Engineering Software

And Other Creative Improbabilities

Sean P. Goggins, Ph.D

Publisher Imprint
Columbia, MO
London, England

Copyright 2020, Sean P. Goggins, Ph.D
All rights reserved.
Draft Assembled on September 8, 2020

“*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.

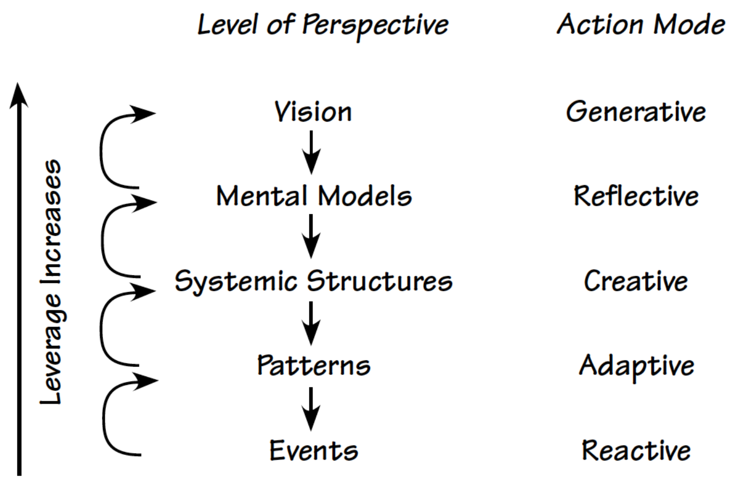
Software engineering is a response, a reaction, to continuous, catastrophic failure in the early days. The steps have been adapted in many ways, and our understanding an SDLC that is sufficient for building apps on our phones is ill equipped for software that functions in safety critical systems like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

"UML has had an impact that far exceeded its original aspirations. Industry has been using it extensively for a wide diversity of applications, it has spawned a whole new area of academic research, and it is taught in practically every software engineering curriculum in the world. Moreover, it has had significant influence to other engineering domains such as systems engineering, business process modeling, and aerospace and defense."
—Bran Selic, President, Malina Software Corp., Canada

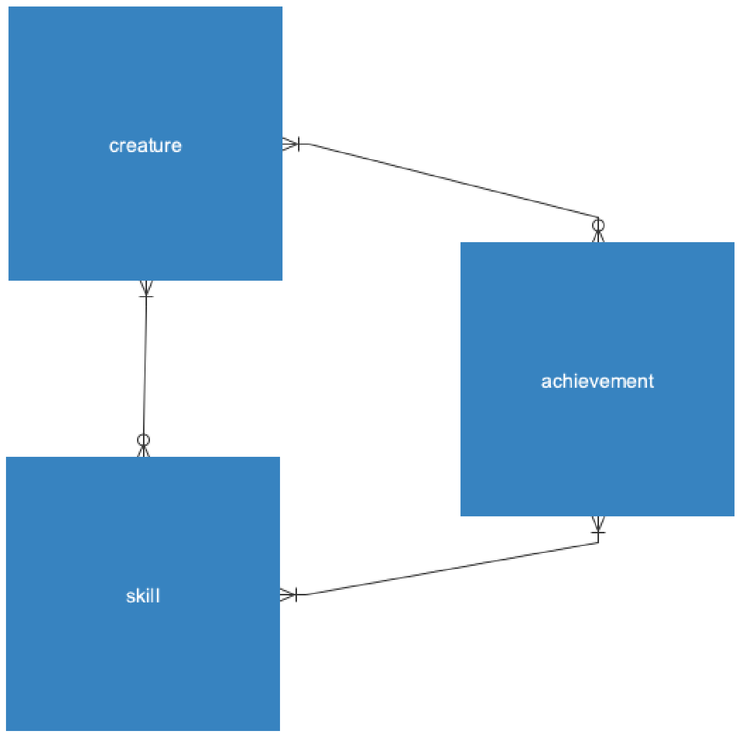
Abstract. Modeling software solves coding problems before you spend a week creating them. Updating models as the code evolves should either be automated, or avoided at the most detailed level, though high level models of components are often useful for new contributors to a system. All software models are static, representing code or data components, or dynamic, representing interactions between components. There is no single, correct way to model software or data, but there are preferred syntax in each case. Unified Modeling Language (UML) is widely used for modeling software. Entity Relationship Diagrams (ERD) are the most common syntax for modeling databases, their tables, and the relationships between them.

4.1 Database Design Work

Data Modeling is Requirements Gathering The effect of identifying the data your application will keep track of demands that you ask your users questions, and it demands that you "think about what without how", "think about data without processing", and "articulate types without instances" Carlis (2000). Database management systems like Postgresql, Oracle, and others do the work of holding the data you identify somewhere. When you think about data as a static collection of things, without considering processes for those things, then you are constraining the task to identifying what it is your application must track, with some precision. Conversations that drift from data modeling toward data processing too early are likely to confuse both you and your stakeholder. Finally, when you articulate types without instances you are giving the user back, in return, the classification of the things you heard them describe as important. From there, the users will question if particular instances fit into these new categories.

"Unique Identifier" is a commonly used term, but it is one word too long. If something is simply an ”identifier”, by its nature it is unique. Most of the tables you design will have one, though a special type of table that maps many to many relationships may not. The physical implementation of a many to many relationship is a ”bridge entity” (in a logical data model), or ”bridge table” (in a physical data model). You will see that fig. 4.1 illustrates how many to many relationships can be modeled conceptually, without bridge entities.

Figure 4.1
A conceptual data model allows us to express high level relationships between concepts that are part of the language whomever a system is being built to serve. One of the dangers of software engineering work you might attend to is delving into details too quickly. This conceptual model shows us that there are always creatures, as indicated by the solid line in the chicken feet Carlis (2000) attached to it, and there are one or many of them. That ”cardinality” is reflected by the presence of the chicken feet instead of a single line. The circles on the chicken feet between creature and skill, and creature and achievement, indicate that each creature may have zero or many skills and achievements.



In fact, conceptual modeling by its nature excludes the names of ”attributes” (logical data models) and ”fields” (physical data models) present in logical and physical model. From a modeling perspective, the conceptual model steps back from considering identifiers and bridge entities/tables, and focuses the mind on inventorying all of the high level ”things” a system will save in its database. You do not want to begin ”in the weeds”, so to speak. Start by standing atop a mountain.

Logical Data Modeling Once you have the conceptual entities, or ”buckets” that you know the system will need to keep track of, and the approximate relationships between them, you can begin logical data modeling. At this stage, you might engage a user or some expert in an exercise of listing examples of all of the entities in the conceptual data model, working one entity at a time. Slowly, perhaps using a spreadsheet, a list of example values will emerge for each creature in our example. Same for each skill, and for each achievement. There is not a single, correct answer in data modeling, but reflecting the users data accurately is the goal. The logical model in fig. 4.2

A next evolution, then, would be to make both columns from creature-skill, and creature-achievement part of a multi-column key. You see this change in the little ”key” symbols in fig. 4.3.

Some achievements may have many levels, like the different colored belts one can earn in the martial arts. ”Karate Belt” could be an achievement, and a creature could attain many different levels. If we wanted, for example, to enable this in our logical data model, we would add an attribute like ”skill level”. This is reflected in our next logical iteration, illustrated in fig. 4.4

Finally, A Physical Data Model After the process of talking with users and other experts about conceptual, and logical data structures, we are ready to implement a physical model. Most often, this is done with a tool that allows you to progress from conceptual, to logical, to physical. The biggest difference between logical and physical is that the datatypes will be specific to the relational database you use for deployment. We see fig. 4.6 implements the database using postgresql.

Many of these tools will also do ”round trip database engineering”, although there are imperfections in each of them. You can connect a modeling tool to an existing relational database, and get a physical model generated in the tool. When, as you are in this case, building a new data model, it will generate SQL that is compliant to the physical platform you chose. However, you will notice that it made the ID’s for each table numeric. For our bridge entities, this is fine because they will have foreign keys for their values in creature, skill, and achievement. Postgres, however, has a datatype called ”Serial8” that will autogenerate new key values when a row is inserted using a sequence object (basically, a counter). This makes programming to our data model significantly easier, and you can see the change in **??**

Figure 4.2

A logical data model allows us to express relationships between creatures, skills, and achievements, while also providing us with the example of where bridge entities, sometimes without their own identifiers, emerge. You can see in fig. 4.2 that two new logical entities, ”creature_skill” and ”creature_achievement exit to make the relationship logically conceivable. Alternately, without a relational structure, we would have one, giant table with all of each creature’s skills and achievements. In this example, you see that the creature_ID is the key in both cases. Does this work? Probably not, because it would limit every creature to ”one achievement”. To overcome this limitation, we assume ”survival” is not each creature’s only achievement, so we need either a multi-column key, which will be easier to query, or an automatically generated identifier like we use for creature, skill and achievement.

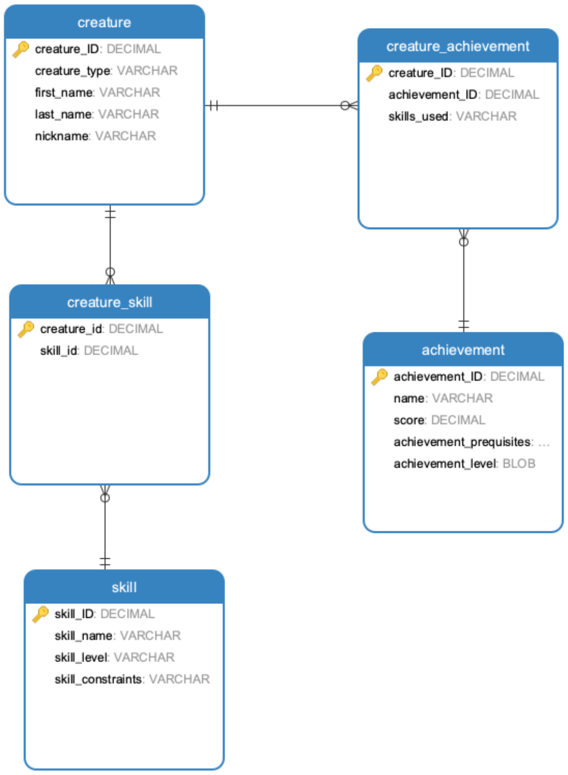


Figure 4.3
A logical data model with a bridge entity containing a multi-column key makes an auto-generated key, like we have in creature, skill, and achievement, unnecessary. It also prevents a creature from having duplicate skills or achievements.

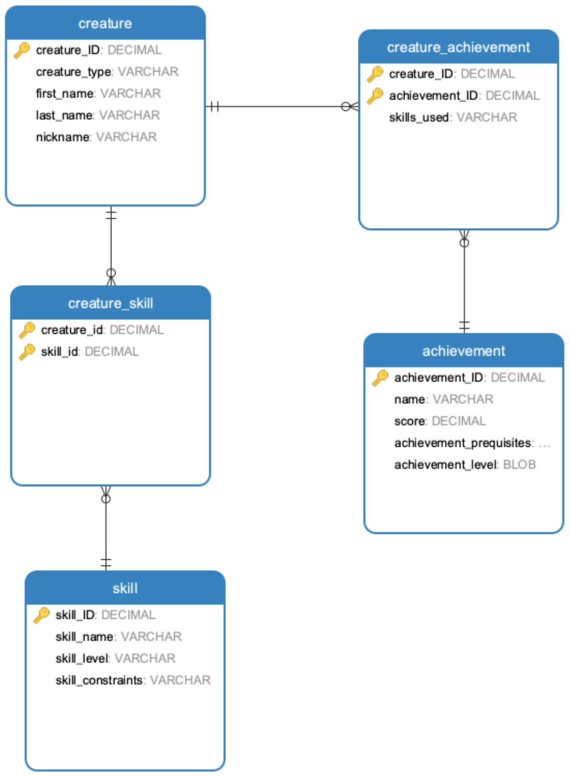


Figure 4.4
A bridge entity with a multi-column key can also add an additional column to the key, to represent levels. Although, as we begin to design like this, an auto-generated key begins to seem like a better, more maintainable choice.

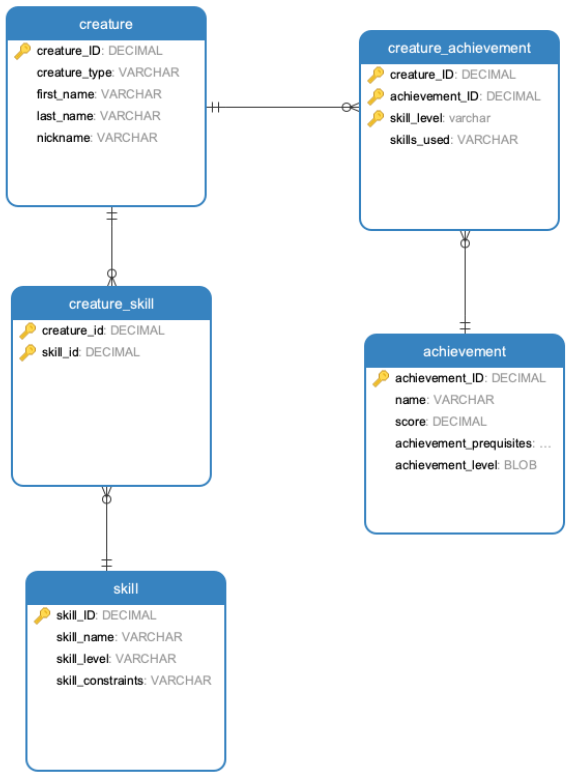


Figure 4.5
A Physical Data Model with a multi-column key can also add an additional column to the key, to represent levels. Although, as we begin to design like this, an auto-generated key begins to seem like a better, more maintainable choice.

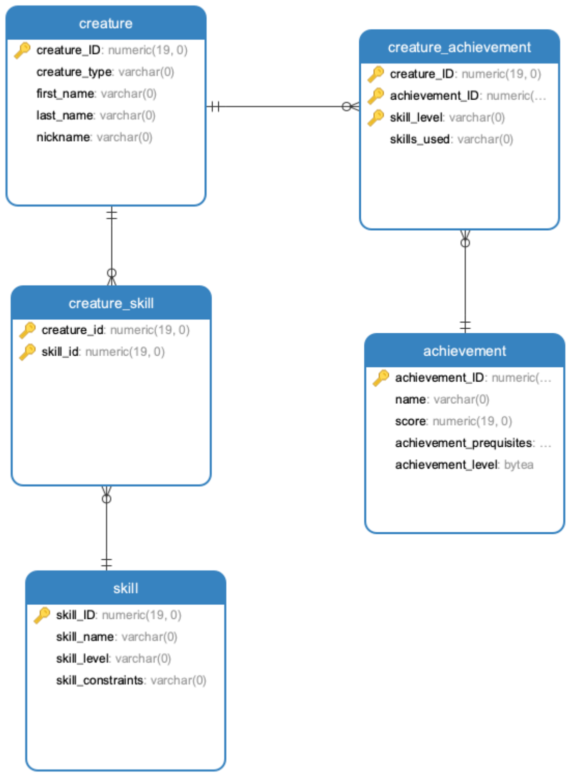
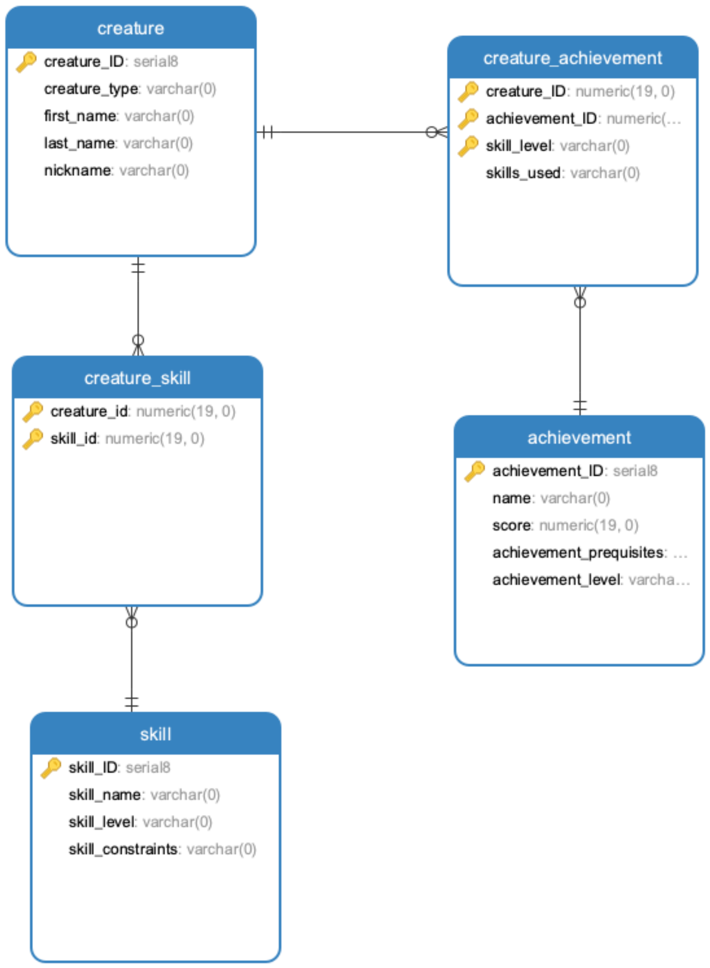


Figure 4.6
A Physical Data Model with a multi-column key can also add an additional column to the key, to represent levels. Although, as we begin to design like this, an auto-generated key begins to seem like a better, more maintainable choice.



The SQL for the physical model in fig. 4.6 is:

```
CREATE TABLE "creature" (
"creature_ID" serial8 NOT NULL,
"creature_type" varchar,
"first_name" varchar,
"last_name" varchar,
"nickname" varchar,
PRIMARY KEY ("creature_ID")
)
WITHOUT OIDS;

CREATE TABLE "skill" (
"skill_ID" serial8 NOT NULL,
"skill_name" varchar,
"skill_level" varchar,
"skill_constraints" varchar,
PRIMARY KEY ("skill_ID")
)
WITHOUT OIDS;

CREATE TABLE "achievement" (
"achievement_ID" serial8 NOT NULL,
"name" varchar,
"score" numeric(19),
"achievement_prequisites" varchar,
"achievement_level" varchar,
PRIMARY KEY ("achievement_ID")
)
WITHOUT OIDS;

CREATE TABLE "creature_skill" (
"creature_id" numeric(19) NOT NULL,
"skill_id" numeric(19) NOT NULL,
PRIMARY KEY ("creature_id", "skill_id")
)
WITHOUT OIDS;

CREATE TABLE "creature_achievement" (
"creature_ID" numeric(19) NOT NULL,
"achievement_ID" numeric(19) NOT NULL,
"skill_level" varchar NOT NULL,
"skills_used" varchar,
PRIMARY KEY ("creature_ID", "achievement_ID", "skill_level" )
)
WITHOUT OIDS;

ALTER TABLE "creature" ADD CONSTRAINT "fk_creature_creature_skill_1" FOREIGN KEY ("creature_ID") REFERENCES "creature_skill" ("creature_id");
ALTER TABLE "skill" ADD CONSTRAINT "fk_skill_creature_skill_1" FOREIGN KEY ("skill_ID") REFERENCES "creature_skill" ("skill_id");
ALTER TABLE "achievement" ADD CONSTRAINT "fk_achievement_creature_achievement_1" FOREIGN KEY ("achievement_ID") REFERENCES "creature_achievement" ("achievement_ID");
ALTER TABLE "creature" ADD CONSTRAINT "fk_creature_creature_achievement_1" FOREIGN KEY ("creature_ID") REFERENCES "creature_achievement" ("creature_ID");
```

Box 4.1
Software for Reading Data, and Writing Data are not the Same

Working for a large publisher some years ago, I joined a project as a ”Software Architect”. There were four of us on a development team of 150, and we reported to a Chief Software Architect. Each of the four software architects were responsible for a different component of this rather large system. Periodically, we would get together with the CSA offsite to spend the day looking at overall progress. Another architect, Candice (not their real name), and I had a question during one of these meetings early on. We asked, how is the information about publication pricing, packaging, and formatting going to be changed as the system evolved? Our goal was to bundle different publications together into subscription sets, and sell them to institutions in new ways. Obviously the company thought they would make a lot of money doing this, because the project had alread spent 20 million dollars when I showed up. When I said, ”I am not sure I understand. We are all here looking over the project and asking questions. But we can’t ask you questions?”

The CSA, Mike (not their real name) blew his stack. He was one of those managers not accustomed to being questioned, and I think he *really* liked being the *Chief* Software Architect. He left the offsite meeting in a bit of a huff. After he left, Candice said, ”Oh my god! You’re not supposed to ask him questions. We think he’s literally insane.”

Oh. That would have been something helpful to know *yesterday*.

Seven months later, at the end of June, during a week I was on vacation, I had 7 voice mails (texting was less common back at that time.). Four were from Candace, and three were from the Vice President in charge of the project. I called Candace first, and she filled me in that they had laid off Mike, 2/3 of the developers, and the other two software architects. We were spared. When I called the VP back, he said the ”RIF’s” (a lovely euphemism for layoffs) were required because the project was about four months from its first live test, and Mike had led him to believe the ”Admin Portal”, which to us was the mysterious piece of software architecture that would edit the data, did not have anyone leading it actively. Apparently, that was Mike’s job.

Do not be afraid of asking hard questions. Some day, doing so may save you. And I lost 50 pounds working our for 90 minutes every morning after that meeting with Mike, mostly because I had never worked for somebody as boisterous and incompetent. It was also a lesson, for me, in taking time and asking more questions before accepting a job offer.

Candace went on to be the CIO of a Fortune 500 company, I started working on my Ph.D so I could be a professor, and the one ”useful” project manager we worked with started his own IT consultancy, and now employs over 300 people. Another point of this little story is that the people who are both competent technically, and effective at communicating, ultimately have an advantage.

4.2 UML Design

UML Modeling is a solution to a problem you will encounter, sooner or later in your career. Probably sooner: Some systems are so complex, that they require careful thinking about what software should be written, and how to structure the components, before coding begins. UML is also helpful for ”fixing” software projects you might encounter, that have become unnecessarily complicated. The distinction between complex and complicated is important in software. When the problem software is being designed to solve is, itself *complex*, then understanding those complexities is essential for discovering the better designs within a sea of bad ideas.

Complex Projects Requirements gathering will help to reveal if the problem is complex, or if its not. For example, an engineering design to manufacturing software system is complex for three reasons. First, engineers typically use computer aided drawing (CAD) software to design a product. Second, there are a limited number of suppliers for many required parts, so sometimes the product design changes, and sometimes the manufacturing process design, which is determined through analysis of the CAD designs and available manufacturing equipment changes. Finally, as soon as the production of the product begins, there is a potential that product modifications will continuously flow through this process as result of market demand. The value of UML models in these cases is high, because they provide an opportunity to ”get it right” at the design stage.

Complicated Projects Unlike complex projects, complicated ones usually arise because of organizational issues that confuse motion with progress. At some point, a group, or groups, started frantically writing code in order to get some technology to market, or accomplish an urgent organizational goal. When non-trivially large projects face this type of pressure, UML models are helpful with either of the two likely outcomes for a complicated project: 1) it crashes and burns, but somebody still has to get it to work, or 2) it launches, and is an expensive to maintain, confusing, and typically poorly documented mess.

UML for Software Design and Software Archaeology In the complex project case, UML helps us make design decisions before we write code. In the complicated project case, UML is helpful for deconstructing a project to understand how to get it back on track. UML for deconstruction is part of what I call software archaeology.

4.3 Static and Dynamic Models

Static Models Class diagrams in UML are especially useful for understanding how a set of components should, or do work with each other. Like with data modeling, we may begin with more conceptual representations of the software, like a component diagram that puts a set of classes that operate as a unit together in a ”package” or a component. Once their, we add the details of state, the internal data structures of a class diagram, and behavior, the methods on a class diagram. Often a classes data and methods will follow software patterns [4], and these patterns are helpful for addressing complexity during the design process, and deconstructing how far a failed or failing process is, and which parts of the existing code pose the most trouble. In a sense, static models in UML and data are both detailed inventories of the software that is written, and how it is intended to work.

Dynamic Models Sequence and activity diagrams in UML are dynamic models. They allow you to design, or visually deconstruct how software is working in practice. What are the interface signatures between classes and modules helps us to more clearly understand the significance, responsibility, and code quality of each component of a system.

[4] Software design patterns provide a general template for how to write software that performs common classes of tasks in software. Common, useful patterns include the singleton, the factory, strategy, observer, builder, adapter, and state patterns. The internet is filled with examples of how to implement these patterns if most languages that are object oriented

Box 4.2
UML in the Coffee Generation: That Day I Met Butch Vig

I went to the University of Wisconsin when some of the more popular bands among college boys were Die Kreuzen, and Killdozer. My friend at the time, a musician named Marques Bovre, was just starting out his career as a kind of sarcastic, funny, folk singer type. He recorded his first tape, ”Living and Dying in the Coffee Generation” with his band, ”The Evil Twins”, in a studio run by Butch Vig, who happened to live in Madison at the time. One night, before Marques performed at the open mic on the Union Terrace at the university, which overlooks Lake Mendota, we were having a beer, and he said, ”You gotta meet the guy who owns the studio I recorded this in. He’s almost famous, but he’s smart as f*** and is going to do big things.” We were young, and we all thought of ourselves and our futures with a certain amount of grandiosity.

I honestly had heard Die Kreuzen, and Killdozer, and thought they were ok, but I did not feel like I was about to meet a person who would actually go on to do amazing things[5]. Butch sat down with Marques and I, bringing along a fresh pitcher of beer, as is the culture and tradition in Wisconsin. What we talked about for a couple hours is mostly lost to the brain cells found by the beer, but he said one thing that stuck with me (surely I am paraphrasing): ”Some bands are a mess, and you have to move them around, play with ’em a bit before you understand how to make a record. Other bands are driven. Sort of on a mission. You have to figure out if the mission they think they are on makes any sense. But whatever band I am working with, I have to have a picture of the Album’s zeitgeist before we start picking tracks. I can get the picture by rearranging and remixing a lot of junk, or I can get it by moving the songs they come in with around a bit, and getting them to hold together.”

Modeling software is a lot like taking the pieces you have to work with, and making them sound the best they can. You might be a drummer, lead singer, bass player, or whatever. Regardless of your situation on a project, make sure somebody has the picture of where this is all going down in a UML diagram. It doesn’t even have to be UML. It can be boxes and lines. Just have a design before you start laying out tracks. Or create a design if you’re trying to reassemble and redesign a collection of mix tapes into a cohesive, working piece of software.

5 Methods: The Software Development Lifecycle

"I made so many recordings with junky mics and crappy mixing consoles, you have to use what your tools are and in some ways that's been inspirational."
—Butch Vig

Abstract. Methods are part of software engineering because every day provides an opportunity to fail, waste your time, or destroy somebody else's work. Waterfall is "the original" software development lifecycle (method), and every method widely used today's adopts each part of waterfall, in much the same way most bands assemble using a small set of instruments. Software methods all include the identification of some problem that can be solved with software, a definition of requirements, software design, software implementation, testing, releasing for use, and software maintenance. The differences between software development lifecycles, or software development methodologies, lies in the presence or absence of sequence, the specific ways each of those component parts are expressed in the method, and the philosophy of software development buried in each of those choices.

5.1 The Waterfall Method

Don't Go Chasing Waterfalls You recognize that phrase as part of a TLC song by now, and I use it here to state clearly that every project you are on, even if its a new project in a company you have worked at for 10 years, is going to use a software development method that is different than the last one you used. Perhaps subtly different, and maybe dramatically different. Practice is idiosyncratic to a group of people, and each group slowly adopts, or reifies, new practices to meet the distinct challenges of a new project (Wenger, 1998), or the evolution of computing languages and the tools we use to do our work as software engineers. The high level steps are visible in fig. 5.1. In this book we also make a special point of addressing software architecture choices, human computer interaction, documentation, and security. Each of these are sub-disciplines of software engineering also essential to the success of most contemporary projects.

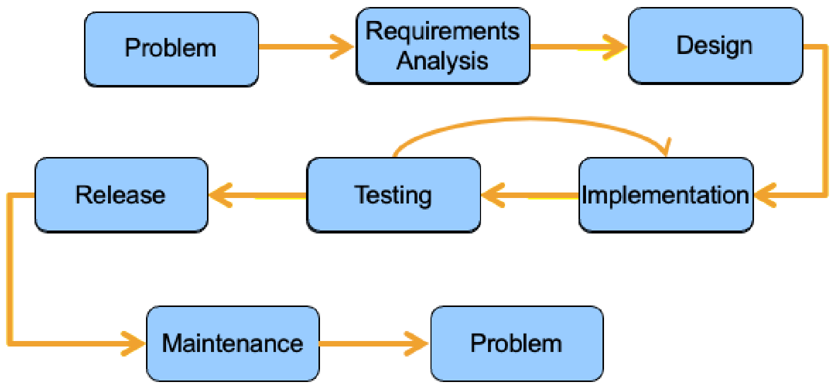
5.2 A Short List of Software Engineering Method Types

The day to day work of software engineering will force you to make small, constant changes to the method your team employs, yet we can define a list of the more common software development methods. This is the author's list, and I encourage feedback and suggestions of your own. First, the methods I have seen widely deployed are all reassembled versions of the Waterfall method that remove, to some degree, the rigorous sequencing of SDLC steps. Removing the constraint that your project cannot go from requirements to design until requirements are ”finished”, for example, is based on the proven software engineering falsehood that requirements are *ever* finished. People are terrible at expressing the design they want, and the most important requirements evolve routinely as they learn more about what is possible or likely in a new or refurbished software system.

Second, existing software engineering methods generally fall into four categories:

1. Heuristic
2. Formal
3. Prototyping
4. Agile

Figure 5.1
The Core Parts of a Software Development Lifecycle (SDLC) and the sequence they occur in, without overlap, in the waterfal method. Not pictured as part of the SDLC, but essential for the success of many projects, are software architecture choices, human computer interaction, documentation, and security. Each of these are sub-disciplines of software engineering.



5.3 Identifiable Methods Within Method Types

Heuristic Methods Take the waterfall method, and reorganize how it is executed by defining specific mechanisms for structuring the process of moving through its phases, while removing the insistence on sequence. Specifically:

1. Structured Analysis and design: Values requirements as an input, and working software as an output. The steps in between are tightly controlled, monitored, and measured. Organizations who hire a lot of MBA's are fond of this method.
2. Data Modeling: Data warehousing, deep learning, data mining, and other projects centered on the accumulation of data from many sources follow a method focused on modeling the data. "Data Science" is one of the term du jour's for data modeling methods. As the founding director, and creator of the core curriculum for the University of Missouri's Data Science Masters program (I committed over 50% of the code for the six core courses, and supervised TA's responsible for 45%.), I speak with some authority when I say that Data Science is a sound and evolving discipline, but its largely a repackaging of methods used for over 50 years, applied to contemporary hardware, software, and data.
3. Object Oriented Analysis and Design: Makes a point of controlling the distinctions between data, and code; as well as the organization of components and classes, and how those pieces of software interact. UML modeling is the central control mechanism for this software engineering method.

Formal Methods Risk is a combination of the likelihood of something bad happening in a context, and the severity of the effects of that bad thing happening. Formal methods are employed in cases where software failures have the potential to kill people. The constraints they introduce focus on testing every possible state a system can be in, which ensures that how a program runs is, as far as possible, deterministic. These methods also include "fail safes", which are system states the software will revert to in the event that an error is detected at runtime. Avionics systems, medical devices, automotive software, and other safety critical systems use formal methods. They are more expensive to implement than all other software engineering methods.

Prototyping Human beings are terrible at expressing what they want the end result of software they will be using to do. Similarly, because software engineering, unlike aerospace engineering, or medicine, does not face any inherent physical or biological limitation. Software is ultimately constrained only by the human imagination. Prototyping, as a method, is about communicating and verifying a shared understanding of software requirements before a real system is built.

Prototypes can be as simple as a sketch, or as complex as a partially working piece of software, depending on the relative importance of cost, schedule, and ultimate software reliability. It seldom is harmful to do some type of prototyping with the user's interaction with a system – the Human Computer Interaction part. There is one *danger* associated with prototyping: If people see a software prototype as progress toward the development of the system they want, they will ultimately feel misled about the schedule.

Box 5.1
Sometimes Method Does Not Matter

My fourth manager when I was a newly minted software engineer was Rachel Haisting, and I was fortunate that she was the best of four very good managers. In our first meeting she said, "I wanted you on my team because your powerpoint presentation explaining what you did to test every major system in our Y2K bunker helped our board recognize that we were, as a company, prepared." Of course I gave the presentation to a bunch of managers, and somebody else took it to the board. And that's fine with me, because I learned my job was more than doing the technical part well.

That experience also showed me that the extremely procedural, formal method of software deployment, upgrading, and testing that I did, alone, in a windowless room for 17 months at a Fortune 500 company was not their only software engineering method. Rachel had to deliver a new data mining system to our marketing organization in four months, and I was going to be her lead software engineer. Its the first software project I led. The first step was finding consultants, because we did not have the engineers on staff to execute a new project like that quickly. "How does choosing consultants put me in charge of a project", I thought. My friends and I had a few beers, and I told them what we were trying to do – use machine learning to identify the market category each of our customers was in – and that my deadline was kind of insane. But beers worked, and my friends identified a small tech company that specialized in using data mining for market analysis. "Perfect!", I thought.

Remembering the impression my powerpoint made, I put together a short one profiling how the firm my beer buddies knew about was the right one for what we needed to get done. Rachel sat back and smiled, "Well, it turns out we have a right of first refusal retainer with one of the big consulting firms, and we need to hear their pitch." My face must have conveyed something along the lines of, "You have got to be fucking kidding me?!" I *am* a terrible poker player. Rachel said, "You can't react like that if you really want the team you say you want. We need to hear their pitch, be engaged, ask questions, and then perform a competitive analysis." Rachel had an MBA. I understood about half the words she was saying at that point in my life. I did as she suggested.

Three hours in a conference room eating catered food that was more delicious than any meal I could afford at that point in my life, listening to a large company pitch their team for *my* project did not build my confidence that the project had a chance if we went with them. So I smiled, nodded, asked questions like I was their best friend, and carefully noted the parts of their answers I would use to submarine them in my competitive analysis. Rachel helped me put together a competitive analysis than sank their boat *hard*.

We kicked off what was now a six week project in early August, 2001. We built a prototype. We did some experiments on the data marketing had assembled, and by early September we were ”cooking with gas”, ready to deliver a working system on September 21, 2001. I came in early every day to check our analysis against what we expected. Everything was going great. We relied pretty heavily on the internet, even then. My team was together by 8am, and we were working. Some time between 8 and 9am the internet *stopped* working! Our HCI consultant came in at 9 and said he heard a plane hit a building in New York. None of us thought anything about it at first. Then the whispers. Then an announcement on the company loudspeaker, which I did not previously know we had. Anyone who wanted to, was invited to watch ”the events in New York” unfold in one of several company auditoriums. We were closest to the largest auditorium, and I said, ”lets go find out what is happening, so we can get back to work.” About a minute after we entered, the first tower fell. My team was in tears. I told them I was going to go order lunch. They could stay here and eat it, and get paid. Nobody was going to get anything done the rest of the day.

Our method did not matter for the rest of the project. We delivered it on time. I bought lunch every day. Our work was something each of us used to manage our shock, and distress. Software engineering methods are important for project success. Remembering that we are all human being, and treating people with respect, kindness, and compassion is more important. That’s what I learned in the wake of 9/11.

Agile Methods Showing end users working software that is relevant to the problem they are trying to solve ever 1-3 weeks builds more trust, and avoids more misunderstandings than any other approach I have used or seen used in my career. XP, Scrum, and other agile software engineering methods share this general belief. Unlike other software engineering methods, agile is guided by a value system focused on building working software.

III Requirements Management

6 Requirements Management Processes and Tools

“”The most important single aspect of software development is to be clear about what you are trying to build.”
—Bjarne Stroustrup

Abstract. There is a great deal written about requirements management. This chapter references two ”canonical” resources on the discipline, and shares a pragmatic approach the author has used for managing requirements to prevent ”scope creep”.

6.1 Introduction

Avalanche Software Requirements is an important, and well covered topic. The act of managing requirements and connecting them to software we build is another sociotechnical combination of art and science. We must understand that how we right things down may be agreed to by our clients and partners, but they may have different understandings of what a requirements ”is”, compared to the team. Agile software methodologies prevent such misunderstandings from transforming a snowball into an avalanche that buries you and requires a beaconing system for you to regain your life. Systematic, heuristic methods can have the same effect. And there is little danger that you will miss important requirements in formal methods, or methods that explicitly keep track of the data you need to track, in cases where that’s a fundamental part of the system.

The software engineering body of knowledge (SWEBOK), version 3, covers the details of software requirements well, and I encourage you to refer to it for a comprehensive understanding (Bourque et al., 2014). Leffingwell and Widrig Leffingwell and Widrig (2000) focus on managing software requirements as a discipline with much detail. You will note that, unlike many technology publications, their book is 20 years old. Few texts in technology survive as useful beyond two or three years. Theirs does largely because managing requirements is not tied to any particular technology. You, and every software engineer, will live it each day, and it is a combination of social interaction and rigorous tracking that leads to success.

6.2 A Model for Requirements Management

Ordering a set of requirements is not a one time activity. You will do it over, and over again. The number of priority codes you have will be a function of the system scope. The example provided in table 6.1 includes the first two priority levels of longer list of requirements available in an example spreadsheet you can download here: TODO. Other columns may prove useful, depending on whether you want to cross reference your requirements with parts of an existing system, a user story, a data model, or other information. Regardless of the SDLC method you use, agreeing to an initial ”cutline” on the prioritized list with stakeholders will create a shared awareness of what we have initially chosen to do, and not do, in a particular sprint, release, or project. *It is a very good idea for the development team to estimate a relative cost for each requirement's implementation. Expressing the cost in effort level is usually best, as it facilitates ”trades” when user priorities change, or new requirements arise. This is especially important if the project's budget is not very flexible.*

Agile Methods Your requirement's spreadsheet will be revisited, added to, and re-prioritized more frequently if your project is using an agile methodology. The reason's are simple: as users begin to see functioning software, they will have new requirement's ideas, or realize that something that had a low priority should be moved up, or something with a high priority moved down. Adding ”effort level” as a column helps decision makers understand how much they may have to ”give up” in order to have their new requirements realized.

Formal Methods If you are using a formal SDLC method, its likely you will describe the result of a successful test case in a column for each requirement, and its likely you will map each requirement to the component or class within the system that implements it. Many organizations will have specialized software for tracking this type of relationship for safety critical systems.

Developing Requirements Lists The SWEBOK Bourque et al. (2014) chapter on requirements makes it clear that your first step is requirements elicitation. When you begin this process, start keeping track of what you learn in the spreadsheet, or one like it, as illuatrated in table 6.1. Do not worry about priority or component at this stage. Simply capture the name and description of the requirement. During requirements analysis, critically examine whether or not *you* understand what the requirement and its description *mean* in terms of software to implement. This will be part of a preliminary version of your software requirements specification, which may also include use cases, that can also be added as a cross reference in your spreadsheet. When you reach this point, go back and ask for clarification and validation from users. This is what the SWEBOK calls ”requirements validation”. *This is also a good time to begin asking users to prioritize requirements.*

Table 6.1
Example Requirements table, ordered by requirement priority.

Component	Priority	Requirement Name	Requirement Description
Customer Portal	1	Enter payment methods	Enable logged in user to enter and save a method of payment.
Catalog	1	Create item	System administrator can create a new item in the catalog.
Catalog	1	Edit item price	System administrator can edit the price of the item.
Catalog	1	Upload item photo	System administrator can upload item photo.
Catalog	1	Create item description	System administrator can create or edit the item description.
Shopping Cart	1	Choose shipping address	Logged in user can select a shipping address
Shopping Cart	1	Choose payment method	Logged in user can select a payment method from saved payment methods, or enter a new payment method. User will have choice about whether or not to save new payment method
Customer Portal	2	Select interests	Provide a list of 20 interests derived from current database, and allow user to select up to 5.
Customer Portal	2	Upload profile picture	Enable a logged in user to upload a profile picture. Preserve old profile pictures for future use.
Customer Portal	2	Change username	Allows user to change their publicly displayed username. Does not change email address assoicated with account
Shopping Cart	2	Gift status	Logged in user will determine if item is a gift, and if so, be able to enter a gift note.
Shopping Cart	2	Apply tax	Before checkout, tax will be applied based on zip code, and user's tax exempt status
Item Display	2	Apply discount code	Logged in user can apply a discount code to an item
Item Display	2	Remove item	Logged in user can remove item from cart

Box 6.1
Requirements Management is Negotiation and Pictures

When I was getting my Ph.D my oldest daughter was in middle school, and I needed a ”real job” that paid enough to support my family, so I took a position as the Director of Product Development at a boutique software company in town. The owner had failed at a few projects recently, and his balance sheet was several hundred thousand dollars in the red. My job was to help him get development contracts for large enough systems, and make them profitable enough to to get his balance sheet ”back in black”. Previously my career was in corporate environments of various sizes, but never anything this ”small scale” that put me so close to understanding how the money worked.

The spreadsheet method I describe in this chapter is something of an ”invention” I came up with to facilitate the types of negotiations that always occur at the end of projects, when customers, inside a company or outside, imagine that the requirements they specified at the beginning include things they only came up with at the end. There’s nothing insidious about this part of human behavior, but without something concrete that your team is working from to refer to, often you will find yourself doing ”free work” to complete a project and maintain a relationship with the people paying for it. Internal, corporate projects are not generally tracked this way, so ”free work” is less costly in that context.

We secured two projects, one with a major sports organization, and one with a medium sized industrial manufacturer. The sports organization wanted all of their information, from schedules, to referees, to player data available dynamically online. Some of it, like editing officials information, player data, and the like needed to be secured behind a login. Schedules were public. The industrial manufacturer needed a marketing website. It had to be ”flashy”, aesthetically pleasing, and information about their complex line of products had to be ”findable”.

Clearly, the projects were different, but each had a requirements spreadsheet. We combined this spreadsheet with mockups of the website look, and ”information architecture”. Until now I left out the ”mockups” and information architecture because we will cover this more in the human computer interaction section. Most people cannot make sense of the spreadsheet style management system shared in this chapter without pictures, and some information about how the website will work. What will be in the menus? How will people find information? These can be mapped into the spreadsheet if its helpful, either formally as a column, or less formally as part of a particular component.

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

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.

Bourque, Pierre, R. E Fairley, and IEEE Computer Society. 2014. *Guide to the software engineering body of knowledge*. OCLC: 973217192.

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.

Carlis, John. 2000. *Mastering data modeling: a user-driven approach*. Addison-Wesley Professional.

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

Leffingwell, Dean, and Don Widrig. 2000. *Managing software requirements: a unified approach*. Addison-Wesley Professional.

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

Wenger, Etienne. 1998. *Communities of Practice: Learning, Meaning and Identity*. New York: Cambridge University Press.