The Pragmatic Programmer Quick Reference Guide[1]

Andy Hunt Dave Thomas

Checklists from *The Pragmatic Programmer*, by Andrew Hunt and David Thomas. Visit www.pragmaticprogrammer.com/ppbook.

[1]Converted to HTML by Jeff Atwood (www.codinghorror.com)
Converted to LaTeX by Matthew J. Miller (www.matthewjmiller.net)

Tips

34. **Use Exceptions for Exceptional Problems** 127
    Exceptions can suffer from all the readability and maintainability problems of classic spaghetti code. Reserve exceptions for exceptional things.

35. **Finish What You Start** 129
    Where possible, the routine or object that allocates a resource should be responsible for deallocating it.

36. **Minimize Coupling Between Modules** 140
    Avoid coupling by writing “shy” code and applying the Law of Demeter.

37. **Configure, Don’t Integrate** 144
    Implement technology choices for an application as configuration options, not through integration or engineering.

38. **Put Abstractions in Code, Details in Metadata** 145
    Program for the general case, and put the specifics outside the compiled code base.

39. **Analyze Workflow to Improve Concurrency** 151
    Exploit concurrency in your user’s workflow.

40. **Design Using Services** 154
    Design in terms of services—independent, concurrent objects behind well-defined, consistent interfaces.

41. **Always Design for Concurrency** 156
    Allow for concurrency, and you’ll design cleaner interfaces with fewer assumptions.

42. **Separate Views from Models** 161
    Gain flexibility at low cost by designing your application in terms of models and views.

43. **Use Blackboards to Coordinate Workflow** 169
    Use blackboards to coordinate disparate facts and agents, while maintaining independence and isolation among participants.

44. **Don’t Program by Coincidence** 175
    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** 181
    Get a feel for how long things are likely to take before you write code.

46. **Test Your Estimates** 182
    Mathematical analysis of algorithms doesn’t tell you everything. Try timing your code in its target environment.

47. **Refactor Early, Refactor Often** 186
    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** 192
    Start thinking about testing before you write a line of code.

49. **Test Your Software, or Your Users Will** 197
    Test ruthlessly. Don't make your users find bugs for you.

50. **Don't Use Wizard Code You Don't Understand** 199
    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** 202
    Requirements rarely lie on the surface. They're buried deep beneath layers of assumptions, misconceptions, and politics.

52. **Work with a User to Think Like a User** 204
    It's the best way to gain insight into how the system will really be used.

53. **Abstractions Live Longer than Details** 209
    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** 210
    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** 213
    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.** 215
    You've been building experience all your life. Don't ignore niggling doubts.

57. **Some Things Are Better Done than Described** 218
    Don't fall into the specification spiral—at some point you need to start coding.

58. **Don't Be a Slave to Formal Methods.** 220
    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** 222
    Beware of vendor hype, industry dogma, and the aura of the price tag. Judge tools on their merits.

60. **Organize Teams Around Functionality** 227
    Don't separate designers from coders, testers from data modelers. Build teams the way you build code.

61. **Don't Use Manual Procedures** 231
    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** 237
    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** 238
    'Nuff said.

64. **Use Saboteurs to Test Your Testing** 244
    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** 245
    Identify and test significant program states. Just testing lines of code isn't enough.

66. **Find Bugs Once** 247
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** 248
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** 248
Documentation created separately from code is less likely to be correct and up to date.

69. **Gently Exceed Your Users' Expectations** 255
Come to understand your users' expectations, then deliver just that little bit more.

70. **Sign Your Work** 258
Craftsmen of an earlier age were proud to sign their work. You should be, too.

Checklists

☐ **Languages To Learn** . . . . page 17
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** . . . . page 20
What 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** . . . . page 34

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

☐ **Things to prototype** . . . . page 53

- 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** . . . . page 55

- 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** . . . . page 98

- 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** .... page 141
An object's method should call only methods belonging to:

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

☐ **How to Program Deliberately** .... page 172

- 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** .... page 185

- 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** .... page 212
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** .... page 237

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