0. Introduction
1. Beginner
2. Learn to Debug
3. How to Debug by Splitting the Problem Space
4. How to Remove an Error
5. How to Debug Using a Log
6. How to Understand Performance Problems
7. How to Fix Performance Problems
8. How to Optimize Loops
9. How to Deal with I/O Expense
10. How to Manage Memory
11. How to Deal with Intermittent Bugs
12. How to Learn Design Skills
13. How to Conduct Experiments
14. Why Estimation is Important
15. How to Estimate Programming Time
16. How to Find Out Information
17. How to Utilize People as Information Sources
18. How to Document Wisely
19. How to Work with Poor Code
20. How to Use Source Code Control
21. How to Unit Test
22. Take Breaks when Stumped
23. How to Recognize When to Go Home
24. How to Deal with Difficult People
25. Intermediate
26. How to Stay Motivated
27. How to be Widely Trusted
28. How to Tradeoff Time vs. Space
29. How to Stress Test
30. How to Balance Brevity and Abstraction
31. How to Learn New Skills
32. Learn to Type
33. How to Do Integration Testing
34. Communication Languages
35. Heavy Tools
36. How to analyze data
37. How to Manage Development Time
38. How to Manage Third-Party Software Risks
39. How to Manage Consultants
40. How to Communicate the Right Amount
41. How to Disagree Honestly and Get Away with It
42. How to Tradeoff Quality Against Development Time
43. How to Manage Software System Dependence
44. How to Decide if Software is Too Immature
45. How to Make a Buy vs. Build Decision
46. How to Grow Professionally
47. How to Evaluate Interviewees
48. How to Know When to Apply Fancy Computer Science

49. How to Talk to Non-Engineers
50. Advanced
51. How to Tell the Hard From the Impossible
52. How to Utilize Embedded Languages
53. Choosing Languages
54. How to Fight Schedule Pressure
55. How to Understand the User
56. How to Get a Promotion
57. How to Develop Talent
58. How to Choose What to Work On
59. How to Get the Most From Your Team-mates
60. How to Divide Problems Up
61. How to Handle Boring Tasks
62. How to Gather Support for a Project
63. How to Grow a System
64. How to Communicate Well
65. How to Tell People Things They Don't Want to Hear
66. How to Deal with Managerial Myths
67. How to Deal with Organizational Chaos
68. Appendix A * Bibliography/Websiteography
69. Appendix B * History (As of January 2016)
70. Appendix C * Contributions (As of January 2016)

How to be a Programmer: Community Version

Robert L. Read with Community

Copyright 2002, 2003, 2016 Robert L. Read

Licensed under Creative Commons Attribution-ShareAlike 4.0 International License.

Introduction

To be a good programmer is difficult and noble. The hardest part of making real a collective vision of a software project is dealing with one's coworkers and customers. Writing computer programs is important and takes great intelligence and skill. But it is really child's play compared to everything else that a good programmer must do to make a software system that succeeds for both the customer and myriad colleagues for whom she is partially responsible. In this essay I attempt to summarize as concisely as possible those things that I wish someone had explained to me when I was twenty-one.

This is very subjective and, therefore, this essay is doomed to be personal and somewhat opinionated. I confine myself to problems that a programmer is very likely to have to face in her work. Many of these problems and their solutions are so general to the human condition that I will probably seem preachy. I hope in spite of this that this essay will be useful.

Computer programming is taught in courses. The excellent books: The Pragmatic Programmer [Prag99], Code Complete [CodeC93], Rapid Development [RDev96], and Extreme Programming Explained [XP99] all teach computer programming and the larger issues of being a good programmer. The essays of Paul Graham [PGSite] and Eric Raymond [Hacker] should certainly be read before or along with this article. This essay differs from those excellent works by emphasizing social problems and comprehensively summarizing the entire set of necessary skills as I see them.

In this essay the term boss is used to refer to whomever gives you projects to do. I use the words business, company, and tribe, synonymously except that business connotes moneymaking, company connotes the modern workplace and tribe is generally the people you share loyalty with.

Welcome to the tribe.

Contents

1. Beginner
    - Personal Skills
        - Learn to Debug
        - How to Debug by Splitting the Problem Space
        - How to Remove an Error
        - How to Debug Using a Log
        - How to Understand Performance Problems
        - How to Fix Performance Problems
        - How to Optimize Loops
        - How to Deal with I/O Expense
        - How to Manage Memory
        - How to Deal with Intermittent Bugs
        - How to Learn Design Skills
        - How to Conduct Experiments
    - Team Skills
        - Why Estimation is Important
        - How to Estimate Programming Time
        - How to Find Out Information
        - How to Utilize People as Information Sources
        - How to Document Wisely
        - How to Work with Poor Code
        - How to Use Source Code Control
        - How to Unit Test
        - Take Breaks when Stumped
        - How to Recognize When to Go Home
        - How to Deal with Difficult People

2. Intermediate
    - Personal Skills
        - How to Stay Motivated
        - How to be Widely Trusted
        - How to Tradeoff Time vs. Space
        - How to Stress Test
        - How to Balance Brevity and Abstraction
        - How to Learn New Skills
        - Learn to Type
        - How to Do Integration Testing
        - Communication Languages
        - Heavy Tools
        - How to analyze data
    - Team Skills
        - How to Manage Development Time
        - How to Manage Third-Party Software Risks
        - How to Manage Consultants
        - How to Communicate the Right Amount
        - How to Disagree Honestly and Get Away with It
    - Judgment
        - How to Tradeoff Quality Against Development Time
        - How to Manage Software System Dependence
        - How to Decide if Software is Too Immature
        - How to Make a Buy vs. Build Decision
        - How to Grow Professionally
        - How to Evaluate Interviewees
        - How to Know When to Apply Fancy Computer Science
        - How to Talk to Non-Engineers
3. Advanced
    - Technological Judgment
        - How to Tell the Hard From the Impossible
        - How to Utilize Embedded Languages
        - Choosing Languages
    - Compromising Wisely
        - How to Fight Schedule Pressure
        - How to Understand the User
        - How to Get a Promotion
    - Serving Your Team
        - How to Develop Talent
        - How to Choose What to Work On
        - How to Get the Most From Your Team-mates

- How to Divide Problems Up
- How to Handle Boring Tasks
- How to Gather Support for a Project
- How to Grow a System
- How to Communicate Well
- How to Tell People Things They Don't Want to Hear
- How to Deal with Managerial Myths
- How to Deal with Organizational Chaos

4. Glossary
5. Appendix A - Bibliography/Websiteography
6. Appendix B - History (As of January 2016)
7. Appendix C - Contributions (As of January 2016)

How To Be A Programmer: Community Version by Robert L. Read with Community is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

1. Beginner

- Personal Skills
  - Learn to Debug
  - How to Debug by Splitting the Problem Space
  - How to Remove an Error
  - How to Debug Using a Log
  - How to Understand Performance Problems
  - How to Fix Performance Problems
  - How to Optimize Loops
  - How to Deal with I/O Expense
  - How to Manage Memory
  - How to Deal with Intermittent Bugs
  - How to Learn Design Skills
  - How to Conduct Experiments
- Team Skills
  - Why Estimation is Important
  - How to Estimate Programming Time
  - How to Find Out Information
  - How to Utilize People as Information Sources
  - How to Document Wisely
  - How to Work with Poor Code
  - How to Use Source Code Control
  - How to Unit Test
  - Take Breaks when Stumped
  - How to Recognize When to Go Home
  - How to Deal with Difficult People

Learn to Debug

Debugging is the cornerstone of being a programmer. The first meaning of the verb "debug" is to remove errors, but the meaning that really matters is to see into the execution of a program by examining it. A programmer that cannot debug effectively is blind.

Idealists, those who think design, analysis, complexity theory, and the like are more fundamental than debugging, are not working programmers. The working programmer does not live in an ideal world. Even if you are perfect, you are surrounded by and must interact with code written by major software companies, organizations like GNU, and your colleagues. Most of this code is imperfect and imperfectly documented. Without the ability to gain visibility into the execution of this code, the slightest bump will throw you permanently. Often this visibility can be gained only by experimentation: that is, debugging.

Debugging is about the running of programs, not programs themselves. If you buy something from a major software company, you usually don't get to see the program. But there will still arise places where the code does not conform to the documentation (crashing your entire machine is a common and spectacular example), or where the documentation is mute. More commonly, you create an error, examine the code you wrote, and have no clue how the error can be occurring. Inevitably, this means some assumption you are making is not quite correct or some condition arises that you did not anticipate. Sometimes, the magic trick of staring into the source code works. When it doesn't, you must debug.

To get visibility into the execution of a program you must be able to execute the code and observe something about it. Sometimes this is visible, like what is being displayed on a screen, or the delay between two events. In many other cases, it involves things that are not meant to be visible, like the state of some variables inside the code, which lines of code are actually being executed, or whether certain assertions hold across a complicated data structure. These hidden things must be revealed.

The common ways of looking into the ‘innards’ of an executing program can be categorized as:

- Using a debugging tool,
- Printlining - Making a temporary modification to the program, typically adding lines that print information out, and
- Logging - Creating a permanent window into the programs execution in the form of a log.

Debugging tools are wonderful when they are stable and available, but printlining and logging are even more important. Debugging tools often lag behind language development, so at any point in time they may not be available. In addition, because the debugging tool may subtly change the way the program executes it may not always be practical. Finally, there are some kinds of debugging, such as checking an assertion against a large data structure, that require writing code and changing the execution of the program. It is good to know how to use debugging tools when they are stable, but it is critical to be able to employ the other two methods.

Some beginners fear debugging when it requires modifying code. This is understandable - it is a little like exploratory surgery. But you have to learn to poke at the code and make it jump; you have to learn to experiment on it and understand that nothing that you temporarily do to it will make it worse. If you feel this fear, seek out a mentor - we lose a lot of good programmers at the delicate onset of their learning to this fear.

Next How to Debug by Splitting the Problem Space