Pragmatic Programmers Summary

Learning is a continuous and ongoing process.

They think beyond the immediate problem, always trying to place it in its larger context, always trying to be aware of the bigger picture.

The greatest of all weaknesses is the fear of appearing weak.

One of the cornerstones of the pragmatic philosophy is the idea of taking responsibility for yourself and your actions in terms of your career advancement, your project, and your day-to-day work.

We can be proud of our abilities, but we must be honest about our shortcomings—our ignorance as well as our mistakes.

Remember the Big Picture.

Don't put all your technical eggs in one basket.

Don't forget the human side of the equation. (Read book invest yourself)

If you can't find the answer yourself, find out who can. Don't let it rest.

Communicate is important.

Be a Listener.

Choose a style.

DRY (Don't repeat yourself)

Make It Easy to Reuse

If it isn't easy, people won't do it. And if you fail to reuse, you risk duplicating knowledge.

An orthogonal approach reduces the risks inherent in any development.

Avoid global data.

The Singleton pattern in Design Patterns is a way of ensuring that there is only one instance of an object of a particular class.

Be careful with singletons—they can also lead to unnecessary linkage.

Duplicate code is a symptom of structural problems. Have a look at the Strategy pattern in Design Patterns for a better implementation.

An orthogonally designed and implemented system is easier to test. There is always more than one way to implement something.

There Are No Final Decisions Code That Glows in the Dark.

We build software prototypes in the same fashion, and for the same

reasons—to analyze and expose risk, and to offer chances for correction

at a greatly reduced cost.

Architecture

New functionality in an existing system.

Structure or contents of external data.

Third-party tools or components.

Performance issues.

User interface design.

Estimate to Avoid Surprises.

All estimates are based on models of the problem.

The normal rules of estimating can break down in the face of the complexities and vagaries of a sizable application development. We find that often the only way to determine the timetable for a project is by gaining experience on that same project. This needn't be a paradox if you practice incremental development, repeating the following steps.

-Check requirements-

-Analyze risk-

-Design, implement, integrate-

-Validate with the users-

Basic Tools: Shell or GUI, Editor, Source Control, Debugging, Code Generators

Choose One Editor : know it thoroughly, and use it for all editing tasks. If you use a single editor (or set of keybindings) across all text editing activities, you don’t have to stop and think to accomplish text manipulation: the necessary keystrokes will be a reflex.

Source Control System: But a source code control system does far more than undo mistakes.

A good SCCS will let you track changes, answering questions such as: Who made changes in this line of code? What’s the difference between the current version and last week’s? How many lines of code did we change in this release? Which files get changed most often? This kind of information is invaluable for bug-tracking, audit, performance, and quality purposes

Always Use Source Code Control

Fix the problem, Not blame.

You Can’t Write Perfect Software. Because perfect software doesn’t exist. No one in the brief history of computing has ever written a piece of perfect software. Don't waste your time following the impossible dream.

Inheritance and polymorphism are the cornerstones of object-oriented

languages and an area where contracts can really shine.

One of the benefits of detecting problems as soon as you can is that you can crash earlier. And many times, crashing your program is the best thing you can do.

"There is a luxury in self-reproach. When we blame ourselves we feel no one else has a right to blame us" -Oscar Wilde

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.

An error handler is a routine that is called when an error is detected.

Finish What You Start

Life doesn’t stand still.

Neither can the code that we write. 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. Otherwise we may find our code quickly becoming outdated, or too brittle to fix, and may ultimately be left behind in the mad dash toward the future.

Traversing relationships between objects directly can quickly lead to a combinatorial explosion of dependency relationships.

"No amount of genius can overcome a preoccupation with detail"

Configure, Don't Integrate

Our goal is to think declaratively (specifying what is to be

done, not how) and create highly dynamic and adaptable programs. We do this by adopting a general rule: program for the general case, and put the specifics somewhere else—outside the compiled code base.

Operating systems already configure themselves to hardware as they boot, and Web browsers update themselves with new components automatically. Well, out here in the real world, species that don't adapt die.

We don't usually approach programming with either of these aspects in mind. When people first sit down to design an architecture or write a program, things tend to be linear. That's the way most people think do this and then always do that. But thinking this way leads to temporal coupling: coupling in time. Method A must always be called before method B; only one report can be run at a time; you must wait for the screen to redraw before the button click is received. Tick must happen before tock

On many projects, we need to model and analyze the users' workflows as part of requirements analysis. We'd like to find out what can happen at the same time, and what must happen in a strict order. One way to do this is to capture their description of workflow using a notation such as the UML activity diagram.

Analyze Workflow to Improve Concurrency

Always Design for Concurrency

Early on we are taught not to write a program as a single big chunk, but that we should "divide and conquer" and separate a program into modules. Each module has its own responsibilities; in fact, a good definition of a module (or class) is that it has a single, well-defined responsibility. But once you separate a program into different modules based on responsibility, you have a new problem. At runtime, how do the objectstalk to each other? How do you manage the logical dependencies between them? That is, how do you synchronize changes in state (or updates to data values) in these different objects? It needs to be done in a clean, flexible manner we don’t want them to know too much about each other.

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.

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.

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.

Use Blackboards to Coordinate Workflow

Program by Coincidence

We should avoid programming by coincidence—relying on luck and accidental successes— in favor of programming deliberately. 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

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)

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

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.

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.

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 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:

Whether the code meet the contract

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.

Writing Unit Tests

By making the test code readily accessible, you are providing developers who may use your code with two invaluable resources:

Examples of how to use all the functionality of your module

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

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.

Don't Use Wizard Code You Don't Understand

Before the project

The Requirements Pit

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

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.

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

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.

Use a Project Glossary

Get the Word Out

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

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.

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?

Not Until You're Ready

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

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.

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.

Some Things Are Better Done than Described

Circles and Arrows

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.

Expensive Tools Do Not Produce Better Designs

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.

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

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.

Check out the source code from the repository

Build the project from scratch (marked with the version number).

Create a distributable image

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

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

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".

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.

Use Saboteurs to Test Your Testing

Testing thoroughly:

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.

Find Bugs Once

It's All Writing

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

Treat English as Just Another Programming Language

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.

Great Expectations

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

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."

Sign Your Work

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.