1. Care about your craft

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

1. Think! About Your Work

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

1. 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.

1. Don’t Live with Broken Windows

Fix bad designs, wrong decisions, and poor code when you see them.

1. Be a Catalyst for Change

You can’t force on people. Instead, show them how the future might be and help them participate in creating it.

1. Remember the Big Picture

Don’t get engrossed in the details that you forget to check what’s happening around you.

1. Make Quality a Requirements Issue

Involve your users in determining the project’s real requirements.

1. Invest Regularly in your Knowledge Portfolio

Make learning a habit.

1. 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.

1. It’s both What you say and the way you say it

There is no point in having great ideas if you don’t communicate them effectively.

1. DRY – Don’t Repeat Yourself

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

1. Make it Easy to Reuse

If it’s easy to reuse, people will. Create an environment that supports reuse.

1. Eliminate Effects Between Unrelated Things

Design components that are self-contained, independent, and have a single, well-defined purpose.

1. 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.

1. 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.

1. Prototype to Learn

Prototyping is a learning experience, its value lies not in the code you produce, but in the lessons you learn.

1. Program Close to the Problem Domain

Design and code in your user’s language.

1. Estimate to Avoid Surprises

Estimate before you start. You’ll spot potential problems up front.

1. Iterate the Schedule with the code

Use experience you gain as you implement to refine the project time scales.

1. Keep Knowledge in Plain Text

Plain text won’t become obsolete. It helps leverage your work and simplifies debugging and testing.

1. Use the Power of command Shells

Use the shell when graphical user interfaces don’t cut it.

1. Use a Single Editor Well

The Editor should be an extension of your hand; make sure your editor is configurable, extensible, and programmable.

1. Always Use Source Code Control

Source code control is a time machine for your work- you can go back.

1. 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.

1. Don’t Panic When Debugging

Take a deep breath and THINK! About what could be causing the bug?

1. “select” isn’t Broken

It is rare to find a bug in the OS or the complier, or even a third-party product or library. The bug is most likely in the application.

1. Don’t Assume It-Prove it

Prove your assumptions in the actual environment-with the real data and boundary conditions.

1. Learn a Text Manipulation Language

You spend a large part of each day working with text. Why not have the computers do some of it for you?

1. Write Code That Writes Code

Code generates increase your productivity and help avoid duplication.

1. You Can’t Write Perfect Software

Software can’t be perfect. Protect your code and users from the inevitable errors.

1. Design with Contracts

Use contracts to document and verify that code does no more and no less than it claims to do.

1. Crash Early

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

1. Use Assertions to Prevent the Impossible

Assertions validate your assumptions. Use them to protect your code from an uncertain world.

1. Use Exceptions for Exceptional Problems

Exceptions can suffer from all the readability and maintainability problems of classic spaghetti code. Reserve exceptions for exceptional things.

1. Finish What You Start

Where possible, the routine or object that allocates a resource should be responsible for reallocating it.

1. Minimize Coupling Between Modules

Avoid coupling by writing “shy “code and applying the Law of Demeter.

1. Configure, Don’t Integrate

Implement technology choices for an application as configuration options, not through integration or engineering.

1. Put Abstractions in Code, Details in Metadata

Program for the general case, and put the specifics outside the complied code base.

1. Analyze Workflow to Improve Concurrency

Exploit concurrency in your user’s workflow.

1. Design Using Services

Design in terms of services---independent, concurrent objects behind well-defined, consistent interfaces.

1. Always Design for Concurrency

Allow for concurrency, and you’ll design cleaner interfaces with fewer assumptions.

1. Separate Views from Models

Gain flexibility at low cost by designing your application in terms of models and views.

1. Use Blackboards to Coordinate Workflow

Use blackboards to coordinate disparate facts and agents. While maintaining independence and isolation among participants.

1. 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.

1. Estimate the Order of Your Algorithms

Get a feel for how long things are likely to take before you write a code.

1. Test Your Estimates

Mathematical analysis of algorithms doesn’t tell you everything. Try timing your code in its target environment.

1. Refractor Early, Refractor Often

Just as u might weed and rearrange a garden, rewrite, rework, and re-architect code when it needs it. Fix the root of the problem.

1. Design to Test

Start thinking about testing before you write a line of code.

1. Test your Software, or Your Users Will

Test Ruthlessly, Don’t make your users find bugs for you.

1. 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.

1. Don’t Gather Requirements-Dig for Them

Requirements rarely lie on the surface. They’re buried deep beneath layers of assumptions, misconceptions, and politics.

1. 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.

1. Abstractions Live Longer than Details

Invest in the abstraction, not the implementation. Abstractions can survive the barrage of changes from different implementations and the new technologies.

1. Use a Project Glossary

Create and maintain a single source of all the specific terms and vocabulary for a project.

1. 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?”

1. Start when you’re Ready

You’ve been building experience all your life. Don’t ignore niggling doubts.

1. Some Things Are Better Done than Described

Don’t fall into the specification spiral—at some point you need to start coding.

1. 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.

1. 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.

1. Organize Teams Around Functionality

Don’t separate designers from coders, testers from data modelers. Build teams the way you build code.

1. Don’t use Manual Procedures

A shell script or batch file will execute the same instructions, in the same order, time after time.

1. Test Early, Test Often, Test Automatically

Tests that run with every build are much more effective than test plans that sit on a shelf.

1. Coding Isn’t Done till all the Tests Run

‘Nuff said.

1. Use Saboteurs to Test Your Testing

Introduce bugs on purpose in a separate copy of the source to verify that testing will catch them.

1. Test State Coverage, Not Code Coverage

Identify and test significant program states. Just testing lines of code isn’t enough.

1. 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.

1. 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.

1. Build Documentation In, Don’t Bolt It On

Documentation created separately from code is less likely to be correct and up to date.

1. Gently Exceed Your User’s Expectations

Come to understand your user’s expectations, and then deliver just that little bit more.

1. Sign Your Work

Craftsmen of an earlier age were proud to sign their work. You should be, too.