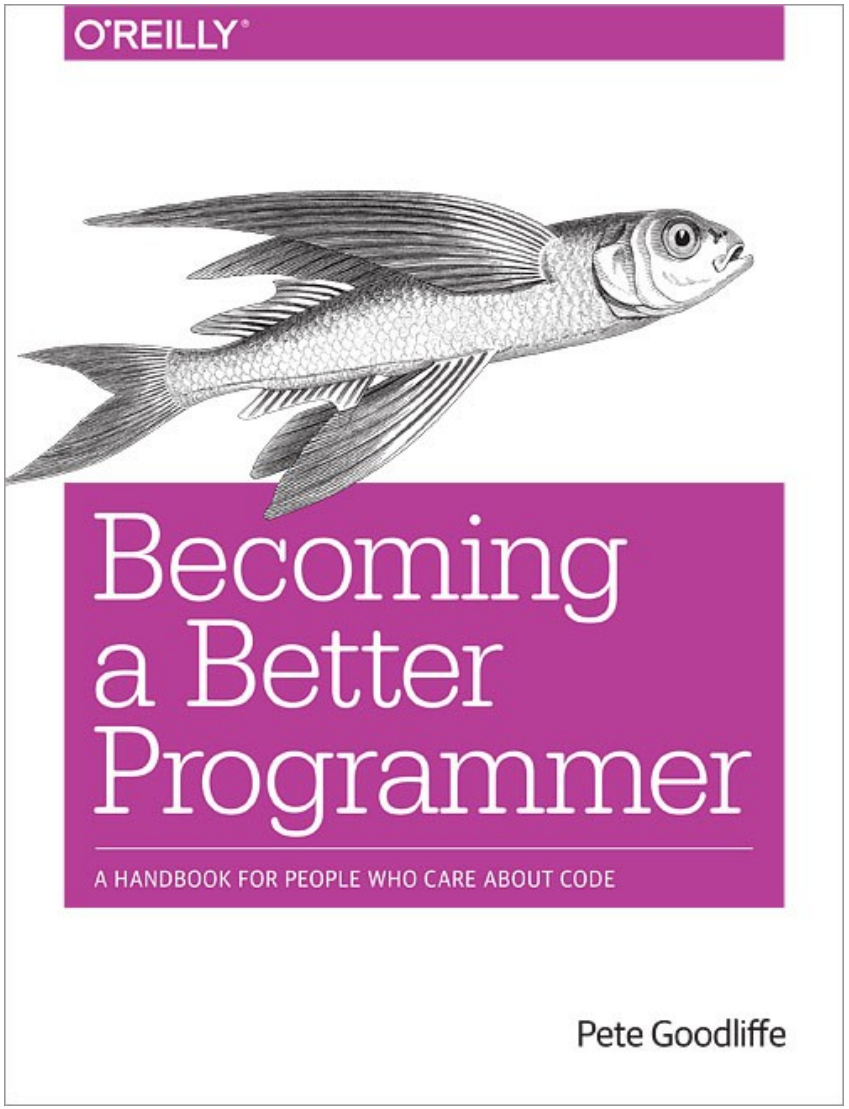
Book review

# *Becoming a better programmer*

by Peter Goodliffe

# Common sense and care

- Becoming a better programmer is not a magic feat
- It takes common sense and care
- Coding is probably 10% of the time of a software engineer
- The rest is where you can make a difference.
- The book describes organization, habits, environment, style, tools, and work patterns



# First, do no harm

- You have to *care*
- Fight the temptation of cutting corners
- Old saying: an amateur…
  - Never has time to do it right
  - Always has time to redo it
- Not just for code, but for all the lifecycle
- If you don't care enough, you *will* harm:
  - Yourself
  - Your organization

# It starts with code

- Code quality makes or breaks a software-dependent company
- Most companies these days depend on software
- Code is written once then maintained forever
- What you want is code that is easy to maintain, thus:
  - Understandable
  - Not too smart
  - Robust (easy to modify in small pieces)
  - Testable
  - With at least some doc

# Don't underestimate presentation

- Code should have a standard presentation (formatting, blanks, style).
- Which one?
  - It doesn't matter, just pick the same one across a project
  - Ban gratuitous style variations (generate dummy changes)
- Have mercy on your maintainers
  - People who open your files shouldn't be shocked
  - Finding what you are looking for in the code should be painless.

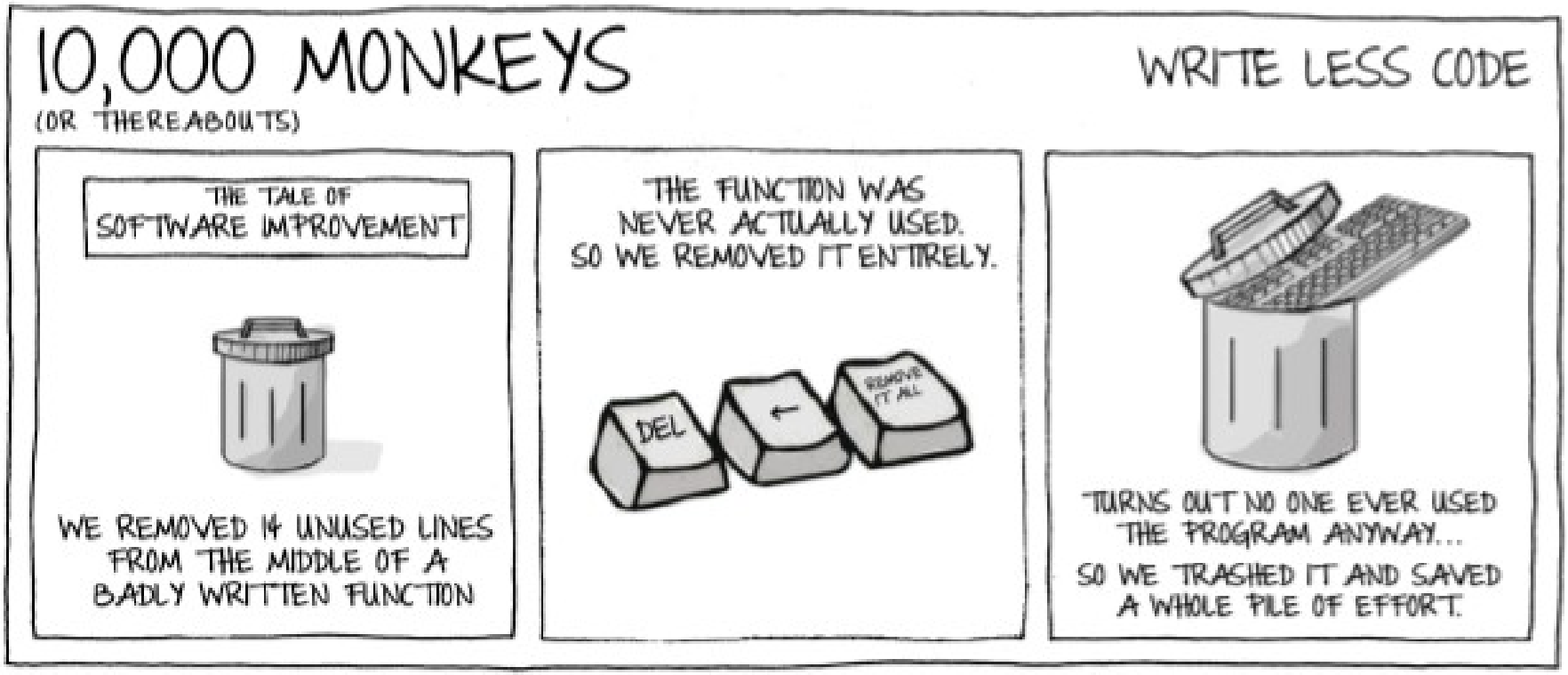
# The magic code improver command

- The best command to improve code in one line:

  ```
  rm
  ```

- Removed code has no bug!
- Remove unnecessary code. Deduplicate, refactor.
- Ban copy-and-paste coding.
- When you first see a certain problem, be certain you didn't see it first.
- Find out existing solutions.
- No existing solution? Neither OSS nor commercial?
  - Either you are on the cutting edge...
  - Or you are doing something wrong.

# The book has cartoons!



# Dealing with existing code

- Existing code can be wonderfully written…
- Or it can be stuff inherited from a cheap contractor that got fired after the original company went under.
- When maintaining existing code, make it testable
- Add unit code, system code, automated build
- Document aberrations and gotchas (and good stuff too!)
- Add order to the chaos
- What about *your* existing code? Have a process to on-board newcomers.
- Good luck.

# Impressing our little friends

- Every coder wants to show off…
- … By writing super-clever code.
- Drawback 1:
  - Few if any people will understand the clever parts
  - So now you will maintain this code for life. Gratz.
- Drawback 2:
  - When you are debugging code, you must be twice as smart as when you write it.
  - So if you write code as smartly as you can, you will never be able to debug it by definition.
- Conclusion: DON'T.

# Look carefully at bad projects

- Engineers love reading about disasters. Why?
- Train wrecks are always entertaining, sure, but there is more.
- Try to extract the root causes or anti-patterns of disasters you come across

- The chemistry of learning:
  Sc + Sw - > X
  (Scars + Sweat - > Experience)
- Read www.thedailywtf.com

# Learn the signs of bad code

- Code inspection should give the smallest possible WPM/LoC (WTFs per minute per line of code)
- Does the code have:
  - Different patterns for the same operation a few lines apart?
  - Inconsistent naming conventions?
  - Booleans that have values TRUE, FALSE or FILE NOT FOUND?
  - Comments like
    `i++; // increments the index`
    but nothing explaining WHY a complex function exists?
  - APIs that read like a bad Greek translation of War and Peace?
  - Exceptions and error codes that are ignored?
- Yep, we got a live one.

# Be prepared

- Only the paranoids survive! → Don't say "it won't happen."
- Things *will* go wrong:
  - Input data *will* be garbage → Sanitize it
  - Network *will* "notwork." → Check for errors and retry
  - API calls *will* fail → Test every call, use exceptions
  - Your code will be killed -9 or run during a power failure → Don't expect a clean state.
- Robust code should expect the worst and be tested in failure conditions.
- But... Don't test for errors you cannot deal with, e.g., logger errors.
- Windows famous message:
  `The error processing code encountered an error.`
  Now what?

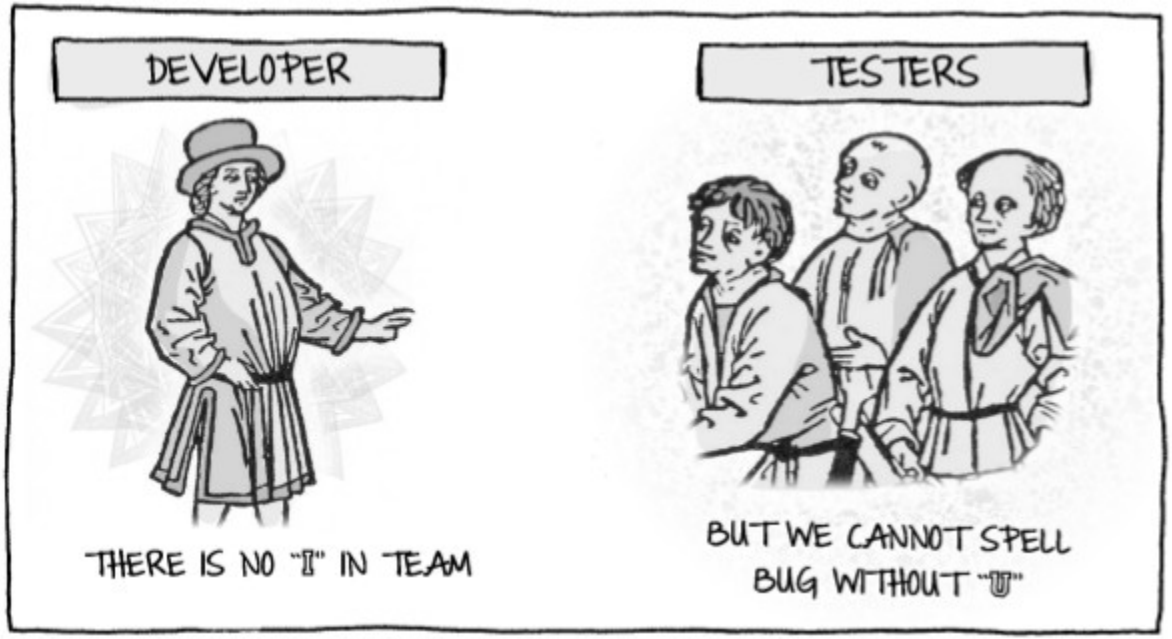
# Test, or be sorry

- Ideally, write tests before code
- TDD: Tests become your working documentation
- So comment each test case!
- Have a strategy to deal with side effects such as writes to DB, file
  - Use mocks or similar
  - Don't test your dependencies
- Tests should be repeatable and push-button easy to run.
- Unit tests: for classes, functions
  - Checks my code
- Integration tests: For subsystems
  - Check my code calls yours correctly
- System test: For full builds
  - End-to-end
  - Simulated inputs

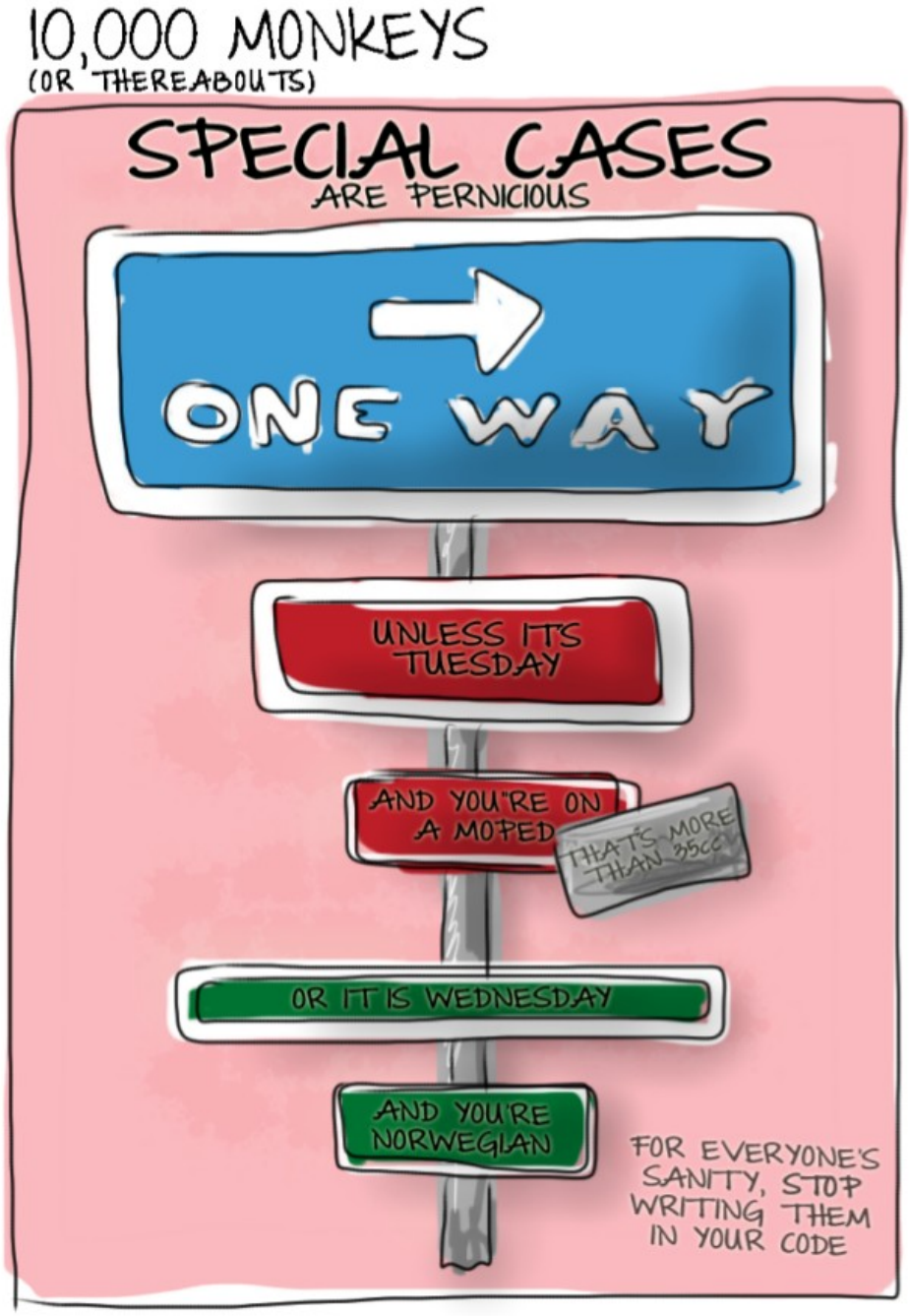
# Why have good tests?

- You will need it to code well
  - Tests solidify your interfaces
  - Bad coder's sign #1 : Tests take too much time to write
- Refactoring and improving? Re-run the tests
  - Make sure you didn't break code
  - So you aren't afraid of changing code.
- QA needs it too

New feature? Add a test!



# The enemy: complexity



# Impressing our little friends

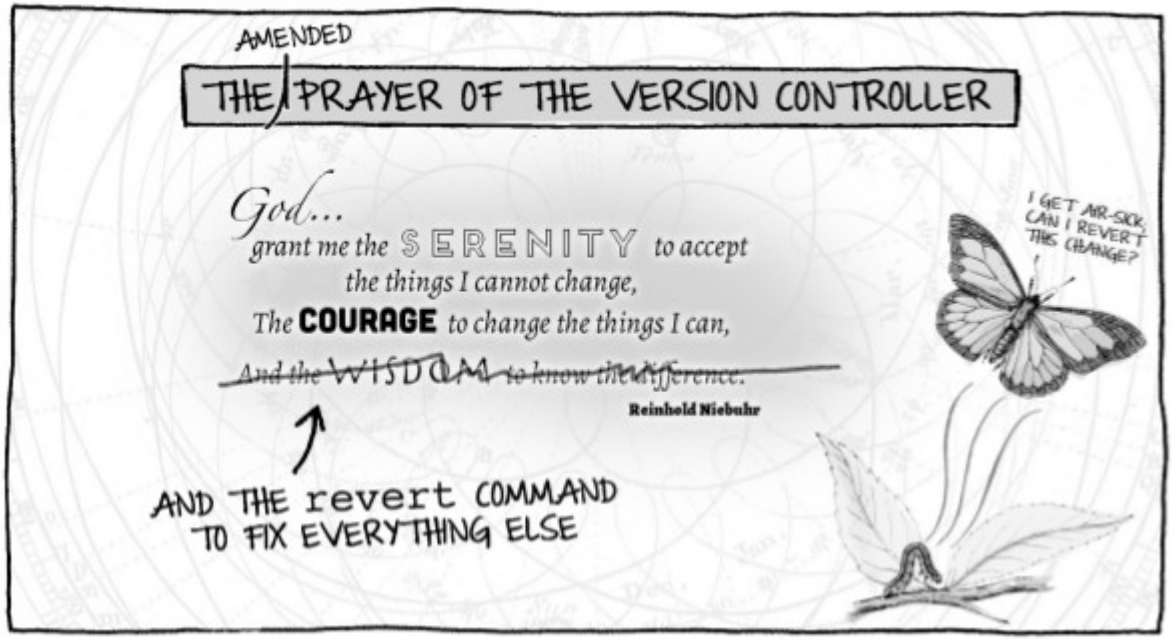
- Every coder wants to show off…
- … By writing super-clever code.
- Drawback 1:
  - Few if any people will understand the clever parts
  - So now you will maintain this code for life. Gratz.
- Drawback 2:
  - When you are debugging code, you must be twice as smart as when you write it.
  - So if you write code as smartly as you can, you will never be able to debug it by definition.
- Conclusion: DON'T.

# Keep it simple, keep it nice

- YAGNI: You ain't gonna need it. Resist the temptation of "in case we decide to do X".
- DRY: Don't repeat yourself. No to copy-paste, yes to code libraries.
- Avoid premature optimization.
- Follow the Boy Scout Rule. Whenever you touch some code leave it better than you found it.
- ... Or at least less infuriating.
- Don't consider that bad stuff is set in stone.
  - Bad stuff doesn't age well. Replacing it with good stuff costs less over time
  - True for code, tools, methods... and people.

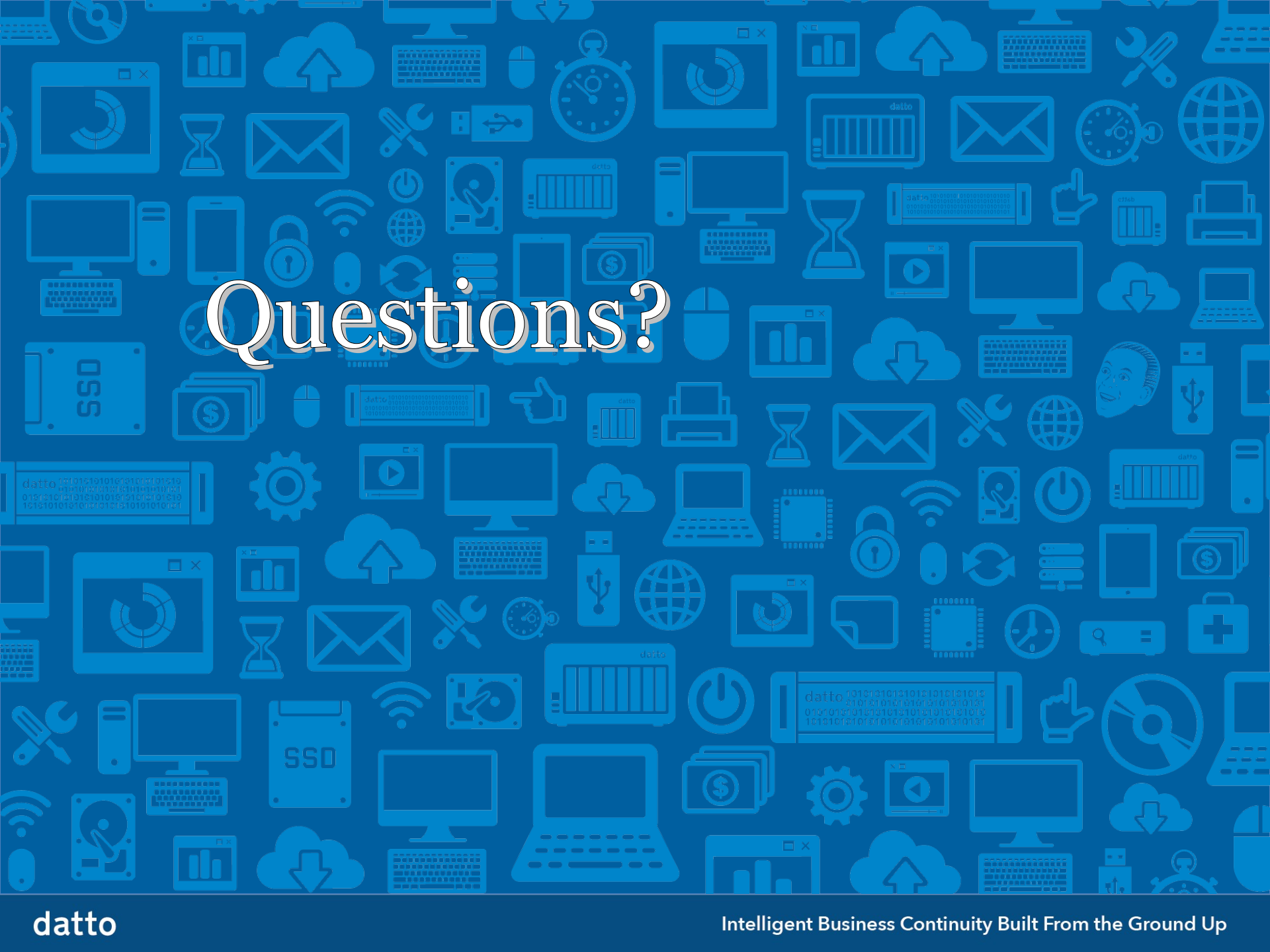
# Use version control

- Anything that takes you more than 5 minutes should be VCed.
- At home:
  - Don't want your own Git server? Use RCS. Or Subversion.
  - If you edit it, version it!
- "What version control do you use?" is question #1 when joining a new team.
- Learn your VC inside and out.



# Intercommunication

- You write code for other people to read
- The computer can deal fine with binary, thank you.
- Nobody is an island. You need good communication.
  - Develop good people skills (can be done. Really.)
  - Be humble. Even Von Braun and Feynman were humble… often.
  - Be nice. Especially when you are right.
  - Close the email client or tab when you are angry. Don't add to the stupid.
  - Maintain email discipline: Be precise, be clear, be concise, give sufficient context, use good grammar.


Questions?