This is my summary of the The Pragmatic Programmer, by Andrew Hunt and David Thomas. I use it while learning and as quick reference. It is not intended to be an standalone substitution of the book so if you really want to learn the concepts here presented, buy and read the book and use this repository as a reference and guide.

If you are the publisher and think this repository should not be public, just write me an email at hugomatilla [at] gmail [dot] com and I will make it private.

Contributions: Issues, comments and pull requests are super welcome 😀 There is a Quick Reference at the end.

Table of Contents

Chapter 1. A Pragmatic Philosophy

Tip 1: Care About Your Craft

Why spend your life developing software unless you care about doing it well?

Tip 2: Think! About Your Work

Turn off the autopilot and take control. Constantly critique and appraise your work.

1.-The Cat Ate My Source Code

Tip 3: 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.

2.-Software Entropy

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.

Tip 4: Don't Live with Broken Windows

Don't mess up the carpet when fixing the broken window.

3.-Stone Soup and Boiled Frogs

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.

Tip 5: 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.

Tip 6: Remember the Big Picture

4.-Good enough soup

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

Tip 7: Make Quality a Requirements Issue

Great software today is often preferable to perfect software tomorrow. **Know When to Stop**

5.-Your Knowledge Portfolio

An investment in knowledge always pays the best interest.

- 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

Building Your Portfolio

- Invest regularly
- Diversify
- Manage risk
- Buy low, sell High
- Review and rebalance

Tip 8: Invest Regularly in Your Knowledge Portfolio

Goals

- 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.

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

Tip 9: Critically Analyze What You Read and Hear

6.-Communicate

- Know what you want to say. Plan what you want to say. Write an outline.
- Know 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: 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.

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

Chapter 2. A Pragmatic Approach

7.-The Evils of Duplication

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.

Tip 11: DRY—Don't Repeat Yourself

Types of duplication:

- **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

8.-Orthogonality

Two or more things are orthogonal if changes in one do not affect any of the others. Also called *cohesion*. Write "shy" code.

Tip 13: Eliminate Effects Between Unrelated Things

Benefits:

- 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
- Project Teams: Functionality is divided
- Design: Easier to design a complete project through its components
- Toolkits and Libraries: Choose wisely to keep orthogonality
- Coding: In order to keep orthogonality when adding code do:
    - Keep your code decoupled
    - Avoid global data
    - Avoid similar functions
- Testing: Orthogonal systems are easier to test.
- Documentation: Also gain quality

9.-Reversibility

Be prepared for changes.

Tip 14: There are no Final Decisions.

10-Tracer Bullets

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.

Tip 15: 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 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.

11.-Prototypes and Post-it Notes

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

Prototype anything that:

- carries risk
- hasn't been tried before
- is absolutely critical to the final system
- is unproven
- is experimental
- is doubtful

Samples:

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

Tip 16: Prototype to Learn

Avoid details:

- 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?

Never deploy the prototype

12.-Domain Languages

Tip 17: Program Close to the Problem domain

13.-Estimating

Tip 18: Estimate to Avoid Surprises

How Accurate Is Accurate Enough?

First: Do they need high accuracy, or are they looking for a ballpark figure?

Second: Scale time estimates properly

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

Where Do Estimates Come From?

Ask someone who's been in a similar situation in the past.

- 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

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

The refinement and confidence in the schedule gets better and better each iteration

Tip 19: Iterate the Schedule with the Code

What to Say When Asked for an Estimate

"I'll get back to you."

Challenges

Start keeping a log of your estimates. For each, track how accurate you turned out to be. If your error was greater than 50%, try to find out where your estimate went wrong.

Chapter 3. The Basic Tools

Tip 20: Keep Knowledge in plain text

14.-The Power of Plain Text

Drawbacks

- more space
- computationally more expensive

The Power of Text

- Insurance against obsolescence: you will always have a chance to be able to use text.
- Leverage: Virtually every tool in the computing can operate on plain text.
- Easier testing

15.-Shell Games

Tip 21: 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.

16.-Power Editing

Tip 22: Use a Single Editor Well

Editor "must" 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)

17.-Source Code Control

Tip 23: Always Use Source Code Control

18.-Debugging

Tip 24: Fix the Problem, Not the Blame

Tip 25: Don't Panic

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.

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

Know 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 26: "select" Isn't Broken

The Element of Surprise

Tip 27: Don't Assume It—Prove It

Debugging Checklist

- Is the problem being reported a direct result of the underlying bug, or merely asymptom?
- 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?

19.-Text Manipulation

Tip 28: Learn a Text Manipulation Language

20.-Code Generators

Tip 29: Write Code That Writes Code Two main 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.
- **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.

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.

Chapter 4. A Pragmatic Paranoia

Tip 30: 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.

21.-Design by Contract

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

- Preconditions
- Postconditions
- Invariants

Tip 31: 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)

DBC enforce Crashing Early

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.

22.-Dead Programs Tell No Lies

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

Tip 32: 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.

23.-Assertive Programming

Tip 33: 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.

24.-When to Use Exceptions

Tip 34: 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

25.-How to Balance Resources

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.

Tip 35: Finish What You Start

Nest Allocations

- 1.-Deallocate resources in the opposite order to that in which you allocate them
- 2.-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.

Chapter 5. Bend or Break

26.-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.

```
book.pages().last().text().

// Instead, we're supposed to go with:

book.textOfLastPage()
```

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

The Law of Demeter for Functions

The Law of Demeter for functions states that any method of an object should call only methods belonging to:

```
class Demeter {
    private A a;
    void m(B b) {
        a.hello();                          //itself
        b.hello();                          //any parameters that were passed to
the method
        new Z().hello();                    // any object it created
        Singleton.INSTANCE.hello();         // any directly held component
    }
}
```

Tip 36: Minimize Coupling Between Modules

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.

27.-Metaprogramming

"Out with the details!" Get them out of the code. While we're at it, we can make our code highly configurable and "soft"—that is, easily adaptable to changes.

Dynamic Configuration

Tip 37: Configure, Don't Integrate

Metadata-Driven Applications

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*

Tip 38: Put Abstractions in Code Details in Metadata

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.

28.- 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

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

Tip 39: Analyze Workflow to Improve Concurrency

Architecture

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 40: Design Using Services

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.

Cleaner Interfaces

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

Tip 41: Always Design for Concurrency

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.

29.-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.

Tip 42: Separate Views from Models

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.

30.-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.

Blackboard Implementations

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.

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.

Organizing Your Blackboard by partitioning it when working on large cases.

Tip 43: Use Blackboards to Coordinate Workflow

Chapter 6. While you are coding

31.-Program by Coincidence

We should avoid programming by coincidence—relying on luck and accidental successes— in favor of programming deliberately. **Tip 44: Don't Program by Coincidence**

How to 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. 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

32.-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)

```
bool IsFirstElementNull(IList<string> elements)
{
    return elements[0] == null;
}
```

- **O(lg(n))**: Logarithmic (binary search) lg(n) = lg2(n)

```
Int BinarySearch(list, target)
{
   lo = 1, hi = size(list)
   while (lo <= hi){
      mid = lo + (hi-lo)/2
      if (list[mid] == target) return mid
      else if (list[mid] < target) lo = mid+1
      else hi = mid-1
   }
}
```

- **O(n)**: Linear: Sequential search

```
bool ContainsValue(IList<string> elements, string value)
{
    foreach (var element in elements)
    {
        if (element == value) return true;
    }

    return false;
}
```

- **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)

```
bool ContainsDuplicates(IList<string> elements)
{
    for (var outer = 0; outer < elements.Count; outer++)
    {
        for (var inner = 0; inner < elements.Count; inner++)
        {
            // Don't compare with self
            if (outer == inner) continue;

            if (elements[outer] == elements[inner]) return true;
        }
    }

    return false;
```

```
    }
```

- **O(n^3)**: Cubic (multiplication of 2 n x n matrices)
- **O(C^n)**: Exponential (travelling salesman problem, set partitioning)

```
int Fibonacci(int number)
{
    if (number <= 1) return number;

    return Fibonacci(number - 2) + Fibonacci(number - 1);
}
```

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)

Tip 45: Estimate the Order of Your Algorithms

Tip 46: Test Your Estimates

Best Isn't Always Best

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.

33.-Refactoring

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

When Should You Refactor?

- Duplication. You've discovered a violation of the DRY principle (The Evils of Duplication).
- Nonorthogonal design. You've discovered some code or design that could be made more orthogonal (Orthogonality).
- 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.

Tip 47: Refactor Early, Refactor Often

How Do You Refactor?

- 1. Don't try to refactor and add functionality at the same time.
- 2. Make sure you have good tests before you begin refactoring.
- 3. Take short, deliberate steps.

34.-Code That's Easy to Test

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

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.

Tip 48: 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.

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

Tip 49: Test Your Software, or Your Users Will

35.-Evil Wizards

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.

Tip 50: Don't Use Wizard Code You Don't Understand

Chapter 7. Before the project

36.-The Requirements Pit

Perfection is achieved, not when there is nothing left to add, but when there is nothing left to take away....

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

Digging for Requirements

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.

Tip 52: 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

Tip 53: Abstractions Live Longer than Details

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.

Maintain a 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.

Tip 54: Use a Project Glossary

Get the Word Out

Publishing project documents to internal Web sites for easy access by all participants.

37.-Solving Impossible Puzzles

Degrees of Freedom

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.

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

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?

38.-Not Until You're Ready

If you sit down to start typing and there's some nagging doubt in your mind, heed it.

Tip 56: Listen to Nagging Doubts—Start When You're Ready

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.

39.-The Specification Trap

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.

Tip 57: Some Things Are Better Done than Described

40.-Circles and Arrows

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

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.

Tip 59: Expensive Tools Do Not Produce Better Designs

Chapter 8. Pragmatic Projects

41.-Pragmatic Teams

Pragmatic techniques that help an individual can work for 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.

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.

Tip 60: 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

42.-Ubiquitous Automation

All on Automatic

Tip 61: 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

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.

43.-Ruthless testing

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 62: 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".

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

3 Main aspects:

1.-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.

2.-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 specialized 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.

Tip 64: Use Saboteurs to Test Your Testing

- Testing thoroughly:

Tip 65: Test State Coverage, Not Code Coverage

3.-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.

Tip 66: Find Bugs Once

44.-It's All Writing

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

Tip 67: Treat English as Just Another Programming Language

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

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.

45.-Great Expectations

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

Tip 69: Gently Exceed Your Users' 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

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."

Tip 70: Sign Your Work

Quick Reference

Tips

Tip 1: Care About Your Craft Why spend your life developing software unless you care about doing it well?

Tip 2: Think! About Your Work Turn off the autopilot and take control. Constantly critique and appraise your work.

Tip 3: 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.

Tip 4: Don't Live with Broken Windows Fix bad designs, wrong decisions, and poor code when you see them.

Tip 5: Be a Catalyst for Change You can't force change on people. Instead, show them how the future might be and help them participate in creating it.

Tip 6: Remember the Big Picture Don't get so engrossed in the details that you forget to check what's happening around you.

Tip 7: Make Quality a Requirements Issue Involve your users in determining the project's real quality requirements.

Tip 8: Invest Regularly in Your Knowledge Portfolio Make learning a habit.

Tip 9: Critically Analyze What You Read and Hear Don't be swayed by vendors, media hype, or dogma. Analyze information in terms of you and your project.

Tip 10: It's Both What You Say and the Way You Say It There's no point in having great ideas if you don't communicate them effectively.

Tip 11: DRY – Don't Repeat Yourself Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Tip 12: Make It Easy to Reuse If it's easy to reuse, people will. Create an environment that supports reuse.

Tip 13: Eliminate Effects Between Unrelated Things Design components that are self-contained, independent, and have a single, well-defined purpose.

Tip 14: There Are No Final Decisions No decision is cast in stone. Instead, consider each as being written in the sand at the beach, and plan for change.

Tip 15: Use Tracer Bullets to Find the Target Tracer bullets let you home in on your target by trying things and seeing how close they land.

Tip 16: Prototype to Learn Prototyping is a learning experience. Its value lies not in the code you produce, but in the lessons you learn.

Tip 17: Program Close to the Problem Domain Design and code in your user's language.

Tip 18: Estimate to Avoid Surprises Estimate before you start. You'll spot potential problems up front.

Tip 19: Iterate the Schedule with the Code Use experience you gain as you implement to refine the project time scales.

Tip 20: Keep Knowledge in Plain Text Plain text won't become obsolete. It helps leverage your work and simplifies debugging and testing.

Tip 21: Use the Power of Command Shells Use the shell when graphical user interfaces don't cut it.

Tip 22: Use a Single Editor Well The editor should be an extension of your hand; make sure your editor is configurable, extensible, and programmable.

Tip 23: Always Use Source Code Control Source code control is a time machine for your work – you can go back.

Tip 24: Fix the Problem, Not the Blame It doesn't really matter whether the bug is your fault or someone else's – it is still your problem, and it still needs to be fixed.

Tip 25: Don't Panic When Debugging Take a deep breath and THINK! about what could be causing the bug.

Tip 26: "select" Isn't Broken. It is rare to find a bug in the OS or the compiler, or even a third-party product or library. The bug is most likely in the application.

Tip 27: Don't Assume It – Prove It Prove your assumptions in the actual environment – with real data and boundary conditions.

Tip 28: Learn a Text Manipulation Language. You spend a large part of each day working with text. Why not have the computer do some of it for you?

Tip 29: Write Code That Writes Code Code generators increase your productivity and help avoid duplication.

Tip 30: You Can't Write Perfect Software Software can't be perfect. Protect your code and users from the inevitable errors.

Tip 31: Design with Contracts Use contracts to document and verify that code does no more and no less than it claims to do.

Tip 32: Crash Early A dead program normally does a lot less damage than a crippled one.

Tip 33: Use Assertions to Prevent the Impossible Assertions validate your assumptions. Use them to protect your code from an uncertain world.

Tip 34: Use Exceptions for Exceptional Problems Exceptions can suffer from all the readability and maintainability problems of classic spaghetti code. Reserve exceptions for exceptional things.

Tip 35: Finish What You Start Where possible, the routine or object that allocates a resource should be responsible for deallocating it.

Tip 36: Minimize Coupling Between Modules Avoid coupling by writing "shy" code and applying the Law of Demeter.

Tip 37: Configure, Don't Integrate Implement technology choices for an application as configuration options, not through integration or engineering.

Tip 38: Put Abstractions in Code, Details in Metadata Program for the general case, and put the specifics outside the compiled code base.

Tip 39: Analyze Workflow to Improve Concurrency Exploit concurrency in your user's workflow.

Tip 40: Design Using Services Design in terms of services – independent, concurrent objects behind well-defined, consistent interfaces.

Tip 41: Always Design for Concurrency Allow for concurrency, and you'll design cleaner interfaces with fewer assumptions.

Tip 42: Separate Views from Models Gain flexibility at low cost by designing your application in terms of models and views.

Tip 43: Use Blackboards to Coordinate Workflow Use blackboards to coordinate disparate facts and agents, while maintaining independence and isolation among participants.

Tip 44: Don't Program by Coincidence Rely only on reliable things. Beware of accidental complexity, and don't confuse a happy coincidence with a purposeful plan.

Tip 45: Estimate the Order of Your Algorithms Get a feel for how long things are likely to take before you write code.

Tip 46: Test Your Estimates Mathematical analysis of algorithms doesn't tell you everything. Try timing your code in its target environment.

Tip 47: Refactor Early, Refactor Often Just as you might weed and rearrange a garden, rewrite, rework, and re-architect code when it needs it. Fix the root of the problem.

Tip 48: Design to Test Start thinking about testing before you write a line of code.

Tip 49: Test Your Software, or Your Users Will Test ruthlessly. Don't make your users find bugs for you.

Tip 50: Don't Use Wizard Code You Don't Understand Wizards can generate reams of code. Make sure you understand all of it before you incorporate it into your project.

Tip 51: Don't Gather Requirements – Dig for Them Requirements rarely lie on the surface. They're buried deep beneath layers of assumptions, misconceptions, and politics.

Tip 52: Work With a User to Think Like a User It's the best way to gain insight into how the system will really be used.

Tip 53: Abstractions Live Longer than Details Invest in the abstraction, not the implementation. Abstractions can survive the barrage of changes from different implementations and new technologies.

Tip 54: Use a Project Glossary Create and maintain a single source of all the specific terms and vocabulary for a project.

Tip 55: Don't Think Outside the Box – Find the Box When faced with an impossible problem, identify the real constraints. Ask yourself: "Does it have to be done this way? Does it have to be done at all?"

Tip 56: Start When You're Ready. You've been building experience all your life. Don't ignore niggling doubts.

Tip 57: Some Things Are Better Done than Described Don't fall into the specification spiral – at some point you need to start coding.

Tip 58: Don't Be a Slave to Formal Methods. Don't blindly adopt any technique without putting it into the context of your development practices and capabilities.

Tip 59: Costly Tools Don't Produce Better Designs Beware of vendor hype, industry dogma, and the aura of the price tag. Judge tools on their merits.

Tip 60: Organize Teams Around Functionality Don't separate designers from coders, testers from data modelers. Build teams the way you build code.

Tip 61: Don't Use Manual Procedures A shell script or batch file will execute the same instructions, in the same order, time after time.

Tip 62: Test Early. Test Often. Test Automatically Tests that run with every build are much more effective than test plans that sit on a shelf.

Tip 63: Coding Ain't Done 'Til All the Tests Run 'Nuff said.

Tip 64: Use Saboteurs to Test Your Testing Introduce bugs on purpose in a separate copy of the source to verify that testing will catch them.

Tip 65: Test State Coverage, Not Code Coverage Identify and test significant program states. Just testing lines of code isn't enough.

Tip 66: Find Bugs Once Once a human tester finds a bug, it should be the last time a human tester finds that bug. Automatic tests should check for it from then on.

Tip 67: English is Just a Programming Language Write documents as you would write code: honor the DRY principle, use metadata, MVC, automatic generation, and so on.

Tip 68: Build Documentation In, Don't Bolt It On Documentation created separately from code is less likely to be correct and up to date.

Tip 69: Gently Exceed Your Users' Expectations Come to understand your users' expectations, then deliver just that little bit more.

Tip 70: Sign Your Work Craftsmen of an earlier age were proud to sign their work. You should be, too.

CheckList

Languages To Learn

Tired of C, C++, and Java? Try the following languages. Each of these languages has different capabilities and a different "flavor." Try a small project at home using one or more of them.

- CLOS
- Dylan
- Eiffel
- Objective C
- Prolog
- Smalltalk
- TOM

The WISDOM Acrostic

- **W**hat do you want them to learn?
- What **i**s their interest in what you've got to say?
- How **s**ophisticated are they?
- How much **d**etail do they want?
- Whom do you want to **o**wn the information?
- How can you **m**otivate them to listen to you?

How to Maintain Orthogonality

- Design independent, well-defined components.
- Keep your code decoupled.
- Avoid global data.
- Refactor similar functions.

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

Architectural Questions

- Are responsibilities well defined?
- Are the collaborations well defined?
- Is coupling minimized?
- Can you identify potential duplication?
- Are interface definitions and constraints acceptable?
- Can modules access needed data – when needed?

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?

Law of Demeter for Functions

An object's method should call only methods belonging to:

- Itself
- Any parameters passed in
- Objects it creates
- Component objects

How to Program Deliberately

- Stay aware of what you're doing.
- Don't code blindfolded.
- Proceed from a plan.
- Rely only on reliable things.

- Document your assumptions.
- Test assumptions as well as code.
- Prioritize your effort.
- Don't be a slave to history.

When to Refactor

- You discover a violation of the DRY principle.
- You find things that could be more orthogonal.
- Your knowledge improves.
- The requirements evolve.
- You need to improve performance.

Cutting the Gordian Knot

When solving *impossible* problems, ask yourself:

- Is there an easier way?
- Am I solving the right problem?
- Why is this a problem?
- What makes it hard?
- Do I have to do it this way?
- Does it have to be done at all?

Aspects of Testing

- Unit testing
- Integration testing
- Validation and verification
- Resource exhaustion, errors, and recovery
- Performance testing
- Usability testing
- Testing the tests themselves