Quick Reference :

Pragmatic Programmer general characteristics :

- Fast adapter
- Inquisitive(Tend to ask questions)
- Critical Thinker
- Realistic
- Jack of all trades

Tip 1 : Care about their work.

Pragmatic Programmers care about whatever work they are doing.

Tip 2 : Think about your work

Never run on auto pilot.Don’t do your work without thinking about it.

Tip 3 : Provide Options, Don't Make Lame Excuses

A Pragmatic Programmer takes charge of his or her own career, and isn't afraid to admit ignorance or error.When you do accept the responsibility for an outcome, you should expect to be held accountable for it. When you make a mistake (as we all do) or an error in judgment, admit it honestly and try to offer options.Don't say it can't be done; explain what can be done to salvage the situation.

Tip 4 : Don’t live with “Broken Windows”

Don't leave "broken windows" (bad designs, wrong decisions, or poor code) unrepaired. Fix each one as soon as it is discovered. If there is insufficient time board it up.Neglecting them accelerates rot faster than any other factor.

If you find yourself on a team and a project where the code is pristinely beautiful—cleanly written, well designed, and elegant—you will likely take extra special care not to mess it up, just like the firefighters. Even if there's a fire raging (deadline, release date, trade show demo, etc.), you don't want to be the first one to make a mess.

Tip 5 : Be a catalyst for the change

You may be in a situation where you know exactly what needs doing and how to do it. The entire system just appears before your eyes—you know it's right. But ask permission to tackle the whole thing and you'll be met with delays and blank stares. People will form committees, budgets will need approval, and things will get complicated. Everyone will guard their own resources. Sometimes this is called "start-up fatigue."

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...." Pretend it's not important. Sit back and wait for them to start asking you to add the functionality you originally wanted. People find it easier to join an ongoing success. Show them a glimpse of the future and you'll get them to rally around.

Tip 6 : Remember the Big Picture

Most software disasters start out too small to notice, and most project overruns happen a day at a time. Keep an eye on the big picture. Constantly review what's happening around you, not just what you personally are doing.

Tip 7 : Make Quality a Requirements Issue

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

Know When To Stop :

Great software today is often preferable to perfect software tomorrow. If you give your users something to play with early, their feedback will often lead you to a better eventual solution.Don't spoil a perfectly good program by overembellishment and over-refinement. Move on, and let your code stand in its own right for a while. It may not be perfect. Don't worry: it could never be perfect.

Tip 8 : Invest regularly in your “Knowledge Portfolio”

Your knowledge and experience are your most important professional assets. Unfortunately, they're expiring assets.Your knowledge becomes out of date as new techniques, languages, and environments are developed. Changing market forces may render your experience obsolete or irrelevant.We want to prevent this from ever happening.

Managing a knowledge portfolio is similar to managing a financial portfolio.

Building Portfolio :

- Invest Regularly - Invest in your knowledge portfolio regularly
- Diversify - The more technologies you are comfortable with, the better you will be able to adjust to change.
- Manage Risk - Don’t put all your technical eggs in one basket
- Buy low, sell high - Learning an emerging technology before it becomes popular can be just as hard as finding an undervalued stock, but the payoff can be just as rewarding.
- Review and Rebalance.

Suggestions to acquire intellectual capital :

- Learn at least one new language every year.
- Read a technical book each quarter
- Read non-technical books too

- Take classes.
- Participate in local groups
- Stay updated

Tip 9 : Critically analyze what you hear and read

You need to ensure that the knowledge in your portfolio is accurate and unswayed by either vendor or media hype.

Tip 10 : It's Both What You Say and the Way You Say It

A large part of our day is spent communicating, so we need to do it well.

- Know what you want to say - Plan what you want to say. Write an outline. Then ask yourself, "Does this get across whatever I'm trying to say?" Refine it until it does.
- Know your Audience - You're communicating only if you're conveying information. To do that, you need to understand the needs, interests, and capabilities of your audience.WISDOM acrostic :
  - What they Want?
  - What is their Interest?
  - How Sophisticated are they?
  - How much Detail they want?
  - Who do you want to Own the information?
  - How can you Motivate them to listen?
- Choose Your Moment - Make what you're saying relevant in time, as well as in content. Sometimes all it takes is the simple question "Is this a good time to talk about...?"
- Choose a Style - Adjust the style of your delivery to suit your audience(Like large bound reports or simple email)
- Make it look good - Your ideas are important. They deserve a good-looking vehicle to convey them to your audience.
- Involve your audience - If possible involve your readers with early drafts of your document. Get their feedback, and pick their brains. You'll build a good working relationship, and you'll probably produce a better document in the process.
- Be a Listener - There's one technique that you must use if you want people to listen to you: listen to them.
- Get back to People - Keeping people informed makes them far more forgiving of the occasional slip, and makes them feel that you haven't forgotten them.

Tips for email communication :

- Proofread before you hit Send.
- Check the spelling.

- Keep the format simple.
- Try to keep quoting to a minimum.
- Don't flame unless you want it to come back and haunt you later.
- Check your list of recipients before sending.
- Archive and organize your e-mail–both the import stuff you receive and the mail you send.

Tip 11 : DRY—Don't Repeat Yourself

We feel that the only way to develop software reliably, and to make our developments easier to understand and maintain we should follow DRY principle.

Most of the duplication we see falls into one of the following categories:

- 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.

Tip 12 : Make it easy to reuse.

Cause if you fail to reuse, you risk duplicating knowledge.

Tip 13 : Eliminate Effects Between Unrelated Things

In computing, the term has come to signify a kind of independence or decoupling. Two or more things are orthogonal if changes in one do not affect any of the others.When components are isolated from one another, you know that you can change one without having to worry about the rest.

You get two major benefits if you write orthogonal systems: increased productivity and reduced risk.

Gain Productivity :

- Changes are localized.
- Promotes reuse
- Get more functionality per unit effort by combining orthogonal components.

Reduce Risk :

- Diseased sections of code are isolated.
- The resulting system is less fragile.
- Probably be better tested.

        - Will not be tightly tied to a particular vendor.

Ways to apply principle of orthogonality :

- Project Teams - Proper division on the basis of functionality.When teams are organized with lots of overlap,members are confused about responsibilites.
- Design - Systems should be composed of a set of cooperating modules , each of which implements functionality independent of the others.
- Toolkits and Libraries - When you bring in a toolkit(or even library from other members of your team), ask yourself whether it imposes changes on your code that shouldn’t be there.
- Coding -
    - Keep your code decoupled - Try the Law of Demeter.If you need to change an object's state, get the object to do it for you. This way your code remains isolated from the other code's implementation and increases the chances that you'll remain orthogonal.
    - Avoid global data - Everytime your code references global data, it ties itself to other components that share the data.
    - Avoid similar functions
- Testing - Every module should have its own unit test built into its code, and that these tests be performed automatically as part of the regular build process.
- Documentation - With truly orthogonal documentation , you should be able to change the appearance without dramatically affecting the content.

Tip 14 : There are no final decisions.

The mistake lies in assuming that any decision is cast in stone—and in not preparing for the contingencies that might arise. Instead of carving decisions in stone, think of them more as being written in the sand at the beach. A big wave can come along and wipe them out at any time.

Tip 15 : Use Tracer Bullets to find the Target.

(Tracer bullets are loaded at intervals on the ammo belt alongside regular ammunition. When they're fired, their phosphorus ignites and leaves a pyrotechnic trail from the gun to whatever they hit. If the tracers are hitting the target, then so are the regular bullets.)

Instead of one big calculation up front ,then shoot and hope; Pragmatic Programmers tend to prefer using tracer bullets.

Tracer code is not disposable: you write it for keeps. It contains all the error checking, structuring, documentation, and self-checking that any piece of production code has. It simply is not fully functional. However, once you have achieved an end-to-end connection among the components of your system, you can check how close to the target you are, adjusting if necessary. Once you're on target, adding functionality is easy. Tracer development is consistent with the idea that a project is never inished: there will always be changes required and functions to add. It is an incremental approach.

Tracer Bullets Don't Always Hit Their Target.You then adjust your aim until they're on target.

Advantages of tracer code approach :

- 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 better feel for progress.

Tip 16 : Prototype to Learn

Prototyping is a learning experience. Its value lies not in the code produced, but in the lessons learned. That's really the point of prototyping.

When used properly, a prototype can save you huge amounts of time, money, pain, and suffering by identifying and correcting potential problem spots early in the development cycle—the time when fixing mistakes is both cheap and easy.

Post-it notes are great for prototyping dynamic things such as workflow and application logic.

When prototyping avoid details :

- Correctness
- Completeness
- Robustness
- Style

Prototype anything that:

- Is doubtful
- Carries risk
- Hasn't been tried before
- Is absolutely critical to the final system
- Is unproven
- Is experimental

Samples:

- Architecture
- New functionality in an existing system
- Structure or contents of external data
- Third-party tools or components
- Performance issues
- User interface design

Prototype vs Tracer Code

With a prototype, you're aiming to explore specific aspects of the final system. With a true prototype, you will throw away whatever you lashed together when trying out the concept, and recode it properly using the lessons you've learned.

The tracer code approach addresses a different problem. You need to know how the application as a whole hangs together. You want to show your users how the interactions will work in practice, and you want to give your developers an architectural skeleton on which to hang code.

Prototyping generates disposable code. Tracer code is lean but complete, and forms part of the skeleton of the final system. Think of prototyping as the reconnaissance and intelligence gathering that takes place before a single tracer bullet is fired.

Tip 17 : Program Close to the Problem domain.

Tip 18 : Estimate to Avoid Surprises.

Scale time estimates as follows :

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

Basic estimating trick is ask someone who’s already done it.
Where do estimates come from ?

- Understand What's Being Asked
- Build a Model of the System - Helps in discovering underlying patterns.
- Break the Model into components - Identify each parameter.
- Give Each Parameter a Value - Once you have the parameters broken out, you can go through and assign each one a value. You expect to introduce some errors in this step. The trick to work out which parameters have the most impact on the result, and concentrate on getting them about right.
- Calculate the Answers

- Keep Track of Your Estimating Prowess - See how close you were with your estimates.

Tip 19 : Iterate the Schedule with the Code

Estimating Project Schedules

The only way to determine the timetable for a project is by gaining experience on that same project. Practice incremental development, repeating the following steps:

- Guess estimation
- Check requirements
- Analyze risk
- Design, implement, integrate
- Validate with the users
- Repeat

Tip 20 : Keep knowledge in a Plain Text

Plain text is made up of printable characters that can be read and understood easily.It can be structured like htm,xml.

Drawbacks of plain text :

- It may take more space to store than a compressed binary format.
- It may be computationally more expensive to interpret and process a plain text file.

Even in situations where you can't use Plain text , it may be acceptable to store metadata about the raw data in plain text.

Tip 21 : Use the power of Command Shells

By gaining familiarity with the shell, you productivity will increase.

Tip 22 : Use a Single Editor well.

Choose an editor, know it thoroughly, and use it for all editing tasks.

Tip 23 : Always use Source Code Control.

Tip 24 : Fix the Problem not the blame.

Tip 25 : Don't Panic

While trying to find the cause of a bug, its very important to step back a pace, and actually think about what could be causing the symptoms that you believe indicate a bug.

Tip 26 : Select isn’t broken

Its generally more profitable to assume that the application code is incorrectly calling into a library than to assume that the library itself is broken.

Debugging Strategies

- 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 - Now what happens before and after.
- Rubber Ducking - Explain the bug to someone else.
- Process of Elimination - It is possible that a bug exists in the OS, the compiler, or a third-party product—but this should not be your first thought.

Tip 27 : Don’t assume it - Prove it.

When faced with a "surprising" failure, you must realize that one or more of your assumptions is wrong. Don't gloss over a routine or piece of code involved in the bug because you "know" it works. Prove it. Prove it in this context, with this data, with these boundary conditions.

Tip 28 : Learn a Text Manipulation Language.

Tip 29 : Write Code That Writes Code

Two types of code generators :

- Passive code generators are run once to produce a result. They are basically parameterized templates, generating a given output from a set of inputs.
  Uses of Passive code generators :
    - Creating new source files.
    - Performing one-off conversions among programming languages.
    - Producing lookup tables and other resources that are expensive to compute at runtime.
- 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.This is not duplication as the derived forms are disposable and are generated by the code generator.Helps in following DRY principle.
  Eg : Active code generator to create code for structure from database schema.

Tip 30: You can't write Perfect Software

Tip 31: Design by Contract.

A correct program is the one which does exactly what it is asked for, no more nor less.

- Preconditions
- Postconditions
- Invariants

Promise as little as prossible.

Implementing DBC :

Simply enumerating at design time:

- What is the input domain range.
- What the boundary conditions are
- What the routine promises to deliver.

Tip 32: Crash Early

A dead program normally does a lot less damage than a crippled one.

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

Tip 33 : If It Can't Happen, Use Assertions to Ensure That It Won't

Assertions check for things that should never happen.Don't use assertions in place of real error handling.

Leave Assertions Turned On. Turning off assertions when you deliver a program to production is like crossing a high wire without a net because you once made it across in practice. There's dramatic value, but it's hard to get life insurance. Even if you do have performance issues, turn off only those assertions that really hit you.

Tip 34 : Use Exceptions for Exceptional Problems

Exceptions should be reserved for unexpected events. Assume that an uncaught exception will terminate your program and ask yourself, "Will this code still run if I remove all the exception handlers?" If the answer is "no," then maybe exceptions are being used in nonexceptional circumstances.

For example, if your code tries to open a file for reading and that file does not exist, should an exception be raised? Our answer is, "It depends." If the file should have been there, then an exception is warranted. Something unexpected happened—a file you were expecting to exist seems to have disappeared. On the other hand, if you have no idea whether the file should exist or not, then it doesn't seem exceptional if you can't find it, and an error return is appropriate.

Tip 35 : Finish what you start

It simply means that the routine or object that allocates a resource should be responsible for deallocating it.

Resource usage usually follows a predictable pattern: you allocate the resource, use it, and then deallocate it.

Deallocate resources in the opposite order to that in which you allocate them. That way you won't orphan resources if one resource contains references to another.

When allocating the same set of resources in different places in your code, always allocate them in the same order. This will reduce the possibility of deadlock.

Tip 36 : Minimize Coupling

Law of Demeter for Functions attempts to minimize coupling. It states that any method of the object should call only methods belonging to :

- Itself
- Any parameters that were passed in the method
- Any objects it created
- Any directly held component objects

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.

Tip 37: Configure, Don't Integrate

We should make our code highly configurable and “soft”,that is easily adaptable to changes.

Tip 38: Put Abstractions in Code Details in Metadata

Metadata is any data that describes the application—how it should run, what resources it should use, and so on. Typically, metadata is accessed and used at runtime, not at compile time.

Program for the general case, and put the specifics somewhere else —outside the compiled code base.

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.

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.

Tip 39: Analyze Workflow to Improve Concurrency

Temporal Coupling - coupling in time. Two aspects of time: Concurrency(things happening at the same time) and Ordering(the relative positions of things in time).

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

Tip 40 : 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.

Tip 41: Always Design for Concurrency

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

- 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.

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

Tip 42 : Seperate Views from Models

Publish/Subscribe :

Use this mechanism to implement a very important design concept: the separation of a model from views of the model.In this sender of the event doesn’t need to have any explicit knowledge of the reciever.

Model-View-Controller :

This is the key concept behind the Model-View-Controller (MVC0 idiom: separating the model from both the GUI that represents it and the controls that manage the view.

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.

Tip 43: Use Blackboards to Coordinate Workflow

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.

Main operations on BlackBoard are read,write,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.)

We can use Blackboard to design algorithms based on a flow of objects,not just data.

If your BlackBoard becomes too large, organize it by partioning.

Tip 44 : Don't Program by Coincidence

How to Program Deliberately :

- Always be aware of what you are doing.
- Don't code blindfolded.
- Proceed with a plan.
- Rely only on reliable things.
- Document your assumptions.(Design by Contract)
- Don't just test your code, but test your assumptions as well. Don't guess. (Assertive Programming)
- Prioritize your effort.
- Don't be a slave to history. Don't let existing code dictate future code.(Refactoring)

Tip 45: Estimate the Order of Your Algorithms

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

Some Common Notations :

- Constant : O(1)
- 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 : $O(C^n)$

Tip 46: Test Your Estimates

Pragmatic Programmers try to cover both the theoretical and practical bases. After all this estimating, the only timing that counts is the speed of your code, running in the production environment, with real data.

Be pragmatic about choosing appropriate algorithms—the fastest one is not always the best for the job. Be wary of premature optimization. Make sure an algorithm really is a bottleneck before investing time improving it.

Tip 47: Refactor Early, Refactor Often

Rewriting, reworking, and re-architecting code is collectively known as refactoring.

When to Refactor :

- Duplication - You've discovered a violation of the DRY principle.
- Nonorthogonal design.
- 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.

How To 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.

Tip 48 : Design To Test

We should build testability into the software from the very beginning, and test each piece thoroughly before trying to wire them together.

- Unit Testing :
  - It is testing done on each module, in isolation, to verify its behavior.
  - The unit tests need to be conveniently located. For small projects, you can embed the unit test for a module in the module itself. For larger projects, we

        suggest moving each test into a subdirectory. Either way, remember that if it isn't easy to find, it won't be used.
    - Using Test Harnesses :

        A test harness can handle common operations such as logging status, analyzing output for expected results, and selecting and running the tests. 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

- Testing Against Contract
    - This will tell us two things: Whether the code meet the contract and Whether the contract means what we think it means.
    - By emphasizing testing against contract, we can try to avoid as many of those downstream disasters as possible.

Tip 49 : Test You software , or your user will

All software you write will be tested—if not by you and your team, then by the eventual users—so you might as well plan on testing it thoroughly. A little forethought can go a long way toward minimizing maintenance costs and help-desk calls.

Tip 50 : Don’t use Wizard code you don’t understand

You won’t be able to maintain it and you will be struggling when it comes to debug.

Tip 51 : Don't Gather Requirements—Dig for Them

Requirements rarely lie on the surface. Normally, they're buried deep beneath layers of assumptions, misconceptions, and politics.

Tip 52 : Work with a User to Think Like a User

It's important to discover the underlying reason why users do a particular thing, rather than just the way they currently do it. At the end of the day, your development has to solve their business problem, not just meet their stated requirements. Documenting the reasons behind requirements will give your team invaluable information when making daily implementation Decisions. There's a simple technique for getting inside your users' requirements that isn't used often enough: become a user.

Tip 53: Abstractions Live Longer than Details

One way of looking at use cases is to emphasize their goal-driven nature.

A big danger in producing a requirements document is being too specific. Good requirements documents remain abstract.

Requirements are not architecture. Requirements are not design, nor are they the user interface. Requirements are need.

Tip 54: Use a Project Glossary

Create and maintain a project glossary—one place that defines all the specific terms and vocabulary used in a project. 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.

Tip 55: Don't Think Outside the Box—Find the Box

It's not whether you think inside the box or outside the box. The problem lies in finding the box—identifying the real constraints.

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.We want to identify the most restrictive constraints first, and fit the remaining constraints within them.

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?

Tip 56: 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.

Tip 57: Some Things Are Better Done than Described

Program specification is the process of taking a requirement and reducing it down to the point where a programmer's skill can take over.

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.

Tip 58: Don't Be a Slave to Formal Methods

We should use Formal Methods. But always remember that formal development methods are just one more tool in the toolbox. If, after careful analysis, you feel you need to use a formal method, then embrace it—but remember who is in chargeNever become a slave to a methodology: circles and arrows make poor masters. Pragmatic Programmers look at methodologies critically, then extract the best from each and meld them into a set of working practices that gets better each month. This is crucial.

Tip 59: Expensive Tools Do Not Produce Better Designs

Try not to think about how much a tool costs when you look at its output.

Tip 60: Organize Around Functionality, Not Job Functions

Pragmatic Teams :

- No Broken Windows
- People assume that someone else is handling an issue.Avoid this.
- DRY
- Orthogonality.

Split teams by functionally.
The project needs at least two heads : Technical and Administrative.

Tip 61: Don't Use Manual Procedures

Manual procedures leave consistency up to chance; repeatability isn't guaranteed, especially if aspects of the procedure are open to interpretation by different people.A great way to ensure both consistency and accuracy is to automate everything the team does.

Compiling the Project

We want to check out, build, test, and ship with a single command

- Generating Code
- Regression Tests

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...

Tip 62: Test Early. Test Often. Test Automatically.

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

Tip 63: Coding Ain't Done til All the Tests Run

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". Test as frequently as you can.

What to Test :

- Unit Testing - It is testing done on each module, in isolation, to verify its behavior.
- Integration Testing - In this you're testing how entire subsystems honor their contracts Integration testing shows that the major subsystems that make up the

project work and play well with each other. With good contracts in place and well tested, any integration issues can be detected easily.
- Validation and Verification - As soon as you have an executable user interface or prototype, you need to answer an all-important question: the users told you what they wanted, but is it what they need?
- Resource Exhaustion, Errors, and Recovery - You need to discover how it will behave under real-world conditions.A few limits you may encounter :
    - Memory
    - Disk space
    - CPU bandwidth
    - Wall-clock time
    - Disk bandwidth
    - Network bandwidth
    - Color Palette
    - Video Resolution
- Performance Testing - Ask yourself if the software meets the performance requirements under real-world conditions—with the expected number of users, or connections, or transactions per second. Is it scalable?
- Usability Testing - It is performed with real users , under real environmental conditions.

Tip 64: Use Saboteurs to Test Your Testing

How to test :

- Regression testing - A regression test compares the output of the current test with previous (or known) values. We can ensure that bugs we fixed today didn't break things that were working yesterday.
- Test data - Where do we get the data to run all these tests? There are only two kinds of data: real-world data and synthetic data. We actually need to use both.
- Testing thoroughly
- Testing the tests - Because we can't write perfect software, it follows that we can't write perfect test software either. We need to test the tests. Think of our set of test suites as an elaborate security system, designed to sound the alarm when a bug shows up. How better to test a security system than to try to break in?

If you are really serious about testing, you might want to appoint a project saboteur. The saboteur's role is to take a separate copy of the source tree, introduce bugs on purpose, and verify that the tests will catch them.

Tip 65: Test State Coverage, Not Code Coverage

These coverage analysis tools watch your code during testing and keep track of which lines of code have been executed and which haven't. These tools help give you a general feel for how comprehensive your testing is, but don't expect to see 100% coverage. Even if you do happen to hit every line of code, that's not the whole picture. What is important is the number of states that your program may have. States are not equivalent to lines of code.

Tip 66: Find Bugs Once

If a bug slips through the net of existing tests, you need to add a new test to trap it next time.

Tip 67: Treat English as Just Another Programming Language

Pragmatic Programmers embrace documentation as an integral part of the overall development process. Writing documentation can be made easier by not duplicating effort or wasting time, and by keeping documentation close at hand—in the code itself, if possible.

Tip 68: Build Documentation In, Don't Bolt It On

Documentation and code are different views of the same underlying model, but the view is all that should be different.

Tip 69: Gently Exceed Your Users' Expectations

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

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.

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

Tip 70: Sign Your Work

Pragmatic Programmers don't shirk from responsibility. Instead, we rejoice in accepting challenges and in making our expertise well known. If we are responsible for a design, or a piece of code, we do a job we can be proud of.

People should see your name on a piece of code and expect it to be solid, well written, tested, and documented. A really professional job. Written by a real professional.
A Pragmatic Programmer.