A Pragmatic Philosophy

Care About Your Craft	Why spend your life developing software unless you care about doing it well?
Think! About Your Work	Turn off the autopilot and take control. Constantly critique and appraise your work.
Provide Options, Don't Make Lame Excuses	Instead of excuses, provide options. Don't say it can't be done; explain what can be done to salvage the situation.

Software Entropy

Don't Live with Broken Windows	Don't mess up the carpet when fixing the broken window.

One broken window, left unrepaired for any substantial length of time, instills in the inhabitants of the building a sense of abandonment—a sense that the powers that be don't care about the building. So another window gets broken. People start littering. Graffiti appears. Serious structural damage begins. In a relatively short space of time, the building becomes damaged beyond the owner's desire to fix it, and the sense of abandonment becomes reality.

Stone Soup and Boiled Frogs

Be a Catalyst for Change	Most software disasters start out too small to notice, and most project overruns happen a day at a time. If you take a frog and drop it into boiling water, it will jump straight back out again. However, if you place the frog in a pan of cold water, then gradually heat it, the frog won't notice the slow increase in temperature and will stay put until cooked. Don't be like the frog. Keep an eye on the big picture.

It's time to bring out the stones. Work out what you can reasonably ask for. Develop it well. Once you've got it, show people, and let them marvel. Then say "of course, it would be better if we added…."

People find it easier to join an ongoing success.

Good enough soup

Make Quality a Requirements Issue	Great software today is often preferable to perfect software tomorrow. Know when to stop.

The scope and quality of the system you produce should be specified as part of that system's requirements.

Your Knowledge Portfolio

Invest Regularly in Your Knowledge Portfolio	Learn at least one new language every year. Read a technical book each quarter. Read nontechnical books, too. Take classes. Participate in local user groups. Experiment with different environments. Stay current. Get wired. An investment in knowledge always pays the best interest.

Your Knowledge Portfolio (cont)

Critically Analyze What You Read and Hear	You need to ensure that the knowledge in your portfolio is accurate and unswayed by either vendor or media hype.
Building Your Portfolio	- Serious investors invest regularly, as a habit - Diversification is the key to long-term success - Smart investors balance their portfolios between conservative and high-risk,high-reward investments - Investors try to buy low and sell high for maximum return - Portfolios should be reviewed and rebalanced periodically

Communicate!

Know your audience (WISDOM)	What do they **Want**? What is their **Interest**? How **Sophisticated** are they? How much **Detail** do they want? Whom do you want to **Own** the information? How can you **Motivate** them to listen?
Choose your moment	Understanding when your audience needs to hear your information.
Choose a style	Just the facts, large bound reports, a simple memo.
Make it look good	Add good-looking vehicle to your important ideas and engage your audience.
Involve your audience	Get their feedback, and pick their brains.
Be a listener	Encourage people to talk by asking questions.
Get back to people	Keep people informed afterwards.

Know what you want to say. Plan what you want to say. Write an outline.
It's Both What You Say and the Way You Say It.

A Pragmatic Approach

Tips and tricks that apply at all levels of software development, ideas that are almost axiomatic.

The Evils of Duplication

DRY—Don't Repeat Yourself	The problem arises when you need to change a representation of things that are across all the code base. Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

The Evils of Duplication (cont)

Make it easy to reuse

Imposed duplication Developers feel they have no choice—the environment seems to require duplication.
Inadvertent duplication Developers don't realize that they are duplicating information.
Impatient duplication Developers get lazy and duplicate because it seems easier.
Interdeveloper duplication Multiple people on a team (or on different teams) duplicate a piece of information.

Orthogonality

Eliminate Effects Between Unrelated Things	Two or more things are orthogonal if changes in one do not affect any of the others. Also called cohesion. Write "shy" code.
Project Teams	Functionality is divided.
Design	Easier to design a complete project through its components.
Toolkits and Libraries	Choose wisely to keep orthogonality.
Testing	Orthogonal systems are easier to test.
Documentation	Also gain quality
Gain Productivity	Changes are localized Promotes reuse M x N orthogonal components do more than M x N non orthogonal components
Reduce Risk	Diseased sections or code are isolated Are better tested Not tied to a product or platform
Coding: In order to keep orthogonality when adding code do:	Keep your code decoupled Avoid global data Avoid similar functions

Reversibility

There are no final decisions.

Tracer Bullets

Use Tracer Bullets to Find the Target	Advantages: Users get to see something working early Developers build a structure to work in You have an integration platform You have something to demonstrate You have a better feel for progress
Tracer Bullets Don't Always Hit Their Target	Tracer bullets show what you're hitting. This may not always be the target. You then adjust your aim until they're on target. That's the point.

Tracer Bullets (cont)

Tracer Code versus Prototyping	With a prototype, you're aiming to explore specific aspects of the final system. Tracer code is used to know how the application as a whole hangs together.
Prototyping generates disposable code	Tracer code is lean but complete, and forms part of the skeleton of the final system.

In new projects your users requirements may be vague. Use of new algorithms, techniques, languages, or libraries unknowns will come. And environment will change over time before you are done.
We're looking for something that gets us from a requirement to some aspect of the final system quickly, visibly, and repeatably.

Prototypes and Post-it Notes

Things to Prototype	Architecture New functionality in an existing system Structure or contents of external data Third-party tools or components Performance issues User interface design
What details can you ignore?	Correctness Completeness Robustness Style
Prototyping Architecture:	Are the responsibilities of the major components well defined and appropriate? Are the collaborations between major components well defined? Is coupling minimized? Can you identify potential sources of duplication? Are interface definitions and constraints acceptable? Does every module have an access path to the data it needs during execution?

We build software prototypes to analyse and expose risk, and to offer chances for correction at a greatly reduced cost.

Prototype anything that carries risk. Anything that hasn’t been tried before, or that is absolutely critical to the final system. Anything unproven, experimental, or doubtful. Anything you aren’t comfortable with.

Prototype to learn, and never deploy the prototype.

Domain Languages

Program close to the problem domain.

Estimating

Estimate to Avoid Surprises	**First:** Do they need high accuracy, or are they looking for a ballpark figure? **Second:** Scale time estimates properly
Where Do Estimates Come From?	Understand What's Being Asked Build a Model of the System Break the Model into Components Give Each Parameter a Value Calculate the Answers Keep Track of Your Estimating Prowess
Iterate the Schedule with the Code	Guess estimation Check requirements Analyze risk Design, implement, integrate Validate with the users Repeat
What to Say When Asked for an Estimate	"I'll get back to you."

Quote estimate in

1-15 days - days
3-8 weeks - weeks
8-30 weeks - months
30+ weeks - think hard before giving an estimate

The Basic Tools

Every craftsman starts his or her journey with a basic set of good quality tools.

The Power of Plain Text

Drawbacks	More space Computationally more expensive
The Power of Text	Insurance against obsolescence Leverage Easier testing

Keep knowledge in plain text.

Shell Games

Use the power of command shells.

Can't you do everything equally well by pointing and clicking in a GUI? **No**. A benefit of GUIs is WYSIWYG - what you see is what you get. The disadvantage is WYSIAYG - what you see is all you get.

Power Editing

Editor Features	Configurable
	Extensible
	Programmable
	Syntax highlighting
	Auto-completion
	Auto-indentation
	Initial code or document boilerplate
	Tie-in to help systems
	IDE-like features (compile, debug, and so on)

Use a single editor well.

Source Code Control

Always use source code control!

Debugging

A Debugging Mindset	Don't waste a single neuron on the train of thought that begins "but that can't happen" because quite clearly it can, and has. Try to discover the root cause of a problem, not just this particular appearance of it.
Where to Start	Before you start, check the warnings or better remove all of them. You first need to be accurate in your observations and data.
Bug Reproduction	The best way to start fixing a bug is to make it reproducible. The second best way is to make it reproducible with a single command.
Visualize Your Data	Use the tools that the debugger offers you. Pen and paper can also help.
Tracing	Know what happens before and after.
Rubber Ducking	Explain the bug to someone else.
Process of Elimination	"select" Isn't Broken
The Element of Surprise	Don’t Assume It—Prove It

Fix the problem, not the blame. And don't panic.

Debugging Checklist

Is the problem being reported a direct result of the underlying bug, or merely a symptom?
Is the bug really in the compiler? Is it in the OS? Or is it in your code?
If you explained this problem in detail to a coworker, what would you say?
If the suspect code passes its unit tests, are the tests complete enough? What happens if you run the unit test with this data?
Do the conditions that caused this bug exist anywhere else in the system?

Text Manipulation

Learn a text manipulation language.

Code Generators

Code Generators Needn't Be Complex	Keep the input format simple, and the code generator becomes simple.
Code Generators Needn't Generate Code	You can use code generators to write just about any output: HTML, XML, plain text - any text that might be an input somewhere else in your project.

Write Code That Writes Code

Passive code generators are run once to produce a result. They are basically parameterized templates, generating a given output from a set of inputs.

Active code generators are used each time their results are required. Take a single representation of some piece of knowledge and convert it into all the forms your application needs.

A Pragmatic Paranoia

You can't write Perfect Software. No one in the brief history of computing has ever written a piece of perfect software. Pragmatic Programmers don't trust themselves, either.

Design by Contract

Design with Contracts	Write "lazy" code: be strict in what you will accept before you begin, and promise as little as possible in return.
Implementing DBC	Simply enumerating at design time: what the input domain range is what the boundary conditions are what the routine promises to deliver (and what it doesn't)
Assertions	You can use assertions to apply DBC in some range. (Assertions are not propagated in subclasses)
Invariants	Loop Invariants: Is true before and during the loop, therefore also when the loop finishes Semantic Invariants: ie the error should be on the side of not processing a transaction rather than processing a duplicate transaction.

DBC enforces crashing early.

A correct program is one that does no more and no less than it claims to do. Use:

Preconditions
Postconditions
Invariants

Dead Programs Tell No Lies

Crash Early	A dead program normally does a lot less damage than a crippled one. When your code discovers that something that was supposed to be impossible just happened, your program is no longer viable.

All errors give you information. Pragmatic Programmers tell themselves that if there is an error, something very, very bad has happened.

Assertive Programming

If It can't happen, use Assertions to ensure that It won't.

Assertions are also useful checks on an algorithm's operation.

Don't use assertions in place of real error handling.

Leave assertions turned on, unless you have critical performance issues.

When to Use Exceptions

Use exceptions for exceptional problems.

What Is Exceptional?	The program must run if all the exception handlers are removed. If your code tries to open a file for reading and that file does not exist, should an exception be raised? Yes: If the file should have been there No: If you have no idea whether the file should exist or not

How to Balance Resources

Finish What You Start	When managing resources: memory, transactions, threads, flies, timers—all kinds of things with limited availability, we have to close, finish, delete, deallocate them when we are done.
Nest Allocations	Deallocate resources in the opposite order to that in which you allocate them When allocating the same set of resources in different places in your code, always allocate them in the same order (prevent deadlocks)
Objects and Exceptions	Use finally to free resources.

Bend, or Break

In order to keep up with today’s near-frantic pace of change, we need to make every effort to write code that’s as loose - as flexible - as possible.

Decoupling and the Law of Demeter

Minimize Coupling	Be careful about how many other modules you interact with and how you came to interact with them. Traversing relationships between objects directly can quickly lead to a combinatorial explosion. Symptoms: 1.Large projects where the command to link a unit test is longer than the test program itself 2."Simple" changes to one module that propagate through unrelated modules in the system 3.Developers who are afraid to change code because they aren't sure what might be affected

Decoupling and the Law of Demeter (cont)

The Law of Demeter for Functions	Any method of an object should call only methods belonging to: - itself - any parameters that were passed in to the method - any objects it created - any directly held component objects
Does It Really Make a Difference?	Using The Law of Demeter will make your code more adaptable and robust, but at a cost: you will be writing a large number of wrapper methods that simply forward the request on to a delegate. imposing both a runtime cost and a space overhead. Balance the pros and cons for your particular application.

Metaprogramming

Put Abstractions in Code, Details in Metadata	We want to configure and drive the application via metadata as much as possible. Program for the general case, and put the specifics somewhere else —outside the compiled code base.
When to Configure	A flexible approach is to write programs that can reload their configuration while they're running. - long-running server process: provide some way to reread and apply metadata while the program is running. - small client GUI application: if restarts quickly no problem.

Metaprogramming (cont)

Benefits
- It forces you to decouple your design, which results in a more flexible and adaptable program.
- It forces you to create a more robust, abstract design by deferring details—deferring them all the way out of the program.
- You can customize the application without recompiling it.
- Metadata can be expressed in a manner that's much closer to the problem domain than a general-purpose programming language might be.
- You may even be able to implement several different projects using the same application engine, but with different metadata.

Configure, don't integrate.

Temporal Coupling

Two aspects of time:

- Concurrency: things happening at the same time
- Ordering: the relative positions of things in time

We need to allow for concurrency and to think about decoupling any time or order dependencies. Reduce any time-based dependencies

Workflow

Analyze Workflow to Improve Concurrency	Use activity diagrams to maximize parallelism by identifying activities that could be performed in parallel, but aren't.

Architecture

Design using services.

Balance load among multiple consumer processes: the hungry consumer model.

In a hungry consumer model, you replace the central scheduler with a number of independent consumer tasks and a centralized work queue. Each consumer task grabs a piece from the work queue and goes on about the business of processing it. As each task finishes its work, it goes back to the queue for some more. This way, if any particular task gets bogged down, the others can pick up the slack, and each individual component can proceed at its own pace. Each component is temporally decoupled from the others.

Design for Concurrency

Programming with threads imposes some design constraints—and that's a good thing.

- Global or static variables must be protected from concurrent access
- Check if you need a global variable in the first place.
- Consistent state information, regardless of the order of calls
- Objects must always be in a valid state when called, and they can be called at the most awkward times. Use class invariants, discussed in Design by Contract.

Thinking about concurrency and time-ordered dependencies can lead you to design cleaner interfaces as well.

Deployment

You can be flexible as to how the application is deployed: standalone, client-server, or n-tier.

If we design to allow for concurrency, we can more easily meet scalability or performance requirements when the time comes—and if the time never comes, we still have the benefit of a cleaner design.

It's Just a View

Publish/Subscribe	Objects should be able to register to receive only the events they need, and should never be sent events they don't need. Use this publish/subscribe mechanism to implement a very important design concept: the separation of a model from views of the model.
Model-View-Controller	Separates the model from both the GUI that represents it and the controls that manage the view. Advantage: - Support multiple views of the same data model. - Use common viewers on many different data models. - Support multiple controllers to provide nontraditional input mechanisms.
Beyond GUIs	The controller is more of a coordination mechanism, and doesn't have to be related to any sort of input device. **Model** The abstract data model representing the target object. The model has no direct knowledge of any views or controllers. **View** A way to interpret the model. It subscribes to changes in the model and logical events from the controller. **Controller** A way to control the view and provide the model with new data. It publishes events to both the model and the view.

Blackboards

A blackboard system lets us decouple our objects from each other completely, providing a forum where knowledge consumers and producers can exchange data anonymously and asynchronously.

With Blackboard systems, you can store active objects - not just data - on the blackboard, and retrieve them by partial matching of fields (via templates and wildcards) or by subtypes.

Blackboards (cont)

Functions that a Blackboard system should have	- **read** Search for and retrieve data from the space. - **write** Put an item into the space. - **take** Similar to read, but removes the item from the space as well. - **notify** Set up a notification to occur whenever an object is written that matches the template. Organize your Blackboard by partitioning it when working on large cases.

Use blackboards to coordinate workflow

While you are coding

We should avoid programming by coincidence—relying on luck and accidental successes— in favor of programming deliberately.**Don't program by coincidence.**

Program by Coincidence

Program Deliberately	Always be aware of what you are doing. Don't code blindfolded. Proceed from a plan. Rely only on reliable things. Document your assumptions. Don't just test your code, but test your assumptions as well. Don't guess Prioritize your effort. Don't be a slave to history. Don't let existing code dictate future code.

Algorithm Speed

Pragmatic Programmers estimate the resources that algorithms use—time, processor, memory, and so on.

Use Big O Notation	**O(1)**: Constant (access element in array, simple statements) **O(lg(n))**: Logarithmic (binary search) lg(n) = lg2(n) **O(n)**: Linear: Sequential search **O(n lg(n))**: Worse than linear but not much worse(average runtime of quickshort, headsort) **O(n^2)**: Square law (selection and insertion sorts) **O(n^3)**: Cubic (multiplication of 2 n x n matrices) **O(C^n)**: Exponential (travelling salesman problem, set partitioning)

Algorithm Speed (cont)

Common Sense Estimation	- Simple loops: O(n) - Nested loops: O(n^2) - Binary chop: O(lg(n)) - Divide and conquer: O(n lg(n)). Algorithms that partition their input, work on the two halves independently, and then combine the result. - Combinatoric: O(C^n)
Estimate the Order of Your Algorithms	
Test Your Estimates	

Refactoring

When Should You Refactor?	- Duplication. You've discovered a violation of the DRY principle. - Nonorthogonal design. You've discovered some code or design that could be made more orthogonal. - Outdated knowledge. Things change, requirements drift, and your knowledge of the problem increases. Code needs to keep up. - Performance. You need to move functionality from one area of the system to another to improve performance.
Refactor Early, Refactor Often	
How Do You Refactor?	- Don't try to refactor and add functionality at the same time. - Make sure you have good tests before you begin refactoring. - Take short, deliberate steps.

Code needs to evolve; it's not a static thing.

Code That's Easy to Test

Unit Testing	Testing done on each module, in isolation, to verify its behavior. A software unit test is code that exercises a module.
Testing Against Contract	This will tell us two things: 1.Whether the code meet the contract 2.Whether the contract means what we think it means.
Design to Test	There's no better way to fix errors than by avoiding them in the first place. Build the tests before you implement the code.

Code That's Easy to Test (cont)

Writing Unit Tests	By making the test code readily accessible, you are providing developers who may use your code with two invaluable resources: 1.Examples of how to use all the functionality of your module 2.A means to build regression tests to validate any future changes to the code You must run them, and run them often.
Using Test Harnesses	Test harnesses should include the following capabilities: - A standard way to specify setup and cleanup - A method for selecting individual tests or all available tests - A means of analyzing output for expected (or unexpected) results - A standardized form of failure reporting
Build a Test Window	- Log files. - Hot-key sequence. - Built-in Web server.
A Culture of Testing	Test your Software, or your users will.

Evil Wizards

Don't Use Wizard Code You Don't Understand	If you do use a wizard, and you don't understand all the code that it produces, you won't be in control of your own application.

Before the project

With these critical issues sorted out before the project gets under way, you can be better positioned to avoid “analysis paralysis” and actually begin your successful project.

The Requirements Pit

Don't Gather Requirements—Dig for Them	Policy may end up as metadata in the application. Gathering requirements in this way naturally leads you to a system that is well factored to support metadata.
Work with a User to Think Like a User	
Documenting Requirements	Use "use cases"
Overspecifying	Requirements are not architecture. Requirements are not design, nor are they the user interface. Requirements are need.
Seeing Further	Abstractions live longer than details.

The Requirements Pit (cont)

Just One More Wafer-Thin Mint…	What can we do to prevent requirements from creeping up on us? The key to managing growth of requirements is to point out each new feature's impact on the schedule to the project sponsors.
Use a Project Glossary	It's very hard to succeed on a project where the users and developers refer to the same thing by different names or, even worse, refer to different things by the same name.
Get the Word Out	Publishing project documents to internal Web sites for easy access by all participants.

Perfection is achieved, not when there is nothing left to add, but when there is nothing left to take away. . . . - Antoine de St. Exupery, Wind, Sand, and Stars, 1939

Solving Impossible Puzzles

Don't Think Outside the Box—Find the Box	The key to solving puzzles is both to recognize the constraints placed on you and to recognize the degrees of freedom you do have, for in those you'll find your solution.
There Must Be an Easier Way!	If you can not find the solution, step back and ask yourself these questions: - Is there an easier way? - Are you trying to solve the right problem, or have you been distracted by a peripheral technicality? - Why is this thing a problem? - What is it that's making it so hard to solve? - Does it have to be done this way? - Does it have to be done at all?

Not Until You're Ready

Listen to Nagging Doubts—Start When You're Ready	If you sit down to start typing and there's some nagging doubt in your mind, heed it.
Good Judgment or Procrastination?	Start prototyping. Choose an area that you feel will be difficult and begin producing some kind of proof of concept, and be sure to remember why you're doing it and that it is a prototype.

The Specification Trap

Some things are better done than described.

Writing a specification is quite a responsibility. You should know when to stop:

- Specification will never capture every detail of a system or its requirement.
- The expressive power of language itself might not be enough to describe a specification
- A design that leaves the coder no room for interpretation robs the programming effort of any skill and art.

Circles and Arrows

Don't Be a Slave to Formal Method	Formal methods have some serious shortcomings: - Diagrams are meaningless to the end users, show the user a prototype and let them play with it. - Formal methods seem to encourage specialization. It may not be possible to have an in-depth grasp of every aspect of a system. - We like to write adaptable, dynamic systems, using metadata to allow us to change the character of applications at runtime, but most current formal methods don't allow it.
Do Methods Pay Off?	Never underestimate the cost of adopting new tools and methods.
Should We Use Formal Methods?	Absolutely, but remember that is just one more tool in the toolbox.

Expensive tools do not produce better designs.

Pragmatic Projects

Pragmatic Teams

No Broken Windows	Quality is a team issue. Teams as a whole should not tolerate broken windows—those small imperfections that no one fixes. Quality can come only from the individual contributions of all team members.
Boiled Frogs	People assume that someone else is handling an issue, or that the team leader must have OK'd a change that your user is requesting. Fight this.

Pragmatic Teams (cont)	
Communicate	The team as an entity needs to communicate clearly with the rest of the world. People look forward to meetings with them, because they know that they'll see a well-prepared performance that makes everyone feel good. There is a simple marketing trick that helps teams communicate as one: generate a brand.
Don't Repeat Yourself	Appoint a member as the project librarian.
Orthogonality	It is a mistake to think that the activities of a project—analysis, design, coding, and testing—can happen in isolation. They can't. These are different views of the same problem, and artificially separating them can cause a boatload of trouble.
Organize Around Functionality, Not Job Functions	Split teams by functionally. Database, UI, API Let the teams organize themselves internally Each team has responsibilities to others in the project (defined by their agreed-upon commitments) We're looking for cohesive, largely self-contained teams of people Organize our resources using the same techniques we use to organize code, using techniques such as contracts (Design by Contract), decoupling (Decoupling and the Law of Demeter), and orthogonality (Orthogonality), and we help isolate the team as a whole from the effects of change.
Automation	Automation is an essential component of every project team.
Know when to stop adding paint.	

Ubiquitous Automation	
All on Automatic	**Don't Use Manual Procedures.** Using cron, we can schedule backups, nightly build, Web site... unattended, automatically.
Compiling the Project	We want to check out, build, test, and ship with a single command: - Generating Code - Regression Tests

Ubiquitous Automation (cont)

Build Automation	A build is a procedure that takes an empty directory (and a known compilation environment) and builds the project from scratch, producing whatever you hope to produce as a final deliverable. 1. Check out the source code from the repository 2. Build the project from scratch (marked with the version number). 3. Create a distributable image 4. Run specified tests When you don't run tests regularly, you may discover that the application broke due to a code change made three months ago. Good luck finding that one. **Nightly build** run it every night. **Final builds** (to ship as products), may have different requirements from the regular nightly build.
Automatic Administrivia	Our goal is to maintain an automatic, unattended, content-driven workflow. Web Site Generation results of the build itself, regression tests, performance statistics, coding metrics... Approval Procedures get marks / *Status: needs_review* /, send email...
The Cobbler's Children	Let the computer do the repetitious, the mundane—it will do a better job of it than we would. We've got more important and more difficult things to do.

Ruthless testing

Test Early. Test Often. Test Automatically.	Tests that run with every build are the most effective. The earlier a bug is found, the cheaper it is to remedy. "Code a little, test a little".

Ruthless testing (cont)

What to Test	- Unit testing: code that exercises a module. - Integration testing: the major subsystems that make up the project work and play well with each other. - Validation and verification: test if you are delivering what users needs. - Resource exhaustion, errors, and recovery: discover how it will behave under real-world conditions. (Memory, Disk, CPU, Screen...) - Performance testing: meets the performance requirements under real-world conditions. - Usability testing: performed with real users, under real environmental conditions.
How to Test	- Regression testing: compares the output of the current test with previous (or known) values. Most of the tests are regression tests. - Test data: there are only two kinds of data: real-world data and synthetic data. - Exercising GUI systems: requires specialised testing tools, based on a simple event capture/playback model. - Testing the tests: After you have written a test to detect a particular bug, cause the bug deliberately and make sure the test complains. Use Saboteurs to Test Your Testing - Testing thoroughly. Test State Coverage, Not Code Coverage
When to Test	As soon as any production code exists, it needs to be tested. Most testing should be done automatically.
Tightening the Net	If a bug slips through the net of existing tests, you need to add a new test to trap it next time. Find Bugs Once.

Coding ain't done 'til all the tests run.

Pragmatic Programmers are driven to find our bugs now, so we don't have to endure the shame of others finding our bugs later.

By **Marco Santos** (marconlsantos)

cheatography.com/marconlsantos/

Published 12th September, 2018.
Last updated 14th February, 2021.

It's All Writing

Comments in Code	In general, comments should discuss why something is done, its purpose and its goal. Remember that you (and others after you) will be reading the code many hundreds of times, but only writing it a few times. Even worse than meaningless names are misleading names. One of the most important pieces of information that should appear in the source file is the author's name—not necessarily who edited the file last, but the owner.
Executable Documents	Create documents that create schemas. The only way to change the schema is to change the document.
Technical Writers	We want the writers to embrace the same basic principles that a Pragmatic Programmer does—especially honoring the DRY principle, orthogonality, the model-view concept, and the use of automation and scripting.
Print It or Weave It	Paper documentation can become out of date as soon as it's printed. Publish it online, on the Web. Remember to put a date stamp or version number on each Web page. Using a markup system, you have the flexibility to implement as many different output formats as you need.
Markup Languages	Documentation and code are different views of the same underlying model, but the view is all that should be different.

Treat English as just another programming language.

Build documentation in, don't bolt it on.

If there's a discrepancy, the code is what matters—for better or worse.

Great Expectations

Communicating Expectations	Users initially come to you with some vision of what they want. You cannot just ignore it. Everyone should understand what's expected and how it will be built.
The Extra Mile	Give users that little bit more than they were expecting. - Balloon or ToolTip help - Keyboard shortcuts - A quick reference guide as a supplement to the user's manual Colorization - Log file analyzers - Automated installation - Tools for checking the integrity of the system - The ability to run multiple versions of the system for training - A splash screen customized for their organization

Great Expectations (cont)

Pride and Prejudice	Pragmatic Programmers don't shirk from responsibility. Instead, we rejoice in accepting challenges and in making our expertise well known. We want to see pride of ownership. "I wrote this, and I stand behind my work."

Sign Your Work.

The success of a project is measured by how well it meets the expectations of its users.

Gently Exceed Your Users' Expectations.

Acknowledgements

Built from Hugo Matilla's summary. **You should still read the book, if you haven't already.**

By **Marco Santos**
(marconlsantos)

cheatography.com/marconlsantos/

Published 12th September, 2018.
Last updated 14th February, 2021.