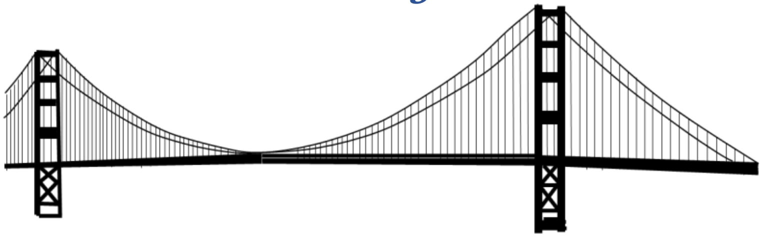
THE GEODESIC MANIFESTO

Essentials of Software Development for the Post-Agile World



Bob Erickson

Contact me at bob@GeodesicManifesto.com.

This first edition needs feedback. Contact me with your comments or visit my blog at https://www.GeodesicManifesto.com. I would appreciate honest reviews on Amazon, or anywhere else reviews are posted.

1 BRIDGES TO SOFTWARE

Humankind has built bridges for thousands of years. I don't mean the metaphorical bridges that span the gap across our political divide; I mean real bridges that make it possible to walk across water without getting that squishing sound in your Converse™ high tops. In Paris there are 37 bridges across the Seine. In London, 32 bridges across the Thames. There are dozens of bridges that cross the Rubicon. We build rope bridges, stone bridges, suspension bridges, railway trestles, and covered bridges with year-round autumn foliage. We build bridges over rivers, canyons, lakes, railroads, roads and the English Channel. Whoops! that's a tunnel.

Bridges don't fail.

OK, occasionally they fail, but we can name the failures because they are so rare: the spectacular Tacoma Narrows bridge failure (look it up on YouTube if you haven't seen it); the Interstate 35W bridge in Minneapolis; the San Francisco Bay Bridge (but there was an earthquake involved). And London Bridge has been "falling down, falling down" for ages. Bridge failures make the news.

But mostly, they don't fail. Cars, trains and people cross them every day without a moment's thought concerning whether this will be their last crossing. If they're about to fail, the authorities close the bridge until it can be repaired. If we measured bridge failures compared to the number of crossings, it would be in the parts per billion. Or less.

Software fails every day.

Like bridges, there are big failures that make the news – the Facebook and Yahoo data breaches, for example – but mostly, software failures are so commonplace we hardly notice them. When our Words with Friends™ app suddenly quits, or our PC spontaneously reboots, or Excel™ freezes in the middle of a calculation, we shake it off, briefly wonder if it was something we did, and then start over, hoping we can pick up where we left off. The coffee machine at the office reminds me every day – I push the buttons for a double decaf (I know – why bother?), the machine pours the first shot, then displays "Ready. Take your Cup," while proceeding to pour the second shot. I guess the decaf QA engineer was too sleepy to notice.

Why is software so bad? Do we need thousands of years of practice before we can get it right? Or is there something inherent in the software problem that makes it harder than building bridges or cars or airplanes? More practice will help, but the software problem is exceedingly more complex than bridge design. And in many cases, the economics don't make it worthwhile to build better software. And software is so easy to change that it's never complete and the next version isn't better, just different. And as the software system grows, it becomes harder and harder to make the next version, until finally we give up and start over, knowing that we have the same chaotic life cycle ahead.

In this book, I will explore the problem of software development – not from an academic perspective, but from the viewpoint of someone who has spent thirty-plus years immersed in software development. My career has gone back and forth between software development and software management, and with each transition, I have learned more about what I should have done better in my previous role. I have made a lot of mistakes and I learned from that. I have done a few things right and I learned from that. I have seen it done well and I've seen it done poorly, and I learned from that. And I learned from writing this book – the act of expressing your ideas will often lead to better ideas.

This is not a book about how to write code – there are plenty of those and I have nothing significant to add. This is not a book about algorithms – how could I hope to out-algorithm Donald Knuth?

This is not a book about Agile, which is an incomplete and often inadequate way of thinking about software that has caught our attention because the Old Way was so bad, and because the founders chose a perfect name. I will play the role of the guileless boy in *The Emperor's New Clothes* who is the only one with the audacity to declare that the Emperor is naked.

I propose a new manifesto, the Geodesic Manifesto, that supplies a wardrobe for the Agile vision and addresses the problems that have blocked the way for many Agile hopefuls. The Geodesic Manifesto forms the backbone of this book. It's short – fits on one 8½ x11 page, but it's dense, with many concepts that take long chapters to describe.

This is a book about why. This book discusses how to think about software. It sets forth several models for evaluating software development and discusses the philosophy I have forged during my career for how to apply those models. It looks at many facets of the problem of software development and attempts to abstract each facet into a few key concepts.

After you've read the book, you will be able to apply the concepts to your own software problem, or at least console yourself with the knowledge that you have a very hard problem. If I could give this book to my twenty-five-year-old self, the entire course of history would have changed. A little.

This book is targeted at anyone who wants to improve how software is developed. Managers will learn about leading their team but also about designing architectures and solving challenging problems. Developers will learn about development processes and their role in the organization.

For the structure of this book, I have chosen this conversational, first-person style. The "you" I am talking to is usually a software manager, but sometimes you're a software developer. I'll try to make it clear when I switch.

The book has a main set of sections that discuss facets of the software problem, and, interspersed throughout the text, are sections that I label "*Non-Sequitur*" that consider related topics that are only loosely connected to the main sections. You can skip these if you want or read them in a different order or you can skip the main thread and only read the non-sequiturs. If you bought, borrowed or stole the book, I thank you. You owe me nothing more.

Each chapter starts with a *non-sequitur* and concludes with a "Key Concepts" section that reviews the material in the chapter and provides cross references back to the discussion. The book is heavily cross-referenced using hyperlinks. If you're reading on paper, you'll have to traverse the hyperlinks the old-fashioned way – use the index.

In this chapter, I will introduce some of the key concepts that serve as background for the rest of the book. Some concepts are pervasive in software development such as requirements and a brief overview of graph theory. I present three models for understanding software that later chapters will reference frequently. I present the Geodesic Manifesto and provide a brief explanation with references to further information in the rest of the book Finally, I list the characters in this story.

A Note on Gender

For years, the pronoun 'he' represented a person of unspecified gender when its antecedent was a person of unspecified gender. In the sixties this usage started to unravel as women rightly objected to the default masculine pronoun. Over the last fifty years we have become increasingly aware of gender to the point that today even assuming that there are only two genders can trigger objection.

But the English language has not evolved. The only singular pronoun with unspecified gender is 'it'. You and I would both object to being referred to as 'it'. Common usage is evolving to use the word 'they' as a singular pronoun referring to a person of unspecified gender. I'm sorry, but the training I received from Sister Eucharia's ruler and Mrs. Hurd's red pen is too deeply embedded for me to adopt that usage. So, until we add a new pronoun (I recommend the word 'che', rhymes with 'the'), I'm stuck.

So, when I have to use a singular pronoun to refer to a person, I randomly choose one of my fictional characters: Jack and Jill, Bonnie and Clyde, or Victoria and Albert. Then I use 'she' or 'he' as appropriate. I hope you can abide that.

1.1 *Non-Sequitur*: Lessons from "The Imitation Game"

Part of my inspiration for this book came from the movie, "The Imitation Game". After watching it, I felt sympathy for Commander Denniston, the poor bastard who was responsible for managing the team of mathematicians, chess masters and puzzle solvers breaking the German Enigma code at Bletchley Park during WWII. Alan Turing, the brilliant mathematician, played by Benedict Cumberbatch, has built a machine that he claims is Britain's best hope to crack the code. The machine impressively whirrs and spins and ka-chunks away but has yet to crack a single coded message. Commander Denniston, played by Charles Dance, is the frustrated manager of this early software project. Of course, he doesn't call it software because the word wasn't applied to computer code until much later, but it has all the earmarks of a software project:

- It's late.
- No one can predict when it will be done.
- All previous predictions have been wrong.
- No one can define the next milestone – in fact, no one thought to ask.
- The system architect (Alan Turing) is a notoriously poor communicator.

Commander Denniston is a successful leader trained in Britain's military. He is an expert in command-and-control organizations and in logistics. He expects that when he gives an order it will be obeyed. He considers that all members of his staff are equivalent -- after all, one soldier is much like another. In fact, uniformity is a requirement; a soldier who is different will infect the entire brigade. The discipline of the British military formed the backbone of the Empire.

Now he finds himself in charge of a mission that everyone thinks is impossible. He has a few thousand people intercepting and decoding messages one by one, and he has a small team trying to break the code. He has never encountered anything like this, where the best efforts of his staff are inadequate to carry out his orders. Command-and-control organizations depend on the leader's understanding of the strategy. The strategy of applying a machine to defeat another machine is not part of his training. He does what he was trained to do – when things aren't going well, he goes back to basics. He pounds the team with the only hammer he knows – he issues orders. First, he delivers a deadline ultimatum. When that fails, he turns to the time-honored tactic of finding someone to blame.

Alan Turing does not match Commander Denniston's idea of a soldier. Turing is not like anyone else, and is not deferential to his commanding officer. Turing's idiosyncrasies both annoy and baffle the commander, and his best model for understanding Turing is a soldier who drinks too much or goes AWOL. The accepted standard for dealing with such a soldier is to throw him in the brig, and if the bad behavior continues, discharge him. So, he tries to remove Turing from the project.

Which brings us back to the earmarks of a software project.

- The manager does not understand the computer, the mathematics behind the solution or the architecture of the system.
- His best tactics for improving things are to re-organize and to apply more brute force, a strategy that clearly (at least in hindsight) can only make things worse.
- He also employs the oft-failed strategy of setting an impossible deadline.

Despite all this, the project succeeds because upper management (Winston Churchill) believes that the best hope is to put the cleverest man in England on the problem, and because the team, fueled by ale and shepherd's pie finds the required breakthrough at just the right time to save the movie and win the war.

Poor Commander Denniston! If only he had read this book, he would have learned:

- That a clear vision and an empowered team are more effective than uninformed orders
- Why software projects are complex and how to overcome their inherent difficulties
- The importance of software architecture and methodology
- How to help the cleverest man in England come to that breakthrough as soon as possible

1.2 Requirements

The term '***requirements***' is ubiquitous in the realm of software development. So, before I start bandying about the word, let's discuss what requirements are, where they come from, and how seriously we need to worry about them.

Requirements are statements about what the software needs to do without regard to how it does it. The term has been used in software development for a long time, and, as you might guess, there are various factions about how precisely they should be specified and how important they are to the development process. I have seen requirements documented with many pages of numbered and cross-referenced statements. I have seen requirements summarized in one PowerPoint slide with big font. Neither of those extremes meets the needs of software development.

The term 'requirement' implies 'necessity', but if some of the requirements are not met at the time we need to ship, we will ship anyway. The requirements are really guidelines with degrees of necessity. I would prefer to use the term 'capability' instead of 'requirement', but the term is so ingrained in the culture that I will continue to use it with this modified definition.

Some requirements are truly critical, and some are so peripheral that only left-handed aardvark keepers need them. Most are somewhere in between. Any product has sets of requirements that are sufficient – multiple sets, because {a,b,c,d} might be sufficient, but also {a,b,e,f}. Agreement on which sufficient set is the primary implementation target is important for the development team.

Some requirements are Boolean – either met or not met, for example, accepting a specific input file format. Other requirements are parametric, with a measurable attribute, for example, the time it takes to process a transaction. Requirement authors often try to transform parametric requirements into Boolean requirements by specifying a threshold. But don't believe it, that threshold is fuzzy at best and will continually move toward zero or infinity, whichever is harder to implement.

Requirements are essential to guide the development team. They define what developers will implement. They help the team set priorities. They guide testing. They provide a vocabulary for the team to use when talking among themselves and with customers.

But because requirements are written in natural language, they are ambiguous. A developer needs to interpret the text and implement the requirement into code. Once implemented, the ambiguity is gone, but there is no guarantee that the developer's interpretation matches what the user meant. Add to that the high probability that the implementation has bugs, and you can see where quality problems come from.

The most common way of reducing requirement ambiguity is to break down requirements into a set of user stories. For example, the requirement for a C++ compiler to have clear error messages – a very ambiguous statement – might include this user story:

- When code illegally references a private member of a class, I want to see both the calling code and the declaration of the private member.

This is a much less ambiguous statement. The clear-error-message requirement might break down into hundreds of these user stories. Although ambiguity is reduced by the user stories, there is no guarantee that the set of stories completely covers the requirement.

Because requirements are often wrong, and always incomplete, software developers must continually make uninformed guesses to answer the question, "What would the user want?" The best practice is for developers to learn to think like users. Then the question transforms to "What would I want?" which is much easier to answer.

Old-style software development expected requirements that were supposed to magically appear at the exact moment they were needed with answers to any question that might be posed. *Should I make the Grpbxtz widget configurable?* The requirements were called “Marketing Requirements” although no one in the Marketing Department would ever use the product. For years, I tried unsuccessfully to get my teams to call them “Market Requirements,” and to instill the notion that everyone is responsible for understanding the market. But the term lived on and they continued to be late and inadequate, and I continued to get questions like, “Why can’t Marketing tell me the marginal benefit of the Grpbxtz widget?” And I wanted to reply, “Because they haven’t got a clue. No one has a clue. And they’re busy putting lipstick on the pig you shipped last year! Just make it configurable.” But I didn’t.

One of the key outcomes of the Agile revolution is that we now recognize that it is impossible to know the complete set of requirements at the beginning of the project. It may be impossible ever to know them. But we need something to guide the team, so, the team should make the best guess they can about the requirements using whatever resources are available, and they should assume the requirements are wrong. Later when they have limping software, they can get feedback from users, refine the requirements and feed the new requirements into the product development.

Assuming the requirements are wrong is the best assumption the team can make. First, it’s unlikely that the team can guess the requirements out of thin air. But more importantly, it creates a mindset that whatever software they write will need to adapt quickly as they discover better information about the requirements.

Since the requirements are wrong, anything that depends on the correct, complete requirements, will also be wrong. In particular, early estimates of software schedules and development cost will always be optimistic because so much of what’s needed has yet to be defined.

The requirements need to be pretty good by the time you release the system or you run the risk of shipping a Clydesdale when you needed a thoroughbred.

1.3 Models for Software Development

Software development is complex, and every development project is different. But there are common themes that apply to all projects. The synthesis of those common themes into models helps us to understand our complex problem.

A model is a metaphor for some aspect of the whole. A good model helps you understand one or more facet of the problem at hand and guides you to make good decisions.

Unfortunately, software development is too complex to be guided by a single model. In the following sections, I present three models for software development:

- The Yin-Yang Model explains why software development is difficult, and gives some guidance on how to combat the forces of evil that cause software problems.
- The Software Thermodynamic Model draws a parallel with classic thermodynamics and explains how software rots.
- The Complexity Model explains how to characterize a software development problem and the best method for solving that problem.

These three models, plus the Geodesic Manifesto that follows this section, form the backbone of the book, which will often refer to them.

1.3.1 The Yin-Yang Model

Software development reflects the classic battle between Good and Evil, light and dark. The forces of Evil do everything they can to make development difficult. But with courage and perseverance, the forces of Good can improve the process and help you create better software sooner.

In this section, I introduce six pairs of Yin-Yang terms. One side of the pair, the Yin, refers to an Evil feature of software that makes the problem hard, and the other side refers to its Good counterpart. Sometimes you can make a conscious tradeoff between them, but often the dark side dominates and your job is to minimize its impact.

1.3.1.1 Complexity vs. Simplicity

Bridge-building is a complex undertaking. A bridge has lots of components - the Golden Gate Bridge has over a million rivets. Bridges interact with the environment – painting the Firth of Forth Bridge took almost 30 years. They often need to be enhanced while remaining open to traffic – widening of a local bridge took over a year, but it was closed for just a few hours.

Software is ***complex.*** A typical system depends on thousands of code statements, sometimes millions. A bridge may have a million rivets, but there are only a few different types of rivets. Every code statement is unique. Each of those statements interacts with other statements, sometimes directly but often indirectly, in ways that are not readily apparent.

To implement that complex system, you need a complex organization of developers, software validators and many others. Coordinating all these activities is also complex.

While there's nothing you can do about the fact that you have a lot of code to manage, you can impose ***simplicity*** to parts of the code and to the process of creating it. You manage complexity with ***Architecture***, that organizes code statements into manageable chunks, and with ***Methodology***, that provides processes and guidelines for the developers who write those statements.

In the section The Complexity Model of Software Development, I will offer a pseudo-formal definition of complexity. Much of the book is dedicated to discussions of architecture and methodology.

1.3.1.2 Opacity vs. Visibility and Clarity

Golden Gate Bridge, from Lincoln Park, San Francisco, California

When I look at the Golden Gate Bridge, I can see how it was built. Its two massive towers of steel and concrete provide the bases of support. The main cables are anchored in concrete at both ends. The suspension cables hang from the main cable and support the roadway. If I move closer, I can see the rivets that hold the steel together. Even though some parts of it are encased in concrete, or hidden from view, it's clear that the design and the embodiment of the bridge are similar.

Software is ***opaque.*** The embodiment of a software system is extremely different from the code that implements it. By observing what it does and how it operates, I can only guess at the high-level architecture of the system. And the details are completely obscured.

The opacity of software makes it difficult to diagnose problems. What you can observe is a mere shadow of the cause of the problem. Not even a shadow – the thermal profile of the shadow.

The opacity of software makes it difficult to enhance. Each enhancement must begin by some developer coming to a new understanding of that part of the implementation – even if that developer was the original author. The code may be unambiguous, but the intentions and interactions are lost in the complexity of the software.

It takes work to overcome opacity with ***visibility***. The purpose of software debuggers is to provide complete visibility, but they're only practical for use by developers. You can add diagnostics that provide a window into the internals of the code; diagnostics are useful for developers and validators. And you can make sure that your architectures and processes encourage ***clarity*** of the code, again only useful for developers.

When users encounter a problem, they are doomed to workarounds based on black magic incantations.

1.3.1.3 Vulnerability vs. Reliability

Bridge builders are acutely aware of what happens if a component fails. Wherever possible, they avoid the possibility of a single point of failure – a place in the design where a single component could fail and bring down the bridge. If a girder could be held in place by two bolts, the designer might use four or six bolts to ensure it never moves. If the designers can't avoid having a single point of failure, like the towers of the Golden Gate Bridge, then they overdesign those components.

Software is ***vulnerable***. Every code statement is a potential single point of failure for a software system, although the severity of the failure may range from annoying to fatal. Since each code statement is written by a human, the likelihood of failure is significant.

To get high-quality software despite its vulnerability, we rely on testing through the opaque interface. The opacity obscures the vulnerability so much that we can't even measure software bugs until after they have been observed. If a bug fails in the forest of code without making any noise, then it doesn't exist.

It is possible to build highly ***reliable*** subsystems, but the cost is very high, so only critical subsystems, for example, the core of the Linux Kernel, can get the required scrutiny. You need to identify the parts of your system that need extra attention and come up with strategies that overcome the vulnerability.

Reliable software has few potential bugs lurking in the code waiting to happen. Developing reliable software is the main theme of the chapter Software Quality.

1.3.1.4 Rigidity vs. Flexibility

As you drive down the interstate, you will observe that most bridges look alike. The highway department has a few flexible designs that they reuse over and over. The design process for a highway bridge is to choose the design type, and then to adjust the parameters to adapt the design to the specific situation. Design reuse saves money and time, and the highway department can concentrate on making those few designs extremely robust.

It's somewhat ironic that software, the most configurable thing in the world, is ***rigid.*** Most software systems are built for a particular purpose and tested for that purpose. Any attempt to use a piece of that system for another purpose is likely to expose problems in the design or implementation. When requirements change, you can't bend the software to comply with the new information, you need to bolt on something new because modifying the existing system will cause it to shatter.

Building ***flexible*** software subsystems that can quickly adapt to changes requires a different mindset from the more common single-use development. You have to think ahead to know what kinds of changes are likely, and then you need to design the software to make those kinds of changes easy, even though you have no idea of the details of those potential changes. Building flexible software may take a little more work at the beginning, but the payback could be huge because often the alternative is starting over.

1.3.1.5 Chaos vs. Repeatability

At six AM, traffic sails across the San Francisco Bay Bridge. At seven AM, it crawls. Somewhere between, a driver tapped the brakes, a bird hit a windshield, a gust of wind pushed a car near the other lane. That perturbation, together with the increased number of cars slowed traffic until later in the day when lighter traffic could clear the bridge. But at seven AM, all you can see are a couple cars in front of you and you wonder what happened.

Chaos theory studies the behavior of complex systems. In a chaotic system, cause and effect are only loosely linked. The famous "butterfly effect" illustrates how a small change in one parameter can lead to an enormous change in the later state of the system.

In a deterministic system, if we know the entire state of the system it is possible to predict the next state of the system. But most of the time, we can only observe a subset of the state of the system, and so the next visible state often appears to be random. I call this the "iceberg effect". When we're caught in a traffic jam, we tend to blame the cars we can see, but they are victims the same as we are.

Software is ***chaotic***; it is subject to the software butterfly effect and the iceberg effect. Small changes to the software, or to the input data, can lead to big changes in the resulting state of the system.

To clarify, chaos and complexity are not the same thing. *Complexity* refers to the code itself and the process for creating that code. *Chaos* refers to the state of the system while the software is running.

The software butterfly effect can have two kinds of manifestation. First, the final state of the program's data can be different. This is common in systems that use complex numerical analysis or optimization algorithms. Second, the modified program state can trigger bugs that have lain dormant for years.

Software users expect ***repeatability***. When they do the same thing over and over, they expect the same result; any other result causes insanity. The opacity of software hides a lot of information from users. When you add the iceberg effect, sometimes software can appear to be non-deterministic.

As an example, the other day, my wife was on the phone with a colleague reviewing a complex spreadsheet used to manage the attendees at a Boy Scout adult training event. They were each looking at the same file on their separate personal computers. My wife referred to the "menu on the right." Her colleague replied, "I don't see that. I used to see it, but I don't anymore." They spent a few minutes trying to figure out how to get that menu, and then gave up. Something in the state of their computers caused them to see different results from the same input.

It takes thought and effort to overcome chaos to meet the expectations of users.

1.3.2 Software Thermodynamic Model

Usually, we think of software at the microscopic level, one code statement at a time. Of course, that's how we develop it and how we modify it. But the system will quickly reach a level of complexity that will benefit from understanding it at a macro level. In this section, I will draw an analogy between software and classical thermodynamics that provides a useful model for understanding software, even though the mathematical rigor is missing.

Classical Thermodynamics

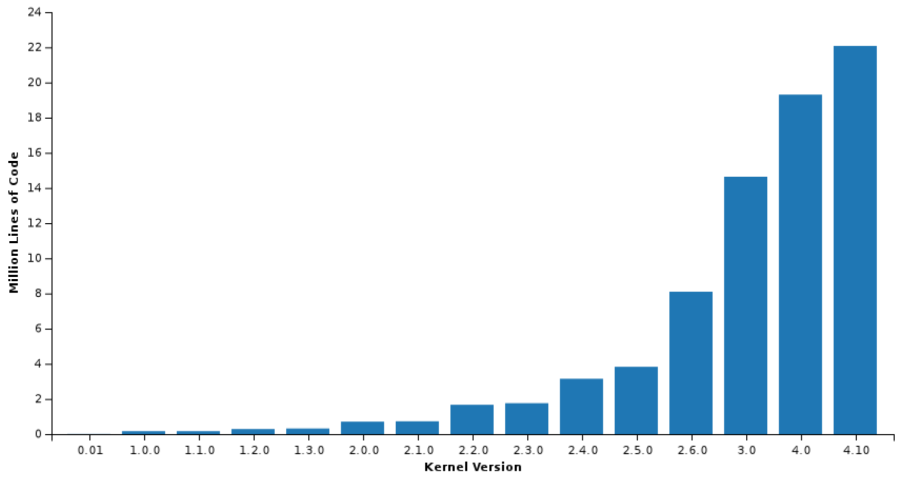
Thermodynamics is a branch of physics that studies the behavior of systems that are too complex to analyze at the component level. It relates heat and temperature to other forms of energy and work. We may not be able to analyze a balloon full of helium at the atomic level, but the statistical behavior of those atoms is consistent, and we can treat the helium in the balloon as a single object with a temperature and pressure, and use that macro-behavior to understand why the balloon rises.

Thermodynamics has four laws, numbered from zero to three. They start at zero not because the scientists who formulated them were computer scientists, but because the well-established laws (one to three) depend on a definition of temperature, so scientists tacked on a zeroth law to rectify that omission. Here is a layman's version of the laws. In these laws, I distinguish between an object, which might be a balloon full of helium, from a system, which includes all the objects in the limited universe of your observations.

0. If two objects are each in thermal equilibrium with a third object, the two objects are in thermal equilibrium with each other. By definition, two objects in thermal equilibrium have the same temperature.
1. Energy can change forms, for example, from kinetic energy to heat energy, but the total energy of a system never changes.
2. The entropy of a system never decreases. Entropy is a measure of the energy of a system that has been lost to heat and can never be converted to another form. It is analogous to disorder. A gas has more entropy than a liquid, which has more entropy than a solid. It's a measure of the energy that's lost and no longer usable.
3. At a temperature of absolute zero, an object has zero entropy. (It's not exactly zero, but zero is close enough for software.)

Software Thermodynamics

Software changes every day. Bug fixes, enhancements, refactoring – they all contribute to the continuous change. And after every change, there's more code. The figure below shows how the amount of code in the Linux Kernel has grown over time.



From https://en.wikipedia.org/wiki/Linux_kernel

With more code there are more bugs. If you search the web for "bugs per line of code", you will find various estimates from 0.1 to 50 bugs per thousand lines of code, depending on lots of factors, including the difficulty of the code, the ability of the developer, and the level of testing. The exact value of the number doesn't matter except that it's not zero. More code means more bugs.

A useful metaphor is to think that all those changes increase the ***entropy*** of the software. But we need to understand what entropy means in the context of software.

It takes work to create and maintain software. Let's define a unit of work, the ***swerg*** (a portmanteau of SoftWare erg), to be one staff hour of work – one average software developer working for one hour.

A software module, some logical collection of code like a function, a class, a source file, a library, etc., has some amount of ***latent work*** W_L needed to make it meet the current requirements of that module with zero bugs. Of course, the requirements will change, but for the definition of latent work, assume the requirements are frozen. The latent work includes work required to meet the requirements W_R and work required to fix all real and latent bugs W_B.

$$(1) \quad W_L = W_R + W_B$$

A *real* bug must be both triggered with the right conditions, and observed. A *latent* bug is bad code that is waiting for the trigger and observation to happen.

Suppose that a module M has 100 swergs of latent work (W_L). An average developer, Jack, applies 20 swergs (W_D) to improving M. You would think that M now has 80 swergs of latent work, but while Alf worked, he created new bugs, so M now has 85 swergs of latent work. It's useful to think of the difference between the expected value and the true value as an increase in the entropy E of module M.

(2) $\Delta E = W_{L2} - (W_{L1} - W_D) = 85 - (100 - 20) = 5$

From this we can define the ***entropy*** of a module to be the amount of work needed to fix all the latent bugs in that module. This definition is intuitively close to thermodynamic entropy since disorder breeds bugs.

Next, we need to define a temperature. Temperature is a property of the module. Conceptually, a ***hot*** module is difficult to modify. Any change to that module will create lots of new bugs and increase the entropy. A ***cool*** module is easier to modify. To be consistent with our conceptual understanding, I will define the module temperature:

(3) $T_M = \frac{\Delta E}{W_D}$

where ΔE is the increase in entropy in the entire system (not just module M), and W_D is the developer work applied to the module. This definition depends on an average developer making an average change, and of course that never happens, because in your organization, all the developers are above average.

One of the factors that determines the temperature of a module is the amount of coupling of that module to other parts of the system. In a hot module, a single software change can greatly increase the entropy. That is because in a hot module, code often has convoluted relationships to other parts of the system. That one change can cause (or expose) new bugs in unchanged parts of the system. The entropy doesn't just add, it multiplies. This is sometimes called, "Software Rot", a common term for the phenomenon that, over time, parts of the system, that previously worked just fine, start to fail even when they have not changed.

In a cool module, software changes are isolated and do not propagate entropy beyond the boundaries of the module where the change is made. When new code is added, entropy is increased, but it doesn't multiply. In a cool system, software rots much more slowly.

Let's look at the extremes. When entropy increase is greater than the developer work applied, you have a poison module. Everything you do to it makes the system worse. Perversely, high temperature modules are both the ones you want to change the most, and are the hardest to change. They become extremely stable because no developer will touch that module, and you are stuck with the current implementation.

On the other hand, you can imagine changes that decrease the system entropy by more than the effort applied – a module with a negative temperature. These changes are rare, and modules with negative temperature don't last very long. Usually, they are removed from the system by re-architecting parts of the system to remove inter-module coupling, or re-writing a critical module that has had a lot of problems. I fondly remember the deep sense of satisfaction I experienced when one of these changes worked.

Fixing the bugs found by testing can decrease the entropy of a system, but testing can't reduce the temperature of the module. If the module is hot, once you've recovered from one change, then the next change you make is likely to introduce new bugs. The only way to reduce the temperature of a module is to improve the architecture to isolate the effects of changes.

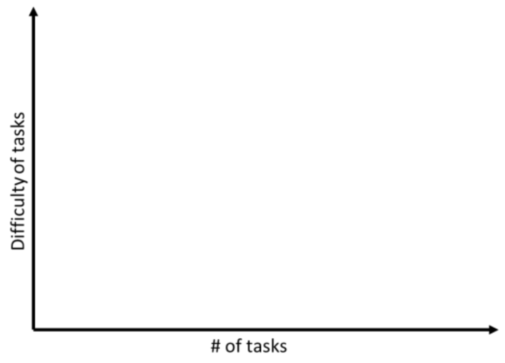
There are no laws of software thermodynamics, but there are tendencies that parallel the laws of classical thermodynamics.

0. A software system won't come to thermal equilibrium, but a hot module is likely to increase the temperature of the modules it interacts with.
1. Software is not a zero-sum game. All the work that goes into making software creates value in the software system. The value of the software is much more than the cost of the swergs that went into it.
2. The entropy of a system always increases. That's because the amount of code in a system always increases. Those rare changes that remove code are offset by many more changes that add code. Less code may have fewer bugs, but most bugs are caused by writing not quite enough code.

   This is not a law. It is possible to reduce the entropy of a system, but it takes work. It is much easier to design a cool system that will have low entropy than to fix a hot system that has high entropy.
3. The latent work of a software system never reaches zero. Changes in the requirements cause changes in the code, which create new bugs. Software is only stable when it's dead.

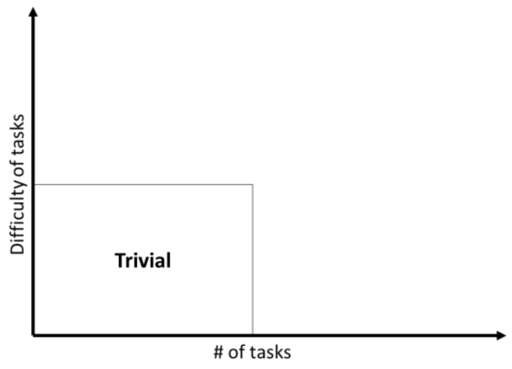
1.3.3 The Complexity Model of Software Development

The Complexity Model of software development has served me well over the past twenty years. It is simple enough that most people can quickly understand it, yet rich enough that it can be applied to a wide range of problems.



Software problems can be characterized by the number of tasks it takes to solve a problem and the difficulty of the tasks. I will divide this space into four quadrants.

1.3.3.1 Trivial Problems



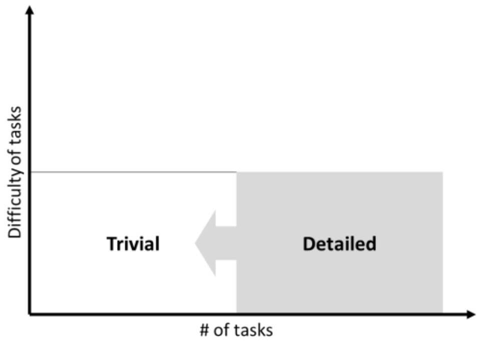
The lower-left quadrant, problems that can be solved with a few easy tasks, is named ***Trivial***. It's a nice place to work, but it doesn't pay very well. However, it is the goal of software development to transform a problem from the other quadrants to the Trivial quadrant. When people talk about "Ease of Use" for a software product, they are really evaluating how close to the origin did the product get.

For example, consider fingerprint security for the iPhone®.

Once it is set up, the use model is trivial – put your finger on the button. But behind the scene, there's a lot of stuff going on. The sensor outputs an array of pixels. Then image processing transforms the pixels into something that represents the fingerprint. And then the fingerprint must be matched to the master fingerprints you set up. And if it finds an acceptable match it lets you use the phone.

On the other hand, the setup of the master fingerprints is a process that takes quite a few tasks. You must follow instructions and respond to arcane feedback before the phone accepts your master fingerprint. This process is tedious and thus, leads us to the next quadrant in the model.

1.3.3.2 Detailed Problems

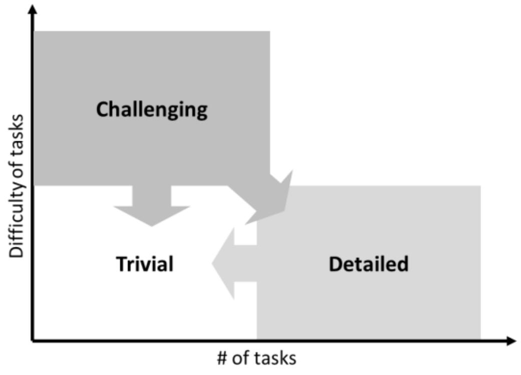


The lower-right quadrant, problems with lots of easy tasks, is named ***Detailed***. When I first came up with this model, I called it ***Tedious*** because it represents the kind of problem I despise. But then I realized that any organization needs people to solve these problems and that, indeed, there are people who love solving these problems.

Detailed problems are solved in two ways. Automation transforms the Detailed problem into a trivial problem. Brute force solves the problem by applying lots of time and/or resources. Automation is the preferred approach if you are going to encounter the same problem often. But don't discount brute force; if you're only going to encounter this problem once, then brute force may be both the most efficient and most expedient way to solve the problem. ("There is no such thing as ineffective brute force, only insufficient." – attributed to Lou Scheffer) And even for problems you plan to automate, using the brute force approach at first may help you understand how to automate the process.

Consider the problem of the Postal Service. They need to deliver millions of items each day. They use automation for a lot of the processing, but the final mile is still served by thousands of postal workers who physically deliver the mail.

1.3.3.3 Challenging Problems



The third quadrant, problems with a few difficult tasks, is named ***Challenging***. This is the reason I became a software developer in the first place. This is the realm of the NP-complete problem, of the problems that are closely tied to mathematics, of statistical analysis, of numerical analysis, of optimization, of cryptography. This is the realm of the PhD dissertation, of the technical journals, of professors who have dedicated their lives to one Challenging problem.

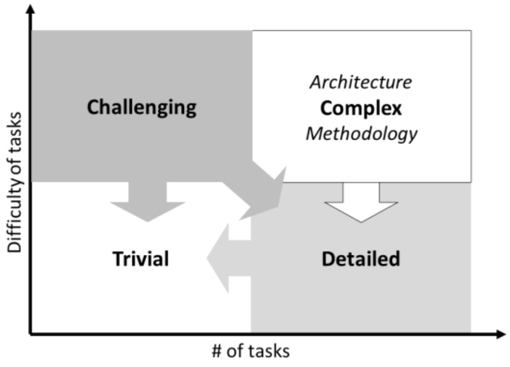
A Challenging problem is solved by transforming it into one or more Detailed or Trivial problems. The key to the solution is invention. The solution may be in the literature, or it may never have been encountered before. The solution may require applying techniques from many sources. The best software developers for these problems understand a wide range of techniques, and are willing to experiment with these techniques until they find the combination that works best for the problem.

Consider Alan Turing's problem with the Enigma code. Let's assume that I needed to solve that problem today (so I don't have to build a computer from scratch). I might try the brute force technique first to understand how bad it is. I would read papers on cryptographic cracking. I would try a few of the techniques in the papers. I would sleep on it and sometimes wake up with new ideas. And eventually, I would have an acceptable solution. I might be able to improve the solution over time.

An important characteristic of Challenging problems is that they are inherently unpredictable. Before starting on the Enigma decryptor, I could fairly accurately predict how long it would take to develop the code to try to decrypt the message. But I would almost certainly be hopelessly optimistic about any guess about when the code would succeed at decryption. And after the acceptable solution is in place, I would be hopelessly wrong about a prediction for when the code would run ten times faster.

There is a later chapter dedicated to a discussion of Challenging Problems.

1.3.3.4 Complex Problems



The final quadrant, problems with lots of difficult tasks, is named ***Complex***. Most problems worth solving fall into this quadrant. No one person can solve a Complex problem – it takes a team. To make the team effective requires two things: an architecture that provides a framework for breaking tasks into manageable chunks, and a software development methodology that makes sure that when those chunks are completed, they will work within the whole.

The essence of software leadership is the ability to develop architectures and methodologies. These two aspects of software development are co-dependent. The best methodology in the world will fail if the architectural underpinnings aren't strong enough to hold it up. The best architecture won't get off the ground without a solid methodology to make sure the pieces can come together to form a useful whole.

There is a later chapter dedicated to Architecture. Most of the rest of the book discusses various aspects of Methodology.

1.4 The Geodesic Manifesto

Unless you studied topology, you probably have only heard the word 'geodesic' applied to Buckminster Fuller's dome. In topology, a geodesic is the shortest path between two points in a generalized space. On a Euclidean plane, it is a line segment. On the surface of a sphere, it is a segment of a great circle. The geodesic from Sacramento to Reno, given the constraint of driving an automobile, goes through the Donner Pass. Dijkstra's algorithm can find a geodesic in a graph, the shortest path between two nodes. In software, the geodesic is a metaphor for the way that a productive software team starts with nothing but a twinkle in someone's eye, and ends up with high-quality working software.

The Geodesic Manifesto describes the Geodesic Philosophy at a high level. Like Agile, it aims to help software development teams create quickly high-quality software in a world where requirements are vague, incomplete and changing. Whereas the Agile Manifesto (discussed in the next chapter) focuses on what not to do, the Geodesic Manifesto describes what to do to achieve your goals. Agile's negative views of process, documentation, planning and leadership have led many development teams into one-way, dead-end tunnels. I hope that the Geodesic Manifesto can free them to use the ways that work, even if they're not "Agile."

This section presents the Manifesto, followed by the annotated Manifesto. On first read, you may want to skip to the annotated version that includes brief discussions with links to the parts of the book that expand on the key concepts.

THE MANIFESTO OF THE GEODESIC PHILOSOPHY OF SOFTWARE DEVELOPMENT

ASSUMPTIONS

We assume these truths:

- Our goal is to create software-based solutions that do what the user wants in the way the user expects.
- Requirements are an inaccurate abstraction of what customers and users want. Requirements cannot be known completely and will change even after they are known. The only true specification of the solution is working software.
- People who develop software have a wide range of ability, motivation and temperament.
- Our problem is to help those people overcome software's inherent difficulties: complexity, opacity, vulnerability, rigidity, and its chaotic nature.

PRINCIPLES

To overcome the limitations of requirements:

- We have frequent interaction with customers and users.
- We frequently deliver working software and actively seek feedback.
- Developers must learn to think like their customers.

To overcome the limitations of people:

- We build synergistic teams that improve the performance of every individual.
- We establish a methodology that fosters creativity and encourages interaction.
- We encourage frequent and open communication among team members.

To overcome the inherent difficulties of software:

- We seek out change from as many sources as possible.
- We plan for change, and change the plan when necessary.
- We embrace change through constant measurement, analysis and improvement of the software and methodology.
- Developers strive for the highest level of technical excellence and assume responsibility for quality.

PILLARS

Success depends on:

- Leadership that:
  - Creates a vision for the solution we're trying to create
  - Organizes the development team to maximize the contribution of each member
  - Develops dynamic plans that answer three questions: When will it be done? What will it contain? What do I do next?
  - Establishes processes and controls that maximize productivity and creativity
- An architecture that organizes the solution into modules and their interactions that is modular, flexible, consistent and sufficient.
- A methodology that defines common processes and controls that foster innovation, empower the development team to create the best solution, and maximize efficiency while minimizing risk
- A decision-making ethos to guide us as we change the vision, plan, architecture and methodology.

1.4.1 The Annotated Geodesic Manifesto

THE MANIFESTO OF THE GEODESIC PHILOSOPHY OF SOFTWARE DEVELOPMENT

❖ *Now that's a pretentious title!*

ASSUMPTIONS

We assume these truths:

- Our goal is to create software-based solutions that do what the user wants in the way the user expects.
  - ❖ *The Manifesto primarily addresses software, but software is often part of a larger solution that may include hardware, data and other stuff.*
  - ❖ *The Quality chapter defines high quality software with this statement: "High Quality Software does what the user wants in the way the user expects."*
- Requirements are an inaccurate abstraction of what customers and users want. Requirements cannot be known completely and will change even after they are known. The only true specification of the solution is working software.
  - ❖ *See the Requirements section earlier in this chapter.*
- People who develop software have a wide range of ability, motivation and temperament.
  - ❖ *The Organization section of the Software Manager chapter discusses these traits of software developers.*
- Our problem is to help those people overcome software's inherent difficulties: complexity, opacity, vulnerability, rigidity, and its chaotic nature.
  - ❖ *These are the five Yin traits from the Yin-Yang Model.*

PRINCIPLES

To overcome the limitations of requirements:

- We have frequent interaction with customers and users.
  - ❖ *The Agile Manifesto's principles recommend daily interaction with "business people", whom I refer to as "customer advocates".*
- We frequently deliver working software and actively seek feedback.
  - ❖ *The chapters on Methodology, starting with Common Processes focus on software delivery.*
- Developers must learn to think like their customers.
  - ❖ *There is a brief discussion of how to think like a customer in Think Like a Customer*

To overcome the limitations of people:

- We build synergistic teams that improve the performance of every individual.
  - ❖ *The Agile Manifesto talks about self-organizing teams. But self-organization is not enough. A team needs leadership and direction to make the transformation from a bunch of people who work together into a team that can move mountains. When that happens, it's a wonder to watch.*

  - ❖ *In Teamwork, I present four values essential to synergistic teams: Mutual Success, Mutual Ownership, Common Understanding and Continuous Improvement.*
- We establish a methodology that fosters creativity and encourages interaction.
  - ❖ *The chapter Common Processes discusses what processes are needed for an effective methodology.*
  - ❖ *See the Methodology Frameworks chapter for examples of high-agility methodologies.*
- We encourage frequent and open communication among team members.
  - ❖ *Organize teams and office space to maximize communication efficiency.*
  - ❖ *Avoid unnecessary documentation; write only what's needed for the use and development of the software.*
  - ❖ *The formality of communication must increase with physical and organizational distance.*

To overcome the inherent difficulties of software:

- We seek out change from as many sources as possible.
  - ❖ *Changes come from customers, users, testing and the creativity of the development team.*
- We plan for change, and change the plan when necessary.
  - ❖ *We need a plan, but only if it can adapt to the changing environment.*
- We embrace change through constant measurement, analysis and improvement of the software and methodology.
  - ❖ *The continuous improvement cycle – measure, analyze, implement – forms the core of every Agile methodology. Lean thinking provides a framework for doing that.*
- Developers strive for the highest level of technical excellence and assume responsibility for quality.
  - ❖ *Technical excellence is one of the principles of the Agile Manifesto.*
  - ❖ *Write code that is easy to understand, easy to extend, and easy to change.*
  - ❖ *In the Quality chapter I propose that developers must assume responsibility for quality because they are the only ones who can affect it. Validators identify problems, but developers created the problems.*
  - ❖ *Quality is achieved through effective development processes that include code review, high coverage test development, frequent measurement, and quick feedback about problems.*

PILLARS

Success depends on:

- Leadership that:
  - Creates a vision for the solution we're trying to create
  - ❖ *The Vision section of the Software Manager chapter discusses how to create and communicate a vision.*
  - Organizes the development team to maximize the contribution of each member

- ❖ *The Organization section of the Software Manager chapter discusses management of the team.*
   - Develops dynamic plans that answer three questions: When will it be done? What will it contain? What do I do next?
- ❖ *The word 'dynamic' implies that the plan needs to change as the project evolves. We can't expect to plan once and then execute. Plan for change and change the plan when necessary*
- ❖ *The Agile Manifesto devalues planning and has given a lot of lazy managers the excuse to avoid it. But planning is necessary because at any point, you need to be able to answer these three questions. See the section of the Software Manager chapter.*
   - `
- ❖ *See the Common Processes chapter and the Control section of the Software Manager chapter.*

- An architecture that organizes the solution into modules and their interactions that is modular, flexible, consistent and sufficient.
   - ❖ *The Architecture chapter briefly discusses how to think about architecture. These five attributes help distinguish good architectures from bad.*
- A methodology that defines common processes and controls that foster innovation, empower the development team to create the best solution, and maximize efficiency while minimizing risk
   - ❖ *The Methodology Frameworks chapter provides examples and analysis of common methodologies.*
- A decision-making ethos to guide us as we change the vision, plan, architecture and methodology.
   - ❖ *The primary purpose of this book is to help you build an ethos – a guide to tell right from wrong in software development.*
   - ❖ *Decision making is discussed in the Leadership section of the Software Manager chapter.*
   - ❖ *As the Agile Manifesto states, "At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly."*

1.5 Graphic Details

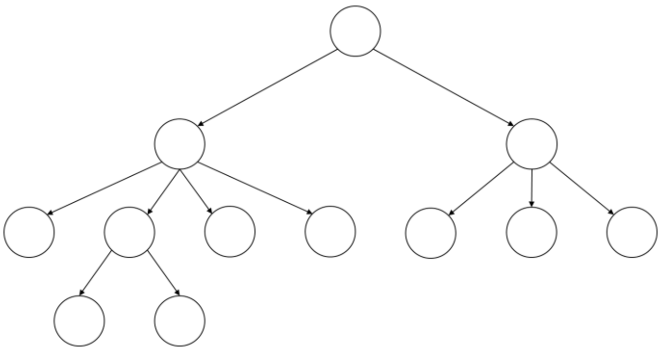
I occasionally use concepts from graph theory to describe parts of the software development process. Since I realize that some of you are unfamiliar with graph theory, this section is a very basic introduction with the definitions of the terms I use. If you are confident in your knowledge, you can skip this section, but you'll miss all the graph theory jokes.

Two nodes, Neil and Sheila, walk into a bar. Why didn't Sheila duck?

A ***graph*** is a set of ***nodes*** (sometimes called vertices, but not by me) and ***edges*** that represent the relationship between nodes. When using a graph, the nodes will represent something, like the corners of a polygon or the modules in a software architecture. The edges may also represent something, like the edges of a polygon or the dependencies in a software architecture. In ***directed graphs***, edges have direction – they point from one node to another. In an ***undirected graph***, edges just represent a relationship without the notion of from-to. The name, "undirected graph", is distinctly unimaginative, but you can see why they didn't call it the "any-which-way graph".

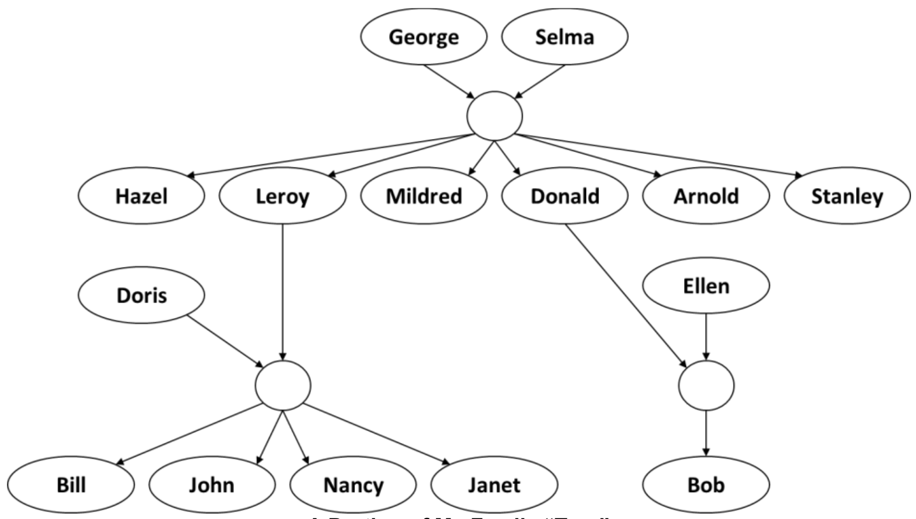
A graph has a ***loop*** (mathematicians prefer 'cycle') if you can leave a node and find your way back to the same node without repeating any edges.

The bartender says, "You're looking a little edgy." Sheila replies, "My edge points to him and his edge points to me. I'm feeling a little loopy." Neil says, "That's a cyclical argument."



Tree Example

A ***tree*** is a directed graph where every node has at most one input edge. There is one root node of the tree that doesn't have any inputs. The outputs of a tree node connect to the child nodes. Nodes without any children are often called ***leaf nodes***. If you draw it carefully, a tree looks like an oak, with all the nodes branching from the root. If I draw it, you can think of it more as a hydroponic tree, because I always draw the root node at the top.

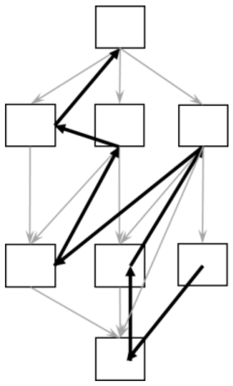


A Portion of My Family "Tree"

There's a special kind of directed graph that has no loops. It's called a ***Directed Acyclic Graph***, or ***DAG***. Your family tree is not a tree; it's a DAG with two kinds of nodes. One node type represents an individual, Uncle Leroy; and the second node type represents a union that produces children, Uncle Leroy married Aunt Doris and they had four children. We know it's a DAG because a loop would mean that you are your own ancestor, which can only occur in bad time travel science fiction.

"Do you have any children?" asked the bartender. "Just one boy," said Sheila. "We call him Dagwood."

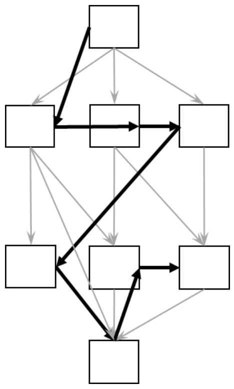
There are two common ways to search a graph: depth-first and breadth-first.



Depth-First Search Example

In ***depth-first search***, you start from the top node and search successor nodes from left to right. If the successor node has not been processed, then search its successor nodes. Once all the successor nodes have been processed, or there are no successor nodes, then process that node. In the example graph above, the nodes are processed in the order shown by the red lines. The advantage of depth-first search is that when you process a node, you know that all its successors have already been processed. One of the uses of depth-first search is to find the longest path from the start node to any leaf node.

“Dagwood is in the navy, specializing in depth-first charges,” said Neil.



Breadth-First Search Example

In ***breadth -first search***, you start from the top node, process it, and then put all its successors in a queue. Take the node from the head of the queue, process it, and then put all its successors in the queue, skipping the ones that have already been processed or are in the queue. In the example above, the nodes are processed in the order shown by the red lines. One of the uses of breadth-first search is to find the shortest path from the start node to any leaf node.

"You've gained a little weight, dear," said Neil. "Maybe you should stand a little taller." "Breadth first, dear," replied Sheila.

I didn't say they were good jokes.

1.6 Dramatis Personae

Software development has a large cast of characters. In this section, I list the characters and the names I use for them throughout the book. Beware that in some organizations these names mean something else.

Software Developer – A person who writes software for a living. Also known as programmer, coder, hacker, software engineer, computer scientist, and many other names.

Software Validator – A person who verifies that software does what it's supposed to. Software must be tested to validate that it delivers the intended reliability and functionality. Developers are responsible for some of the testing, but independent system testing is done by the validation team. I use this term instead of the more traditional "quality assurance" (QA) because over time QA acquired a bad reputation as a job where failed developers ended up.

Architect – A developer who is responsible for maintaining the integrity of the architecture.

Tech Writer – a person who writes product documentation. Users require documentation to effectively use the product. The required amount and quality of documentation depends on many factors like the complexity of the product, the sophistication of users and level of support that's available.

Manager – a person who manages a group of software developers. This person is responsible for making sure the software gets done with all the required features, on the committed schedule, and with high quality. The manager is also responsible for the welfare of his team.

User – a person who uses the software.

Customer – the person (or entity, if it's a company or other organization) who pays for the software. Sometimes this is the user. More often, this is an entity who provides the software to the users. For example, if I buy PhotoShop™, I am both the user and the customer. If IBM buys photoshop, IBM is the customer. There will be many users within IBM who have no idea how much, or under what terms, IBM paid for PhotoShop.

Customer Advocate – someone within your organization who can speak for the customer. This might be someone in marketing, sales, or support.

Executive – someone up the chain of management from the Manager, typically the Manager's boss's boss or higher. Also known as upper management.

Knobs, Inc. – a purveyor of door handles that have well known requirements and never need an upgrade.

1.7 Key Concepts

Requirements

- Requirements are an abstract specification of what the software should do.
- Requirements aren't required; they are guidelines for capabilities that have a degree of necessity.
- Requirements cannot be known completely and will change over time.

The Yin-Yang Model

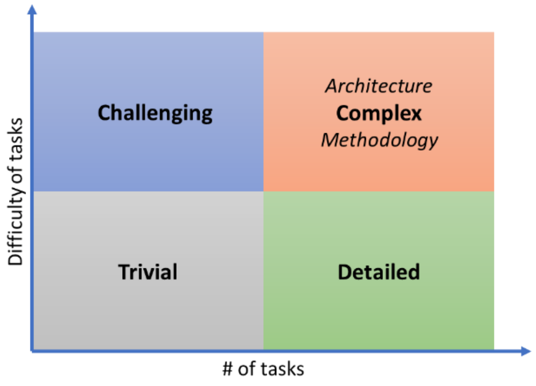
- The Yin-Yang Model presents five pairs of good vs. evil traits of software:

Yin	Yang
Complexity	Simplicity
Opacity	Visibility and Clarity
Vulnerability	Reliability
Rigidity	Flexibility
Chaos	Repeatability

Software Thermodynamics

- The Software Thermodynamics Model is an analog to classical thermodynamics that attempts to define entropy and temperature of software.
- Entropy represents the development work that is wasted because it creates bugs or needs to be redone.
- The temperature of a module represents the tendency of work done on that module to create entropy. Work done on low temperature modules tends to create little entropy, while work done on high temperature modules tends to create a lot of bugs, and therefore a lot of entropy.

Complexity Model



Complexity Model

- The Complexity Model partitions software problems into four quadrants based on the number of tasks required to solve the problem, and the difficulty of those tasks.
- Complex problems are solved with a combination of Architecture and Methodology.

The Geodesic Manifesto

- The Geodesic Manifesto presents the key concepts that will be addressed in this book.
- It presents a way to achieve the agility that organizations want, together with the predictability and quality that they need.

1.8 About the Author

Bob with the Birds

Bob Erickson was born and raised in northern Minnesota, where he walked to school in forty below zero weather uphill and against the wind in both directions. He received a BA in Physics and Electrical Engineering from Rice University, and an MSEE from Stanford. After a few years as a hardware designer at HP, he wandered into the Electronic Design Automation industry, working for Silicon Compilers and with Mentor Graphics after the acquisition. He alternated between management roles and development roles for many years before becoming VP of Engineering at startup Synplicity and later VP of Software at startup Tabula. He returned to the Synplicity group after its acquisition by Synopsys, where he worked mostly as a software developer and architect. He was named Synopsys Distinguished Architect before his retirement in 2017.

Bob welcomes feedback and questions. Contact Bob at bob@geodesicManifesto.com.