# The essence of "Pragmatic Programmer"

A heavily paraphrased summary of the book

Andrew Hunt, David Thomas: The Pragmatic Programmer: From Journeyman to Master, Addison-Wesley Professional 1999 (321 pages)

(Lutz Prechelt, 2013)

## Ch. 1: A Pragmatic Philosophy

"Pragmatic Programmers […] think beyond the immediate problem, always trying to place it in its larger context, always trying to be aware of the bigger picture." They think critically.

They take responsibility for everything they do, refuse doing it if they cannot, and have no fear of appearing weak or incompetent.

If something goes wrong, they act constructively and offer options, not excuses. When they see how things ought to be, they work as catalysts to make it happen (e.g. using the stone soup trick).

They understand the context in which they work and so understand what is sufficient: What makes *good-enough software*. They explicitly trigger requirements discussion on quality levels.

But they will immediately repair (or at least board up) any "broken window" they find: anything in the software that is not clear and orderly.

They keep learning explicitly all the time and also continually strive to become better at communicating and understanding their audience. They treat learning like financial investment, with notions of diversification, making both low-risk and high-risk investments, buying low and selling high, and rebalancing the portfolio.

## Ch. 2: A Pragmatic Approach

There are some ideas how to approach software development that apply at many different levels and in any software development domain:

7. "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.", or shorter: Don't repeat yourself (DRY).

Some duplication may seem unavoidable (e.g. because the same thing must be expressed in more than one notation), sometimes it happens because of inattentiveness or laziness of one developer or because several people introduce the same knowledge independently, but all of these types can be avoided: e.g. by code generators, by recognizing that duplication will usually become costly later on, by making modules easy to reuse, and by a sound software architecture and strong communication.

8. Orthogonality: Eliminate effects of one thing onto an unrelated thing. This will make changes local, make parts testable individually and more reusable, make the overall system more robust. Even team members can be more or less orthogonal to each other. Also, avoid relying on things you cannot control. Use refactoring to constantly move towards more nearly orthogonal code.

Testing and bugfixing provide good indicators of successful orthogonality: Can you easily write tests that test only a single module? Does fixing a bug usually involve a single file only? Then the system's orthogonality is high.

9. Reversibility: There are no final decisions. Make sure that reversing a decision is cheap. Introducing suitable abstractions goes a long way for that.

10. Tracer Bullets: Write your system in such a way that your code helps to find out quickly and easily how close to the target you are – and (unlike prototyping) supports getting closer. The actual solution must be operational *long* before it is fully functional: to provide something to show the users, to provide an architecture and integration platform for the developers, and to provide a definite measure of progress.

11. Prototypes: Prototypes are vehicles for understanding a few particular aspects of a system, e.g. of visual design, a workflow, a critical performance issue, or the behavior of some technology. They are built in the cheapest possible manner (which needs not involve program code) and are thrown away after the question is solved, because they are about the lesson learned only. Make sure everybody involved understands they will be thrown away.

12. Domain-specific languages: Program close to the application domain. Use domain

vocabulary at least. If domain experts use unambiguous language, emulate its semantics and perhaps also its syntax (either for specification only or even in executable fashion). Consider different mini-languages for different types of users. There can be mere data languages (often using rather simple formats), executable languages, or metaprogramming (generating or manipulating parts of the application).

13. Estimating: Make it a habit to estimate how large things are going to be: Memory requirements, disk space requirements, bandwidth requirements, run times, development times, event frequencies (both at run time and in the development process) and so on. This avoids surprises.

Consider the accuracy required and use suitable units. Draw on the experience of others if possible. Make assumptions explicit. Build models. If the estimate is difficult but important, produce multiple estimates with different approaches. For project estimates, that very same project can be a source of estimation knowledge if incremental development is used. When asked for an estimate, answer "I'll get back to you" and take your time.

## Ch. 3: The Basic Tools

Every craftsman needs high-quality tools. Their skilled use improves only over time but you should still constantly look for better tools, too.

14. Plain text: Our material is knowledge, its best representation is human-understandable structured or semi-structured plain text, because that is best for analysis and manipulation (except sometimes when small size and high processing speed of binary format prevails), is most interoperable, does not become obsolete, and is best supported by tools.

15. Shell: The shell is to a programmer what a work bench is to a woodworker: The center of work. GUI tools are just too inflexible to make the shell obsolete. On MS Windows, use Cygwin.

16. Editor: Know one editor really well; it should be multi-platform, extensible, and programmable.

17. Version control: Is required to undo multiple days of changes when needed, to find out who changed what when, to measure the amount of change over time, to find hotspots of change, to automate builds, etc. Use it for all development (even throw-away stuff) and version all relevant files, not only source code.

18. Debugging: Debugging is just problem solving; treat it as such.

- Focus on the problem, not on blaming.
- Don't panic. Think.
- Fix the cause of the failure, not its symptoms.
- Fix any compiler warnings first. Reproduce the failure then (get help if you cannot) and automate the reproduction.
- Preferably use a good debugger program with data visualization capabilities.
- Use binary search to narrow problems down.
- Use tracing/logging where the debugger does not work well and use or make software that helps to wade through the tracing output.
- Explain partial insights to someone else to complete them.
- Suspect your project's code first, not compilers or external libraries, but changes to those (or the OS) may break your code without you doing anything.
- Don't assume, check.
- Once you found the problem, add the test that would have caught it and look for further similar problems. If the failure happens far away from the defect, add more integrity checks to the code.
- If the defect was due to a misunderstanding, make sure to clear that up with the author and find a way to avoid similar misunderstandings in the future.
- If debugging took long, reflect why.

19. Text manipulation: Learn Python, Ruby, or Perl for sifting through and processing plain text. Use it for automating many things.

20. Code generators: Write code that writes code, either for subsequent manual editing (passive generators: for convenience; the code needs not be complete or perfect) or for immediate use (active generators: for following the DRY principle).

## Ch. 4: Pragmatic Paranoia

You cannot write perfect software. Therefore, do not waste energy trying; be pragmatic.

Code defensively: Don't trust the code and data of others -- nor your own!

21. Design by contract: Specify preconditions and (simplified) postconditions explicitly. Perform run-time checking for them, using assert if you have nothing else or using a

stronger mechanism (that includes inheritance and object invariants) if available for your language. Preprocessors tend to be messy.

22. Crash early: Check many things that "cannot happen" or absolutely must not happen and crash the program if they happen.

23. Assertions: Check many things that "cannot happen" or absolutely must not happen and do not be intimidated by the runtime overhead prematurely. Turn off only those assertions that are really too slow. Make very sure your assertions have no side effects.

24. Exceptions: Use exceptions to free the code from too much intermingled problem handling in order to make the main execution path clearly visible. Use exceptions for unwanted conditions that are at least somewhat surprising, not for fully regular ones.

25. Resource management: Whenever you can, "finish what you start", i.e. the routine that allocates a resource should be responsible for deallocating it. Deallocate in the opposite order of allocation. Use standard allocation orders to avoid deadlock. Wrap resource use in classes so the destructor can clean up left-over resources. In Java, `finally` is your friend for cleaning up reliably. Where "finish what you start" is not applicable, the resource should become part of some container that is responsible for deallocation.

## Ch. 5: Bend or Break

The world changes constantly, so code must be flexible, too.

26. Decoupling and the Law of Demeter: Couple your classes to no more other classes than reasonably necessary: your instance variables, method arguments, and new local objects (the Law of Demeter). Have those objects perform a complete service for you rather than giving you an object with which *you* can perform the service. This will require many delegation-only methods, though.

27. Metaprogramming: Provide many configuration option to avoid change programming. Put abstractions in code, details in metadata. If you drive this far enough, you may even be able to implement different systems using the same application engine, just with different metadata. Business rules and workflows are good candidates for business data. Great applications can change them even without requiring a restart.

28. Temporal coupling: Design for maximal concurrency, avoid introduce unneeded ordering constraints on steps. This may help your design quality, too, e.g. because you may ask yourself why that global variable that you now need to lock exists at all.

29. Events and views: Event-based control is a good decoupling mechanism, e.g. in a publish/subscribe (observer) structure. In particular, separate views from models, e.g. in a model-view-controller (MVC) structure, whether in the context of GUIs or elsewhere. You can stack them: One structure's view becomes the next-higher structure's model.

30. Blackboards: An even stronger form of decoupling, where only data structures (or objects) are shared but no call coupling is explicit is a blackboard storage (tuple space, e.g. JavaSpace), asynchronous and possibly transactional, were events are created by the fact that an object with certain properties appears in the storage (written by some other participant). Often combined with rules engines to coordinate workflows.

## Ch. 6: While You Are Coding

31. Programming by coincidence: To make sure your program works tomorrow, you must thoroughly understand why it works today. If you don't, your code may be slow, confusing, only partially correct, error-prone to change, and prone to collapse if the objects change that it is calling. Know what you are relying on. Don't rely on anything you need not rely on.

32: Algorithm speed: O-Notation and runtime complexity classes. Complexity estimation rules-of-thumb. Estimate. Then test your estimates. Be pragmatic about algorithm choice.

33. Refactoring: Building SW is more like gardening than like construction; a constant process of monitoring and care. Regularly refactor your SW to push back duplication, non-orthogonal design, outdated knowledge, and performance degradation. Refactor early, refactor often – and avoid telling your clients you do it. Make small steps and do not change functionality at the same time. Have automated tests to safeguard your changes.

34. Code that's easy to test: Write automated tests for each module that test against its contract (self-testing code). Use assertions in the code. Test lowest-level modules first and

higher-level modules later (to simplify defect localization). Co-design code and its tests. Store the test code close to the module code. Tests also serve as documentation. Use a test harness. Do not throw away the ad-hoc tests you invented during debugging. Make sure you can test your software during production, too. Log files and semi-official debugging console windows or built-in webservers are helpful. Establish a standardized test culture (as e.g. on the Perl platform).

35. Evil wizards: If you use a Code Generation Wizard, make sure you understand the code produced, because it will be interwoven with your application.

## Ch. 7: Before the Project

36. The requirements pit: "Requirements rarely lie on the surface. Normally, they're buried deep beneath layers of assumptions, misconceptions,and politics." Identify policies (e.g. access privilege rules) that come as part of requirements and expect them to be volatile; make them configurable. Identify user interface details that come as part of requirements and initially treat them as manner of speaking only. Understand and document why users want certain things, not just what. To understand the domain, become a user yourself for a week – it helps build trust and rapport, too. To document requirements, use a suitable Use Case format. Do not overspecify, stick to what's strictly needed. Track all requirements changes to avoid creeping featurism. Maintain a glossary and stick to those terms. Use hypertext and internally publish the requirements.

37. Solving impossible puzzles: The key to coping with seemingly impossible problems is discriminating real constraints from perceived ones. Are you even solving the right problem? Then, enumerate all conceivable (not: possible) routes and carefully explain for each why it cannot work. Find your weak arguments: There are your possibilities.

38. Not until you're ready: Don't start as long as you have doubts, but start when you are ready. How to discriminate real doubts from mere procrastination? Prototyping will often reveal the problem behind your doubts or quickly get you to readiness.

39. The specification trap: Don't write highly detailed specifications.

40. Circles and arrows: Don't become a slave to formalized methods. Beware of requirements specification notations your end users do not understand, beware of developer overspecialization, beware of methods that restrict the flexibility of your designs (e.g. regarding the use of metadata for configuring behaviors). Never underestimate the learning cost for a new method. Each method should be a tool in your toolbox, selected and used when appropriate and its use constantly refined.

## Ch. 8: Pragmatic Projects

41. Pragmatic teams: All the above advice applies even more strongly at the team level. Teams must not accept broken windows, must constantly look out for deteriorating conditions, avoid duplication. A team should create a brand for itself and communicate consistently to the outside. Appoint topic experts. Use groupware. Communicate and discuss lively within the team. Organize around functionality, not job roles. Isolate sub-teams from each other by design by contract, Law of Demeter, orthogonality. Even good teams need a technical head and an administrative head, larger ones also a librarian and a tool builder.

42. Ubiquitous automation: Avoid manual procedures. Automation is more efficient, more consistent, and more accurate. Use scripts (such as buildfiles) to automate routines and cron to automate even their occurrence. Apply this even to tasks with manual aspects in them, e.g. by manually annotating code with "needs review" and then automating the review management only.

43. Ruthless testing: Test early, test often, test automatically. Use unit tests to catch local defects and integration tests to catch non-local ones. A good project may have more test code than production code. Besides functional testing there are requirements validation tests(*), error-and-recovery tests, performance tests, load tests, usability tests(*), and others. Except for (*), they are automatable: automate them. Use artificial as well as real data. Avoid testing at the GUI level much. Test your tests by planting defects intentionally and see how many are caught. Assess your tests' coverage, preferably state coverage, not just code coverage. Add an automated test for it when you manually found a defect.

44. It's all writing: Treat documentation as an integral part of your project. Apply all the other principles: Treat English as just another programming language, avoid duplication,

automate, etc. Code comments should explain WHY something is done and anything else that is not obvious. Consider generating code or operations from documents, not just the other way round. Prefer documentation based on plain text formats (e.g. DocBook) over complex word processor formats.

45. Great expectations: In reality, project success means to fulfill the users' expectations; some of them too high, others too conservative. Understand and then groom and manage these expectations. Avoid big surprises but try to surprise and delight them a little: go the extra mile.

46. Pride and prejudice: Sign your work. This guards against sloppiness. Collective code ownerships works well in XP (because of pair programming) but is problematic elsewhere. But don't become territorial either.

## Further material

The book contains a quick reference guide with a three-line summary of each of the 70 tips contained in the book.

The guide also provides 11 short, itemized checklists representing criteria embedded in some of the tips.

The book contains a number of exercises (with solutions in an appendix).

It suggests a number of books for reading and explains why. It suggests becoming a member of ACM and IEEE (and perhaps a national association) and reading several specific ones of their magazines.

It recommends a number of specific tools [but that list is fairly dated by now].