Book Report

Student Name: Xuechen Wang

NetID: xuechen5

Book Title: The Pragmatic Programmer

Author: Dave Thomas & Andy Hunt

Date of Report: Dec 5, 2020

Number of Pages: 449

Point of View: Software Engineering Student

Edition: 2nd

Personal Evaluation

The book is composed of a collection of self-contained short topics that can be read in any order because there are cross reverences to put each topic in context. But I read the chapters and topics sequentially, and will share my thoughts for the same order as I read.

I didn’t know what a pragmatic programmer is until I learned form the book that pragmatic Programmers distinguish themselves by an attitude, a style, a philosophy of approaching problems and their solutions. They think beyond the immediate problem, placing it in its larger context and seeking out the bigger picture.

As the book promotes the concept of modularity which applies to code, designs, documentation, and team organization. Modularity aims to increase the dependency inside a module, and decrease the coupling outside a module. Keep our applications decoupled and modular, will make them more amenable to change. In my opinion, one advantage of object oriented programming languages over functional and is using modules to makes code reuse available. The more decoupled code, the more reusable it becomes.

The book also introduced lots of good coding style. I found that ETC principle (Easier to Change) and DRY(Don’t Repeat Yourself) are especially useful to me when working on team project. ETC Is a Value, Not a Rule.

In our team project, because each programmer works on a set of different files, we are trying to make minimal changes to the existing file, and don’t create unnecessary classes and functions if we can reuse what’s already in the repo. Because even with good communication in the team, duplication in code can still cause confusion and debugging nightmares.

One kind of coupling we encountered in the project is a chain of method calls. It is also a type of code smell. I remember assigning a variable through a chain of method calls, some of the return value type of one method are not the same with the next method’s argument type. Caused the program to crash.

A violation to ETC is using global variables. One function changed the global vairble by accident, the whole program goes wrong. That’s why we use paramemters to pass values through functions, instead of using global variables whenever we can.

Orthogonality is closely related to the DRY principle. With DRY, you’re looking to minimize duplication within a system, whereas with orthogonality you reduce the interdependency among the system’s components. We want to design components that are self-contained: independent, and with a single, well-defined purpose. In our project, we made new functions for existing classes if such classes exist, and only add new classes when necessary, and place them in the folders following the project’s logic.

Programming needs lots of concentration and logical thinking, for me, if the environment is noisy, I will have a hard time to concentrate. The book even suggests that finding a comfortable working condition is also being responsible to our work.

The book suggested that we produce software that’s good enough—good enough for our users, for future maintainers, for our own peace of mind. I came from business background. In economics, there is there about cost and profit, especially concepts called “profit margin” and “opportunity cost”. Like businesses, software also has the concepts of optimal and perfect. In business, the total sales and profits are not totally parallel. When the quantity reached to a certain level, fixed cost will step up to a much higher level, and making the business unprofitable or even have losses rather than profit. Similarly, in software development, perfection costs, and sometimes are not necessary. Negligible improvements costs the most time and effort. Same theory applies to testing using exhaustive values.

All the time when we write programs, we should keep ETC in mind. They are all heuristics towards writing good code, and being pragmatic.

The book is a summary with concepts of what I learned and did in Software Engineering class and other computer science classes, and covered interpersonal skills for a programmer in a team. Like how to communicate time using appropriate units (days, weeks, month, etc), and how to estimate progress.

Some recommendations for using and configuring command shell. I took linux shell programming class in college and knows how to customize a prompt, set aliases and shell functions, but I didn’t even know that I can set color themes until I read this book.

One thing bad about a blue background color, is when you edit java or xml files, the commented code will be in red color, making them hard to read in a blue background. Config backgroud color: makes eyes more adjustable to the interface, and more productive.

The book introduced Pragmatic Starter Kit that covers three critical and interrelated topics: version control, regression testing, and full automation.

Since I was never worked as a software engineer that require git on a day-to-day baisis, and due to limited online resources/ tutorials I found / could find about Gitlab, I only knew basis pull and push requests on main branch. I only know that Version control systems can track changes I made in my source code and documentation, and with a properly configuration, I can always go back to a previous version of my software. However, I didn’t know that I can see who made changes in which line of code, compare difference between the current version and last week’s, and see which files get changed most often until I worked on the team project, and used all these features.

We used version control as a project hub. GitLab is our central repository for archiving all our code. It allow two or more of our team members to work concurrently on the same set of files as well as making concurrent changes in the same file. I was glad to see that VCS can manage the merging of concurrent changes when the files are sent back to the repository.

Another thing I learned about version control systems is that I can isolate a version of development into branches at any point in my project's history, so that any work I do in that branch will be isolated from all other branches. At some time in the future I can merge the branch I'm working on back into another branch, so the target branch now contains the changes I made in my branch.

I never heard of branches before I only used git to submit homeworks on main branch. Luckily my teammates used git in their work, and introduced me to branches before I read this book and learned about it. In our team project repository on Gitlab, we created branches for each UC, so the pair of programmers working on the each UC will not interfere with other UC, and no one edits the main branch until the UC branches are done coding and testing, and ready to merge with the main branch. Branches are at the heart of our team's project workflow.

In addition, Gitlab also supports our team communications since I got emails when pull requests are created on my branch I author, so I can know what is happening.

Quality and robust code must endure testing. The book mostly talked about two kinds of tests: unit test and integration test.

Unit test is code that exercises a module. Typically, the unit test will establish some kind of artificial environment, then invoke routines in the module being tested. It then checks the
results that are returned, either against known values or against the results from previous runs of the same test (regression testing).

Unit test measures code coverage to see if all conditional branches of code are being tested. In turn, code coverage can discover if there are any dead code in your module. My code editor has coverage analysis tools that keep track of which lines of code have been executed and which haven't during testing.

However, what's more important than test for code coverage is to test state coverage. Ideally, we should identify all possible states of the program. However that's not realistic or necessary in practice. So in the spirit of "good enough" the book advocated, I always only do enough testing for unit tests, which means testing boundary and extreme values in addition to check code coverage.

Integration testing shows that the major subsystems that make up the project work and play well with each other. In my opinion, if unit tests test on how each module honor their contracts(what the expected input and output of a module), integration testing is an extension of unit test, and testing how the entire subsystems honor their contracts. Integration issues can be detected easily, if we have good contracts in place and well tested.

The book emphasised the importance of regression testing in property-based tests, in refactoring, and in debugging releases. We also used regression testing in our team project before I read this book and learned this term and its importance.

I learned from the book that property-based tests can generate random values that get passed to tests, so there's no guarantee that the same values will be used the next time when I run tests. So to reproduce the bug, a unit test is needed to forces those values to be used to ensures that this bug won't slip through. Unit tests always act as regression tests.

To produce high quality code, we need to accompany our block of code with unit tests all the time. As we refactor code as needed, the utility of a function or class may change too. So rename variables are also needed when the old names of the variable no longer expresses the intent, or is misleading or confusing. Because we have unit tests accompany modules of our code, we need to update the unite tests with the new names. And the full regression tests, will in turn catch any instances we may have missed. In other words, maintaining good regression tests is the key to refactoring safely.

Regression testing are also used to verify if a new release is bug-free. Combined with debugging methods like binary-chop, it can catch bugs introduced in new releases. And the plain-text output from regression tests can be trivially analyzed with shell commands or a simple script, a new trick I learned from the book.

In our team project, we also did selenium test, a good way to test web pages by manipulating values for HTML elements by code, but not applicable to all kinds of program though. The book talked about good documentation about code, comments, and testing, and selenium test is a good way to document tests for web pages. Instead of writing an instruction of which button was pressed, and what value is entered for the input field.

The book suggested that we test early, test often, and test automatically. In our project repository, we have more test code than production code.

Automation is an essential component of every project team. In this class, we learned how to deploy java projects manually, step by step. We also learned how to use Maven to deploy java projects automatically. The manual approach is slower and more error prone than automatic build using maven. When doing the team project, we also learned how to let "mvn install" also run all tests automatically when building the project. The automatic build covers several major types of software testing: unit testing; integration testing; validation and verification, and selenium testing. Automating everything the team does id great way to ensure both consistency and accuracy.

Many teams have their VCS configured so that a push to a particular branch will automatically build the system, run the tests, and if successful, deploy the new code into production. However, our team hasn't come to this step.

Some debugging techniques introduced in the book I also used in my debugging process including reproduce bugs, binary-chop, Logging and/or Tracing, and Rubber Ducking.

It never happened to our team project that a new version of the code introduced a bug, but it's still good to know that if the team has introduced a bug during a set of releases, we can create a test that causes the current release to fail. Then choose a half-way release between now and the last known working version, and use binary-chop and regression testing to find the bug.

Tracing statements are those little diagnostic lines printed to the screen to help debug. I always use tracing statement to print to console to see if a block of code in conditional statements are executed, and also to check if a value that is returned from a function and later passed in to another function get the correct value. I also used tracing statements to check which value in an array or loop crashed the program. When doing the team project, since I'm dealing with webpages, I used tracing statements to print to the the screen in stead of console to check the variable values.

I also feel that rubber ducking is a particularly effective and useful technique when debugging as mentioned in the book. When pair-programming our use case for the team project, I was struggling about how to implement a feature for activating / deactivating pre-registered patients.
As I explain to my partner about my design for the interface, I realized that it could have been done in a different way to avoid the bug. The process of telling other people about the problem makes me think more and deeper about my design choices.

A right process of elimination helps debug easier. In most projects, the code we are debugging may be a mixture of application code written by us and others on our project team, third-party products (database, connectivity, web framework, specialized communications or algorithms, and so on) and the platform environment (operating system, system libraries, and compilers). It is generally more profitable to assume that the application code is incorrectly calling into a library than to assume that the library itself is broken.

In my other class, the autograder uses an older version of python, and I have the newer version of python on my laptop. Everything runs fine and passed when I test the program on my laptop, but crashed on the autograder’s VM. The solution was to change the functions to the corresponding function in the older version of python, so that the autograder can parse.

The book reminded me that don’t over trust our code, and “Don’t gloss over a routine or piece of code involved in the bug because you “know” it works. Prove it in this context, with this data, with these boundary conditions.” In my past experience, often the code I copied from other functions or files that has worked fine in its original block, will introduce bugs to the new code. I need to double-check the variable names used, the counter variable are consistent with what’s used in the current program. It caused bugs for me in the past. In nested loop copied from my existing code, the counter variable used for outer loop and inner loop are reversed, but I didn’t realize, and it caused “Index Of Bounds” exception. Sometimes, just read and find and replace the variable names are not enough. You need to read through the block of code, and “rehearse” the execution in your mind.

And also a good advice from the book for being a pragmatic programmer is that when we come across a surprise bug, beyond merely fixing it, we need to determine why this failure wasn’t caught earlier. And we need to amend the unit or other tests so that they would have caught it.

Also I learned from experience that fix bug as well as code smell as soon as we find them, or it will haunt us in the future. Those are the things I have been doing when coding, but the book put them in a systematic way.

The booked talked about updating your knowledge base as well as your developing tools. When we bump into some apparent limitation of the editor are’re using, search around for an extension that will do the job. The chances are that you are not alone in needing that capability. I’ve been using Eclipse to write Java code for the team project, the copy the files to VM to deploy until I did pair programming with my partner for the project, my partner introduced me to VS Code with SSH extension. VS Code saved me lots of time unrelated to coding, and also increased our code quality because I could focus more on programming.

Conclusion

The book is good to read, and thank you for recommending it to us. On one hand, the book summarized the concepts and techniques I’ve been using when coding and debugging but didn’t aware of, on the other hand, it introduced new knowledge and skills I didn’t know of, and can use and practice in my future programming works.