The Pragmatic Programmer

20th Anniversary Edition | David Thomas and Andrew Hunt

A Literature Review

[SE1 - 2020]

Authors:

Roman Shevchuk

Kiryl Volkau

Illia Manzhela

Mohd Wafa

M Hasibur Rahman

Instructor: Maciej Bednarz

Content

1 Summary

1.1 Introduction

The book illustrates the best approaches and practices of many different aspects of software development. It further extends to provide advice and practices that help in improving your personal productivity and the push for a growth mindset which is crucial for job satisfaction and perseverance in this ever-growing and improving field. Essentially, the book dives into how to approach the specialization and technicalities of modern software development in a pragmatic way. Furthermore, it endulves into topics from personal responsibility and career satisfaction mentalities to architectural techniques (writing clean, flexible and reusable codes).

Physically this book is split into 9 chapters, 53 topics along with several tips in the form of 'reference cards', which stands out in the whole page allowing the reader for a quick revisit to the book for perhaps a bit of advice or motivation.

From the first short tip we realize the author's emphasis and introduction to the pragmatic philosophy and set the tone of the book that is, narrating the debatable statement that software engineering is an art and engineers in this field are craftsmen. Just as any craftsman, mastery and self development is needed to hone your craft which this book highly stresses on.

The split of 9 chapters with the first chapter introducing the pragmatic way from an individual point of view to the last chapter introducing the bigger picture and how this pragmatic way of life can be applied to a bigger scene (projects). In the following section, we will discuss criticisms about this pragmatic way of thinking and its criticisms. Moreover, we will talk in detail

about specific chapters and highlight their main messages and its pragmatic elements illustrated by the authors.

1.2 Main Message

Each chapter contains its own gems and teachings and due to its step by step flow of writing with each chapter related to its previous one, we will adhere to flow style writing of the authors for the following paragraphs. Thus we will use Chapter 1: 'A Pragmatic Philosophy' as our starting point and express in short the latter chapter's main messages and justify the points with references in the book from our personal interpretations.

The 'Pragmatic Philosophy' chapter presents the author's ideas and philosophy regarding software development, mainly focusing around the actions to take responsibility for your career and the consequences of your actions, some of them even include technical debt, being a person used for change, trade-offs in quality, and managing your knowledge portfolio; and finishes off covering different types of communication and, how to do it well.

The chapter, 'Pragmatic Approach' builds on the previous chapter('Pragmatic Philosophy'), but also engages with issues involved in coding, which is explained more in detail with example Ruby code. It claims 'There are certain tips and tricks that work at all levels of software development, processes that are virtually universal, and ideas that are almost fundamental' and that this chapter displays them to you. It covers principles of design, strategies to deal with the fear of change, the start of projects, domain languages and estimation.

The chapter, 'Basic Tools' builds on the programmer-as-artisan metaphor where a creator starts with a basic set of tools. It is heavily biased towards the

Unix way of completing tasks, which is to implement the importance of plain text, command shells, and text manipulation languages. More universal is its recommendation of maintaining Engineering Daybooks. The Unix command line tools have been around for a long time – and, thanks to the FSF, will hopefully be available for free for a long time.

The chapter, 'Pragmatic Paranoia' is all about dealing with imperfections. The first three topics : Design by Contract, Dead programs tell no lies, and assertive programming, all help with the construction of the proper software. The topic, 'How to Balance Resources' is all about resource management and how it interacts with scope and exceptions. The final topic 'Don't Outrun Your Headlights' exponents taking small steps to avoid trying to implement tasks that are not possible to approximate.

The chapter, 'Bend or Break' is mostly about focusing on easy-readable code and avoiding the creation of brittle code. In particular, it discusses decoupling, building upon this book's 'Easier to Change' principle. It teaches a tip, Avoid Global Data and we really think that the related principle Parameterise from Above should have been explained. In the 'Juggling the Real World' topic the ability to be reactive to external change / events covers finite state machines, Publish/Subscribe, and Reactive Programming and Streams. So there are multiple things to learn from this particular chapter. The topic, Transforming Programming raises command-lines and the importance of taking care of some problems as a case of inputting data, changing it, and then giving the output. The Inheritance Tax topic is not enthusiastic of inheritance and finds other alternatives. It was surprising that the SOLID acronym wasn't explained. Finally, it looks at the role of arrangement for flexibility. The chapter, 'Concurrency' debate temporal

coupling and the role of time in software architectures – 'Concurrent and Parallel code used to be exotic.

The 'While You Are Coding' chapter is a combination of 8 main topics and 37 useful tips on coding and is full of useful and implementable advice.

The chapter, 'Before the Project' discusses basic stuff needed and how developers can gather the requirements for a project from the users and then build on and correctly implement that information. It also gives tips on how to solve 'impossible' puzzles. One topic in the chapter, 'Working Together' exponent Pair Programming for the creation of better quality software. The final topic takes into account the essence of agile software development.

The last chapter, 'Pragmatic Projects', is about the main area and how tips and tricks from this book can be applied to any project. In its 'Pragmatic Starter Kit' topic in the chapter, it covers three main areas, that are – version control, regression testing and automation of building and testing. The topic, 'delight the users' strengthens the need for developers to be bare to multiple aspects of the customer's organisation and needs.

2 Targeted audience & Overview

From the statement in the foreword section of the book, "*This book is a guide to becoming a better programmer efficiently*", we can formulate that the targeted audience is programmers and developers. Furthermore, the subheading, "*your journey to mastery*" allows us to presume that the book is for new programmers who just started their journey. However the content of the book suggests that it is appropriate for more experienced developers as it consists of remainders of things a developer might intuitively know but not

be fully aware of. Moreover, highlight some aspects of their work that are underappreciated.

The book is very well written with information and real-life examples that a developer can relate to. The anecdote, guidelines and ‘reference cards’ as mentioned before allows for great reminders or lessons to reflect upon. The book’s pragmatic and simple flow of writing allows readers to understand the book better. Unlike other software books, The Pragmatic Programmer does not get too technical and present things from a perspective which is not heavily influenced by teachings that suit the corporate world and big bureaucratic companies. This way of writing indeed justifies the book’s subheading and reinforces that this book is more like a journeyman's guide. Thus its content and its pragmatic wisdoms can also be beneficial to audiences beyond the scope of people with title ‘programmers’ such as freelancers, small business workers, or even for programmers working in large companies, who we usually presume to know it all.

With several simple real life examples i.e. anecdotes, the author is able to deliver timeless advice on being a true pragmatic programmer and hence attains the label of being a classic book as it convinces readers to refer back to the book to gain advice through the anecdotes presented in the book for a current similar real world problems one might be facing.

Along with features such as ‘reference cards’ in the book, the writers also allow engagement with the reader by providing exercises and challenges at the end of some chapters. It allows the readers to experience the pure satisfaction of confirming your success(or not) with the answers provided at the last pages of the book.

The author uses thoughtful code examples which are written in up to date modern programming languages. They use these examples to demonstrate the best and worst practices(which is to be avoided). Along the side of providing examples, the author most of the time also provides an opposite respective example to further explain a statement. We can see this in page 32 when the author provides a code snippet that demonstrates Duplication in code. Thereafter giving an example on how to rectify such duplication in code. This coherent and pairwise use of examples to display both aspects of a topic gives readers a clarity of the discussed topic.

3 Critical Discussions

After browsing through many articles and forums, it was apparent that most viewed this book rather a self-help book in the sense that it's a feel good book. It has been referred to as a book that is not practical - lacks actionable guidance. Experienced software developers argue that any beginner whom have or have not read this book would definitely still continue to make the same mistakes highlighted by the book. This could be perhaps because of trivial tips such as "Think! About Your Work" and the rant of paragraphs thereafter in page explaining the tip with subtle humor or anecdotes does not really provide actionable guidance. Some refer to the book written well in terms of flowery language and that sometimes authors saying a lot without actually saying a lot i.e. unnecessary acronyms complicating explained concepts. Hence, entertainment is achieved rather than focusing on efforts to explain concepts.

The book is referred to be more as a "why" book rather than a "how" book due to its fair share of examples and tips that is rather obvious. The book covers the hows in a bird eye perspective and does not really dig too deep in the

topics(maybe example). The mentioned bad practices outweigh the good practices and thus gives the feel of a dictation of what not to do rather than here is how not to make such mistakes. For most tips and mentioned practices, the book could be distilled down into a couple of pages of bullet points and not lose much value. This statement has been rather affirmed in this review in the main message section where the chapters were filtered and main points were pointed out within a few pages.

As newbies in this field and without viewing this book as an experienced developer on the internet, we would say that it contains good snippets of information and the slow successive flow of information presented in the book does give a sense of a journey to mastery which allows for a direction to improvement in one's skills. We do recognise that this book is mostly for new developers. For more experienced developers, this book does rather serve gentle reminders and guidelines to what one should be doing.

4 Personal Findings

The programming part of the world has not had a normal approach to itself for a long time, for outsiders. This is why so many programmers are mostly self-taught. If the authors of this book continue to revise and improve this work, and fix all the little mistakes in it, I would go so far as to say this might be the classic for introducing practical-minded outsiders to the complicated universe of computer programming.

The Pragmatic Programmer is not a “talkative” book. It assumes you are intelligent and can think for yourself about problems, and need help solving them, but not necessarily to be told how to solve them. So much of "teaching"

software development is just faulty teachers trying to explain implicit software paradigms on incomplete problems and expecting you, the learner, to be not only satisfied with the completeness of their answer, but to understand why they picked the particular subjective approach that they did.

I used to think it was all about coding, but it really isn't. It's about problem solving, and not just formal problem solving, but real-world and practical problem solving. What problem does your software solve, and does it solve it the most effective way you can think of? Because of this, I am planning to broaden my horizon to not just towards more and different types of languages, but also towards trying to find easily implementable solutions towards real-life problems.

[Kiryl]
As to me the book was useful. However, there were some not so significant points I wanted to argue about with the authors - for instance, they brought up as an example “The broken glass theory” with reference to the NYPD actions during the 80s. The theory was compromised and original authors told about it in their paper dating 1984, however, as no media decided to publish it (as it contained massive criticism of police actions), the theory continues to live now. As to me this is a little signal about the thoroughness of the preparation and the responsibility.

Putting aside this little error, the rest of the book felt useful to me - especially parts on code generators and metaprogramming, as those features were either introduced or greatly improved in the new version of .NET (.NET 5). However, the first parts about “Pragmatism”, “Specification”, “Team Work” and “Discipline” felt very impractical and general. The book is also kind of old

(as it was written 15 years ago), and it is left to the reader to figure out, which principles are still somewhat useful and which are already outdated.

5 Further reading

1. Code Complete by Steve McConnell (https://en.wikipedia.org/wiki/Code_Complete)

2. Refactoring: Improving the Design of Existing Code by Kent Beck (https://martinfowler.com/books/refactoring.html)

3. Programming Ruby by Chad Fowler (https://en.wikipedia.org/wiki/Programming_Ruby)

4. Working Effectively with Legacy Code by by Michael C. Feathers (https://www.oreilly.com/library/view/working-effectively-with/0131177052/)
5. The Art of Computer Programming by Donald Knuth (https://en.wikipedia.org/wiki/The_Art_of_Computer_Programming)