**The Pragmatic Programmer Quick Reference Guide**

For more information about The Pragmatic Programmers LLC, source code for the examples, up-to-date pointers to Web resources, and an online bibiography, visit us at [www.pragmaticprogrammer.com](http://www.pragmaticprogrammer.com/)

1. **Care About Your Craft**  
   Why spend your life developing software unless you care about doing it well?
2. **Think! About Your Work**  
   Turn off the autopilot and take control. Constantly critique and appraise your work.
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.
4. **Don't Live with Broken Windows**  
   Fix bad designs, wrong decisions, and poor code when you see them.
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.
6. **Remember the Big Picture**  
   Don't get so engrossed in the details that you forget to check what's happening around you.
7. **Make Quality a Requirements Issue**  
   Involve your users in determining the project's real quality requirements.
8. **Invest Regularly in Your Knowledge Portfolio**  
   Make learning a habit.
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.
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.
11. **DRY–Don't Repeat Yourself**  
    Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.
12. **Make It Easy to Reuse**  
    If it's easy to reuse, people will. Create an environment that supports reuse.
13. **Eliminate Effects Between Unrelated Things**  
    Design components that are self-contained. independent, and have a single, well-defined purpose.
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.
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.
16. **Prototype to Learn**  
    Prototyping is a learning experience. Its value lies not in the code you produce, but in the lessons you learn.
17. **Program Close to the Problem Domain**  
    Design and code in your user's language.
18. **Estimate to Avoid Surprises**  
    Estimate before you start. You'll spot potential problems up front.
19. **Iterate the Schedule with the Code**  
    Use experience you gain as you implement to refine the project time scales.
20. **Keep Knowledge in Plain Text**  
    Plain text won't become obsolete. It helps leverage your work and simplifies debugging and testing.
21. **Use the Power of Command Shells**  
    Use the shell when graphical user interfaces don't cut it.
22. **Use a Single Editor Well**  
    The editor should be an extension of your hand; make sure your editor is configurable, extensible, and programmable.
23. **Always Use Source Code Control**  
    Source code control is a time machine for your work—you can go back.
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.
25. **Don't Panic When Debugging**  
    Take a deep breath and THINK! about what could be causing the bug.
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.
27. **Don't Assume It—Prove It**  
    Prove your assumptions in the actual environment-- with real data and boundary conditions.
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?
29. **Write Code That Writes Code**  
    Code generators increase your productivity and help avoid duplication.
30. **You Can't Write Perfect Software**  
    Software can't be perfect. Protect your code and users from the inevitable errors.
31. **Design with Contracts**  
    Use contracts to document and verify that code does no more and no less than it claims to do.
32. **Crash Early**  
    A dead program normally does a lot less damage than a crippled one.
33. **Use Assertions to Prevent the Impossible**  
    Assertions validate your assumptions. Use them to protect your code from an uncertain world.
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.
35. **Finish What You Start**  
    Where possible, the routine or object that allocates a resource should be responsible for deallocating it.
36. **Minimize Coupling Between Modules**  
    Avoid coupling by writing "shy" code and applying the Law of Demeter.
37. **Configure, Don't Integrate**  
    Implement technology choices for an application as configuration options, not through integration or engineering.
38. **Put Abstractions in Code, Details in Metadata**  
    Program for the general case, and put the specifics outside the compiled code base.
39. **Analyze Workflow to Improve Concurrency**  
    Exploit concurrency in your user's workflow.
40. **Design Using Services**  
    Design in terms of services—independent, concurrent objects behind well-defined, consistent interfaces.
41. **Always Design for Concurrency**  
    Allow for concurrency, and you'll design cleaner interfaces with fewer assumptions.
42. **Separate Views from Models**  
    Gain flexibility at low cost by designing your application in terms of models and views.
43. **Use Blackboards to Coordinate Workflow**  
    Use blackboards to coordinate disparate facts and agents, while maintaining independence and isolation among participants.
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.
45. **Estimate the Order of Your Algorithms**  
    Get a feel for how long things are likely to take before you write code.
46. **Test Your Estimates**  
    Mathematical analysis of algorithms doesn't tell you everything. Try timing your code in its target environment.
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.
48. **Design to Test**  
    Start thinking about testing before you write a line of code.
49. **Test Your Software, or Your Users Will**  
    Test ruthlessly. Don't make your users find bugs for you.
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.
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.
52. **Workwith a User to Think Like a User**  
    It's the best way to gain insight into how the system will really be used.
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.
54. **Use a Project Glossary**  
    Create and maintain a single source of all the specific terms and vocabulary for a project.
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?"
56. **Start When You're Ready.**  
    You've been building experience all your life. Don't ignore niggling doubts.
57. **Some Things Are Better Done than Described**  
    Don't fall into the specification spiral—at some point you need to start coding.
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.
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.
60. **Organize Teams Around Functionality**  
    Don't separate designers from coders, testers from data modelers. Build teams the way you build code.
61. **Don't Use Manual Procedures**  
    A shell script or batch file will execute the same instructions, in the same order, time after time.
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.
63. **Coding Ain't Done 'Til All the Tests Run**  
    'Nuff said.
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.
65. **Test State Coverage, Not Code Coverage**  
    Identify and test significant program states. Just testing lines of code isn't enough.
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.
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.
68. **Build Documentation In, Don't Bolt It On**  
    Documentation created separately from code is less likely to be correct and up to date.
69. **Gently Exceed Your Users' Expectations**  
    Come to understand your users' expectations, then deliver just that little bit more.
70. **Sign Your Work**  
    Craftsmen of an earlier age were proud to sign their work. You should be, too.

**Languages To Learn**

Tired of C, C++, and Java? Try CLOS, Dylan, Eiffel, Objective C, Prolog, Smalltalk, or TOM. Each of these languages has different capabilities and a different "flavor." Try a small project at home using one or more of them.

**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