**The Pragmatic Programmer**

There is no such thing as a best solution, be it a tool, a language, or an operating system. There can only be systems that are more appropriate in a particular set of circumstances.

You shouldn’t be wedded to any particular technology, but have a broad enough background and experience base to allow you to choose good solutions in particular situations.

Your background stems from an understanding of the basic principles of computer science, and your experience comes from a wide range of practical projects. Theory and practice combine to make you strong. Your confidence is born of experience.

Good goals: 1) become more efficient, 2) write code that’s easier to maintain, and 3) spend less time in meetings.

**Quarry Worker’s Creed** – *we who cut mere stones must always be envisioning cathedrals*.

One hundred years from now, our engineering may seem as archaic as the techniques used by medieval cathedral builders seem to today’s civil engineers, while our craftsmanship will still be honored.

Common Traits of the Pragmatic Programmer

* Early and Fast Adopter
* Inquisitive
* Critical Thinker
* Realistic
* Jack of All Trades

**Tip 1** Care about your craft.

**Tip 2** Think! About your work.

***Kaizen*** is a Japanese term that captures the concept of continuously making many small improvements.

If you do come across a term you haven’t seen before, please don’t skip over it. Take time to look it up.

**Chapter 1 A Pragmatic Philosophy**

**The Cat Ate My Source Code**

**Take Responsibility**

**Software Entropy**

**Putting Out Fires**

**Stone Soup and Boiled Frogs**

**The Villagers’ Side**

**Good-Enough Software**

**Involve Your Users in the Trade-Off**

**Know When to Stop**

**Your Knowledge Portfolio**

**Your Knowledge Portfolio**

**Building Your Portfolio**

Invest regularly

Diversify

Manage risk

Buy low, sell high

Review and rebalance

**Tip 8** Invest regularly in your knowledge portfolio

**Goals**

Learn at least one new language every year.

Read a technical book each quarter.

Read non-technical books, too.

Take classes.

Participate in local user groups.

Experiment with different environments.

Stay current.

Get wired.

**Opportunities for Learning**

**Critical Thinking**

**Tip 9** Critically analyze what you read and hear

**Communicate!**

**Know What You Want to Say**

**Know Your Audience**

**Choose Your Moment**

**Choose a Style**

**Make It Look Good**

**Involve Your Audience**

**Be a Listener**

**Get Back to People**

**Chapter 2 A Pragmatic Approach**

**The Evils of Duplication**

* **Imposed Duplication**
* **Inadvertent Duplication**
* **Impatient Duplication**
* **Interdeveloper Duplication**

**Orthogonality**

**Reversibility**

**Tracer Bullets**

**Prototypes and Post-It Notes**

Prototyping is much cheaper than full-scale production.

Different prototypes can be used to test different parts of the system.

Prototyping costs can be reduced further by using computer models in-place of physical prototypes.

All of this effort is intended to mitigate the risk and uncertainty prior to committing to build the real item.

Post-it notes are great for prototyping dynamic things such as workflow and application logic.

A UI can be prototyped as a drawing on a whiteboard, as a non-functional mock-up drawn with a paint program, or with an interface builder.

If you’re designing a GUI then the exact details of the computational or performance model may not matter as much. Conversely if computation or performance is critical, then poor GUI or even no GUI at all may suffice. On the other hand, if you cannot sacrifice the details, then prototyping is most likely not the correct approach.

What are good situations where prototyping is useful:

* Anything that carries risk.
* Anything that hasn’t been tried before, or that is absolutely critical to the final system.
* Anything unproven, experimental, or doubtful.
* Anything you aren’t comfortable with.

Prototyping is a learning experience. Its value is not in the code produced, but in the lessons learned.

Details you can ignore in a prototype:

* Correctness – using dummy data for example.
* Completeness
* Robustness
* Style – maybe original code has little to no documentation.

**Domain Languages**

**Estimating**

**Chapter 3 The Basic Tools**

**Chapter 4 Pragmatic Paranoia**

**Chapter 5 Bend, or Break**

**Chapter 6 While You Are Coding**

**Chapter 7 Before the Project**

**Chapter 8 Pragmatic Projects**