The Pragmatic Programmer

**Chapter 1 Tips: Preface**

1. Care about your craft
   1. Continuous process, small amount of daily care
2. Think! About Your Work
3. Provide Options, Don’t Make Lame Excuses
   1. Take Responsibility
   2. Always have a solution in hand aka “Options”
4. Don’t Live with broken windows
   1. Fix bad designs / wrong decisions / poor code as soon as you spot it
   2. If its clean, people will keep it clean >> if its shit, people won’t take the time to keep it clean
5. Be a Catalyst for Change
   1. Start the charge and get people to help with small little things that add up
      1. If you give people a huge checklist, they checkout, but if you give them something small they will knock it out
   2. People find it easier to join an ongoing success
6. Remember the Big Picture
   1. Keep an eye on the big picture, constantly review what happening around you, not just what you’re personally doing
7. Make Quality A Requirements Issue
   1. Many users would rather use software with some rough edges today than wait ayear for the multimedia version
   2. Know when to stop
8. Invest Regularly in your knowledge Portfolio
   1. Learn one new language every year
   2. Read a technical book each quarter
   3. Read a non-technical books too
   4. Take classes
   5. Participate in local user groups
   6. Experiment with different environment
   7. Stay current
   8. Get wired
   9. Talk to people to gain expertise
9. Critically Analyze What You Read and Hear
   1. Think Critically about what you hear and read
      1. Question validity
   2. Tips for asking
      1. Be specific when you ask for help
      2. Frame your question carefully and politely
10. Its both what you say and the way you say it
    1. Know what you want to say
       1. Does this get across whatever I’m trying to say? >> Refine it until it does
    2. Know your audience
       1. Empathize from their point of view
    3. Choose your moment
       1. Timing is important
    4. Choose a style
       1. Just the facts or long, wide-ranging chat before getting down to business
    5. Make it look good
    6. Involve your Audience
       1. Get the end users feedback early and often
    7. Be a listener
       1. Encourage people to talk by asking question
    8. Get back to people
       1. “I’ll get back to you later” rather than ignoring the email

**Chapter 2 Tips: A Pragmatic Approach**

1. DRY – Don’t Repeat Yourself
   1. Multiple representation of information
   2. Documentation in code & Documentation and code
      1. To ensure tests accurately reflected the specification, the team generated them programmatically from the document itself
      2. When the client amended their specification, the test suite changed automatically
2. Make it Easy to Reuse
   1. Setup group or forum to discuss common problems, exchange ideas and ask questions
3. Eliminate Effects between unrelated Things
   1. Have Orthogonal Systems >> i.e: change in database code won’t affect user interface code
   2. Design components that are self-contained & independent
      1. Single Well-defined purpose
   3. Keep Orthogonal Teams
      1. Separating infrastructure from application
      2. Separating obvious division of application functionality
   4. Tips for Being Orthogonal
      1. Keep your code decoupled
      2. Avoid Global Data
      3. Avoid Similar Functions
   5. Testing
      1. Every Module has its own unit test built into its code. These test should be performed automatically as part of the regular build process

\*\*Orthogonal Algorithms\*\* >> Predict at each step in the process before it makes its end decision

1. There are no Final Decisions
   1. Keep Reversibility in mind when designing solutions
   2. Don’t assume any decision is cast in stone and not have a contingency plan for change
   3. Using vendor\_A database and changing to vendor\_B.
      1. Don’t keep all your code reliant on using vendor\_A where you cant reverse and change plans
   4. Flexible Architecture
2. Use Tracer Bullets to find the target
   1. Create the hooks / wireframe for the system to make sure it works
      1. Keep it to a simple use case
      2. Build the structure and have the end-to-end for continuous integration
3. Prototype to learn
   1. Disposable code just to test out ideas and learn
4. Program Close to the Problem Domain
   1. Give a dummy language for user to write requirements
      1. Then they can give you something you can convert to actual language
         1. Parse it to language of the app
5. Estimate to Avoid Surprises
   1. Based on the duration of how far out, terminology can change the expectation
      1. Determine which units you want use when giving an estimate
   2. Estimating Project Schedules
      1. Check Requirements
      2. Analyze risk
      3. Design, implement, integrate
      4. Validate with the users
6. Iterate the Schedule with Code
   1. Be Agile, track your sprints, and refine your schedule based on iterations