.e. invention.

Invention is the coming up with a ***new idea and proving it can be done***. Once an idea has been proved, invention is over and

the part that is often harder begins; turning that invention into innovation.

Innovation is taking the proof that an invention can work, and making it work at scale. *At scale* usually means others can apply it and get value from doing so.

Modern businesses value innovation because it is where the real money often is. While invention is recognised as important, especially from a legal and intellectual property perspective, it is ***turning that invention into an innovation that can be monetised*** that is where the real interest is for a business.

Enabling Innovation

It may seem an odd starting point, but let me begin this section by introducing a true gent, by father, Laurence Miles:

Laurie Miles, otherwise known as "Dad"

Laurie Miles was a recognised expert at running successful businesses. He was also a care salesman, but don't hold that against him!

One of the keys to Laurie's success was that he could, time and again, get the best out of people. He could take individuals that were, at best, performing ok or, at worst, grossly under performing and give them a platform to make a positive impact to the business.

When asked how Laurie did this, the answer was straightforward: by enabling innovation through an environment that was open to it. Laurie knew that ideas and innovation wouldn't come from any special 'innovation group', even less so the actual Directors of the company.

Laurie knew that the best innovations, the ones that would have great impacts to the business, came from harnessing the creativity of the individuals in the business. By removing blocks to that creativity and giving just enough guidance so that they could experiment in safety with their potential innovations.

And as you can see below, Laurie Miles enjoyed a high degree of success with this simple approach.

Success is a good hot tub

The question is: How does a business enable innovation so that it can join the 'hot-tub' elite in success? This book isn't a book on how to foster creativity in your people, but it is a book that intends to enable software that can embrace change and stressors that are important to your systems, and so it is natural to consider the stressors that occur when innovation is valued and how we can thrive on them.

Technical Innovation Stressors

Technology on its own cannot spark innovation, but it can be a force that supports or resists it.

As it happens, innovation is adversely affected by technology for many of the same reasons that change itself can be resisted. When attempting to innovate you want technology that encourages your ideas and attempts, and then supports taking those ideas that are most successful into full innovations that provide real business value.

On the Origin of Services...

The keys to technology support of innovation can be summarised as:

TBD Image of keys. i.e. (Experimentation, Mutation), (Natural Selection, Optionality, Competitiveness), Granularity

The first key to enabling innovation is to enable *experimentation*. Experimentation through languages, frameworks, functionality. If people are going to try new things, they need to the freedom to *try*. Remember, software development is *Research &* Development, experimentation is critical to research.

One ***barrier to experimentation is the size of the change necessary*** to be tested throughout your experimentation. If you are working with monoliths of software that take days, weeks, months or even years to replace then the ability to experiment with better becomes, at best, a board-level, big-budget decision. At worst it is just never contemplated and is left as an unchanging millstone around your company's neck.

This leads us to the second underpinning of innovation that technology can support, which is *Granularity*. ***Technology needs to enable small experiments that can be attempted quickly in a controlled and safe way***.

The freedom to try at an appropriate level of granularity leads to the freedom to *select*. The technology needs to support the selection of the best *option* for a given solution. The more choice you have, the greater flexibility you have in selecting the best parts to build a better system.

Innovation requires a system to ***support the freedom to exper-***

iment, the ***granularity to support highly flexible change***, and the ***ability to then select the best*** of the available options.

These characteristics exactly underpin an antifragile microservices-based architecture.

Antifragile Systems of Microservices are an Engine for Innovation

A microservices-based system provides a level of granularity, small, single-purpose reusable services, that supports the granularity and experimentation underpinnings of invocation.

Given simple microserives then the re-implementation of a microservice can often happen in as little time as an hour to demonstrate a new idea. This ***supports experimentation with new languages and frameworks as well as new business functionality*** that was just not possible in a monolithic view of the world where the choice of implementation was a *big decision.*

The move then towards ***selection of a new innovation for production*** is often just a simple step of taking a new collection of microservices and making that selectable as part of the production mix. Once again, it is the granularity of microservices, along with the antifragile properties of the links between services, that make this possible.

There are a lot of other benefits to applying microservices and antifragility that are being rapidly discovered by a number of businesses already, including being able to support larger collections of teams through the granularity of the microservices

real-estate. But the ***biggest bang for your buck so far comes from reducing the negative frictions on the stressors of change that come from the need to innovate***.

Innovation, critical to your business, is supported best by microservices and antifragility.

That's the key payback.

It's now time to show how that's achieved in ***real code*** in the remainder of this book.

Remember & Apply

I've always found it ironic that we often call the last section of presentations and papers and books a 'Summary'. It feels a little stale, and really what you want is to do a lot more than just recap what's been discussed. One day I'd like to do a study with the hypothesis that the majority of people's brains switch off at the very mention of the word 'Summary'. Any takers from the research community?

Rather than buy into the trend and call this section of Book 1 "Summary", I thought I'd turn it into something far more *active*. I didn't want to simply leave a section that could be ignored, I wanted to emphasise that this is where the nuggets of importance can also be found, not just a fluffy ending to the first book.

In a nutshell, here's what you need to know to begin your journey through Book 2:

- There are 6 axioms that are crucial to modern software development.
- Modern software systems need to embrace change and all sorts of stressors that were not there in such magnitude in the past.
- Antifragility is a key system property that helps us embrace and even thrive on those stressors.
- Simplicity is crucial to human comprehension, and is therefore a foundational underpinning to handling design-time stressors.

- Microservices are one architectural style that supports both antifragility and simplicity.
- Microservices and Antifragility support Research & Development and the accompanying stressors of Innovation that are critical to your business beating the competition.

Agile software development requires your software architecture and design to be agile too, that's what we are enabling here. Take a moment to rest after Book 1 if you're reading this from front-to-back. Book 1 contains a lot of important buildup, and it's important that you have that mastered before you dive into building new, or converting old, systems towards microservices and antifragility.

Ready? Rested? Then let's get going…

Book II - Architecture, Design & Implementation for Antifragility

Skin in the Game

Approach to Implementation - Antifragile Ethics - Show & Share - Rules of Honesty and Being Critical

I'm sitting on the benches outside the Pivotal[17] offices in London enjoying the early morning sun before presenting a course. I'm pondering how to honestly and authentically write a book on antifragile software. Specifically how do I share my knowledge of the different characteristic patterns of these systems with readers of this book without sounding like a ***high-level, ivory-tower architect fraud***.

This is more than just an academic moment of authorial doubt, this is a serious concern! I'm writing a book that features antifragility as a key theme and one thing that Nassim Nicholas Taleb hammers home in his essays on ethics in the Antifragile book[18] is that I should avoid, at all times, being a fraud.

I should not just talk about it but ***live it; walk the walk***. Most importantly, I shoud have ***skin in the game***.

So how the heck do I do that?

[17] https://pivotal.io

[18] http://www.amazon.co.uk/Antifragile-Things-that-Gain-Disorder/dp/0141038225

Ethics: Do It, Risk It

> ***"Thou shalt not have antifragility, at the expense of the fragility of others"*** - *Taleb, Antifragile*

The intention of this book is to present ***a new way of looking at software systems*** where they can ***embrace, or even thrive, on the key stressors that they naturally encounter*** as we conduct this strange activity of ***software research and development***. ***Learn and thrive on these challenges*** as opposed to crumbling to pieces as most systems would do currently.

The opportunity for fraudulent claims when introducing new ideas and concepts is as ever-present as it always is in our industry.

This book is intended to be a big step forward and it requires the reader to go on quite a journey! I *could* present those concepts in a *"high-faluting"*, *"wink-wink"*, *"I know, trust me"*, *"do it and I promise it will work"* way with patterns, pictures, diagrams and stories alone. This is sadly the way of a lot of new-concept-introduction and architectural books in software today.

My ethics are a little different. ***I don't want to recommend***

anything that I haven't personally taken a risk on. I don't want others trying out the ideas that I've stated authoritatively but not actually executed or applied myself, using their ***experiences and inherent fragility to bolster up my own ideas***.

When I recommend something, it's for real and you can see it and I've risked my own projects and professional reputation on it.

That's why I'm approaching this architecture, design and implementation section of the book on antifragile software differently than you might usually expect. I'm going to share the ***successes, pains and epiphanies of building a real world, commercial project as I build it with you***.

I'm going to do this for real with my own skin in the game before you even consider placing yours on the line.

Show, Share, Make Mistakes, Learn and Improve

At every step I'm going to share with you my own antifragile thinking as the product architecture, design and implementation is rapidly iterated upon.

I'm going to share with you how I deal with the various ***challenges that occur***, ***mistakes I make*** and ***lessons we learn together***. I'm going to show you the ***patterns and implementation choices*** I make along the way, and share ***why I made those choices***.

I'll also be taking advantage of the fact that this is a LeanPub book to keep things up-to-date going forward. Montaigne continued to extend and adapt his great work, "Essais", until he died[19]; it was a living embodiment of knowledge. I'm aiming for this book to be very similar in intention, we could even call it ***"essays in the architecture and design of antifragile software systems"*** and this would accurately meet the definition of an *essay*, that of being an *attempt*.

By the end of this book you'll know all the ***awkward and painful truths of building a real-world product in an antifragile manner***, unedited and there ***to be learned from***.

[19] And in fact beyond with several editions collected and published posthumously.

On Being Critical

> ***"Never complain unless you are taking risks doing it.***. - Taleb, Facebook[20]

Honesty demands that I will share the patterns, designs and tools that facilitate the development of this system. However this ***isn't limited to just the ones that helped***. I will likely criticise tools, techniques, patterns that conflict with the goals of an antifragile software system and I'll ***take the risk of making that criticism public here***.

As Nassim Nicholas Taleb also points out, ***"If you see fraud and do not say fraud, you are a fraud"***. Through an honest discursive on how I build a real-world, antifragile, commercial software product, and sharing all my dirty laundry throughout the journey as I go, the aim is for you my dear reader to get a better learning journey and for me to stay true to my antifragile ethics of maintaining ***skin in the game***.

It's time to ***get that journey started now***.

[20] https://www.facebook.com/pages/Nassim-Nicholas-Taleb/13012333374

Fear, Loathing & *Promise* in Uncertainty-Ville

> ***"Just keep swimming, just keep swimming..."*** -
> *Dory, Finding Nemo*

I don't know what I'm doing.

That's the thought that haunts me.

I really don't know what I'm doing!

This isn't a game. I'm not just playing with code in a sandbox somewhere, I'm going to do a project and it's going to be in the public domain and ***I've got bsolutely no plan!***

This is very real, visceral, tangible ***panic***.

It's the beginning of the project and I'm trying to practice what I preach. I'm approaching the *possibility* of a new software product to be built for this book, and I haven't figured out yet what it is supposed to do, and this is plaguing me.

I don't have a detailed spec; I don't have an understanding of who my user is… heck, I don't even know if I'll have any users!

I just have an inkling that there is ***something needed that could be built that might have an impact*** in this ever-more-jaded world, and I'm willing to chase that down for the purposes of building something useful for this book.

Something that can show what antifragility is supposed to gain us software research and developers.

And then it hits me.

Of course I don't know what I'm doing!!! That's the point!

We never do.

We don't know what we're doing, and the user doesn't know what they want… yet. That should be every software research and developer's mantra:

"We don't know what we're doing, and that's situation-normal!"

The key is to keep the options open. To, as Chris Matts and the Real Options[21] crew would say, *"Realise that you have options, and that those options can expire"*. To *"make my decisions at the most responsibile moment, not at the last responsible one"*.

My options right now look either pretty open, or completely closed depending on your point of view. I don't *have* anyting. So where do I start?

Start with an idea, or better still, in software, start with a ***question***.

[21] http://www.infoq.com/articles/real-options-enhance-agility

Idea 1: *An Open Cloud Index*

Wouldn't it be nice when deciding where to run your applications or services you could go to a single source and ask something like ***"Given I have these things to run, where should I put them today to get the best return?"***.

In a world of ***limited (computing) resources***, and where ***cost and access to those resources can be measured in minutes***, this question is becoming rapidly more important. But as far as I'm aware there's nowhere to get a great answer.

So an idea starts with a need, or more accurately a question in this case.

Can I build a simple way of trawling the various cloud options to come up with a simple, but hopefully useful, answer to the question "Where should I run my stuff right now?".

Let's call it the ***Open Cloud Index***...

... mainly because I own the *.com* url ;)

Phone, Or Have a Drink with, A Friend

Gawain

The waning sun tries valiantly to break through the trees of Clapham Common while being fought off by the *"British Summer"* weather and the fact that it is 9pm and the horizon is trying to stake its claim to it being night now.

In the garden of the Windmill pub, keeping a careful eye on my motorbike, I sit with an old friend, Gawain, sharing my idea and, more importantly, my worries.

I just don't know where to start. How do you get going when you don't know what to build?

One of the reasons I meet up with Gawain is that, frankly, the man is hilarious. Things happen in Gawain-land that don't really happen to anyone else I know, such as being gamely hunted by wild dogs while in India, and so meeting up with Gawain is usually a great way to shirk off any worries of the day and just listen to him reminisce on some event or other that will leave you baffled, possibly shocked and, at a minimum, laughing.

The man is also, frankly, a technical genius. He's humble enough not to know that too, which helps! Which is why I'm confessing to not knowing where to start.

Gawain's answer is perfect. He recommends I ***start with what I do know***, the language of the problem domain and then ***explore what I'd like to possibly know about that domain***.

He recommends I start with the ***Events***.

Ubiquitous Language & Domain Objects?

Start with Events?! It's fair to say I'm confused, ***this is not how I've designed software in the past***.

To get to know a new problem or domain it's usually best to ***establish some common lexicon*** to provide the foundation to discuss the domain with technical and non-technical people.

In Domain Driven Design, this lexicon is called the ***ubiquitous langugage***.

The ubiquitous language of a domain ***describes the key concepts*** that non-technical stakeholders can recognise, and hopefully then provides the ***foundation of what is technically implemented*** with as little translation between the two as possible.

In the past I've used the ubiquitous language to establish a set of classes called ***Domain Objects*** that I've then used *everywhere* in my system because, well, they're ubiquitous, right? This felt in the past like the application of a ***best practice***.

My working definition of a best practice is that it is ***"what everyone is doing, it's just right, so just do it"***.

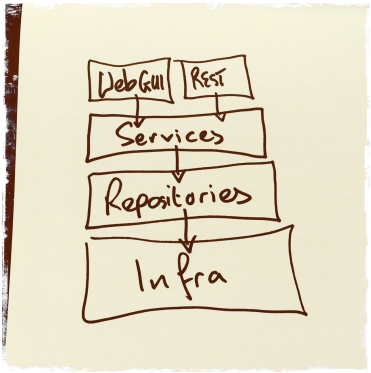
Turns out my use of domain objects was ***dead wrong***. Worse, it would likely have had Fred Brooks in fits of laughter.

The Ills of the *Layered Architecture*

The problem starts with the *layered architecture*, which is just ***an approach to visualising a software system***.

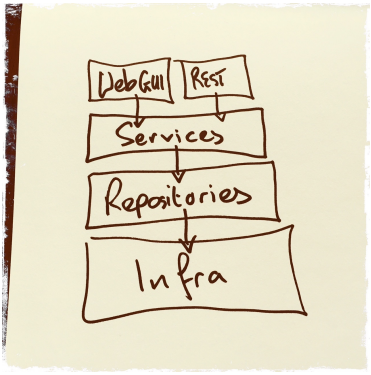
In a typical layered architecture sketch, the system is broken down into pizza boxes that are stacked one on top of the other to attempt to group together interfaces to the outside world, services where business logic resides, repositories or other data access components that are responsible for persisting and retrieving data and an ugly box at the bottom that reflects all the infrastructure.

Finally we join the whole lot with some nice, straight arrows, because we're architects and we like the order these straight lines convey.



Layered Architecture Pizza Boxes

Then we get a little nervous, realising that many of the other areas have some dependencies o nthe infrastrucrure of the system, so in order to continue to pretend that there's some order we introduce some dotted lines (dotted means not important, right..?) to show this.



Layered Architecture Pizza Boxes and Dotted (unimportant, honest...) arrows

How do we judge if this is a ***good and reasonable diagram***?

My own rules for what makes a good diagram are:

- It must help me ***answer some great questions***
- It must help me ***ask some great questions***

Unfortunately the layered architectural ***scores poorly on both these counts***.

It ***doesn't tell me much about the system***, other than there *might* be layers in it, and it certainly ***doesn't ask any useful questions***.

To make matters worse, the ***layered architecture rarely actually resembles the underlying software in any useful sense***. It tends to be entirely separated from reality and therefore any answers or questions it asks should be seen as highly questionable to begin with.

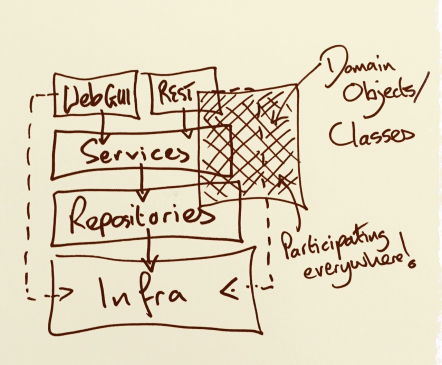
Unfortunately the layered architecture sketch is tenacious in our industry. The layered architecture is a favourite of ivory tower architects and software trainers that need something that will fit on a powerpoint slide.

Given those criticisms of the layered architecture sketch alone,I resolve to do better than the layered architecture for any visualisations used in this book.

But in fact the layered architecture sketch is ***a lot worse than just not being very useful***; ***it's dangerous***. ***A force for evil in software architecture and design*** because it leads us to a specific anti-pattern in software design…

The *Domain Objects at the Centre of the Universe* Anti-Pattern

There's a box missing from the layered architecture diagram that usually gets added. This is the wonder of ***crippling tight-coupling that is the Domain Objects***.



Domain Objects Everywhere

It's the domain objects that cause much of the problems that software modularity is supposed to solve. They are a ***force for brittleness within the application architecture***.

What happens is that, initially, the domain objects are simply

implementations of the ubiquitous language and used to communicate between the levels of the design. All fine so far.

All fine until ***they are used for the specific and orthogonal purposes of each layer***.

The Repository layer attempts to turn the domain objects into storable entities, often using something like an Object-Relational Mapping system.

The Services have their own pressures and want the domain objects to be amenable to those, but the Repository layer will often force a design compromise in order to effectively store the data in the objects with an entirely different relational schema in the database.

Then top-level integration layers will then add their own pressures to the mix.

I've seen domain objects shoe-horned into acting as Form Backing Objects (FBOs) for web layer forms as well as being used to represent payloads where they are marshalled to and from XML, JSON, PDFs … you name it!

Suddenly these poor ***domain objects are being used, and abused, by everything***! Rather than being the point of clarity and comprehension that the implementation of the ubiquitous language should be, the domain objects are a hideous franken-

classes that try to be all things to all layers.

How your Domain Objects begin to look

But the impediment to comprehension is only the start of the problems for these highly promiscuous, compromising and crowd-pleasing domain objects; we haven't even touched on the biggest problem!

Noone wants to *change the Domain Objects*...

Rapidly the *Domain Objects at the centre of the universe* become a source of contention in your design. This is best evidenced by the presence of the ***40 file commit anti-pattern***.

Have you ever changed a domain object's class, just a little, and your IDE runs through a refactoring where you end up with ~40 files to commit to your version control system? You'd be forgiven for thinking ***"Wow, look how productive I am"*** as each change leads to an ever-larger commit log.

The IDE is catching your changes as you go, and the great refactoring tools in your IDE are helping you edit those domain objects too.

Unfortunately those great refactoring tools are, rather than just helping, trying to tell you that ***there's a real problem afoot***. Every change that results in many-file commits is quietly suggesting that ***your design has got a nasty secret***.

Fred Brooks would have recognised it as the ***Ripple Effect***.

Promiscous and Incorrect **Domain Objects and the** ***Ripple Effect***

In this design your ***domain objects are everwhere***. They have more roles than Madonna has changes of clothes at a concert.

Domain Objects are being entirely used and abused.

On top of all these conflicting pressures, domain objects are ***also supposed to be representative of the ubiquitous language*** of your problem domain.

No wonder domain objects end up being a ***frightful compromise of all these pressures*** and having very little left in common with the actual problem domain's definition. It is the ***ubiquitous language that is most often compromised*** as these domain objects become more and more promiscuous.

The ripple effect is simply reminding us of that promiscuity.

But if the ubiquitous language is so poorly represented by domain objects, and the layered architecture so brittle a result, what alternative do we have?

Not the *Things*, the *Things that happen*

Time to turn back to the *wisdom of Gawain.*

It turns out that when discussing a system with non-technical stakeholders there is often an ***entirely different emphasis to the dialogue***. They don't discuss the pieces, or things, involved in a given domain. Non-technical stakeholders often revert to talking about ***what has to happen***.

"This happens, then this happens, then this is the final result."

Non-technical stakeholders aren't talking about objects, or even components and services! They are tallking about ***things that happen and their results***.

They are talking about ***Events***.

***Events*, a Better Ubiquitous Language**

When I have a conversation with a non-technical stakeholder I could attempt to talk about what services I'm going to create, what architectural structures are going to be present and how it will all knit together into a platonic perfection[22] but they are rarely interested in that unless they are quite technical themselves.

Instead the conversation veers on to ***what the system will do***. What is the behaviour? ***What behaviour in their users will be enabled?***

As that conversation drills down into more and more detail the dialogue refines to:

"This happens, then that happens".

These are not structures, services or otherwise. They are describing ***events***. When talking to non-technical stakeholders, it is the ***events that tend to arise first and foremost as the ubiquitous language***.

The problem is that *event* as a term means lots of things to different people. In the context of our software, what are the properties of these fundamental building blocks that harmoniously capture the ubiquitous lanaguge of our domain?

What actually are ***events?***

[22]... for today...

Past, Unchangeable *Facts*

Events are ***things that happen in the past***.

Events are named as Facts.

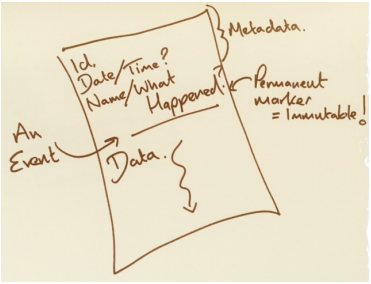
> *For example:* ***User Log-in Requested***, *rather than* ***UserLoginRequest***.

Events are Unchangeable, ***Immutable***.

Events have an ***identity***, ***often a time/date***, and possibly a sequence number.

Events are the minimal set of information about a datum; ***events are metadata.***

There are events that are intended to change something, these are called ***Commands***. There are events that ask a question, these are called ***Queries***.



Events are the foundational building blocks of comprehension of a system and of microservices-based architectures.

Understanding Events - 90% Accountant, 10% Rock Star

Nicholas Nassim Taleb suggests that to practice antifragility you should be ***90% Accountant, 10% Rock Star***[23]. This might be meant a little-tongue-in-cheek, but there is real sense here.

As it happens it is accountants that understand the important of events best!

Accountants understand events because their entire business is based on them. They record events, account entries ideally double-entried, and everything else is a projection across those events.

They wouldn't dream of going back and changing those events (immutable)! But they will issue new events to establish corrections.

Real life is a sequence of immutable events, so why shouldn't our software gain from mirroring it!

That turns our attention to the ***sequences of events*** themselves; what are they and how do they manifest in our antifragile software systems?

[23] http://knowledge.wharton.upenn.edu/article/nassim-nicholas-taleb-on-accepting-uncertainty-embracing-volatility/

Event *Streams*

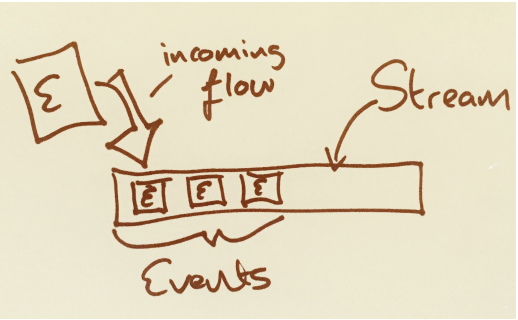
I am now watching the great British summer really unfold; in other words, ***it's raining again***.

As I cover up my motorbike I glance down at the puddles and small streams forming in the pathways to our house and another philosopher comes to mind, Thales.

Thales believed that water was the first principle[24]. All things were changing

When things happen they are passing as well as unchangeable. ***Events need a communication medium***.

That medium is streams that are ordered sequences of events, and they are just as important as the events themselves.



[24] https://en.wikipedia.org/wiki/Thales#Water_as_a_first_principle

As a child[25] there was nothing I loved more than taking control of the forces of nature while on a beach. I would happily dig rivulets and streams around medieval castles in the sand, watching the reversing tide water run back towards the receding sea.

Across those streams I would create moats, meanders, anything I wanted. I could do anything with the right streams.

Once I have some events I can ***create streams that start to give me the real power*** I'm looking for.

[25] Ok, I still do this with my own kids actually

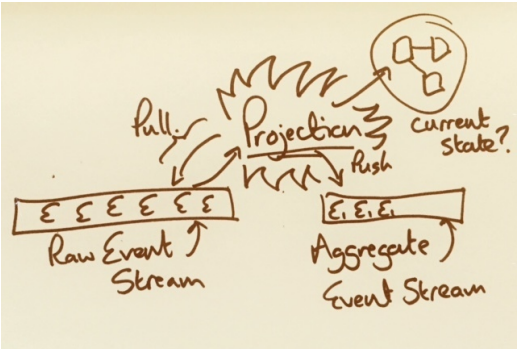
Comprehending Event Streams with *Projections & Aggregates*

Once I have a stream of events, or facts as I like to think of them, either from an event store or in a live stream then it's time to work those events into something we can use.

The next stage is to ***process, or summarise, the raw events into something that gives me a more useful perspective on the world*** that is a derivative of the raw events. At this point, I'm really only interested in the ***command events***, the ones that communicate that something has changed, rather than any query events as I don't have any means of answering a question yet!

If I want to know the current average price of a particular cloud option, say the average spot price for a small instance in the past week in the various regions, then I want a stream of events that are the changing prices of those options and I can then apply an averaging projection as shown in the following sketch.

The output of a projection across the raw events is an Aggregate.


Pull
Projection
Push
current state?
Raw Event Stream
Aggregate Event Stream

An Aggregate is some ***current picture based on the events observed*** in one or more streams.

Often an aggregate is the ***result of some processng, or projection, applied to an event stream***.

An aggregate is ***optimised for update***; it must consume the events in the streams it is interest in as quickly as possible to ensure it is as up-to-date as possible.

However an aggregate is an intermediate step. ***It still doesn't give me an answer***.

An aggregate often simply produces another stream of more refined events!

Microservices can be aggregates and projections, especially if you need those aggregates and projections to evolve at their own pace.

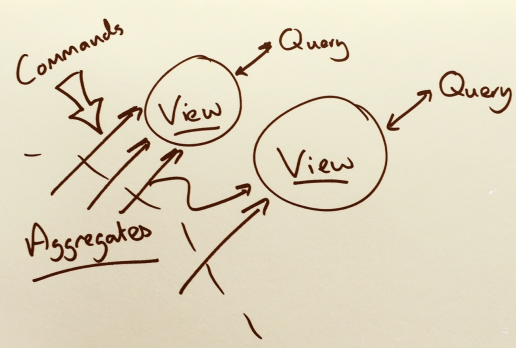
That's all well and good, but when am I going to be able to get a useful answer our of this flow of events?

Answering the Question* with *Views

Views turn your aggregates into an answer.

In contrast to an aggregate, which is optimised to achieve a particular view on the state based on the raw command events, ***views are optimised for a very specific answer that will be in response to the other key class of events: Queries.***

Typically a ***view will only store the current version of that answer according to the events it has seen*** from one or more aggregates. A view might be ***listening to events from a number of aggregate views*** in order to figure out its answer.



If you have a popular view chances are you will scale it out and, since its ***state is simply a sum of the events it has seen similar to an aggregate***, you can ***spin up a new view instance to handle load by playing in those events***.

If you need new answers to your questions, you'll simply create a new view.

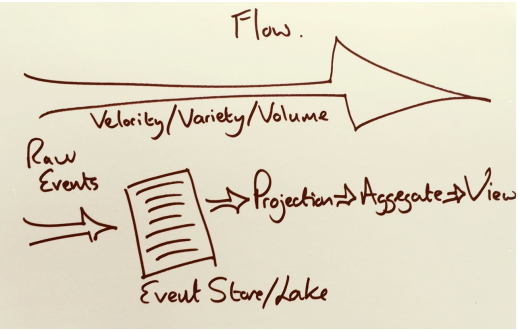
Views are microservices if they need to evolve, come-and-go at their own rates as new functionality is tested and accepted within the system.

Fast* and *Slow Flows

There is now the basis for a ***flow through the system***.

Flows start with new command events coming into the system and they are often channeled to an ***Event Lake*** which is a collection of systems that contain, at a miminum, an ordered sequence of all the raw command events, called the ***Event Store***.

From the Event Store, projections can do their work to produce aggregates, before aggregates can then produce the secondary command events that are useful for views to maintain their answers. Views then service the incoming queries of the system.

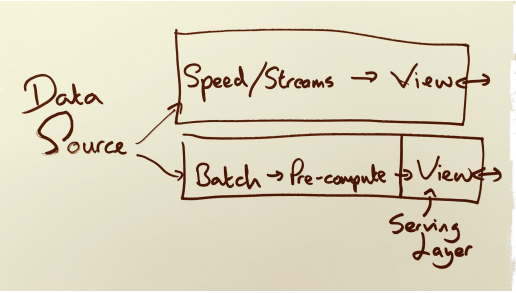


Depending on the velocity needed of that answer, and the volume and variability of the events necessary for a projection to produce it, the system will be made up of fast and slow, sometimes called hot and cold, event flows.

All this is starting to look remarkably similar to…

Lambdas!?

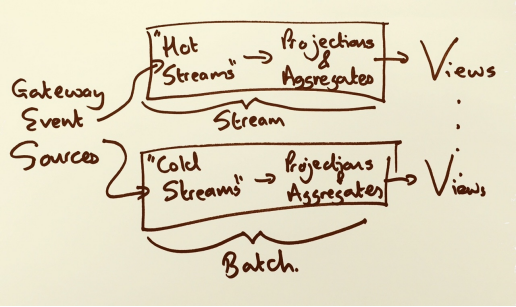
The Lambda Architecture appeared in the excellent "Big Data: Principles and best practices of scalable realtime data systems"[26] by Nathan Marz and James Warren. It's main aim is to provide an architecture to support both streams and batch data processing applications, but that's not where its impact necessarily needs to stop!



Two main areas are identified, a Batch or Slow-data area with pre-computed views that are then used to produce a Serviing Layer with actual views for querying, and a Speed or Fast-Data area that provides direct flows to Views.

When we overlay this architecture with the patterns in this chapter, we can start to see that there's a lot of commonality between the two.

[26] http://www.amazon.co.uk/Big-Data-Principles-practices-scalable/dp/1617290343



The design patterns I've described in this chapter are slightly more detailed but they fit perfectly into the remit of the Lambda architecture. This is not necessarily surprising as both systems borrow a little from a shared, common parent in the form of the Enterprise Integration Patterns[27].

It turns out that the Lambda architecture is not limited at all to Big Data problems, but rather with the more detailed design patterns that we've encountered in this chapter we have the basis for a general-purpose architecture that is going to benefit from applying microservices in order to achieve antifragility.

[27] http://www.amazon.co.uk/Enterprise-Integration-Patterns-Designing-Addison-Wesley/dp/0321200683

What no *REST* yet?

Harking back to the common fallacies of Microservices, it might come as a surprise to some that we haven't yet discussed REST APIs or event entity/resource manipulation semantics!

For this I turn back to my mentor on this journey, Gawain. His response was simply:

"Not yet".

We haven't gotten down to that level yet.

REST is one option, don't rush it.

REST might be the right way to manipulate a projection, an aggregate or a view, or even a Gateway, but we don't know that yet!

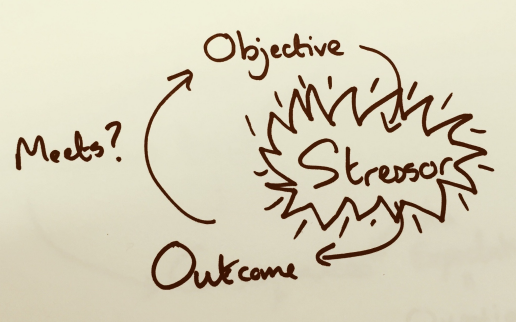
Events are the foundational units of your service APIs, they are good enough to get going.

Don't rush the decision to the exact mechanism of event distribution, let that emerge as you implement.

Software Antifragility patterns with *Stressors*

Antifragility starts when you begin to ***imagine the conditions that could harm, and ideally improve, a system***; stressors and other factors.

You need to have an idea ***what those stressors are before you can begin to thrive on them***.



One stressor on software we are already intimately aware of is change. Changing requirements on your software that mean that it becomes in conflict with the newly discovered reality, this is the basis of the Elephant in the Standup after all.

Stressor's are very similar to the Scientific Method in this respect, each stressor being a hypothesis about the system and its' ability to cope with the unexpected and forming part of a feedback loop of discovery in terms of how a system evolves to cope with that stressor.

There will be many other stressors to consider but as it happens it's usually best to ***consider stressors on a system as the system actually evolves***.

This is a great thing as if we were forced to come up with all the possible stressors in advance we'd likely end up in a strange version of ***analysis paralysis*** where we never built anything for want of exploring all the options.

Instead it's now time to get started. To build some of the idea: an Open Cloud Index. Once we have something real, we can begin to look at the stressors that will affect the system in parallel and evolve our ideas as we go.

For now, it's time to build something. So where do we start?

Time … to Build

"I've… seen things… you people wouldn't believe. Attack ships on fire off the shoulder of Orion; I watched c-beams glitter in the dark near the Tannhäuser Gate… All those… moments… will be lost, in time, like tears… in… rain. Time… to die" *- Much improvised monologue by Rutget Hauer in Ridley Scott's Bladerunner*

… Not quite! …

For us, it's ***time… to start!***

The Problem of *Belonging to Tomorrow*

I'm in anguish. I'm staring at my laptop, wondering what IDE or tooling to use. I still don't know what I'm doing, but somehow that should be ok…

I'm procrastinating at a professional level. Procrastinating direcyly somes from the Latin ***Procrastinationem***, *"a putting off from day to day"*. I'm thinking about things that are ***procrastinus***, *"belonging to tomorrow"*.

I'm worrying about all of sorts of concerns that business-people, and I'm one of them, would call ***strategic***.

Strategic decisions are ***Big Decisions***; ones that are worth worrying about for a long time to get right. Aren't they?

Questions that I spend a long time worrying over sounds a little like I'm searching for the perfect answer, a little Platonic if you will. Instead of getting going, I'm in a haze of different strategies about how to approach the problem both technically and politically.

This is insane. I've not even built anything yet.

Procrastination is the enemy of doing.

Antifragility is in direct conflict with Procrastination[28]:

"Antifragility has a singular property of allowing us to deal with the unknown, to do things without understanding them-and we do them well. Let me be more aggressive: we are largely better at doing than we are at thinking, thanks to antifragility. I'd rather be dumb and antifragile than extremely smart and fragile, and time." - *Taleb, Antifragile*

How do I ***become a doer*** when there are so ***many big decisions I think I need to make first*** before I get going?

[28]Including Over-Design, that dangerously wast-inducing form of procrastination most popular to any intellectual product field, but most especially software development.

Reducing the fear of *Big Decisions*

Time is ticking away and I'm aware that I've written very little software, and even less prose for this book.

Here are some of the options I'm considering:

- Should I use Java?
- Should I use Go Lang (I have a sweet spot for this little language and ecosystem)?
- Should I use Docker?
- Should I use Cloud Foundry?
- What about Heroku?
- What about Frameworks?
- Am I accidentally endorsing a product?
- What about …. ***AAAAAAARGH!***

All of these questions are valid, but I'm forgetting something. One of the benefits of microservices in the service of antifragile system construction is supposed to be adaptability; adaptability in the face change.

Including a change of mind on technologies.

Martin Fowler famously and pithily stated that architecture was about the ***Big Decisions***, but in the case of microservices and antifragility we're trying to make those decisions smaller and all up-side.

Microservices can, in themselves, often be re-written in a very short amount of time[29]. The reduction of big decisions into smaller, frequent decisions is a key factor in the adaptability, and therefore potential anifragility, of the microservice-based approach.

In addition, microservice-based systems are polyglot by definition with each microservice loosely coupled with its collaborators and running and evolving in its own space. So questions of which technology that would have deserved the monicker of being ***the big decisions*** shouldn't be any more. They are scoped to the specific service.

[29]This is a side-effect, not necessarily a goal or measure to decide whether something is a microservice or not though.

Right now the big decision should only be ***what can I construct right now to do something useful?***

And then, and only then, ***what technology can I use to do this quickly and simply?***

Forget big decisions, let's just make some small ones and be a doer.

In the immortal cannon of Richard Branson:

"Screw it, just do it."

Eating an Elephant with the Life Preserver

A Tale of Three Teams - The Life Preserver

Story of late night conference calls. I hate them but this one just got interesting…

What is the best way to approach a new project based on Microservices?

How do I take a monolith, which has become a bit of an elwephant in the standup, and evolve it towards antifragile microservices?

essentially, How do I eat the elephant?

The answer to both questions is simply: Create the *right* monolith,

But the story of how we got ot that answer, and the detail of achieving the *right* monolith, is more complicated…

Taking a short but topical detour[30] this chapter starts with a little story.

[30]We'll be coming back to the design of systems using the patterns from the last chapter in just a bit!

A Tale of Three Teams

There's very little real scientific experimentation these days in software development.

In the main we tend to look to other industries for inspiration and then, dangerously, copy those superficially-appropriate techniques that appeal to our own individual sensibilities.

Much pain and expense has been accrued in the software development industry from the naive dogmatic adoption of techniques from other industries.

Back to *"Skin in the Game"*

Take a risk, do an experiment as best we can.

What is the right approach to building a system?

Threee hypothesis:

Microservices first? Green field? Only do microserivces or…

Layered monolith like before… do it as you would always do it.

Brown-field? Suggested by my colleague, David Dawson.

Do a better monolith, and then break apart because it is amenable to it based on stressors.

The Process and Tool: *The Life Preserver*

Life Preserver Process for working with an existing monolith both in terms of integration with the microservices-based system, and gradually evolving the legacy forward.

NOTE: Design for change. Visualise and rationalise about the system considering change.

What makes a good diagram. Ask and answer questions. What questions?

Doing O.R.E.: Organise, Reduce & Encapsulate

The Life Preserver *Process*: Asking and Answering Questions

Questions motivation

Just Enough Design

Iterative, sometimes sequential, often in parallel

Where does my component go?

What should my component do?

Limit to the patterns

Complex Relationships, Simple Components

Group Together that Change Together

When should I split my component?

Discover the Domain

Events, not Domain Objects

What happens, not what is there. It's the ubiquitous language.

What goes in an Event?

Event Storming to elicit Events

Process, outputs etc.

Thinking about Integration & De-Coupling

Focus on Contract First

What needs to be known and agreed between the two components

How should my Components Communicate? Design Events

Communicate intention.

How should my Events be transmitted? Design Dispatch

How should my Events be Encoded? Design Payload

Discuss CQRS

Commands, Queries, Events

Thinking Stressors

Creating Stressors

Consider Circuit Breakers

Consider Bulkheads

12-Factor Services?

Consider Technologies *Early*

Bounded Contexts, Boundaries & Fracture Lines with Tectonic Plates

What 'could' change

Naive is ok, at first

Having a Conversation about Change

Seeing Inter-Plate and -Component Communication

Friction at the Fault Lines

Focus on where the Friction is Greatest

Testing your Inter-Plate Change Coupling

The Dangers of Inappropriate Inter-Plate Intimacy

Flows across the Life Preserver

Reiterate patterns from prior chapter, including streams.

Continuous Refinement: *Emergent Design*

A design approach that encompasses the fact that we are doing research and development.

Scaling Life Preservers

Organisational, bounded area of business (reference to Sam Newman), System, individual systems and even Microservices.

Examples of each as sketches.

Big Payback: *Objectivitity*

Story of Snowflake and the outcome.

Simplicity and the Life Preserver gives just enough constraints and shared goal to come up with objectively improved designs.

ManifestNO for Antifragile Software

Change and Ossification

Evil Manifestos of Years Past

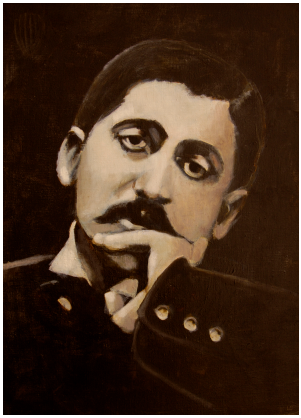
Why a Manifesto at all?

Dodging the Bullet

Why we're not going to be having one.

Anti-manifesto description and discussion

Why Microservices are not just SOA



Marcel Proust; not known for his love of cliché.

Recap to extinguish this argument.

The poverty of SOA. Cliche and the superficial articulation of a good idea.

Remember & Apply

The aim of this book is no less than to realign our thinking and calling us to action in how we build our software and so rather than a dull and static 'Summary', it would only be fair to instead re-emphasise what those calls to arms were. Ideally as vociferously and supportively as possible! In my presentations and writings I call that the 'Remember & Apply' section, as it accurately communicates what I want you to do.

So with no further ado, here are the main things to remember and apply from "Antifragile Software":

Suggested Further Reading

No technique in software development is an island and Simplicity underpins many of the more recent development in software development techniques. No doubt the following list will grow as time goes by and more attention is focussed on solving real problems of software development.

Included in this list for further investigation and learning are techniques such as:

- **Domain Driven Design, CQRS and Event Sourcing** - Human Comprehensibility is king and nothing brings this to the fore like the work of Eric Evans and Greg Young who have given us Domain Driven Design: Tackling Complexity in the Heart of Software[31] and CQRS and Event Sourcing[32] respectively. Unfortunately Greg is still working on his book but we're all rooting for that to be out soon as, from Greg's talks and very popular courses, this will be crucially important material for simplifying software and increasing adaptability.
- **Clean Code** - Human Comprehensibility is king but it goes hand-in-hand with clarity. Clean Coding techniques

[31] http://www.amazon.co.uk/Domain-driven-Design-Tackling-Complexity-Software/dp/0321125215

[32] http://www.amazon.com/Event-Centric-Simplicity-Addison-Wesley-Signature/dp/0321768221

really deliver on that clarity and Uncle Bob Martin has written the canonical books on the subjects: "Clean Code" and "The Clean Coder". Also check out the Clean Coders[33] site for some excellent videos on these topics.

- **Test and Behaviour Driven Development (TDD and BDD)** - One of the tenets of Test and Behaviour Driven Development is to write "only the code that is necessary to pass the test". This maxim helps us to effectively simplify and minimise the amount of code we allow ourselves to write in solving a given problem, and as such TDD and BDD is discussed in this context in more detail as specific patterns for simplifying your software in this book. In addition, TDD and BDD can have a much broader impact in how you think and design your code and so we recommend the excellent books by Kent Beck[34] and Uncle Bob Martin[35] for a deeper exploration of the subject.
- **Impact Mapping** - Impact Mapping is a tool for exploring the main options, and assumptions underlying those options, in order to select the simplest route to manifesting some value. Impact Mapping was coined by Gojko Adzic in his excellent "Impact Mapping: Making a Big Impact with Software Products and Projects"[36]. Impact Mapping is discussed in some more in this book as it is such a

[33] http://cleancoders.com

[34] http://www.amazon.co.uk/Driven-Development-Addison-Wesley-Signature-Series/dp/0321146530/ref=sr_1_1?ie=UTF8&qid=1381233393&sr=8-1&keywords=kent+beck+tdd

[35] http://www.amazon.co.uk/Agile-JavaTM-Test-Driven-Development-ebook/dp/B003DVG7QO/ref=sr_1_1?ie=UTF8&qid=1381233419&sr=8-1&keywords=Robert+Martin+TDD

[36] http://www.amazon.co.uk/Impact-Mapping-software-products-ebook/dp/B009KWDKVA/ref=sr_1_1?ie=UTF8&qid=1381233463&sr=8-1&keywords=Impact+Mapping

powerful technique for working towards "delivering the right thing".

- **Real Options** - Real Options in IT is the work of the excellent Chris Matts who, along with Olav Maassen and Chris Geary, has written a wonderful graphic novel called "Commitment"[37] that illustrates the technique to perfection. Well worth a read, and I think we'd all love more technical subjects to be explained with a great story and great art like this!

[37] http://commitment-thebook.com