Engineering Software

And Other Creative Improbabilities

Sean P. Goggins, Ph.D

Publisher Imprint
Columbia, MO
London, England

Draft Assembled on 2020-08-25

"*Sooner or later I'm going to die. But I'm not going to retire.*"
—Margaret Mead

Contents

List of Figures

Preface

There are many good software engineering collections out there. Sommerville, especially, comes to mind. Whatever this idiomatic, idiosyncratic collection of words, sentences, paragraphs (the usual) becomes (or is now) will not replace any of them. My aim here is to produce something that informs and entertains. Yes, entertainment is one of my aims. Otherwise what you are holding on your tablet now is merely a less expensive ambien.

Within these pages I state my opinions and all are welcome to disagree with them (The world needs ditch diggers, too). In fact, in the tradition of open source software, I welcome contributions, edits, corrections, alternate views and the like. Such contributions will be recognized in future editions of this ”book”. There. I did it. I called it a book. But its not, yet. Contributors will be credited as long as we understand I will be the one invited to speak at the United Nations and visit Stockholm for the Nobel; as soon as there's one for satirically kind of interesting software engineering literature.

I earnestly want to integrate what I know about team science, open source software metrics and culture, social computing, social media, online discourse, software engineering and progressive rock into this volume. Not because I need one book to connect all the things I think about, but because they are, I believe, more interconnected and relevant to each other than has been noted so far (progressive rock is a stretch, I know). That's what this book aims for right now. This preface could change as it evolves.

Sean P. Goggins
Spring, 2020

I Introduction to Software Engineering

1 Software Engineering as Systems Thinking

"Systems Thinking is the fusion of analysis and synthesis, depending on whether our objective is knowledge or understanding."
—Russell Ackoff

Abstract. Its helpful to know the specific problems that software engineering solves. First, it provides a framework, and a language that you can use to communicate with other software engineers. Second, software construction is well known as work fraught with financial risk for any organization that takes it on. Understanding the systems of people and practice that you will be building software for will help you avoid failure, first and foremost. Avoiding failure begins with "building the right thing". Once you have mastered that practice, which is a decidedly human one, we focus on processes and practices for "building the thing right". Working within a system that your team agrees to will provide you with references, signposts, and ultimately instincts to guide your success. Building software is creative work with social and technical components, often described under the umbrella of "sociotechnical systems". With that in mind, we begin our journey with some highlights of systems theory.

1.1 Introduction

Software engineering defines milestones in a process whose intent it is to provide a clear picture of what our code does before we write it. The process includes well documented stages: requirements, design, implementation, testing, deployment, maintenance. In that order. You will encounter methodologies, like "agile programming", that cycle through these stages over a period of weeks, and others like the classic "waterfall method", that takes years. The problem is that you cannot read a software engineering textbook, follow the steps and become a software engineer. Or engineer software. If you have a belief system that demands we need to hold a Hamilton-Burr like dual so you might defend

software engineering's honor, I advise you to consult science as well, and consider the possibility that your own passions can betray you. [1].

If you want to write code you can do that without taking a software engineering course or reading a book. Write good code and find an open source project that interests you. If you have a team of people and a non-trivially complex problem to solve, chances are members of your team will need to specialize. When that happens you will need to coordinate your efforts so that when you are making progress you do not step on each other. You will also need to collaborate to sift through complexity[2] and identify a pool of potential solutions to explore and ultimately use to derive your solutions, expressed in code. I used the concepts "coordinate" and "collaborate" right next to each other and defined the differences on purpose by the way, because its a common misconception that "collaborate" and "coordinate" are interchangeable concepts. They are not.

1.2 Reacting to Failed Software Projects: A Brief History

Born of a Crisis Software engineering, then, is a set of practices and a process that assists groups of technologists and other stakeholders working together to build solutions to solve some important, non-trivially complex problem. You will be surprised how difficult beginnings and endings are to delineate from each other in the practice of software engineering.

> "Since when," he asked, "Are the first line and last line of any poem Where the poem begins and ends?" — Seamus Heaney

Box 1.1
Story: An Adverse Events System, and Sexism in Software Engineering

I often make the claim that I am 16-1 on the software projects I led during my career in industry. Its a pretty impressive record, and I amassed it not by thinking I am awesome, or even good at computer programming. leadership, design, or influencing people. Mostly, I was bold with regards to seeking the advice and insight of others, almost constantly. One time, I was assigned a new project with two other software engineers. The manager entered the room, half attending to his pager, half attending to his palm pilot, and as he sat down he took a brief moment to figure

[1] "I have never met anyone so passionate and dedicated to a belief as you. It's so intense that sometimes it's blinding." — Dana Scully (Gillian Anderson)

[2] Prior to the age of the global pandemic, I advised students to use white boards, walk around, talk, and in other ways work the problem together. We will now need to use methods you are less familiar with, but which are popular in Open Source Software. Use technologies like Zoom, Slack, Github Issues, and your phone's camera to discuss a problem and its complexity with each other. Collaboration is part brainstorming and part design. Its all collective problem solving.

out what this meeting was about. He was unprepared, and I had badgers tearing a hole in my stomach because this was my first project at a new job, and I had just left a more lucrative one involving heavy travel, to make sure my daughter would recognize me when she turned three.

Alice and Helen (not their real names), my two colleagues, had each worked for this medical device company a number of years, and been promoted at least once. Our obviously unprepared manager surrendered to his inability to figure out the purpose of the meeting and turned to me, "So, tell me about this project and give me a ballpark estimate on how long it will take for the three of you to get it done." I looked at Alice and Helen. I knew the project involved improving adverse events[3] identification speed, and some data. The badger clawed at my stomach more fiercely. "Chad" (not his real name), I said, "I have been at the company for a week, and when the director met us during orientation, she and I talked about my desire for a challenging assignment, and my request to be on an experienced team." I turned to Alice and Helen, "Hi, experienced team. What can I do to help answer Chad's question?"

Helen was the clinical analyst who knew the data well, "Start by back tracing the data in the current reports with the original data we get from these 7 sources. I think, somewhere, we are integrating the data wrong. I can show you how to access those systems." Alice, the software development savant in our group volunteered, "We either have tests that are passing in error, or we are not testing something. There's no way the adverse event clusters the system is identifying have anything to do with each other. They are almost random collections of device[3] failures in the field. I'll check the code." Alice concluded, "The team who built this did not check it with the design engineers or field support team who retrieves the failed devices, and logs the events with doctors. We need access to both of those teams, and 2 weeks to figure out what needs to be done to undo this clusterfuck."

A moment passed, Chad looked again at his pager and said, "OK. Lets meet again in two weeks. Alice, send me updates if you have any before that meeting. Lets fix it! Its all about shareholder value!" Chad left the room, and my face betrayed the "what the fuck?" I felt after the meeting. I looked across the table to my team mates and asked, "What the fuck?" They looked at me, rolled their eyes, and as if in rehearsed unison, said, "Yeah." Helen closed by asked me to come back to her desk so we could make sure I had access to the data I needed, and the various other pieces of software that touched it prior to its arrival in the adverse events system.

We spent a month identifying and correcting a series of data translation errors, mostly introduced when source data was moved into a data warehouse, and field mappings were simply wrong between the source and target. The system we fixed in a month had been in service for 2 weeks before it obviously did not work, and had taken 18 months, and several million dollars to build.

The larger the system, the longer it takes to build, the more it costs, and greater the risk that our efforts fail. This was realized early on when computing systems were only affordable by governments and very large businesses. Banks, armies, and scientists were

[3] Adverse events are when an implanted device fails. To prevent device recalls, the company needs to find the root cause, and especially patterns of similar adverse events quickly when they occur in the field.

the first with access to computing hardware whose extraordinary cost was justified by the problems it could solve, given the working software.

Working software was seldom a given, and the frequent failures in the "software part" came to the attention of powerful people who controlled the money used to fund them. To the people writing the checks in the late 1960's, the term *software crisis* was the long way of saying, *software*. Failure took the forms of performance failure against expectations or project failure against budgetary and time constraints (Buxton and Randell, 1970; McClure, 1968). Testifying to the role of governments and armies in early software work, the North Atlantic Treaty Organization (NATO), commissioned two reports on the topic in 1968 and 1969. Software engineering as a discipline was born from those discussions and reports.

Cognitive Limitations of Humans: We are terrible at risk assessment for large scale systems You will read about the Software Development Lifecycle (SDLC) on any cursory search around the subject of software engineering. The structure an SDLC provides deliberately seeks to overcome individual cognitive limitations for risk assessment. Myriad human disasters are attributable to ignoring readily available information, and often the informed, passionate judgment of experts.

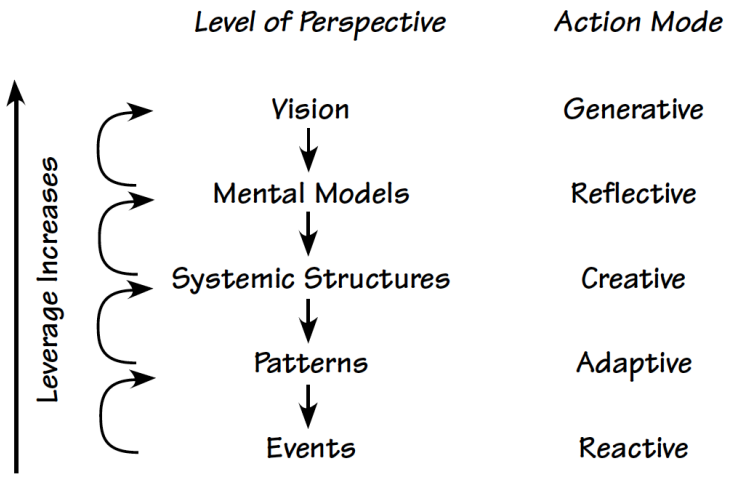
Software engineering is a response, a reaction, to continuous, catastrophic failure in the early days. The steps have been adapted in many ways, and our understanding an SDLC that is sufficient for building apps on our phones is ill equipped for software that functions in safety critical systems like airplanes, automobiles, and medical devices.

1.3 A Systems Theory Lens

From Reactive to Generative Though what software engineering "is" has evolved mightily in the past 50 years it cannot escape its origins in crisis. When an unforeseen crisis emerges, systems must react, and reaction is the least strategic and most stressful way to build and evolve systems. The stepwise, "Waterfall" method was a reaction to failure. As a system stabilizes, the sociotechnical system can begin to operate more strategically. The agile software movement, which I elaborate on at some length later, was I think the first significant advance of practice, enabling our evolution from reactiveness toward a reflective way of thinking about the structuring of large scale software projects. Boehm's Boehm (1988) spiral model is the first SDLC to demonstrate the idea of checking each phase against the prior one in the SDLC, and allowing for modification of a design, for example, based on what we learned in early construction. The process is inherently reflective.

What we call "agile software development" today is a descendant of Boehm's spiral model.

Figure 1.1
Systems theory accounts for, among other factors, the conditions under which people are making decisions, then maps those conditions to particular ways of viewing the world. When you are reacting, you are taking in events. Like computer scientists and practitioners in the late 1960s. In other moments, these same people surely operated in higher leverage modes as described in this figure from Kim (2000).



1.4 Systems Thinking

The central aim of my use of systems theory in this book is to apply a lens for understanding software engineering and the reasons it exists at is does. Through that understanding I think you will be better able to apply the most sound aspects of software engineering to a particular situation.

Know How and Understanding Ackoff et al. (1997) pointed out the critical importance of distinguishing between thought that seeks to make sense of how things work the way they do, and *why things work the way they do*. The difference is subtle and significant in my view for competent software engineering practice. Begin your journey by considering how you react to different software engineering challenges placed before you with the figure outlining the modes that provide you, and your mates, more leverage.

2 Software Engineering's Origin Stories

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Software Engineering became a discipline largely in response to failure. Over the past 50 years, it evolved into a complex collection of methods, and processes that share a fundamental checklist with each other: Problem or need identification, requirements gathering, design, implementation, release, maintenance, and adaptation to new problems. The differences between different approaches are primarily the amount of time, overlap, and scope of delivered, working software that emerges from the completion of a cycle.

Albert Einstein is famously quoted to argue that "the universe if made of stories, not atoms. Today, arguably, the universe, or at least our portion of it, is made of stories and code. Understanding a bit about what can go wrong on a grand, or local scale, is best achieved through stories. Some are substantial, like the NATO reports of 1968 and 1969. Others, less so grand, but possibly easier to relate to in a contemporary context.

2.1 Problems on a Grand Scale

Box 2.1
The NATO Reports

If one reads the NATO reports, how participants describe the problem at the time is interesting, because this was a period of requirements gathering for computer science. We created significant technology, but effectively were incapable of using programming languages to build reliable systems. I have a few observations:

1. It is interesting to note that some viewed the problem as restricted to a certain types of projects: large, complex, and/or critical importance (life and death): There are many areas where there is no such thing as a crisis: sort routines, payroll applications, for example.
2. Dealing with the risks and uncertainty of developing *new* software: "In computing, the research, development, and production phases are often telescoped into one process. In the competitive rush to make available the latest techniques, such as on-line consoles served by time-shared computers, we strive to take great forward leaps across gulfs of unknown width and depth. In the cold light of day, we know that a step-by-step approach separating research and development from production is less risky and more likely to be successful. Experience indeed indicates that for software tasks similar to previous ones, estimates are accurate to within 10–30 percent in many cases. This situation is familiar in all fields lacking a firm theoretical base. Thus, there are good reasons why software tasks that include novel concepts involve not only uncalculated but also incalculable risks."
3. Projects running over-budget
4. Projects running over-time
5. Software was very inefficient
6. Software was of low quality
7. Achieving reliability
8. "seemingly unavoidable fallibility of large software"
9. Software often did not meet requirements
10. difficulties meeting specifications
11. Projects were unmanageable and code difficult to maintain
12. Regarding hospital systems: "This kind of system is very sensitive to weaknesses in the software, particular as regards the inability to maintain the system and to extend it freely."
13. Software was never delivered

Of equal interest is the ideas the NATO reports have for addressing the crisis as they saw it. For example:

1. Note about NATO 1968: They argued about terminology and distinctions between design, production and service. Some of their terms are different than what we would say today. sometimes they got sidetracked in minutia. Some of what they were focused on does not apply today as hardware and system software has changed, but most of it does.
2. Notes about NATO 1969: There was a large communication gap among participants that had an effect on the effectiveness of the conference itself. Perhaps this is analogous to what can happen in large software projects. Further, time was also spent discussing non-software engineering solutions needed to some of the problems, such as ways to reduce hardware dependencies.
3. CORE IDEA: Develop theory and practice for an engineering approach to software manufacture.

4. ”The phrase ‘software engineering’ was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering.”
5. They believed progress could be made by giving attention to the program design process
   They abstracted the software project activities they were already following:
   They diagramed different views of the activities of a software project and indicated terminology.
6. In another place they listed steps to follow in a logical order.
   REQUIREMENTS/SPECIFICATIONS
   1969: One person proposed an iterative method to get all the specifications up front, but received pushback from other participants that this didn’t work on large projects.
   PLANNING/DESIGN:
   Importance of designing before coding was mentioned several times.
   There was emphasis on decomposing the problem in a structured programming approach. Modules, subroutines, etc..
   Should adhere to fundamental design concepts for maintainability: modularity, rigid specification of the interfaces, and generality
   Design was a big topic in 1968: they discussed top-down, bottom-up, and fit-and-try approaches to designing components. Each approach had advantages and disadvantages.
   It was said that the design must continue to give service in the presence of hardware and software errors.
   The 1969 conference talked about the importance of planning the architecture, how the pieces fit together.
   1969 also discussed designing for portability : considering interfaces and degree of coupling, etc. One suggestion is what we would call a wrapper to increase adaptability.
   CONSTRUCTION: One suggestion: build the software in modules or components, build one and add to it.
   TESTING was a major theme
   an environment for testing
   need a simulation of run time conditions
   testing programs for different abstraction levels: unit, integration, system
   validate design before sending it out
   ”a software system can best be designed if the testing is interlaced with the designing instead of being used after the design.”
   They talked about simulation in testing, using dummy modules.
   Testing should be automated. All parts of the system should be exercised.
   Customers need to test on-site. Should have redress if things don’t work.

Many additional observations emerged from these conferences where software engineering was, in effect, ”born”. A few ideas culled from the reading of the resulting reports worth note: - Measurement of software in production: performance, conformance to requirements, acceptability

-in 1969, they also discussed formal proofs of correctness, but there was pushback on the practicality

1. To address understanding new problems, quality and meeting requirements, FEEDBACK was emphasized:
2. Collecting data / feedback: "One must collect data on system performance, for use in future improvements."
3. One approach was Development/improvement cycles: produce working product, then re-design
4. There needs to be feedback between "external" and "internal" design or we might say between the requirements and the design. Does the design match the requirements? If not, which needs adjusting?
5. Part of the issue was the need to go back and re-design when it is clear the problem was not understood.
6. COMMUNICATION was stressed as essential for many of the issues.
7. They considered how the people should be organized, and when, how, and between who communication occurred
8. Documentation as a vehicle of communication: must be clear and organized.
9. Documentation tailored to the audience: end user, technical manuals. Maintenance of documentation should be automated.
10. They proposed that there be standards for documentation and diagrams to aid in communication
11. NOTATIONS: A concise mathematical notation to express structures and relationships would be useful. For example, Boolean Algebra is used for CPUs need something similar for software notation.
12. they discussed clear communication, use of automated tools for communication, having "friends" be team-mates, teams not too big.
13. STANDARDIZATION was often proposed
14. common or standardized program structures and program flow
15. standardized notation systems
16. standard processes
17. Using a common language for projects increased programmer productivity
18. the need for standards to increase portability
19. TOOLS to increase efficiency and reduce error were proposed
20. They proposed the development of tools, such as general purpose compilers, assemblers, loaders
21. A common area file storage with backups

22. It was observed that many problems were MANAGEMENT problems and not software engineering problems per se.
23. To address problems with time and budget:
24. ”Programming management will continue to deserve its current poor reputation for cost and schedule effectiveness until such time as a more complete understanding of the program design process is achieved.”
25. it was noted that many aspects of the process were usually underestimated, only the best case situation was planned for, and that documentation not being up to date caused delays, and rushing into development before program blocks and their interfaces were fully planned. The implication is that better estimation and planning techniques are needed.
26. 1969 had a working paper discussing various aspects and approaches to estimation.
27. A lot of measurement is geared toward gathering data to enable better future estimations.
28. Huge productivity differences in programmers. This leads to issues in estimation.
29. They discussed the difficulties of how to know where a project is and how it is going. How to measure progress? Hard to decipher real versus apparent progress.
30. Note which groups of ”stakeholders” each issue was most a concern for
31. The idea I’m laying the ground for is that many of the agile concerns address developer needs and gloss over business/management needs especially in less flexible environments such as government.
32. Primary Stakeholder concerns:
    Business/Management:
    Projects running over-budget
    Projects running over-time
    Software was never delivered
    Users
    Software was very inefficient
    Software was of low quality
    Software often did not meet requirements
    Developers
    Projects were unmanageable and code difficult to maintain
    Users don’t know what they want or have trouble communicating it

The NATO conferences, and the resulting reports produced a set of goals for software engineering. First, and foremost: ”Make good software.” This seems like the computing equivalent of ”Bill and Ted’s Excellent Adventure”, in a way, as its mantra was ”Be excellent to each other.” Both phrases smack a bit of being delightful platitudes. However, in the case of software engineering, a few suggestions are boiled down from the reports, and provide a high level guide for what one must do to ”make good software”:

1. The application of a systematic, disciplined, quantifiable approach to all aspects of software production from problem definition through maintenance.

2. Goals of SWE
   - Dependability
   - Maintainability
   - Efficiency
   - Acceptability
   - Security

2.2 Problems on a localized Scale

Software engineering is the intersection of the skills involved in computer science, human relations, working in teams, and understanding systems involving both people and technology. When a computer programmer, or software engineer goes to work every day, its not just programming skill that determines their short, and long term success.

Box 2.2
A Handheld Bridge Inspection System

I wrote software professionally for seven years before I ever had a course in software engineering. In my first programming job, I was the only programmer in a small railroad bridge engineering firm, and much of our business was the inspection of existing railroad bridges. My process included no version control system, but other parts of following a software engineering process were somewhat natural in a small context.

First, the engineers used a series of paper forms for inspection before they had a computerized, handheld system for recording information. So, the first thing I did was spend weeks in the field dangling off of bridges, wearing fall protection, and talking with the engineers about what they looked for and how they proceeded with each inspection. I learned there are three main types of bridge structures: Wood, concrete, and steel. Each required different information.

Second, I chose steel as the first type of structure to write software for more automated inspections. I designed a simple system that automated part of inspection work. Then I returned to the field to see how that software helped, or hindered, the inspection process. Mostly I learned about hindrances.

Third, I used the insights gained from that first attempt, which I now recognize more as a throw-away prototype than software, to design a system that was more natural to the way engineers did steel bridge inspection. Fourth, I built, and bench tested the software, and fifth I field tested the software again.

The second field test, this time with a working piece of software, exposed some small changes needed, but largely formed the foundation for a system that evolved to include inspection of wooden and concrete structures as well.

Later in my career, when I finally took a software engineering course at the University of Minnesota, I recognized many of steps from seven years of practice. In a small irony, the bridge I crossed in my car to get to class in Minneapolis, collapsed, leading to the extensive inspection of automotive bridges [1]

.

[1] https://en.wikipedia.org/wiki/I-35W_Mississippi_River_bridge

3 Distributed Version Control

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Version control systems are central to software engineering practice. Without knowing how to use one in the context of team oriented software engineering, you will struggle to make your contributions to a project without smashing another computer programmer's efforts. Or, perhaps, spending hours or days reconciling your changes with those of others.

3.1 Centralized Version Control

Before 2010, nearly all widely used version control systems for software were centralized. Systems like "CVS", and later "Subversion" were ubiquitous in academic, corporate, and government software development. Many systems developed before 2010 continue to use centralized systems for version control. The central feature of these types of systems is that there is one, centralized location for the management of software versions. It is a client-server type of system. To work on code, you must check it out of that system, and check it back in when you are finished. The opportunity to test your code against other changes being made by other developers, on different files at the same time does not exist. You know whether or not your code "works" in the system, as its revised daily, is revealed by a "nightly build". These types of systems are non-trivially harsh, because its often embarrassing to "break the build", even though you have no way of really knowing if your new code is going to interact poorly with some other person's new code from the same day.

3.2 Distributed Version Control

In the early 2010's, Linus Torvalds developed a distributed version control system called "Git", because he was tired of having to reconcile the errors produced when integrating code from dozens of developers regularly. His intentions were simply to allow each devel-

oper to test their code against other developers locally, and solve these types of integration problems themselves.

If you ”fork” a repository on GitHub, and spend a few days making a change, you can determine whether your change will work within the modified development branch of the server's copy of the repository by following two, basic steps:

1. Merge the latest version of the main repository that you forked – probably the dev branch – into your local repository.
2. Compile and run the system. If it works as you expect, you then merge your new code into the same branch of the dev repository on a server.

Each of these ”merging” steps, is implemented differently by different ”Git” Platforms. GitHub calls it a pull request, GitLab calls it a ”Merge Request”, and Gerrit calls it a ”Change”. Each platform has a different name for the mechanisms they use to perform exactly the same functions in the underlying technology they share, called ”Git”.

II Software Engineering Models and Methods

4 Modeling Software

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

4.1 A

1. Database design work
2. Writing and Coding
3. Architectural and Component Modeling at MDCon

4.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

5 Methods: The Software Development Lifecycle

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

5.1 A

1. SIP:
2. Waterfall horror stories MDC
3. Project Velocity G

5.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

III Requirements Management

6 Requirements Management Processes and Tools

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

6.1 A

1. Sportsorg - and TNc - Spreadsheets. Clarity for estimates, tools for trade offs
2. OPen source, issues and market demand

6.2 B

1. A
2. B
3. C
   - C.1

Exercises

1. x

2. Cyb

a) test

b) test

7 Working with Use Cases

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

7.1 A

1. First time with use cases in MN. What are they? How do they help?
2. OTS - Use cases to help reveal a project you don not want to take.

7.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

IV Systems Theory and Ethics

8 Systems Theory in Software Engineering

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

8.1 A

1. Pets - 23 plants with database version issues.
2. SSO - competing interests .. not always above board
3. Adducci

8.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

9 Ethics in Software Engineering

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

9.1 A

1. The Millpic - valuing developers, charging for jerks
2. DSA Program

9.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

10 Collaboration, Coordination and Group Work

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

10.1 A

1. GSOC
2. COVID
3. Do not tolerate poor behavior
4. Professional Ethics references

10.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

11 Conflict and Communication

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

11.1 A

1. The dangers of text based communication: Emails, texts, and feelings
2. Can the conflict be avoided? "Is this a hill worth dying on?"
3. Stigmergic practice in open source

11.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

12 Metrics and Measurement

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

12.1 CHAOSS

1. Activity
2. Long Range Health
3. Negotiating metric definitions

12.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

13 Community Building

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

13.1 A

1. Getting started with Augur
2. Basic installation stuff

13.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

14 Software as a Complex System

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

14.1 A

1. Research software with Introne
2. Adapting to the evolution of products outside your immediate portfolio
3. Dependencies

14.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

V Technology Architecture and Software Design

15 An Overview of Technology Architecture and Software Design

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

15.1 A

1. Augur architecture
2. Architecture of large projects: Velocity

15.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

16 Sketching

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

16.1 A

1. Augur and Sketching the original
2. Augur refactorings

16.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

17 Prototyping

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

17.1 A

1. xmatch: Anti-patterns: When the prototype becomes the system
2. People suck and defining what they want

17.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

18 Design Modeling

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

18.1 A

1. MDC: Admin portal. Everybody is creating data, and believing some magical team in the sky is going to provide them an editor for that data .. nope.
2. Treating plant process control systems: Too much modeling, not enough building.

18.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

VI Human Computer Action and Social Computing

19 People

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

19.1 A

1. The systems we build can cause pain, and my decision to become an academic emerges from that experience.
2. A/B Testing
3. Use sketching and prototyping to draw people out.. draw out their requirements.

19.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

20 HCI: Human Computer Interaction

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

20.1 A

1. Auggie and push notifications
2. Task analysis
3. Game design

20.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

21 Social Computing

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

21.1 A

1. Gud: Marketing system
2. Email overload and Slack
3. Social media and influence / Eva paper.

21.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

VII Construction

22 Software Frameworks

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

22.1 A

1. Everybody wants to build a framework, nobody wants to use one.
2. J2EE: MDC
3. Peoplesoft
4. javascript

22.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

23 Programming Languages, Databases, and Algorithms

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

23.1 A

1. Teaching myself C for a project
2. Putting everything together while you are learning about them.

23.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

24 Continuous Integration

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

24.1 A

1. Augur: Release Struggles solved via travis-ci
2. Don't break the build!

24.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

25 Releasing Software

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

25.1 A

1. Declaring a project finished and seeing it released six months later is a shell game. Avoid it.
2. Call center: The initial release has to work, and you need a rollback plan.
3. Subsequent releases: Don't take things away. Small moves. Small changes.

25.2 B

1. A
2. B
3. C
   - C.1

Exercises

1. x

2. Cyb

a) test

b) test

VIII Testing

26 Test Driven Development

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

26.1 A

1. Write the tests first so you know everyone agrees on how the components interface. Gets the API down ... IX
2. Test first exposes undocumented features and things you forgot.

26.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

27 Writing Software Tests and Evaluating Coverage

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

27.1 A

1. GD: In safety critical systems every possible state needs to be covered. You also need "fail safes".
2. What result are you looking for in a test of a non-deterministic system like your Facebook feed?

27.2 B

1. A
2. B
3. C
   - C.1

Exercises

1. x

2. Cyb

a) test

b) test

28 Continuous Integration and Testing

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

28.1 A

1. Travis-CI or other continuous integration tools.
2. Always

28.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

IX Maintenance and Evolution

29 Modifying Released Software

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

29.1 A

1. WebMD

29.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

30 Managing Dependencies

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

30.1 A

1. Libraries.io
2. The working group Risk for CHAOSS

30.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

31 Evaluating Impact of Proposed Changes

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

31.1 A

1. Question why the change is needed guid ... y2k
2. Small changes mean small impact. Augur iterations.

31.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

X Software Engineering Projects

XI Essential Orthogonal Disciplines

32 Documentation

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

32.1 A

1. readthedocs.io and augur
2. user documentation v developer documentation

32.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

33 Security

"Don't go chasing waterfalls Please stick to the rivers and the lakes that you're used to I know that you're gonna have it your way or nothing at all But I think you're moving too fast"
—TLC

Abstract. Long. Short. Sequential. Iterative. Different types.

33.1 A

1. Open SSL
2. Equifax Breach
3. Apps that send your data back somewhere. Tik Tok.

33.2 B

1. A
2. B
3. C
    - C.1

Exercises

1. x

2. Cyb

a) test

b) test

Bibliography

Ackoff, Russell Lincoln, Lauren. Johnson, and Systems Thinking in Action Conference. 1997. *From mechanistic to social systemic thinking : a digest of a talk by Russell Ackoff*. Cambridge, Mass.: Pegasus Communications. https://www.youtube.com/watch?v=yGN5DBpW93g.

Boehm, Barry W. 1988. A spiral model of software development and enhancement. *Computer* 21 (5): 61–72.

Buxton, John N, and Brian Randell. 1970. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*. NATO Science Committee; available from Scientific Affairs Division, NATO.

Kim, Daniel H. 2000. Introduction to Systems Thinking.

McClure, Robert M. 1968. *NATO SOFTWARE ENGINEERING CONFERENCE 1968*.