Engineering Software

And Other Creative Improbabilities

Sean P. Goggins, Ph.D

Publisher Imprint
Columbia, MO
London, England

Draft Assembled on 2019-08-20

"*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 stretech, I know). That's what this book aims for right now. This preface could change as it evolves.

Sean P. Goggins
Spring, 2019

1 Systems Thinking and Software Engineering

"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 by "building the right thing". Working within a system that your team agrees to will keep that work bounded and on track. With that in mind, you will benefit knowing a little more about systems theory.

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 is highly specified: requirements, design, implementation, testing, deployment, maintenance. In that order. Either fast ("agile") or slow ("waterfall"). 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 instead[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

[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)

need to collaborate to sift through complexity[2] and identify a pool of potential solutions to explore and ultimately draw from. I used the concepts "coordinate" and "collaborate" right next to each other and defined the differences on purpose by the way[3].

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 write code that solves an important, non-trivially complex problem. To understand why software engineering is the way it is, however, its helpful if you know that software engineering is a response, a reaction, to continuous, catastrophic failure. The way its taught and the way it is implemented holds artifacts of these catastrophes gone by in all the same ways that Minnesota Vikings fans begin each NFL season[4] curious how their team will fail to win the Lombardi Trophy[5]. As computers grew in their importance for banking, defense and other vital interests of modern nation states in the 1960s so, too, did the failure of these systems. Failure took the forms of performance failure against expections or project failure against budgetary and time constraints (Buxton and Randell, 1970; McClure, 1968).

1.1 A Systems Theory Lens

Recoil Though what software engineering "is" has evolved mightily in the past 50 years it cannot escapte its origins in crisis. The agile software movement, which I elaborate on at some length later, was I think the first significant recoiling of practice against rigid processes and structures. Boehm's Boehm (1988) spiral model is, I think, substantially the origin of the agile movement. When I was part of early-ish conversations about agile methods in safety critical systems in Banff, Alberta in early 2003 I talked with a guy who worked with Boehm and he supports the Boehm-agile intellectual provenance assertion I am making. If you look at the agile software methodology chapters later and contrast them with Boehm's spiral method I think you will see the connections as I and this guy in Banff do [6].

Caveat Systems Theory You should know I am not a systems theory scholar or a person whose life is devoted to viewing the world through a systems theory lens. Rather, I enjoy conversations about systems theory and value its utility for making sense of the world as

[2] Use white boards. Walk around. Talk. Try to explain the problem and complexity to each other. Collaboration is part brainstorming and part design. Its all collective problem solving.

[3] Its a common misconception that "collaborate" and "coordinate" are interchangeable concepts. They are not. Please contact me if you would like to order a coffee mug with this paragraph on it.

[4] For non-North Americans, if you're a hockey fan, the way Finland begins every Olympiad is an apt idiomatic substitute.

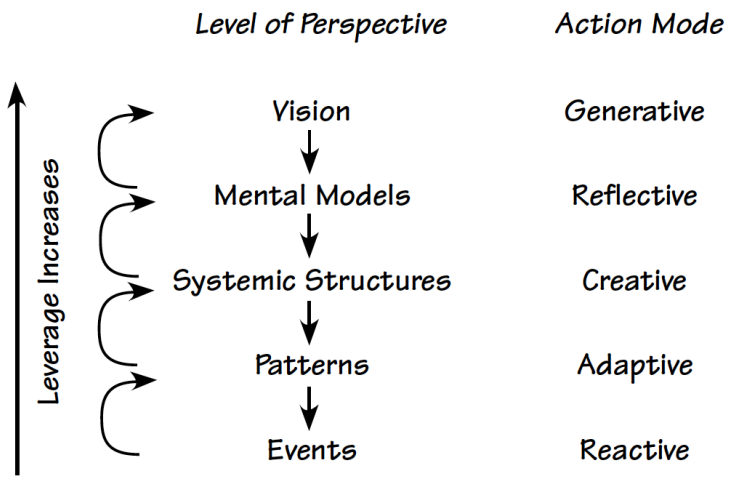
[5] "Gold Medal", Finnish hockey fans.

[6] Why can only stories about sports involve vague references to experts or authoritative software engineering "Sports Center" aficionados?

it is, with so many interconnected people and collections of people. I find systems theory useful for making deliberate decisions about how to solve particular problems. *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.*[7]

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.

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).



[7] My systems theory impostor syndrome could dissipate through the process of connecting software engineering to it, which would alter this paragraph in future editions. Make sure to keep your original PDF, just in case there comes a time they are traded like college professor baseball cards.

2 Introduction

"Probably the most dangerous thing about college education, at least in my own case, is that it enables my tendency to over-intellectualize stuff, to get lost in abstract arguments inside my head instead of simply paying attention to what's going on right in front of me."
—David Foster Wallace

Abstract. Most courses on software engineering are called "software engineering". The rearrangement of words here shifts the construct from a noun to a verb. "Software Engineering" is a "thing". "Engineering software" is an action. This chapter provides an initial overview.

2.1 Introduction to Engineering Software

2.2 Topics

1. Systems Theory and Engineering Software x
2. Ethics in Engineering Software x
3. The Software Development Lifecycle x
   - Requirements x
   - Design x
   - Implementation x
   - Testing x
   - Release x
   - Operations x
   - Maintenance
4. Flavors of Software Development Lifecycle x
   - Waterfall x
   - Agile x
   - Open Source x
5. Computing Disciplines Needed to Engineer Software
   - Computer Human Interaction x

- Database Management Systems x
- Algorithms x
- Computing Languages x

6. Types of Technological Systems for Which We Engineer Software
   - Real Time Systems x
   - Safety Critical Systems x
   - Service Oriented Architectures x
   - Web Systems and Frameworks x
   - Mobile Systems x
   - Data Scientific Systems x
7. Essential Orthogonal Disciplines
   - Documentation
   - Version Control and Configuration Management x
   - Request Tracking x
   - Architecture x
   - Commercial Off the Shelf (COTS) Software x
   - Security x
8. The Social System of Engineering Software x
   - Collaboration x
   - Coordination x
   - Group Work x
   - The Science of Team Science x
   - Conflict x
   - Communication x
   - Metrics and Measurement x
   - Community Building x

Exercises

1. If you have not already, create an account on GitHub.

2. create a new github repository under your account, with your pawprint as the name.

a) Create a fork of the repository https://www.github.com/chaoss/augur into your account

b) Clone the project to your local machine.

c) If you are a Windows computer, please see our operating system support statement in the Syllabus. Even the Ubuntu subsystems on the latest version of windows do not function like other Linux based operating systems and do not support software engineering using open source tools. We are unable to support Windows OS specific issues.

d) Take a Screen Shot of the application directory listing from your operating systems command line.

e) Use git commands (see link in syllabus) to Switch to the dev branch.

f) Take a Screen shot of the application directory listing in from your operating systems command line for the dev branch.

3. Setup a Development Environment

a) 1. Update Python to 3.7:

b) - 'sudo apt-get install python3.7'

c) - 'sudo update-alternatives –install /usr/bin/python3 python3 /usr/bin/python3.7 2'

d) - 'sudo update-alternatives –config python3'

e) - Select the option for Python 3.7

f) 2. Install Python's Virtual Environment Tool: 'pip3 install virtualenv'

g) 3. Change to your "home directory" for the next step. For all the operating systems we are aware of, you accomplish this simply by typing 'cd' and then pressing enter at the command line.

h) 4. Create a Python 3 virtual environment for Augur: 'virtualenv –python=python3 newaugur'

i) 5. Activate your virtual environment 'source newaugur/bin/activate' (In the case of Ubuntu, you get the 'source' command automatically put into your path using the 'bash' shell. So, if you get an error, type 'bash' and then hit the enter key and try again.)

j) 6. Make sure you have NodeJS and NPM installed. 'sudo dnf install nodejs' on Fedora or 'sudo apt-get install nodejs' on Ubuntu. Or 'brew install nodejs' on a Mac.

k) Take a picture of your screen showing these pieces of software are installed. Basically, issue the commands at a terminal and screen shot those.

4. Commit all screen shots to the GitHub repository named after your pawprint, and make sure they are pushed to GitHub under an "assignment-one" folder.

5. submit a link to your repository on canvas under assignment one.

3 Ethics in Software Engineering

"In law a man is guilty when he violates the rights of others. In ethics he is guilty if he only thinks of doing so."
—Immanuel Kant

Abstract. Most courses on software engineering are called "software engineering". The rearrangement of words here shifts the construct from a noun to a verb. "Software Engineering" is a "thing". "Engineering software" is an action. This chapter provides an initial overview.

3.1 Introduction

Things.

I The Software Development Lifecycle

II Flavors of the Software Development Lifecycle

III Computing Disciplines Needed to Engineer Software

IV Types of Technological Systems for Which We Engineer Software

V Essential Orthogonal Disciplines

VI The Social System of Engineering Software

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*.