Extreme Reliability

Programming like your life depends on it

Jules May

July 10, 2025

Extreme Reliability: Programming like your life depends on it

First published 2023 by Two Little Ducks Publishing, Montrose, Scotland

This version was published: July 10, 2025

Contact the author at:

julesmay.co.uk,

www.linkedin.com/in/jules-may/, or

medium.com/@julesmay

Contents

Table of Recommendations

To-do list

Acknowledgements

Despite the singular name on the cover, a book like this is never the sole work of one author, and it doesn't just appear, fully-formed, out of nothing. No, it builds on the totality of human knowledge (or at least, as much of it as one person can see), and then takes hundreds of people and thousands of hours to reach, if not completion, at least a natural hiatus.

Over a long career in the software business, I have had the pleasure of working with developers, engineers, and architects in locations across the US, Europe, and Asia, in domains from chip design up to enterprise architecture, many of them intensely safety-critical. Some of those people were truly talented, and were members of some of the highest-functioning and most effective teams on the planet. It was my privilege to have been part of their efforts; to be able to observe them in their natural habitats as they did what they did. From you, I learned not just what a high-functioning team was, but how you created code of the very highest quality. Because of you, this book's message exists. I wish I could identify each of you individually, but I'm afraid you must remain nameless. Thank you.

I have also been involved with teams and projects which, despite the undoubted skills of some of their members, were not so performant — some were positively delinquent! I noticed that many otherwise spectacular developers were trapped, unaware of what was going wrong, unable to change things, and constantly wishing for better. From you, too, I learned a lot. I learned what distinguishes high-functioning teams from everybody else, I

learned that poor quality was not merely an absence of good quality. Most of all I learned that you never gave up hope of a better way, and you began each new chapter filled with the optimism that this time it could be different. Because of you, this book has a reason to exist. Again, you must remain nameless, but: thank you.

To be fair, I've worked, too, with some really rubbish developers. From the clueless to the blindly dogmatic, from delusions of adequacy to hubris bordering on the Napoleonic: I have watched you repeatedly and systematically undermine the efforts of your more talented colleagues, without ever realising that's what you were doing. I learned from you the antipatterns that confound attempts at quality, and I learned how even the most idiotic and corrosive diktats can achieve virality. You, too, will remain nameless (I suspect, even you won't know who you are!) but, rest assured, this book couldn't have existed without you. In spite of everything: thank you.

As the ideas in this book took shape, I started to teach workshops and courses about them. Over several years, based on the feedback from my attendees, I learned the best way to express my insights, and the best way to construct my argument. Your feedback, both from the courses and from your experiences deploying these techniques, have helped shape this book into what it is today. Because of you, this book has evolved to its present shape. Thank you.

Will and I have known each other over forty years, since he was my first editor when I was writing for *.EXE* magazine. Back then, when I was still learning the craft of writing, he helped me to find my voice, and naturally it was to him I turned when I needed to hone the present work. Because of you, the words make sense. Thank you.

And finally: Samantha stuck by me throughout this book's extended gestation (nearly ten years!) When I was losing faith in the whole project, and allowing myself to be distracted by other schemes, it was you who egged me on, who said you just wanted to see it on the shelves and speaking to the world. Because of you, this book exists at all. Thank you.

Foreword

It's morning. You grab a cup of coffee, reach for your smart phone, and catch up on the latest news. Afterwards you check your email, look at today's schedule on the calendar app, and head off to work. Sounds like a standard morning — until your coffee maker suddenly quits in the middle of making your espresso, your news app only shows yesterday's news, the email app crashes every time it opens, and your calendar app shows no meetings for the day (yeah, right). Then your world is turned upside down. Why? Because we expect things like this to work — all the time.

We live in a world where we rely on technology for almost everything, and consequently we need and expect reliable software. However, writing reliable software isn't as easy as it seems. It takes practice, patience, skill, knowledge of patterns and anti-patterns, attention to details, and most of all, a different way of thinking about writing source code.

As a software architect I focus a lot on creating reliable architectures — those that have high levels of availability, fault tolerance, data integrity, and consistency. But architecture alone cannot create reliable systems — it requires the source code to be reliable as well.

You might think the topic of code quality and reliability is not relevant in the age of LLMs and AI code generation. On the contrary, this topic becomes even more important. Does your AI-generated code have any defects? Does it work for all possible scenarios, including those specific to your business problem? It's the responsibility of a developer not only to write reliable source code, but also to verify and revise AI-generated code to make sure it has the reliability you need. And that's where this book comes in.

Extreme Reliability is a book that takes you on a deep-dive journey into the art of creating reliable software. Unlike other code quality books, this is not a collection of recipes or lists to help you create more reliable code. Rather, this book changes your way of thinking about how to write reliable software.

I know what you're thinking — this is all well and good, but who has the time to focus on source code quality and reliability with increased pressure to deliver code faster and cheaper? Turns out we've been doing it wrong all along. This book shows you that accumulating technical debt is the same as introducing expensive defects into your code, and that refocusing and changing the way you think about writing highly reliable source code makes developing software easier and cheaper.

Enjoy your journey into writing high quality, defect-free code that works — all the time.

Mark Richards
Founder, https://developertoarchitect.com
October, 2023

Preface

One of the benefits of working (as I do) as an itinerant consultant is that I get to see the inside of a lot of companies. It has given me a breadth of experience — moreover, a perspective — which, as an employee, one could never achieve.

Every client will tell me that their business is unique; that its problems (which are unique to it) derive from its irreducible commercial uniqueness. They challenge me to invent the unique solutions which they – the experts in their uniqueness – have so far failed to find.

I agree! Every company *is* unique — just like all the others! Yet, when I scratch their surfaces, underneath that highly-specialised veneer I generally find a mere handful of recurring programming pathologies. Prime among them is the programming team which feels the need to introduce their efforts with apologies — the code is disorganised, or buggy, or too complex to handle, or just generally *smelly* — and this, they feel, lies at the heart of their manifest problems.

Perhaps my clients are a self-selecting cohort (after all, nobody hires a consultant unless they have a problem so hard that their own efforts have failed to alleviate it), but I have never encountered a team who believed they were operating at optimum efficiency, or that their product was in any way exemplary, or that they were actualising a technical ideal that each developer carried inside them. In contrast: most felt they were operating not as precision engineers, but as if on a battlefield, with shifting goals, unclear strategies, and incomprehensible and incoherent matériel.

Dear reader: does this sound familiar? Is it just my clients, or do most teams operate this way?

What I have noticed is that, in among the noise and chaos of most commercial development, programmers almost universally express regret for the mess, and carry a hope for a better world. After more than half a century of experience developing software (and half a century of thought-leaders writing about software development), we can mostly agree about the principles that good code and good development practice should follow. And yet, in the fog of real-world development, our code is still a mess (albeit a much larger mess than a generation ago). What's going wrong? Surely, software can be developed better than this?

What this book is about (and what it is not)

My message in this book — my promise to you — is: yes, there is a better way. There really are techniques and methodologies which will survive the exigencies and emergences of industrial-scale software development. Tucked away in unregarded corners of our industry, there are developers turning out high-quality software in a scalable fashion. I've worked with them, and studied them, and I've learned how they do what they do. Presented here is: this is how other great programmers have understood (and solved) their reliability problems, and this is what we can learn from them. Or: when disaster strikes other peoples' projects, this is how it happens.

There is no shortage of books about the craft of writing software. Possibly the two best known are *Clean Code* [Mar08] and *Code Complete*, [McC04] and there are many others. But we are not talking about craftsmanship here. For one thing: if you're reading this, then I'm going to assume that you already take your craft seriously, you have developed some experience, you know the difference between well- and badly-written code, and you are pretty settled in your tastes. You will not find anything along the lines of "Follow these conventions" – they're just conventions, no more significant than the choice of whiteboard or blackboard to discuss your solutions, and you're welcome to your opinion. But a craftsman's feel for software, while necessary to produce good, maintainable code, manifestly doesn't guarantee reliability either over the short term or (especially) the long. If we want to achieve the highest standards of reliability, we need something else as well.

There's more to reliability than just writing well-crafted code. Architecture is vitally important too (and, in today's dogmatically Agile world, is often regarded — wrongly — as an anachronism). But this book is not about architecture.[1] it's not even very much about architecture's little brother: Software Engineering (though in parts we will stray into some Software Engineering territory).

The environment in which code is written is also important. To what extent does the office environment assist or impede development? By "office environment" I'm referring to more than just the colour-palette of the bean-bags or the comprehensiveness of the cafe's menu; it includes things like the flavour of agility being used (and its authenticity), the test and automation strategies, and the values of the development managers. Get these wrong, and of course the code will suffer. But this book is not about those things either. Screeds have already been written on these matters, and I'm reluctant to add to that rising mountain of paper. As far as I can see, most of this dialogue boils down not to one method being more correct or even more effective than another; but more preferable, more or less to the taste of their participants.

This book is quite specific. Since, ultimately, the quality and reliability of our programs is controlled by the nature of the code we write, the book deals with the properties that code needs to have to be reliable and maintainable, and with the process of conceiving and thinking about that kind of code. This is a discussion of what code can be, and how to reason about it, much more than what the finished product needs to look like or how you go about making it.

My argument proceeds in five parts:

Part 1: Preliminaries talks about the absolute basics of software quality: the things that need to be true or meaningful for the concept of software quality even to make sense. Little of this is about code itself; it's more about how

[1]This one is: [May]

we approach the task of programming and what we conceive software to be.

Part 2: Principles addresses code more directly. It builds on code-craftsmanship ideas to set out some measures by which we can judge code and with which we can design it.

Part 3: Pathologies answers the question posed above: "What's going wrong?" The central message of this section is: what we have all believed to be best-practice (or even, only-available-practice) is actually a source of toxicity. It asks you to develop an intolerance of "what everybody knows".

Part 4: Patterns is the antidote to the previous section. It shows some higher-level patterns and strategies which, by their nature, promote highly-readable, highly-robust software systems.

Part 5: Practicalities talks about what happens in real development environments. After complaining that previously-mandated code-craftsmanship seemed not to translate into real workplaces, it seems only fair that I explain why these techniques are immune from the same fate.

Sprinkled throughout this discussion are a number of recommendations: "Do this", or "This is better than that". Taken individually, I'm sure that you'll find your programming will become easier and your product will become more reliable and more maintainable — you'll find that you can write better code faster. Taken all together, you'll see a coherent picture emerge of how you can develop *all* your software as if your life will depend on it, and what software could be if only we have the courage to reject the presupposition that software is so inherently suffused with original sin that it must invariably be defective.

Part I

Preliminaries

What needs to be true for “Software reliability” even to make sense?

1 Good code matters

"Ladies and gentlemen, this is your captain speaking. Welcome to the maiden commercial flight of the X700, the world's first entirely pilotless aircraft. No fallible humans on the flight deck here: everything on-board, from take-off to landing, is controlled entirely by computer, using software written to the highest standards by <*your company*>. Nothing can go wrong. Your flight plan is now programmed, so I'll return to the terminal and leave you in the capable care of the X700. Please sit back, enjoy the flight, and we'll speak again after your return."

How would you feel, sitting on that robotic aircraft? Confident? Nervous? I suspect "Get me off this thing" would be high in your consciousness.

The first time I heard this example was at a conference about software reliability. Of course, nobody felt secure, except one chap, a professor at one of the local universities. "I'd be entirely happy on that plane." he said. "If my students wrote the software, it would never get off the ground!"

Programming, let's face it, is tough.

Since we first started constructing software (in 1843!) we have been aware that the practice of programming is much more difficult than the theory. Time and again, our ability to imagine software has far exceeded our ability to make it. Over the years, there have been attempts to leverage our knowledge in other fields to make the process easier or more reliable, each time arguing that software is, in some way, like something we already know how to do. Thus we invented software engineering, software architecture, and most recently, agile (which really is software whittling). None has worked. Software is unlike anything else we do and remains, resolutely, difficult.

There are, I think, two main reasons for this. Firstly, the scale of a software project is bounded by nothing other than our ability to conceive. In the engineering disciplines, we can imagine all kinds of fanciful devices and technologies — from spaceships to teleporters — but if it's to be engineered, sooner or later we've got to make something, and at that point we find ourselves beset on all sides by basic physics, materials properties, and the costs and availability of resources. In short, while science-fiction authors can imagine transporting a million tons of matériel into outer space and constructing an orbiting hotel, engineers have to be concerned with the practicalities — how is all this to be accomplished?

Not so with software! We can imagine designing software at almost any ludicrous scale, and the raw materials are available in abundance. Even today, during the last sputterings of Moore's Law, we are constrained neither by memory, storage space, transmission speed, nor (except when confronting the most fundamental laws of mathematics) compute speed. In a very real sense, software specification *is* science-fiction.

Secondly (and partly — but not only — because of the previous point) we are limited in our ability to leverage skills from other fields because software is fundamentally unlike anything else we make. A machine from before the digital age might have a few hundred, or even a few thousand moving parts. In our digital age, machines have drastically fewer moving parts, and yet they perform far more flexibly, because the strain is now taken by software embedded into the heart of the machine. A typical control program might contain (to take a rough software equivalent of moving parts) several million decision points, typically one per line of code.

Each decision point in a program, of course, is capable of more significant consequences than a single hinge or bearing operating inside an engine — in fact, our decision points are more like mechanical failures than they are like mechanical joints — can you imagine a machine with a million failures every second? The fact is, we programmers are handling a level of fragility and complexity which has no parallel in any other technology.

Programming looks easy — no tooling, no heavy machinery, just a load of people in jeans pounding away at keyboards. And yet, we invest thousands or millions of man-years into creating our software products, only to see them malfunction (or, what is worse, lie ignored and unused) once they're released into the wild.

Can we be honest with each other, here in the privacy of these pages? Seen from the outside, we're not actually very good at our jobs. Software is a by-word for unreliability and unusability. Time and cost estimation of software projects is so notoriously difficult and arbitrary that some extreme versions of Agile counsel that we do entirely without either! We plan our projects in terms of releases, without even noticing that what is implied by that is: we're *expecting* to construct a substandard product, and we can't even tell how substandard it's going to be. We tell ourselves that ours is a young and immature field — there are many more lessons still to be learned than have so far been discovered — which may be so, but if the helicopter industry or the biotechnology industry (both of which are younger than ours) had our reputation, those industries couldn't possibly continue to exist.

And yet, for all its problems, software is extremely successful commercially. The world has a voracious appetite for software: some 13 million people are employed creating it, and perhaps 12 million more do it for fun.[1] The amount of software IP in the world (that is, original software, not copies of copies) is increasing at 30% per year, and rising. Within ten years, there will be more programmers than Elvis impersonators.

We have all become so used to the way software is built that we barely even notice any more how pathological it all is. Every user of technology simply expects it to malfunction on a regular basis. Even QA champions will admit, sadly: "We can never make it perfect" and then tell you to turn it off and then on again. it's not for want of aspiration: we live our professional lives in a perpetual state of dissatisfaction. I've never met a programmer who didn't think: "There must be a better way!"

Attempting to ease our pain, there is a plethora of books about clean coding or performance-driven development or some other quality-based trope which promises to make the whole field a little less ghastly. They all take

[1] https://www.developernation.net/developer-reports/de20

an approach something like this: programming is hard, so a bit of discipline will make it easier — whether it's naming your variables `_npExternalCounter` or your methods `somethingFactory`, formatting your comments so programs can read them, or obeying all the rules in a linter. These conventions are worth having: they help to co-ordinate the efforts of semi-independent programmers, they serve to label and propagate a team culture,[2] and to some extent lubricate the wheels of programming teams. I agree: they make teamwork easier.

But ultimately they represent largely arbitrary conventions, such as which currency we buy our lunch with, or which side of the road we drive on. None goes to the heart of why programming is hard, and manifestly none has ever made the difference between success and failure. That's the basic reason why, no matter how assiduously we study the regulations in these books, and no matter how carefully we review each others' code, when we're faced with encroaching deadlines and ill-conceived demands, we discard in a moment all this received virtue.[3] Suddenly, we regard religious rule-following as "gold-plating", something that impedes software development rather than supporting it. We know we're building-up technical debt, and we promise, one day, to repay it, even though we know we never will. Ultimately, the piety we all desire so much gets sacrificed on the altar of expediency.

And yet…

We know, because we have built TV studios and aircraft the size of small towns and constellations of satellites and continent-wide electrical power systems, that it is possible to build very large, very complex, highly-stressed systems that work pretty much perfectly, pretty much all the time. We know, because we have built engine management computers, and factory automation systems, and hospitals full of life-support machines, and (our crowning glory!) the Internet, that we can even build software that operates to the same exquisite standard. Even the commercial aircraft with which we started this chapter are controlled almost entirely by software. When it really matters, when peoples' safety depends on well-functioning software, we can do it. We can build software to which we can entrust our lives.

[2]Which of course is a good thing! See page 57.

[3]Go on! Admit it!

Quality matters, but so does quantity. Can we make good software quickly enough? If is it simply a question of taking enough time to do a proper job, then we're in a lot of trouble! We — the programming industry — are expected to turn out trillions of lines of software every year. How can we originate software, to the highest standards of correctness, reliability, and maintainability, at industrial scale? What do they do, the designers and craftsmen behind these almost-perfect systems, that the rest of us don't? Are there simple lessons which we can learn from them?

It turns out: there is an answer — there really is one! And, luckily for us, (as you will see) quality code doesn't demand enormous quantities of time or resource. Quite the reverse, in fact: good code costs less.

In 1993, Steve Maguire wrote "Writing Solid Code" [Mag93]. In it, he expressed his belief that it was possible to write perfectly-functioning code in plain old simple C, and then he explained, plainly and simply, how to do it.

Who would know better than the 10X and the 50X coders? These are the people who are not just faster than their colleagues, they're orders of magnitude faster. [McC; Tor23] How do they do it? For the most part: extremely reliably. The secret of their productivity is precisely their reliability.

What the 50-times programmers know, what Maguire said, and what I'm telling you here is this: you have to change the task of programming so that programming itself becomes easy and forgiving. Think about your programs differently. Use your tools differently. If the programming you're doing is too difficult, find something different to do that's easier. That's it!

Recommendation 1

Software quality is not a luxury

Be a yardstick of quality. Some people aren't used to an environment where excellence is expected.

Steve Jobs

I have worked in programming shops where managers actively encouraged their developers to write sub-standard code. They regarded their programmers' wish for decent code as being no more than an aesthetic revulsion to "code smell", they regarded bug-fixing as a drain on resources, and they regarded code clean-ups as "gold plating". Modern Agile encourages us to spend half our time refactoring, but how many shops actually do so?

Where does the idea of gold-plated code come from? Most project managers talk in terms of the "Iron Triangle", or the "Triple constraint". They draw diagrams like Figure 1.1 and say "You can improve any two of these at the expense of the third."

But, as with so many engineering principles which are extrapolated to soft-

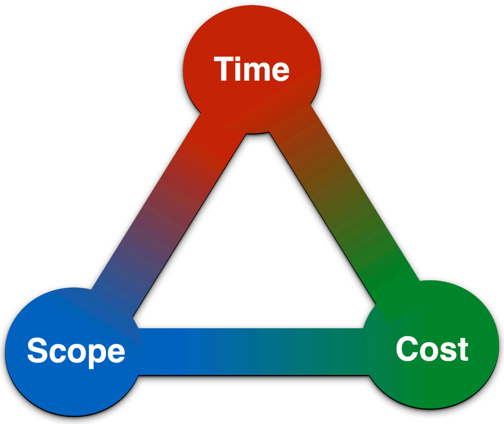


Figure 1.1: The Iron Triangle: You can have it all, you can have it now, or you can have it cheap. Pick any two.

ware, this doesn't really work for us. For one thing, there are only three constraints here: cost, time, and scope. Product quality doesn't appear among them.[4] In any other branch of engineering, a project which delivered an unacceptable quality is, quite simply, a failed project.

Our first mistake in the software business is: we generally don't define what acceptable quality is. How could we? Most quality specifications speak in terms like "Mean time between failures" or "Rejects per million products", but that statistical language makes no sense in software because software doesn't fail in the same statistically-predictable way that parts do in an engine. What on earth would a software quality specification even look like?

The second, and more serious mistake is: improved software quality doesn't translate into higher development cost. In fact, all my experience has overwhelmingly convinced me that the reverse is true — a proper job costs considerably less than a hack: doing it right takes less time and costs less money and is in every respect superior to knowingly doing it wrong.

Good code costs less

Most of us recognise poor software when it malfunctions — when we see a bug. But bug rates have shown themselves to be a very poor metric for software quality, mostly because the bug rate in most projects seems to stay pretty constant no matter how many of them you fix.[5] It turns out, in fact, that no matter how much you debug poor-quality code, you don't produce good-quality code; you just repeatedly demonstrate how bad the code is.

We will see over the course of this book where bugs come from (and if you want a sneak preview, take a look at page 192). For now, notice that bugs exist in programs because somebody put them there, usually by making a mistake of some kind. It is well-established, both by experience and research, that the longer the gap between making a mistake and discovering

[4] Some commentators introduce quality as part of scope — that is, the quality expected of a product is part of its requirements. Other commentators simply conflate the two, and call it "Goodness" or "Better". Yet others show quality inside the triangle, as if quality is a natural consequence of over-specifying, over-spending, and overrunning.

[5] See, for example, [Dor22]

How methodologies differ

Agile processes are constrained by the Iron Triangle just as much as any other process. For example,

- Scrum fixes the time available for development (in each sprint), modifies the scope sprint-by-sprint, and leaves the (lifetime) cost entirely unconstrained — the team keeps sprinting (and burning cash) as long as the backlog still exists.
- Waterfall, in contrast, fixes the scope, allows some flexibility in the cost (especially in cost-plus or time-billed contracts), but fails to constrain the time — it takes as long as it takes, and delays are just a part of the project.
- Finally, a very early-stage startup (which, incidentally requires minimal coordination), may have a fixed cost (the founders' savings), flexible scope (because they don't yet know what they want to build), but they want to optimise the time it takes to become investable.

Archetypally, there are six combinations of fixed, flexible, and unconstrained factors, and each one leads to a different methodology:

Time	**Cost**	**Scope**	Example methodology
Fixed	Flexible		Deadline[a]
Flexible	**Fixed**		Using up a budget
Fixed		Flexible	Scrum
Flexible		**Fixed**	Kanban
	Fixed	Flexible	Early-stage startup
	Flexible	**Fixed**	Waterfall

Which you choose is governed by the constraints imposed on the development by its environment. Agile methodologies aren't always the best.

[a]Such as preparation for an event

it, the more costly will be its remediation. (See, for example [Mar08, page 29]).

A common expression of this truth is depicted in pictures such as figure 1.2, which shows that an error which is introduced by a programmer, and which is immediately detected and fixed (that's the green bar in the middle block), might cost a few minutes additional work, but the same error which isn't discovered until after the software has been released into the wild (that's the green bar on the right) might cost a hundred or a thousand times as much to fix (note the logarithmic cost scale). A defect introduced at the requirements stage, and not discovered until after deployment, will cost something of the order of the entire project's budget. Hardly surprising: you've built the wrong thing!

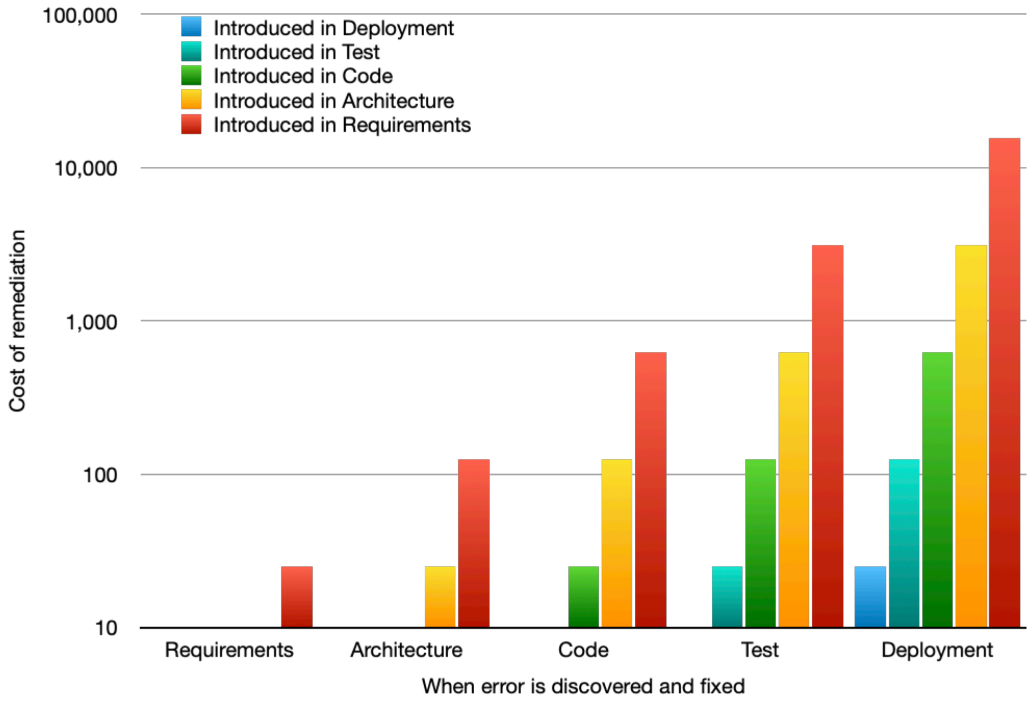


Figure 1.2: The longer a defect lies undiscovered, the more expensive will be its remediation

You may cavil about the names or number of project stages depicted in the diagram (they must seem anachronistically Waterfall to modern, Agile-adapted eyes), and you may quibble about the multiplication factor as you move forward in time (as [Bos15] did so convincingly in Chapter 9 and Appendix B), but the overall message is undeniable: hidden defects cost real money, and the better they're hidden, the more they're likely to cost.

It's worse than that. If your program is full of bugs, you're going to be awakening more and more of them as you develop elsewhere. They will reduce both your maintenance and your new-feature velocity. In other words, bugs get together and compound. That's why we call them "Technical debt".

The bugs that survive the entire process, and don't manifest until after deployment: they are the most costly of all. That's because in addition to the ballooning remediation cost, there's also the cost of the bug itself. You see: almost never is the point of a programming exercise to produce a piece of software. The software is there to supply some other need: an e-commerce system or a payroll system or an industrial control system or a telemetry system: the software is only one small part of the overall system. Even if no actual lives are at risk, these activities have value ranging from the significant to the overwhelming. If your e-commerce system leaks personal data about its customers; if the payroll system doesn't pay (or double-pays) your employees; if your Post Office system randomly deletes transactions from offices' accounts; if the industrial control system crashes your robots into each other; if your telemetry system is garbage collecting at the very moment the significant event happens: these malfunctions will impact your business's reputation, deliverables, and even survival. These are real and significant costs.

Fixing bugs is expensive and unreliable. Not fixing them can be disastrous. A comparatively modest investment in producing bug-free code will pay off many times over in reduced maintenance costs and fewer squealing customers. Defending your code's quality is never gold-plating, and if you can't afford to produce the best quality code, you certainly can't afford second-best!

1.1 What is “Software quality”?

So, if software quality is such an important desideratum, what exactly is it? It is certainly something rather more sophisticated than simply “absence of bugs”, and compliance with coding-standard legislation manifestly doesn’t help.

“Quality” is a vague and overloaded term, so it’s worth unpacking it. As with all machines, we want our code to take responsibility for doing something. As long as it does so, quietly and reliably, we have no problem. But if the code fails to perform its function, or if it stops what it’s doing and seeks human intervention to help it perform its task, we perceive that as a lack of reliability. It’s a failure of trust.

What we care about is compliance with some (explicit or implicit) functional specification. We expect the code to do *this*. Part of the problem with software is that *this* is not merely poorly-defined, it is also changing — week-to-week, month-to-month, and year-to-year. It’s not enough that the software behaves as we expect it to; it must do so even when placed under stress, and even when our expectations of what it is supposed to do are changing and inconsistent.

Defects+malfunctions = bugs

To a first approximation, we estimate the quality of a piece of software by the number of bugs we encounter. To be more precise, we have an idea of how the code should behave, and when it behaves differently, we perceive that it has malfunctioned. That malfunction is what we call a bug. But notice this: we think of a bug as a thing, and yet malfunctioning is clearly a process. The malfunction itself cannot be the bug.

Obviously, every malfunction is traceable to a defect — an error of some kind — contained somewhere in the code; maybe more than one. That defect is the cause of the malfunction. But it’s not the bug either.

A bug is actually the combination of a defect with its associated malfunction. To get a bug, you need both.

This distinction matters for two reasons. Firstly, it tells us something about why we're so bad at debugging! It seems that no amount of debugging improves the quality of our software (see page 373) because most debugging concentrates on the malfunctions. We add code to suppress the malfunctions, when we should be chasing back to the defects and removing the code which contains them.

Secondly, while every malfunction is associated with a defect somewhere,

Why defects matter

If bugs are malfunctions which are caused by defects, then there's a corollary: anything that could cause a malfunction, but hasn't done so yet, is, by definition, a defect.

Figure 1.3: The Christmas Pudding model of bugs. Only a few malfunctions are visible from the outside; most defects are hidden away inside

not every defect causes a malfunction — in fact, most programs contain very many more defects than malfunctions, and most defects lie undiscovered and unsuspected for years before some unanticipated condition or enhancement triggers one. Those hidden defects, they're not bugs, because they haven't yet caused any malfunctions. But they go right to the heart of our sense of code quality.

To illustrate the distinction, consider the following code. The idea is to write a procedure which computes the average of an array of integers:

```
double average (array <int> nn) {
    int t=0;
    foreach (n in nn) t += n;
    return t / nn.count;
}
```

and we can test this function with cases such as:

```
average ({1, 2, 3, 4}); // returns 2.5
average ({4}); // returns 4.0
average ({0.001, −0.001}); // returns 0.0
```

When we deploy this into our program, it works entirely as expected: the containing program always computes the correct arrays of integers, and average always works properly. There's no malfunction, here; there's no bug. We should be happy!

But there is a defect. What happens if we test our function with cases such as this:

```
average ({});
```

or this:

```
average (null);
```

In these cases, the function will fail at (respectively) the division, or the enumerator. But this doesn't matter to us, does it? Because the program never invokes the function this way.

But now, imagine somebody modifies our fully-working program such that, by accident or design, it calls average in one of these pathological ways. It doesn't fail at the site of the change, and it's not guaranteed to fail at all (if the rest of the program usually gives it what it wants). It will fail randomly, occasionally, and not at the site of the change which triggered the malfunction. This is an example of the kind of sneaky bugs that we (or worse, our customers) discover every day.

Now, rewind again, to before we've made the change, to that nostalgic era when, by good fortune, we're always calling this function correctly. What we have, here, is not a bug, because there's no malfunction. What we have is the *potential* for a malfunction, a malfunction-in-waiting. A defect.

There is an unspoken rule, here: average must be called not merely with an array <int>, it must be called with a non-null, non-empty array<int>. And (unless you're using contracts — see page 99) there's no way to express that requirement to the compiler. It would seem that the core problem we have here is not with the body of the function, it's with its signature. What we *really* want to say is:

```
double average (array <int> nn | nn.length >= 1) {
   … etc
}
```

One way to cure this defect (and as we'll see, it is far from a good way) is to provide guards at the top of the procedure. Like this:

```
nullable <double> average (array <int> nn) {
      if (nn==null) return null;
      if (nn == {}) return #NaN; // or, possibly, exception
      int t=0;
      foreach (n in nn) t += n;
      return t/nn.count;
}
```

Note that we've had to change the return type of the function. If we call average(null) then the only possible, sensible return must also be null.[6] This

[6]If you're not immediately persuaded, the complete explanation starts on page 247

change will ripple through the rest of the program like falling dominoes, which shows just how significant this defect was.

The other change we've made is to return #NaN[7] when the list is empty. It wouldn't be appropriate to return null for this case, because it's not a null. The answer really is a double; we just don't know which one.[8] #NaN is entirely the correct way to express this uncertainty.

Exterminating defects

Counting defects (if only we could see them) would be a much better estimate of code quality than counting malfunctions, and infinitely better than just gesturing vaguely towards some notion of technical debt. But we're presently unable to conceive any repeatable measure of code quality. The problems are, of course:

- that we have no way to measure technical debt;
- that testing can elicit only malfunctions (and, in practice, it's not even very good at doing that — see page 332); and
- that we can't see the defects at all.

If we're to create defect-free code, we have to do it not by taking defects out, but by not putting them there in the first place.

I'm not the only person saying this. "Build Quality In" is the second of the Poppendeick's seven principles of Lean software development (right after "1: Eliminate Waste"). They explain:

> "Your goal is to build quality into the code from the start, not test it in later. You don't focus on putting defects into a tracking system; you avoid creating defects in the first place."[PP06, p25]

[7] We'll talk much more about #NaN and what it means on page 256

[8] To understand why, consider this: When we divide zero by zero, we're trying to find x such that $x = \frac{0}{0}$. That means that $0x = 0$, and of course every x satisfies that. So, $\frac{0}{0}$ is a number, but we don't know which one.

Did you notice the *other* defect in the sample code?

And that’s an idea with an illustrious history. No less an authority than W. Edwards Deming (the inspiration for the Total Quality Management movement) counsels, in the third of his fourteen points for management:

> “Cease dependence on inspection to achieve quality. Eliminate the need for inspection on a mass basis by building quality into the product in the first place.”[Dem88]

How to do that? In one sense, that’s what the rest of this book is about. But, for now, I’m going to introduce four core principles: “Zero tolerance for defects”, “The best code doesn’t exist”, “Dangling”, and “Failure is not an option”.

Recommendation 2

Zero tolerance for defects: the Broken Window principle

Maybe we should be sweating the small stuff

Jeff Attwood, "*The Broken Window Theory*", Coding Horror,
https://blog.codinghorror.com/the-broken-window-theory/

In 1982 two economists, James Wilson and George Kelling, introduced the "Broken Window" principle [WK82]: an observation that the decline of once-proud neighbourhoods into crime-sodden ghettos begins with minor infractions which aren't remediated. Locals can't take pride in neighbourhoods that are already run-down, and decline feeds off itself. So, if the streets are already full of litter, then one more discard makes no difference — then one more, then one more, until eventually the river of litter is how the neighbourhood identifies itself. The conclusion: ghettoisation begins with a few broken windows.

Starting in 1985, a number of officials in New York tested the theory by introducing zero-tolerance policies for minor infractions, first on the subway and eventually across the city. It worked: a 2001 study showed that crime of all kinds had fallen sharply and significantly. Similar results were achieved in other cities around the world.

Bad software is exactly the same: it's constructed one defect at a time. If we regard each defect as a broken window, and if we take a zero-tolerance approach to these defects, then such decline cannot take root. We can make sure our programs remain nice places in which to live and work.

But to do that, we need to give up one of our more cherished and fashionable ideas. Technical debt: it's a really good way to explain to managers the need for code maintenance, but it will do you no favours if you make the mistake of believing in it! For one thing, how on earth do you measure technical debt? Malfunctions per hour? '*Yeuchs*' per module? Employee turnover? The term suggests, too, that technical debt is a fungible mass, like sand or horse poo, which you can add or remove by the shovelful, but never totally mop-up.

Technical debt is the aggregation of defects, more like “litter” than it is like discards, more the character of the code than the individual misdemeanours of developers. But the fact is: it is made up of discrete defects, each one unique in its nature, camouflage, and destructive power. By the time you’ve got enough of them to be worthy of the name “Technical debt”, you’ve already got a problem.

The thought: “This kludge is introducing a bit of technical debt” — especially if the code is already carrying a mass of the stuff — elicits a very different reaction to the thought: “it’s introducing fourteen different defects”. And it’s a different reaction again if you think your code is (or is nearly) defect-free: you’re going to be very reluctant to introduce even one. That’s how the broken windows principle works. And that’s why good code demands not that you minimise technical debt (neither today nor tomorrow) but that you address (and root out) each and every defect individually, before you even put it in.

Recommendation 3

The very best code doesn't exist

It seems that perfection is attained not when there is nothing more to add, but when there is nothing more to remove

Antouine de Saint Exupéry, Terre des Hommes, 1939

So, quality code is not absence of *malfunctions* but rather absence of *defects*. But if most defects are invisible, how can we tell how defective the code is? Answer: good code has fewer defects because it has fewer places where defects can possibly hide. This is not about comprehensive testing or code-coverage or any of that. This is a much more basic idea: if we select our languages and our tools and our programming hygiene such as never to introduce defects or (having introduced them) to pinpoint them immediately — such that it is not even *possible* to introduce defects unknowingly — then our code will necessarily be defect-free. We may have to change slightly our notion of what programming is in order to achieve that, but it is trivially true that, if we can't put defects into our programs, there won't be any in there.

Defects live in code. The less code there is, the fewer defects. The most reliable code, and the most performant, is the code that doesn't even exist, because non-existent code never needs to run, doesn't allocate memory, cannot harbour defects, and cannot malfunction. The highly-productive programmers that we mentioned earlier: the way they do it is to write much less than everybody else. They are constantly seeking code that doesn't need to be there, and they ruthlessly cut it away.

Kurt Vonnegut [Von85] described the process of writing thus:

> Every sentence must do one of two things — reveal character or advance the action.
>
> Be merciless on yourself. If a sentence, no matter how excellent, does not illuminate your subject in some new and useful way, scratch it out.

The same advice applies to programming. Be brutal! Be ruthless! Every deletion sweeps away more defects, and as your program becomes leaner, the core problem will gradually materialise before your eyes, with its irreducible detail and intrinsic complexity clearly visible. You'll get to really understand your problem. Good programming is good editing, and while really good programmers can write more code in a day than anyone else, the world's very best developers take code out.

Recommendation 4

Good code dangles

Notice that the stiffest tree is most easily cracked, while the bamboo or willow survives by bending with the wind

Bruce Lee

Eliminating the fat from our programs helps to reduce defects, but probably won't eradicate them entirely. But so what? Plenty of imperfect things work just fine, because they're designed to flex when they're stressed. Cars, for example, are designed such that when they roll over a bump in the road, the steering might deflect a little, but the car itself will push back against the bump, and in very short order the car will return to its original course. You can even take your hands off the steering wheel, and most cars will recover their stability all by themselves. Aircraft are designed the same way: after being disturbed by pockets of air, they will return to straight and level flight, even if the pilot is in the toilet.

But our programs aren't like this. Most programs are extremely brittle, and even minor disturbances will knock them completely out of control. Just one null pointer in an irrelevant backwater of the code can bring an entire program crashing to a halt. That's unforgivable! Why don't we assume that malfunctions will happen (either because of internal defects or because of external stressors), and why don't we design our programs to be stable in the presence of those malfunctions?[9]

Edsgar Dijkstra invented the idea of **self-stabilising code**.[10] The idea here is that programs may experience odd and unexpected conditions for a whole variety of reasons, but the action, on discovering an error, is to try to put it right — not major system-wide surgery, just correcting the instant error.

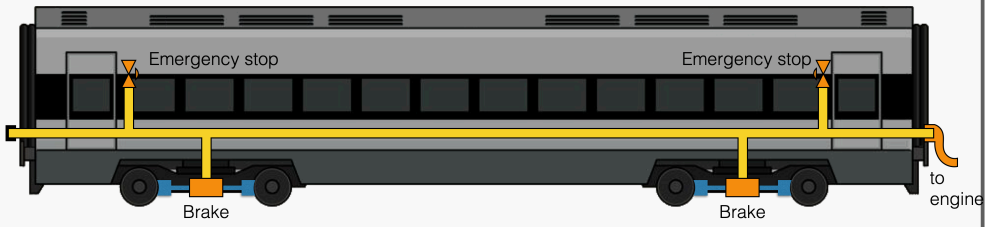
[9]Oh, we do! We throw exceptions. See page 174 to see why exceptions are such a terrible idea.

[10]Not to be confused with Robust Programming, which is actually highly defensive programming, that distrusts everything outside its boundary and throws exceptions at the first sign of naughtiness. This is just as brittle as conventional programming, but it does at least allow a programmer to say "You broke it! Not me!"

The concept of fail-safety

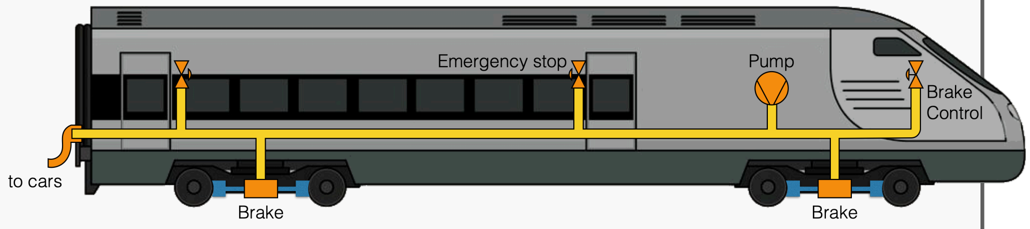
Has it ever occurred to you to wonder why train carriages don't roll away when they're left alone? It's because they have fail-safe braking. It works like this: there's a pipe that runs the whole length of every carriage, and is open at each end. When the carriages are joined into trains, the pipes are all connected together to make one long pipe that runs the whole length of the train. The open end at the back is stopped off, and the open end at the front is joined to the braking system in the locomotive.

Now, when the driver wants to move the train, he engages an air pump which pressurises the air in the pipe, and the pressure disengages the brakes in all the carriages. Now the train is free to move. When the driver wants to slow the train, he lets a little air out of the pipe, and the brakes re-engage in every carriage along the whole length of the train. If one brake calliper fails, there's another dozen taking up the slack. And if, for any reason, a link breaks or a carriage becomes disengaged, the pressure in the pipe fails and all the brakes come on in all the carriages, instantly.



This is what fail-safe looks like. It's a system which normally sits in a guaranteed-safe ground condition. Normal operation will lift it out of that safe condition, but if there's a failure somewhere then the rest of the system naturally falls back into the safe zone. Fail-safes, to make the point absolutely clear, are not additional devices; they are what the system does when the additional devices don't work properly. Train brakes are fail-safe; car brakes (which use pressure to apply the brakes, rather than to release them) are not.

You can see how this applies to software. Most software systems are very much not fail-safe — they tend to propagate their errors and amplify malfunctions. That's why software feels brittle. But if the ground-state of the software is a low-complexity, safe state, and it is the operation of the software that lifts it out of that state, then malfunctions will be stabilised and made-safe before any real damage gets done. In other words, it's the happy path which should be exceptional, not the exigencies. It might not work perfectly, but at least it won't break.



It's the "boy-scout principle" that you already know from programming — leave it a little cleaner than you found it — but this time we're talking about the behaviour of the program, not the behaviour of the programmer.

A similar idea resurfaced in the approach which became known as **contract-based programming** [Mey97]. Here the program declares explicitly what properties a system needs in order to be stable, and the contracts ensure that disturbances from those properties are either minimised or contained.

More recently Nicholas Taleb introduced the concept of "**anti-fragility**". [Tal12] He distinguished between:

- fragility — the propensity of a system to malfunction under stress (what we call brittleness);
- robustness — the propensity of a system to keep going regardless under stress (which in our terms is just brittleness with someone else to blame); and
- anti-fragility — the propensity of a system to accept an exigency and absorb it and become stronger under its influence (which is something like Dijkstra's self-stabilisation, but not quite).

We discuss the idea of **fail-safety** in the sidebar.

All these things are philosophical approaches to achieve the same idea: we want to make our programs stable in the presence of errors and exigencies and malfunctions, not to exist always in fear of them.

To illustrate these ideas, consider the following:

```
class Employee {
    Guid id;
    HumanName name;
    Employee manager;

    string toLongString = "{id}: {name.fullName}, reports to
         {manager.name.fullName}";
}
```

There's a lot we could discuss about this tiny fragment, but for now, I want to concentrate on one obvious defect: that manager may be null. In this case, toLongString is obviously unsafe.

The traditional way to handle this would be to add a null-testing safety-interlock, like this:

```
class Employee {
      [ … ]
      string toLongString = {
            string rv = ”{id}: {name.fullName}”;
            if (manager != null) rv += ” reports to {manager.name.fullName}”;
            return rv;
      }
}
```

which shows that this 'special case' has actually tripled the size of the function, and has hidden the otherwise perfectly-simple straight-line logic in a tangle of string processing and conditionals. It's also not reliable: we have to add the same special-casing to every function which uses manager, and there's no way to be sure that we've got every one. Ubiquitous this approach may be, but it's not good.

There is another approach, though. What kind of an employee would have no manager? I can think of lots of examples: a founder; a board member, a new-recruit who hasn't been fully on-boarded. My point is: not having a manager is not a special case at all. It's actually the base, safe case, and it's only when you start building reporting lines that things start to get complicated.

One way to express this (and, again, as we'll see, it's far from a good way) is to encapsulate the idea that an employee with no manager is responsible only to himself. This is a simple example of a fail-safe:

```
class Employee {
    Guid id;
    HumanName name;
    Employee manager = this; // initialised at construction time unless
        overridden
    […]

    string toLongString = ”{id}: {name.fullName}, reports to
        {manager.name.fullName}”;
}
```

or we could encapsulate an altogether more sinister idea: that the employee is responsible to nobody at all:

```
class Employee {
    static Employee nobody {// null object. Vide page 127
        id = Guid.zero;
        name = HumanName.rockStarName (”Nobody”);
    };

    Guid id;
    HumanName name;
    Employee manager = nobody;

    […]
    string toLongString = ”{id}: {name.fullName}, reports to
        {manager.name.fullName}”;
}
```

In these models, manager is never null. It always refers to a real Employee[11] and the employee it refers to expresses quite well the philosophical idea of responsibility that the organisation believes in. There is now no special-case: the default position is that every Employee is responsible to somebody, even if that somebody is his own conscience. That’s fail-safe.

[11] Well, not unless some fool explicitly sets it to null. This simplistic example is merely fail-safe, not completely idiot-proof or un-vandalisable.

Summary

Whichever philosophy you hang it off (self-stabilisation, contracts, anti-fragility, or fail-safety), the principle is simple, and obvious: if your program may be unbalanced by something, it should be able to absorb the disturbance, and then spring back into shape, or get back on-course, with the minimum ceremony. This basic principle has been known to every other branch of engineering for generations, as has the alternative — that the system instead of absorbing the impact, is broken by it.

One way to visualise this is to think of your code as a column: a series of bricks placed one above the other. You know, without being told, that as you pile the bricks up, the slightest shove will bring the whole edifice crashing to the ground – Jenga! That's because each brick relies on the one below it for support — a small deviation near the bottom will get magnified towards the top of the tower, until the whole thing collapses.

But the tower can also be constructed the other way, with each brick *hanging* from the one above. Now, a small perturbation at the bottom might affect the one above slightly, but generally the perturbations will die away relatively quickly. Even a perturbation near the top will be suppressed by the weight of the blocks below.

Code doesn't need to be perfect for it to work perfectly. It needs to be *stable*. Code that's written to be inherently stable needs only to be "good enough". It will tolerate both internal defects and external exigencies and return to correct operation without any further intervention.

Recommendation 5

Failure is not an option

Never give up. Never, never, never.

Winston Churchill

Programming, as we've discussed, is difficult. The main reason it's so difficult is because there's a brittle quality to it: it feels fragile, touchy, and unpredictable. Everything needs to be exactly right for the program to work properly. We've all experienced "Well, it works on my machine", followed by a frantic search to find out what needs to be changed on the production machine to satisfy the program's demands.

I think there's a reason for this brittleness, and it's got to do with the unique nature of the computers on which the software runs. The thing that made computers revolutionary, that other kinds of machine do only seldom, and what gives software its power, is: the ability to take decisions. At its heart a computer executes its instructions, one after the other, taking data from *here* and placing results *there*, until it comes to a decision point. At that moment, it needs to choose: keep going as it has been doing, or divert to somewhere else entirely. This diversion may be trivial, or it may completely change the nature of the task being executed. These branches, they're how a computer can react to its environment, but there's no limit to their *over*-reaction. Decisions take the place of moving parts in a physical machine, but they don't work like moving parts, they work more like breakages, and they happen millions of times a second! Can you imagine driving a car which had a million breakages every second? No wonder software is so brittle!

Every other engineering discipline carefully designs its products to be stable; variation-diminishing. When you drive a car over a bump in the road, you'll feel the bump, but generally the car will return to its original course without any input from you. An aircraft, disturbed by a pocket of air, will return to its straight-and-level flight even if the pilot is in the loo. A bridge, buffeted by gales, will absorb the energy and damp the resulting vibrations. These things are designed to be stable under stress, to recover from anticipable impacts.

But software is designed to do exactly the opposite — it's actually carefully contrived to go wrong on the merest pretext! Pretty much every time we call some function to perform some action, we need to check that the function has completed properly.[12] But, if it turns out the function has errored: what do we do with that knowledge? Do we try to stabilise the system? No: having been let down by one of our dependencies, we in turn have to renege on our promise. We return an error of our own, or we throw an exception to our caller, or something like that. In other words, we divert onto a completely different track, we propagate the malfunction further into the program, until something, somewhere is prepared to abandon the task altogether and report the error to the user or to the system's operators. This is, without any trace of irony, considered "graceful" error handling!

Let me show you an example of what I mean. Listing 1.1 shows a small code fragment showing a typical modern approach to opening and reading a file.

Let's take just one line from this. Suppose the file-opening doesn't work properly. That's a failure, and thus the file-reading task that depends on it must also fail. In this view, an exception is entirely the correct way to signal that failure.

But there's a problem with this approach. This failure (and all those other things that can go wrong with the file): why should they be the caller's job to fix? Even if the FileReader's errors were fixable, by the time the exception has bubbled up to readFile (or to whatever has called it), the chance to fix it is long-gone: FileReader (and all the intervening state) has been obliterated by the stack's tear-down. The only option available to the exception handler is to abandon the task, perhaps attempt to restart it from scratch, and send some cryptic, system-level message to the user.[13]

[12] We don't check the result itself: we check some metadata that's been provided to us by the function — an error code, for example — so we have to trust that the function is reporting honestly.

[13] And if that isn't enough to convince you that exceptions are a terrible way to solve this problem, there's a lot more about this pathology starting on page 185.

Listing 1.1 An example of a "well-checked" file reader

```
void readFile (string filename) {
    reader = FileReader(); // might throw: out of memory,
    reader.open (filename); // might throw: not found, permissions error, etc
    try {
        while (!reader.atEof()) { // might throw: file disappeared
            line = reader.readLine (); // might throw:
                // file has been altered by someone else,
                // or, the file is binary and we can't find eol markers
            processLine (line);
        }
    }
    catch (e) {
        rethrow e; // tell the caller about the failure
    }
    finally {
        reader.close(); // let's hope this doesn't throw!
    }
}
```

This, right here, is how errors domino into brittleness — interpreting perfectly reasonable states as errors, amplifying them into failures, and propagating them outwards, beyond any possibility of remediating them.

Surely being unable to read a file is a perfectly anticipable state for the overall system to be in? Not being able to open a file should not be a failure; it shouldn't even be an error. It should be part of normal running. We may interpret it as a bump in the road, but our main aim should be to get back to normal operation at the earliest possible opportunity, just as we discussed on page 25. Ideally, readFile shouldn't even need to care whether the file opened correctly or not!

How would it be if a unopened file behaved identically to an empty file — that is: it asserted eof immediately? Now, the above code reduces to this:

```
void readFile (string filename) {
    reader = FileReader();
    reader.open (filename); // always works, even if no file or no permissions
    while (!reader.atEof()) { // returns true if any error
        line = reader.readLine (); // returns "" if any error
        processLine (line);
    }
    reader.close();
}
```

In this code, there is no sense of failure, nor even of an error: every different eventuality is handled on the same happy path. No failures means there's nothing to start a crack in the program, and no exceptions means there's no way to propagate them even if they do start.

To be absolutely clear: we're not pretending that nothing can go wrong. A client that cares why a file appears to be blank can ask it (which we'll explore in more detail beginning at page 59). But checking it is entirely optional because, even if something does go wrong, we already have logic in place to handle the "fail-safe" case.

Conventional software is brittle because, deep down in its DNA, it's filled with tiny errors constantly happening. Our normal error-handling approaches treat every error as a show-stopper, give up whatever they were supposed to be doing, and amplify each error, like cracks propagating through a wine glass until the whole thing shatters.

If we construct our programs such that those micro-cracks don't start, there's nothing to be amplified; if we design the handling of whatever errors remain so as to get back on the original course as soon as possible, the crack can't propagate any further. This is the very essence of reliability.

If you take only one thing from this book, let it be this: software failures happen because we design our software to fail: we deliberately and carefully put failure modes into our code — they're not even like bugs, which we put there accidentally!

But we have a choice: we can instead design software to not fail (and in reliable code, it's not even a choice: failure is not an option). To do that, we need to remove the failure modes from our code, and stabilise any that we inherit from our dependencies. Instead of giving up at the first sign of trouble, we need our programs to keep going anyway. It's really as simple as that!