Bugs with Long Tails

By Stuart Wray

scw@stuartwray.net

17 March 2013

** DRAFT **

Abstract

Underestimation of risk is a danger in software development as much as in financial markets, and for similar reasons: evidence shows that the structure of software, and the pattern of faults in software, both follow a fractal distribution. This being the case, even if we didn't already believe it, we would expect the problems of software projects to mirror the problems of financial markets, with widespread misjudgement of risk and apparently safe projects derailed by "Black Swan" disasters. This article discusses fractal patterns in software development and explores some factors which tend to exacerbate our problems. However, the picture is not entirely bleak, and although we can never hope to escape entirely from the "long tail," this article also suggests some measures which we can take to ameliorate our problems once we have recognised them.

Introduction

Years before the recent disasters, Nassim Nicholas Taleb wrote a book which caused widespread controversy in the financial world. Described by one reviewer as "The book that rolled down Wall Street like a hand grenade," *Fooled by Randomness* (Taleb 2001) explained how the statistical methods used by financial traders were fatally flawed. Although the traders believed that they were making measured, controlled bets, Taleb said they were in fact playing with fire and routinely risking ruin. Although Taleb was almost a lone voice, he received inspiration and support from Benoit Mandelbrot, whose name is synonymous with fractal geometry and who as early as the 1960s had pointed out fractal patterns in market prices (Mandelbrot 2004).

At the time, the financial wizards were making too much money to pay very much attention to Taleb or Mandelbrot. However, as Taleb pointed out, their trading was like a game of Russian Roulette: it doesn't matter how much money you make in the short term, if you face certain ruin in the longer term. So what exactly is the difference between the world the financial traders thought they inhabited and the world Taleb and Mandelbrot said they actually inhabited?

Taleb coined the terms "Mediocristan" and "Extremistan" as a shorthand for these two worlds (Taleb 2008). In Mediocristan, everything you meet falls within a short distance from some average, and the chances of finding anything very far from the average are

negligible. For example, consider the height of adult humans. If you measured the heights of a few people taken at random, you could make a very good estimate of the true average height for the whole population. Not only that, but with these few measurements, you could also make a good estimate of the range of typical heights. Based on that estimate, you could be confident that the chance of ever meeting any adult less than 20cm tall or more than 20m tall was negligible.

Extremistan, on the other hand, is different. You cannot, with a small number of measurements, be so sure of either the average or the typical range. For example, consider what happens if instead of height we measure wealth. If you measured the wealth of a few people taken at random, you could calculate an average, but this could be very far from the true average for a population which contained billionaires like Bill Gates. Extremistan contains "Grey Swans": individuals who are rare, but who have a large overall impact. Until you come across them, you don't know how many there are or how large their impact will be, and you don't come across them very often. This is why you need to take many more measurements in Extremistan before you can claim with any confidence that you know what possible outcomes can be safely ignored. (Extremistan may also contain "Black Swans", which are *extraordinarily* rare, but which still have a huge overall impact.)

The financial traders liked to pretend that they were in Mediocristan because that let them use Gaussian distributions to model their risks, and thus to claim that their risks were moderate. However, it should have been clear to everyone that they were wrong, because financial events occurred a few times per decade which, according to the Gaussian calculations, should each have happened only once in tens of thousands of years. Mandelbrot commented that:

> "For more than a century, financiers and economists have been striving to analyze risk in capital markets, to explain it, to quantify it, and, ultimately to profit from it. I believe that most of the theorists have been going down the wrong track. The odds of financial ruin in a free, global-market economy have been grossly underestimated." (Mandelbrot 2004, p4)

To what extent have we, as software developers, unwittingly committed the same sin? We do not attempt to place bets directly like the financial traders, but in a real sense every software development project is a bet — we bet that we can deliver the planned functionality with a certain effort in a certain time. We know that when we build our systems that they will contain many faults, but we bet that we can remove enough of these faults before delivery — enough that the end-users will see a working rather than a failing system. We know that in placing these bets we run risks, but what if, like the financiers, we have misunderstood the nature of those risks?

Software is Fractal

If software systems are from Extremistan then we ought to find evidence of fractal or "scale-free" structure in these systems. There has been surprisingly little interest in looking for such evidence, but when researchers have looked, they have found it. For example, Potanin et. al. (2005) investigated the pattern of references between objects in dozens of programs written in four different object-oriented languages (Java, C++, Self and Smalltalk). The conventional theory of object-oriented design envisions that programs of any size should be constructed in essentially the same way:

> "... by encapsulating complexity within objects at one level of abstraction and then composing these objects together at the next. Thus all objects should appear to be the same size and complexity; larger programs merely use more objects and more levels of abstraction. We have found the exact opposite in our corpus of OO snapshots." (Potanin et. al. 2005, p102)

Potanin et. al. measured the "size" of objects created at runtime in terms of the number of incoming or outgoing references. (Interestingly, they found no objects with large numbers of both incoming *and* outgoing references.) Rather than the expected pattern of uniform size and complexity, they found that, although most objects in their systems were of the smallest size, there were many objects of larger and larger sizes, their numbers decreasing with size in a power-law distribution. Contrary to expectations, there was no "typical" size of object: the pattern of object references was "scale free". (A fact which, as Potanin et. al. point out, demonstrates that the conventional theory of object-oriented design is wrong.)

Further evidence of fractal structure in software systems comes from Hatton (2009a, 2010, 2011). Hatton examines the component-size distribution in 75 substantial programs written in Ada, C, C++, Fortran, Java and Tcl-Tk, totalling approximately 34 million lines of code. Hatton's "components" are source-code functions and "size" is measured in terms of lines of code and also in terms of distinct tokens. (These two measures are related approximately linearly in the code that he examined.) Again, except for the very smallest components, there is strong empirical evidence for a power-law distribution of component sizes across many different applications and several languages (Hatton 2011). Not only that, but this scale-free structure was present in the earliest releases of the systems he examined and was conserved as the systems were maintained and grew over several years (Hatton 2010). Hatton also gives an information-theoretic argument as to why our systems should have a scale-free structure, despite their authors being oblivious to this fact (Hatton 2009a, 2010).

When we accept that this scale-free structure is real, it explains some other properties concerning faults and failures of programs. (These terms are used here with a specific meaning: a *fault* is a coding or design mistake in a program, such that part of the program behaves in a way that was not intended by its authors. A *failure* is an incident where the program as a whole is observed to behave in a way that was not expected by its users. The term *defect* is used to mean a known, discovered fault, as opposed to a

"latent" or undiscovered fault.)

It has been recognised for decades that although a program may have a great many latent faults, only a very small proportion of these lead to observed failures, and that most of the observed failures are traced back to a very small number of latent faults (Adams 1984). We can now explain this quite simply in terms of the scale-free structure of our programs, because, as noted by Potanin et. al (2005), scale-free networks are very robust in the face of damage. This is because most nodes in a scale-free network have few connections, so if they are damaged the effect on the whole network is minimal. Viewing faults as instances of damage, we would therefore expect most faults in our scale-free programs to make little difference to the whole system. Only a fault which damages one of the few well-connected "hub" nodes is very likely to cause an observable failure.

Of course, a slight subtlety here is that the "size" of a run-time object measured by references in or out need have no correlation with "size" measured in lines of code. A small component in terms of code size might be large in terms of the runtime connectivity graph, and vice versa. However, this disparity might in turn explain the phenomenon reported by Fenton & Ohlsson (2000) where the number of faults detected in a module during pre-release testing and inspection was *inversely* correlated with post-release failure due to faults in that module. This would be the case if pre-release testing and inspection were spread evenly across lines of code, rather than concentrated on the code associated with the well-connected "hub" nodes in the runtime connectivity graph.

Another related question is whether or not there is an optimal size for components: a size at which the fault-density is at a minimum. (If there were such a size, this would constitute evidence that our programs were not fractal, because they would not be scale-free.) Fenton & Neil (1999) coined the term "Goldilocks conjecture" for the widely believed idea that there is an optimum component-size which is "not too big nor too small". Although widely believed, this still remains a conjecture and, as they point out, if the conjecture is false, then:

> "Virtually all of the work done in software engineering extending the fundamental concepts, like modularity and information-hiding, to methods, like object-oriented and structured design would be suspect because all of them rely on some notion of decomposition. If decomposition doesn't work then there would be no good reason for doing it." (Fenton & Neil 1999, p7)

Evidence from Fenton & Ohlsson (2000), supported by Hatton (2011) suggests that there is no systematic correlation between fault-density and lines of code, cyclomatic complexity or any other measure of component-size. Hatton (1997) once believed in the Goldilocks conjecture, but has now repudiated it (Hatton 2009b), suggesting on information-theoretic grounds that in mature systems, the larger components may in fact have a higher fault-density (Hatton 2009a). However, Koru et. al. (2008), following Basili & Perricone (1984), suggest to the contrary that smaller components have a higher fault-density. Taken as a whole, this mixed evidence certainly does not favour the

Goldilocks conjecture. On the contrary, we see further evidence for scale-free structure in the widely accepted idea that defects cluster in software systems. For example, Hatton (2010), Fenton & Ohlsson (2000) and Boehm & Basili (2001) all report a distribution of defects amongst components consistent with a power-law. The evidence overwhelmingly indicates that software is fractal.

So we live in Extremistan and therefore we will have to learn to cope with the Grey Swans: exceptional events from the long-tails of our distributions which nevertheless have a large overall impact. As we know, a single misplaced character can cause a rocket to crash, or a bank to lose millions of dollars. Simple mistakes cause small faults which very occasionally lead to large failures. Most of the danger, most of the risk, comes from these rare but large failures, not from an accumulation of many small failures.

We know this, but we would like it to be otherwise. Perhaps like the financiers, by believing a little too much in how we would like it to be, we unwittingly run risks without being aware of it. We believe in object oriented design. We believe in the Goldilocks conjecture. We would like to believe that by gathering statistics we can estimate the risk of future failures. What if we can't? Taleb (2008) recommends that when we realise that we are in Extremistan that we forget about prediction and invest in preparedness instead. Maybe we should take his advice.

But worse dangers may lurk in the darkness: Black Swans. Extraordinarily rare but utterly devastating events. Since we live in Extremistan, there *will* be Grey Swans, but that doesn't mean that there will necessarily be Black Swans. In the rest of this article I want to discuss three particular areas where we can improve our prospects if we take care, but if we are thoughtless we can make things much worse. These areas are: computational completeness, prospect theory, and variation in people.

Computational Completeness

There is a big difference between the code that we write deliberately and the code that happens by accident. The code that we write deliberately is different because “programs must be written for people to read, and only incidentally for machines to execute” (Abelson et. al. 1996). Programs for people to read are very much larger and better structured, because this structure gives their readers a handrail to guide their understanding. This is of course not the case with code that happens by accident. We know from computation theory that most random code is dense: there is usually no more compact description of what random code does than the code itself, and the only way in principle to know what an arbitrary piece of code does is to run it.

Now, we are accustomed to thinking of code at one level of a system as forming a substrate or virtual-machine for code at a higher level. For example an interpreter forms a virtual-machine for the interpreted source-code, and in turn the hardware of a

processor forms a virtual-machine for the machine-code instructions of the interpreter. However, we are less used to thinking of our programs as virtual-machines for the faults which
accidentally inhabit them. But when we turn around and look at our programs that way, it can be quite revealing.

Think of a particular fault as the difference between the code that should have been written and the code that was actually written. Now regard this difference *itself* as a form of code: accidentally produced code which runs on the rather bizarre and unintended virtual-machine formed by the rest of your program. Given the previously mentioned results concerning the unpredictability of faults, we have every reason to suppose that this accidentally produced code is effectively random, and since the languages we use to write our programs are usually computationally complete, even a small a fault can encode very complex behaviour. For example, we are all familiar with the occasional fault which, when we finally locate it, happens to be the reason that our program has so-far appeared to work, and when we remove this fault the whole program suffers an immediate and spectacular failure.

Worse than that, it is even possible for a small fault to lead to the most complex possible behaviour: a fault can turn a program into an unintended Turing-complete interpreter on a "program" embedded in its input data. As demonstrated by Bratus et. al. (2011), many apparently disparate security problems are in fact instances of the same underlying design flaw: an inappropriate choice of grammar for the data carried over an interface to a software component. This common design flaw provides a fertile ground into which faults may fall, faults which effectively encode "weird machines" — computationally complete interpreters into which outside attackers can feed carefully crafted programs masquerading as data.

Countermeasure: Starve the Turing Beast

The countermeasure proposed by Sassaman et. al. (2011) against such weird machines should also be effective against faults in general, reducing their impact and perhaps turning some Black Swans into Grey Swans. This countermeasure is to "Starve the Turing Beast" by using the simplest possible language at an interface between components, whether those components are peers communicating at the same architectural level or one of them acts as a substrate or virtual-machine for the other.

Wherever two components communicate there is a de-facto interface, with a de-facto grammar, but very often these grammars are more complex than necessary. Every component must embody a recogniser which accepts good inputs and rejects bad inputs, but some grammars are so complex that telling good from bad is undecidable. Those recognisers are therefore impossibly difficult to test and they make comfortable homes for undiscovered faults. Some grammars are so complex that checking whether two implementations are equivalent is undecidable. (We can only be sure of proving that regular expressions and context-free grammars are equivalent. Context-sensitive and recursively-enumerable grammars are too powerful.) Again, choosing an inappropriate

grammar gives faults an easy hiding place and creates the potential for arbitrarily complex behaviour.

Perhaps this is the reason that domain-specific languages (DSLs) are so useful a technique: if we use a DSL which is not computationally complete, we cannot accidentally create a weird machine from a fault in any code written in that DSL. For example it is safer, if we can, to implement a particular component as code for a state-machine DSL, rather than as equivalent code for some computationally complete language, because if there is a fault in the state-machine DSL code, the consequences are much more limited. This is the difference between Grey Swans and Black Swans: limited versus unlimited consequences. Computationally complete interfaces have unlimited consequences.

Unfortunately, we all too often yield to temptation and introduce computationally complete DSLs where we don't need to: for example, SQL and HTML have become computationally complete where once they were not. Naturally as programmers we like to take the easiest path to a solution, and sometimes that means embedding our favourite scripting language in an application, or extending a previously limited DSL with more general capabilities. However, this is unwise, and for safety it would be better to be more restrictive. (For this reason we should also be suspicious about a "DSL" which is actually implemented as a library within another computationally complete language. There is too much potential for leakage between the two supposedly separate levels.)

Prospect Theory

We often find it hard to estimate how much time it will take to deliver working software. Project managers would like to believe that there exists a purely mechanical way to make time estimates, a way that will always give an accurate answer, a way not based on human judgement and intuition. Unfortunately, this appears to be a forlorn hope. Lewis (2001) uses results from algorithmic information theory to show that attempts to formalise software estimation are futile, because there cannot (even in principle) exist a formal process *guaranteed* to predict program complexity, and hence there cannot be a formal process which will reliably predict development time. In support of these theoretical arguments, Lewis also notes that his conclusion is supported by practical experience:

> "Clearly the parts of a project that are similar to previous projects will be estimated more accurately. The remaining parts, even if they are small, can be problematic. … Most experienced programmers have encountered projects where an apparently trivial subproblem turns out to be more difficult than the major anticipated problems." (Lewis 2001, p57)

So, instead of being able to depend on a formal process, software estimation will always rely on human intuition and experience. It is therefore essential that we understand how

human intuition can go wrong when we make judgements in the face of uncertainly. Psychology has, in recent decades, made substantial advances in this area, and we can now draw on the insights of "prospect theory" to help us make better decisions.

The origins of prospect theory date back to the 1970s, when Daniel Kahneman and Amos Tversky described and explained various phenomena which according to the standard economics of "rational expectations" should not happen (Kahneman & Tversky 1979). The core insight of prospect theory is that "people attach values to gains and losses, rather than to wealth, and the decision weights that they assign to outcomes are different from probabilities" (Kahneman 2011, p316).

Prospect theory explained previously mysterious phenomena such as the "endowment effect", where people value things more when they have them than they did beforehand, when they did not have them. For example, Richard Thaler — who coined the term "endowment effect" — noticed one of his economics professors behaving in a way contrary to that predicted by the "rational expectations" economic theory of the time (Kahneman 2011, p292). Professor R, a wine buff, would would be willing to sell bottles from his collection, but only for more than $100. (This was a lot of money at the time, in 1975.) However, he was not willing to pay more than $35 at auction for wine of that same quality. At prices between $35 and $100 he would neither buy nor sell. This was inconsistent with the economic theory of the time, which maintained that there should be a single price, perhaps around $50, above which he would sell, below which he would buy. But in practice that wasn't the case.

The same phenomenon has been found reliably in laboratory experiments. For example, in one experiment, some participants chosen at random are given coffee mugs, which they are told they can take away with them. They are later asked how much they would accept to sell their mug. At the same time, participants who were not given mugs are also asked how much *they* would pay to *buy* a mug. The mug-owners invariably value their mugs more highly than the mug-less prospective buyers, sometimes at double the price.

The explanation from prospect theory is that we do not make decisions based on absolute value, but that we have a reference point, compared to which we regard changes in terms of gains and losses. However, we do not value gains and losses equally: in general, losses loom larger. Thus the fair price to give up our mug is more than the fair price to gain a mug we do not have. (There are subtleties, to do with whether we hold something for our own use or to trade. Experienced traders learn to avoid the endowment effect for the things that they are trading.)

Prospect theory explains how we make decisions intuitively in the face of gains and losses, but it also explains how we make decisions when those losses and gains are not certain. The weights that we give to different options are not what would be chosen by a "rational actor" who simply decided based on the average value of the outcome, the value given by summing up values multiplied by probabilities. Instead we tend to over-weight outcomes which are nearly certain (but not quite) and outcomes which are

unlikely (but possible). When we put all these factors together, the "fourfold pattern" of prospect theory emerges, illustrated in the following diagram. Each cell in the table contains in its first two lines a scenario; on the next line, the dominant emotion which people feel in that scenario; and lastly, how people overwhelmingly behave in that scenario: whether they are risk seeking or risk-averse.

	GAINS	LOSSES
HIGH PROBABILITY "Certainty Effect"	95% chance to win $10,000 (5% chance to win nothing) Fear of disappointment RISK-AVERSE (e.g. Choose to accept offer of smaller but certain payout.)	95% chance to lose $10,000 (5% chance to lose nothing) Hope to avoid large loss RISK SEEKING (e.g. Refuse option to suffer smaller but certain loss.)
LOW PROBABILITY "Possibility Effect"	5% chance to win $10,000 (95% chance to win nothing) Hope of large gain RISK SEEKING (e.g. Buy a lottery ticket.)	5% chance to lose $10,000 (95% chance to lose nothing) Fear of large loss RISK-AVERSE (e.g. Buy a insurance.)

The Fourfold Pattern *After Kahneman (2011, p317)*

When prospect theory predicted this pattern, the behaviour in three of the four cells was in line with earlier expectations, but the behaviour in the top right cell was new and unexpected. Thus, the the bottom left cell confirms the popularity of lotteries, and the bottom right cell, of insurance. The top left cell says that we prefer to lock-in a nearly sure gain, even if we don't gain as much as we might, that "a bird in the hand is worth two in the bush." However, in the top right cell we see something different, something which before Kahneman and Tversky was not predicted by any theory. But as Kahneman notes:

> "Many unfortunate human situations unfold in the top right cell. This is where people who face very bad options take desperate gambles, accepting a high probability of making things worse in exchange for a small hope of avoiding a large loss. Risk taking of this kind often turns manageable failures into disasters." (Kahneman 2011, p318)

If that was not bad enough, we are also prone to “framing effects” where the same situation can be viewed in terms of losses or gains. For example, the standard example of a framing effect is “The Asian disease problem” (Kahneman & Tversky 1984).

> Imagine that the United States is preparing for the outbreak of an unusual Asian disease, which is expected to kill 600 people. Two alternatives have been proposed. Assume that the exact scientific estimates of the consequences of the programs are as follows:
>
> - If program A is adopted, 200 people will be saved.
> - If program B is adopted, there is a one-third probability that 600 people will be saved and a two-thirds probability that no people will be saved.

Most people (75%) when asked by Kahneman and Tversky decided to go for the sure thing rather than gamble, and chose option A. However, when another group of people was presented with an alternative version of the problem, they chose differently. The alternative version offered instead a choice between these two options:

- If program C is adopted, 400 people will be die.
- If program D is adopted, there is a one-third probability that nobody will die and a two-thirds probability that 600 people will die.

This alternative version of the problem is framed in terms of losses rather than gains. As we would now expect, when presented with two bad outcomes, rather than swallow a certain loss, most people (75%) decided to gamble and chose option D. The frightening thing about this is that the actual situation and the possible outcomes in both versions of the problem are identical. They are just framed differently: one in terms of gains, the other in terms losses. The reference point, compared with which we regard outcomes as gains or losses, can be rather easily moved, simply by framing the same situation in different terms.

How does this apply to software projects? At a risk of labouring the now painfully obvious, consider a project which has suffered a schedule slippage. The work has taken longer than expected. What does the project manager do? If they are fully invested in the planned schedule, any slippage will feel to them like a loss. Their set-point is the firm expectation of the promised functionality on the promised date. They are in the top right box, and so they do what prospect theory would predict: they roll the dice and hope for the best. “Let's not bother with those tests,” they say. “We don't have time. Let's just go live on the main system.” And so they trade-up from disappointment to disaster, from a Grey Swan to a Black Swan.

Countermeasure: Mindfulness and The Outside View

The most important thing to remember about prospect theory is that it is a description of the intuitive choices we make without deliberate, rational thought. Modern psychology understands that it is vital to draw a distinction between “conscious” and “unconscious”

action. However, these words do not have the same meaning as the older and more popular usage of the same terms, so sometimes modern psychologists use alternative words to emphasise that distinction — for example, Kahneman (2011) uses the terms "System 1" and "System 2", with the following properties:

System 1	System 2
"Unconscious"	"Conscious"
Quick	Slow
Intuitively	Deliberate
Effortless	Effortful

The relationship between System 1 and System 2 is somewhat like the relationship between the many subsystems of a modern airliner and the pilot "flying" it: System 1 takes care of most things, and it works continuously, quickly and in parallel; System 2 attends to things out of the ordinary, one at a time, like the pilot alerted to a problem by a flashing light on the control-panel.

Most of our ordinary decisions are made by System 1. A few are taken deliberately by System 2, but when if are in a hurry, or tired, or stressed, we don't bother to engage System 2. Using System 2 is always an effort. System 1 is always willing to step in and give an effortless intuitive answer. At the time it feels right. System 1 is generally very competent, but its quick rules-of-thumb — and prospect theory is just a description of some of these rules-of-thumb — sometimes do not select the best choice.

So how can we escape from the bad decisions which prospect theory predicts that we will tend to make in some circumstances? The first thing is that we must do is to exercise what is sometimes called "mindfulness": we must engage System 2 and deliberately attend to what we are thinking. Don't let System 1 make all the decisions: stop and think.

Having stopped to think, ask yourself: "What is the right frame for my problem?" We have seen in the "Asian disease problem" that a different frame can lead to a different intuitive decision. With deliberate, System 2 thinking, we can frame the problem in different ways. For example, in the experiment where people were given mugs then asked how much they would accept to sell them, the "endowment effect" can be almost completely negated by telling the participants to "think like a trader". A similar re-framing, which works against many pressure-selling techniques, is to consider what you would have done at the start if you had been presented then with the choice you are now faced with at the end (Cialdini 2009, p89).

The generalisation of these ideas is to take the "Outside View" (Kahneman 2011, p245-254) of a problem. That is to say, to call to mind the class of situations that most resemble your current one, a reference class within which you will then try to place your current situation. But first, you must assess the distribution of outcomes across the reference class. What usually happens? Only after you have answered that question

should you attempt to make an intuitive estimate of where in the class to place your current situation. (An estimate which can be substantially improved by more knowledge about those other situations, but only slightly improved by more knowledge about your current situation. You will always know a lot about your current situation, which always tends to lead to the feeling that “this time is special”. It usually isn't.) Your place in the reference class then gives the best indication of the likely outcome.

But even when you are mindful and you try to take the outside view, it can be hard to make the rational choice. System 1 carries powerful feelings of rightness with its intuitive choices. However, we don't have to let System 1 decide everything.

Variation in People

It's a commonplace, part of the folk-wisdom of software development, that programmers work at different rates and that the difference between individuals can be a factor of ten or more. (The contrary assumption, that all programmers work at essentially the same rate, is so clearly not true that no programmer even suggests it.) The evidence for this assumed 10× productivity range is not entirely clear-cut. Individual productivity is hard to measure, and Lewis (2001) argues, based on results from algorithmic complexity theory, that it is not possible even in principle to define an objective measure of individual productivity.

However, we do have results which are difficult to explain other than as examples of individual variation. Amongst several examples, McConnell (2011) notes the startling difference between the development of Lotus123v3 and Microsoft Excel3.0. These were two directly competing products, developed at the same time by different companies, but with essentially the same functionality. The productivity of the Lotus team was 1,500 lines per person per year, while the Microsoft team wrote 13,000 lines per person per year. It appears that the Microsoft code was not especially more verbose or worse in any systematic way. They just wrote it faster.

Although this kind of evidence is not entirely unambiguous, it does lend support to our individual experiences: whenever two programmers work closely together on the same thing, it soon becomes clear to both of them who is quickest at that thing. Some people are quicker at one thing, some at another, and some people are quicker at very many things, maybe by a factor of ten or more. McConnell recounts an anecdote in which “Boeing … moved most of the 80 people off that project and brought in *one guy* who finished all the coding and delivered the software on time.” (McConnell 2011, p568). Rather unusual, but not implausible.

Less often remarked, but still a factor in software development, are the small number of programmers with unusually low productivity, perhaps even negative productivity. People who when you add them to a team, actually make progress run backwards. (I think that most programmers would certainly place a few *managers* in this category.)

People so bad, they don't know how bad they are. Or worse, people who have an idea how bad they are, but who are able to claim credit for lucky successes and deflect the blame for failure on to others. This does sometimes happen, particularly when programmers never work together closely on the same tasks, so they never have a clear idea of the true competence of other members of their team.

What are the implications of this wide variation between individuals? And what should be the composition of the ideal team? Would it be best to have a small team of super-stars, or a large team of more run-of-the-mill programmers? Unfortunately, either of these extreme choices tends to exacerbate the problems which we have noted earlier, though in different ways.

The well-known problem with large teams is that they become victims of Conway's Law, which states that the organisation of a system always mirrors the organisation of the people who designed it (Conway 1968). This is because wherever there is a system, there is a design committee, wherever there is a subsystem, there is a design sub-committee, and where subsystems need to communicate, then clearly so do their design sub-committees. How else could the interfaces be agreed?

Now, software development is in a real sense “design all the way down”. There is no part which is clerical or routine. We have outsourced the routine work to our compilers and interpreters. So, if a large number of people are to work simultaneously on a system, the structure of the whole system must be fractured at the outset into a large number of subsystems. This initial decomposition is unlikely to be the best choice, but it will be very hard to improve it after a bad start, because changes will always involve negotiation between many people, rather than choices in the head of one person. With the decomposition of functionality into subsystems relatively fixed, the interfaces between subsystems will tend to become more complex. Rather than “starving the Turing-beast” the designers will tend to feed it, since sometimes the only way to get the job done will be to introduce an over-powerful interface between two subsystems which were separated at birth but now need to work closely together.

Not only will a system designed by a large number of people be worse, but a large number of people needs more coordination, so there will be more managers and relatively fewer people designing software and writing code. When we count these extra heads, this lowers overall productivity still further. In the end, the large team may make such slow progress that their software is never delivered, or fails to meet its real requirements. (And in the face of such disappointment, one of the substantial body of managers may be tempted to make the kind of poor choice predicted by prospect theory, and turn a Grey Swan into a Black Swan.)

Given these problems, the idea of employing a small team of super-stars starts to look appealing. The smaller the number of people building a system, the easier it is to globally optimise the whole system. If two subsystems are designed and built by the same person, it is far easier to move functionality between the parts until it is in the right place. Part of this movement is the redesign of interfaces, and with the

functionality in the right place, these interfaces can be simpler and safer.

However, the problem with the super-star strategy is that there is a potential for at least a Grey Swan event if one of these very good programmers becomes unavailable for any reason. A programmer who is ten times more productive is still just as likely to be hit by the proverbial bus, to get cancer, or to be on holiday when you vitally need them. They might, if anything, be *more* likely to leave for another job. If they leave or are otherwise unavailable, they take with them a much larger fraction of the development team compared with the loss of a single developer from a larger group of slower individuals. Furthermore, if the departing super-star had sole responsibility for some subsystems, then their loss could halt progress on the entire project.

Countermeasure: Masters, apprentices and synthetic people.

So: what should be the organisation of the ideal team? The answer is not difficult to see once we accept that software is actually a craft activity, not an industrial process where interchangeable workers can be slotted into a software assembly-line. In a craft activity, masters work alongside apprentices: making, teaching and learning. Simpler work is done by apprentices, more sophisticated work by masters, but they both work side-by-side on the same things. In this way, the ideal team could contain both super-stars *and* less able programmers, and that way avoid many of the problems described above. This sounds like a laudable aim, but how exactly would it work? What does working "side-by-side" look like in practice?

The most plausible solution would appear to be "pair programming", where all deliverables are produced by pairs of developers working together. Although sometimes misunderstood, pair programming can be very effective if done properly (Wray 2010). Circulating pairs within a larger group work together for only a few hours before splitting up and reforming in different pairs. In a form of Action Projection (Wegner 2002, p187), code is effectively created by "synthetic people" formed from these dynamically changing pairs. Experiments have shown that pair programming especially improves the performance of less experienced programmers, effectively emulating a better programmer with two less able programmers. Pairs which include experts do not tend to work faster, but when only one of them is an expert, this is exactly "working side-by-side": the resulting work is about the same as the master would do on their own, but the apprentice also *learns*.

Since pairs swap regularly and everyone works on several parts of the system, in a team like this the main risk from employing super-stars evaporates: there is no part of the system that only one person knows. Although the best person in the team might leave, they will not take all their knowledge and experience with them. While they are with the team, their knowledge about the system and also their general programming skills will be picked up by other members of the team. Of course, a faster programmer might feel that they were being slowed down by having to work with slower people, but perhaps they could be encouraged to see the virtues of building and nurturing a team which will preserve and extend their code when they have left. As the open source community has

shown, the key to the reliability and long term success of a piece of software is the small group of people who look after it day-to-day. Perhaps we should accept that for system maintenance in general, we should think not of maintaining a working collection of code, but of maintaining a working collection of people.

Conclusion

Software comes from Extremistan, where a few rare events cause most of the trouble. I have outlined three areas — computational completeness, prospect theory and variation in people — where we can apply our understanding, act differently, and make those events less dangerous. There are probably other areas where we might similarly improve our prospects, implementing Taleb's advice that we should invest in preparedness, not prediction (Taleb 2008). I don't believe that there are any software development methods which will lead us to the comfortable realms of Mediocristan, but I think that we can and should take steps to help defend ourselves from the worst disasters of Extremistan.

We should also follow another piece of advice from Taleb (2008) and we should forget about trying to predict the future with precise quantitative models. With a power-law distribution, you need enough data to find the power-law cut-off and also the exponent in the "long tail". However, in practice "enough" data is going to be both very, very large and also be determined by the exponent that you are trying to find. In particular, you will tend to over-estimate the magnitude of the exponent, so extreme events will appear less likely than they really are, simply because you need a lot of data to have a good chance of seeing sufficiently extreme events to properly calibrate your model. (The immediate problem when gathering statistics in software research is that if people wrongly assume Gaussian distributions, then calculate p-values, these will be very misleading.)

Finally, we should turn back to the world of finance where we started. The danger of Black Swans in software is also in the modern world a threat to the *financial* world and hence to us all. This is because of the steadily increasing burden of dreadful software in the financial world (Kelion 2013). Unless the financial world takes steps to to avoid software Black Swans, it's only a matter of time before one of them arrives and causes a real-world catastrophe.

References

Abelson, Harold, Gerald Jay Sussman & Julie Sussman, 1996. *Structure and Interpretation of Computer Programs (2nd Ed.).* MIT Press, Cambridge, Mass.

Adams, Edward N., 1984. Optimizing Preventive Service of Software Products. *IBM Research Journal,* vol.28 no.1, pp.2-14.

Basili, Victor R. & Barry T. Perricone, 1984. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, vol.27, no.1, pp.42-52.

Boehm, Barry & Victor R. Basili, 2001. Software Defect Reduction Top 10 List. *IEEE Computer,* vol.34, no.1, pp.135-137.

Bratus, Sergey, Michael E. Locasto, Meredith L. Patterson, Len Sassaman & Anna Shubina 2011. Exploit Programming: from Buffer Overflows to "Weird Machines" and Theory of Computation. *;login*, vol.36, no.6, pp.13-22.

Cialdini, Robert B., 2009. *Influence (5th Ed.).* Pearson Education, Boston, Mass.

Conway, Melvin E., 1968. How Do Committees Invent? *Datamation.* April, 1968.

Fenton, Norman & Martin Neil, 1999. A Critique of Software Defect Prediction Models. *IEEE Transactions on Software Engineering*, vol.25, no.3, pp.1-15.

Fenton, Norman & Niclas Ohlsson, 2000. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering*, vol.26, no.8, pp797-814.

Hatton, Les, 1997. Reexamining the Fault Density - Component Size Connection. *IEEE Software,* vol.14, no.2, pp.89-97.

Hatton, Les, 2009a. Power-Law Distributions of Component Size in General Software Systems. *IEEE Transactions on Software Engineering,* vol.35, no.4, pp.566-572.

Hatton, Les, 2009b. *Re-examining the Fault Density - Component Size Connection.* Online: http://www.leshatton.org/1996/05/re-examining-the-fault-density-component-size-connection

Hatton, Les, 2010. *Power-laws, persistence and the distribution of information in software systems.* Online: http://www.leshatton.org/2010/01/power-laws-persistence-and-the-distribution-of-information-in-software-systems

Hatton, Les, 2011. *Defects, Scientific Computation and the Scientific Method*. Online: http://www.leshatton.org/2011/10/defects-scientific-computation-and-the-scientific-method

Kahneman, Daniel & Amos Tversky, 1979. Prospect Theory: An Analysis of Decision under Risk. *Econometrica,* vol.47, no.2, pp.263-291.

Kahneman, Daniel & Amos Tversky, 1984. Choices, Values and Frames. *American Psychologist,* vol.39, no.4, pp.341-350.

Kahneman, Daniel, 2011. *Thinking, Fast and Slow.* Allen Lane, London.

Kelion, Leo, 2013. Why banks are likely to face more software glitches in 2013. *BBC News website,* 1 February 2013. Online: http://www.bbc.co.uk/news/technology-21280943.

Koru, A. Güneş, Khaled El Emam, Dongsong Zhang, Hongfang Liu & Divya Mathew, 2008. Theory of relative defect proneness. *Empirical Software Engineering*, vol.13, pp.473-498.

Lewis, J.P., 2001. Large Limits to Software Estimation. *ACM Software Engineering Notes,* vol.26, no.4, pp.54-59.

Mandelbrot, Benoit B., 2004. *The (mis)Behaviour of Markets: A Fractal View of Risk, Ruin and Reward.* Profile Books, London.

McConnell, Steve, 2011. What does 10× mean? Measuring variations in programmer productivity. In: Oram, Andy & Wilson, Greg (Eds.). *Making Software: What really works and why we believe it.* O'Reilly, 2011. (pp 567-573.)

Potanin, Alex, James Noble, Marcus R. Frean & Robert Biddle, 2005. Scale-free geometry in OO programs. *Communications of The ACM*, vol.48, no.5, pp.99-103.

Sassaman, Len, Meredith L. Patterson, Sergey Bratus &Anna Shubina, 2011. The Halting Problems of Network Stack Insecurity. *;login*, vol.36, no.6, pp.22-32.

Taleb, Nassim Nicholas, 2001. *Fooled by Randomness: The Hidden Role of Chance in Life and the Markets*. Random House, New York.

Taleb, Nassim Nicholas, 2008. *The Black Swan: The Impact of the Highly Improbable*. Penguin, London.

Wray, Stuart, 2010. How Pair Programming Really Works. *IEEE Software,* vol.27, no.1, pp.50-55.