Best Practices in Software Development

Axel Naumann, CERN PH-SFT

Openlab Summer Student Lectures, 2014-08-21

Bugs!

Axel Naumann, CERN PH-SFT
Openlab Summer Student Lectures, 2014-08-21

Coding

- Part of XYZ or on top of XYZ (or replacing XYZ!)
- Language
  - “community” knowledge
  - your knowledge
  - practicality

Practices

- More than one dev or more than one user: need to agree on “how”
- CERN has decades of piles of code, lessons learned:

    1. be reasonable!
    2. enforce!
    3. fix rules early, adapt slowly

Best Practices

Best Practices

- Don’t follow today’s best Best Practices blindly
  - it will be ridiculed in a month anyway
- But having them is simpler than arguing for / reminding of each rule’s motivation

Motivation

- Simpler, consistent read
    - improved communication with fellow coders
    - less ambiguities means more correct code
- Less bugs; better maintenance
- Best practices win against experimental coding

Menu

- Coding convention
- Interface jargon
- Change management
- Multi-platform support
- Tests: code-correctness, functionality, static analysis, performance

Disclaimer

- I am not your best practices superhero
- Focus on C++
  - experience, usage, need

Coding Convention

Coding Convention

- What is this?

```
func(val);
```

Coding Convention

- It's a counter-example!

```
func(val);
```

- func: Member function? Data member / function pointer? Some global function pulled in from header?

- val: local variable declared 100 lines up in the same function? Or member? Or enum constant? And where can I find it's declaration?

Coding Convention

```
fFunc(fgVal);
```

- It’s ROOT - you can tell from the names!
- It’s a function call
- fFunc is a member - so it’s a function pointer!
- fgVal is a static data member; must be in same class (or base)

Coding Convention

- Obvious case of improved clarity
- For APIs, user friendly:
    - get_track(), getTrack(), GetTrack() - or Track()?
- Almost all projects employ it

Coding Convention

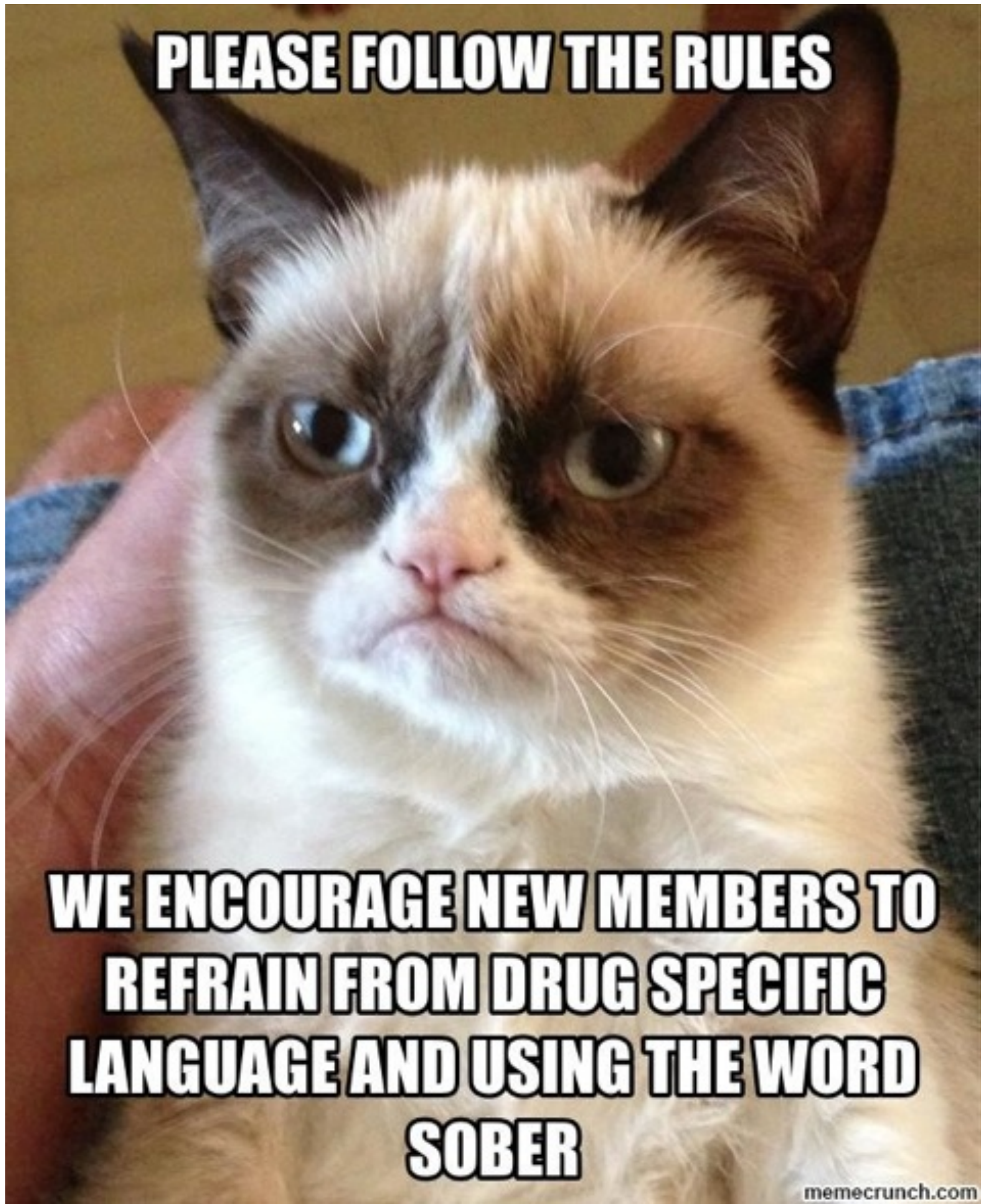
- Typical current examples for C++:
  - Joint Strike Fighter Air Vehicle C++ Coding Standards
  - MISRA C++
- Both absurd for reasonable environments
- Both have very reasonable ingredients: pick yours!

Coding Convention

- Enforcing needs checkers
- Non-trivial; checker must understand C++: what is a function, what is a member etc
- Many C-coding convention checkers (indentation!), few C++, even less open source

Interface Jargon

Interface Jargon



Interface Jargon

- Consistency - we know that already
- Safe code through good APIs!
  - unique_ptr / shared_ptr instead of Type* where ownership is managed; never require “new Type()”, “delete var”
  - document also parameter pre- and post-condition: arg1 must be != 0; arg2 will contain…

Interface Jargon

- Maintain common idioms throughout API; example C++ std library:
    - iterators; functor; make_XYZ; allocator etc
- Don’t screw with your users
    - if interface looks like A, don’t make it do B even if it’s better for you. Change the interface instead.

Threading Support

- Different levels
  - starts threads to compute faster [*multithreaded*]
  - function can be used on same object in multiple, concurrent threads without side-effects [*reentrant*]
  - function can be used on different objects in multiple, concurrent threads without side-effects (no statics)
  - must be locked when accessed through multiple threads [*no threading support*]

Threading Support

- All kinds need to be clearly documented
- Reentrant part of API needs to be visible
- Common contract nowadays:
  - const API means it's reentrant: no mutables! no caches! no hidden state changes!
  - no (unlocked) static variables! State is passed as arguments

Threading Support

- Thus threading support is to a large extend interface jargon

- This is work in progress; has changed rather recently

  - expect further changes; constexpr might play a bigger role soon

  - exposing to >64 threads might change requirements (Amdahl's law!) + style

Interface Jargon + Threading Support

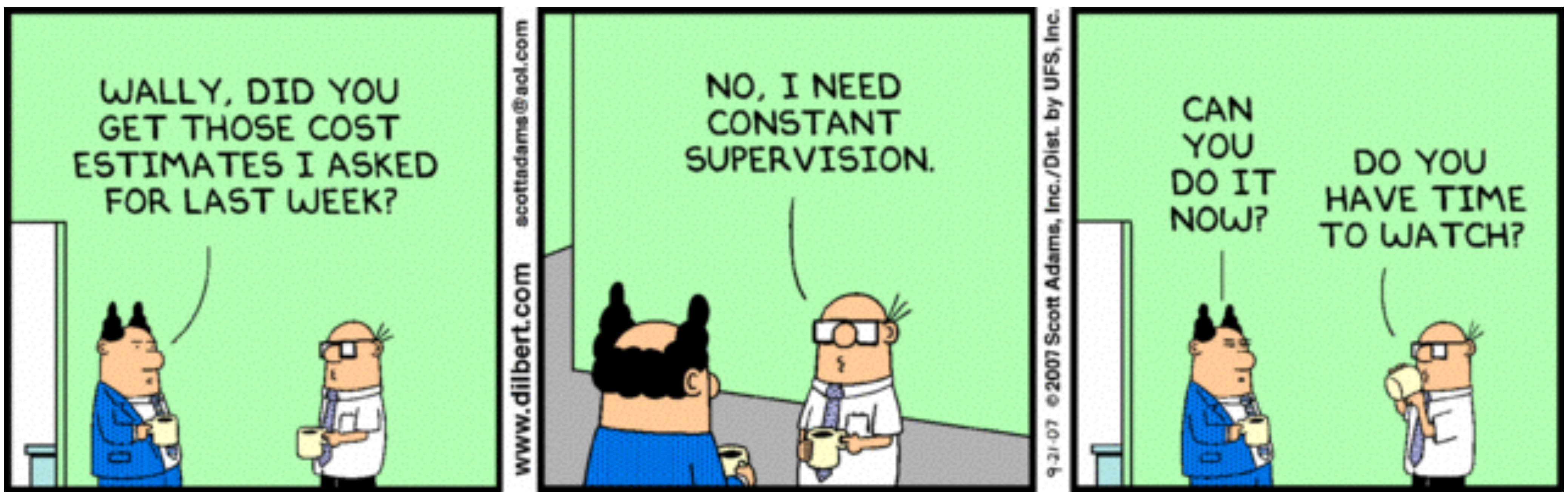
- Automated checking (beyond coding convention) almost impossible
  - requires design work / understanding of the interfaces
- Employ change management instead!

Change Management

Change Management

- Monitor by a second pair of eyes: two brains are better than one
- Avoids bugs creeping in
- Also exposes code, new features to additional / backup developers
- Exposes changes to larger horizon: we all think of changes in different contexts

Change Management



Change Management

- Pre-publication
  - package tags / tag collector (dying concept); instead: change merge as package owner action
  - formalized patch review
  - pair programming
- Post-publication
  - commit review by package owner

Multi-Platform Support

Multi-Platform Support

- Problems:
    - big- versus little-endian
    - OS API
    - lack of language support in compiler
- Developers will get a feeling for what’s causing problems

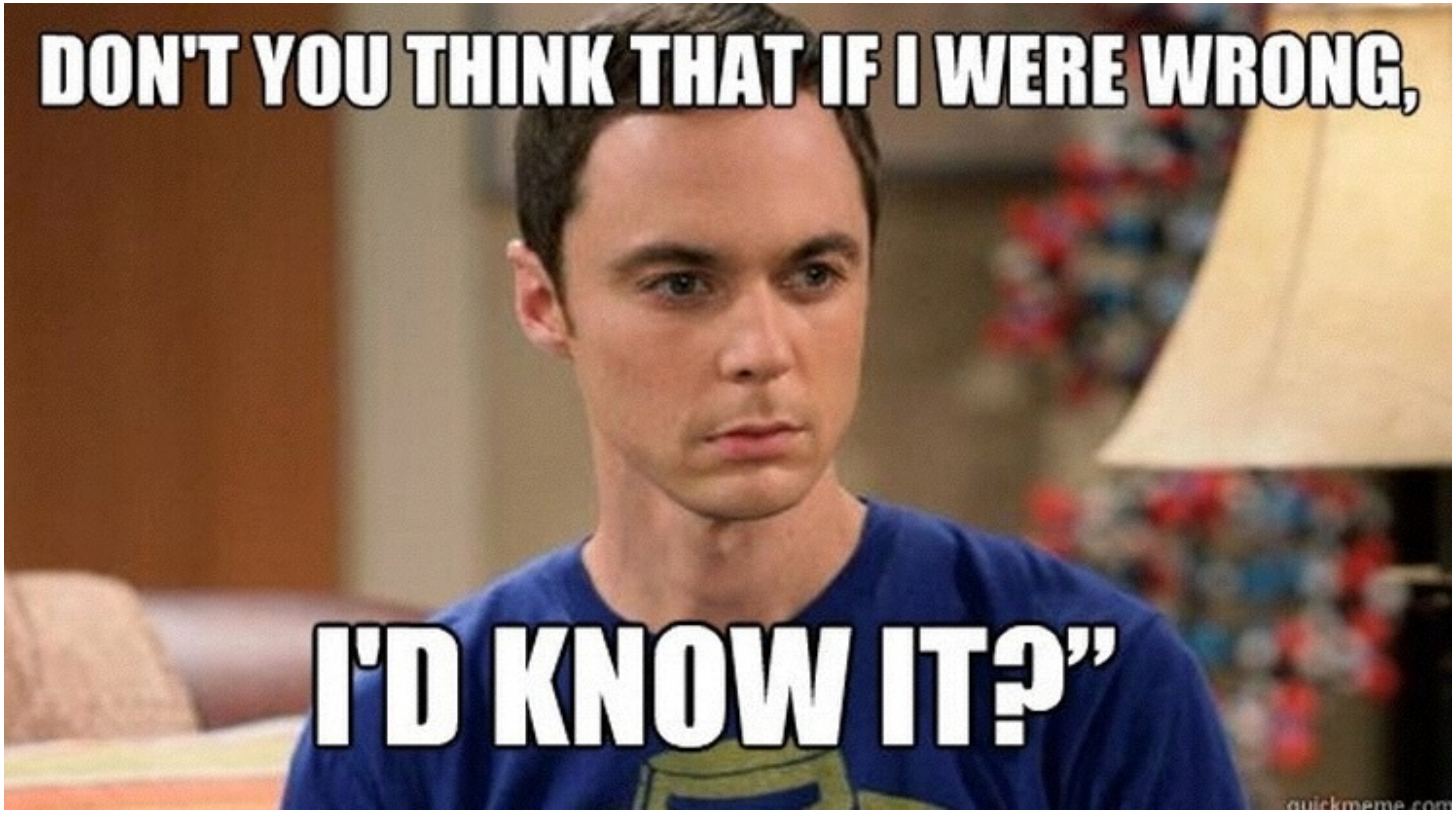
Multi-Platform Support

- Advantages
  - general robustness
  - easier to follow architecture changes
  - will x86_64 be the instruction set of 2030?
  - more compilers = more opinions on code, more warnings (that’s a good thing!)

Multi-Platform Support

- Checking by building on many platforms, regularly
  - Code Correctness Tests!

Tests


DON'T YOU THINK THAT IF I WERE WRONG,
I'D KNOW IT?"

Code Correctness Tests

- Large matrix of builds
  - build on all supported platforms
  - build with all supported configurations
- Ideally after every change
  - helps pinpoint culprits
- Current common grounds: the HEAD works.

Code Correctness Tests

- Run build (incremental or full)
  - check for errors versus platform
  - also check for warnings!
- Run tests
- Build snapshot binaries
  - continuous delivery or bug fix verification

Code Correctness Tests

- Needs automation
- Typical tools: Jenkins / Hudson; Bamboo; TeamCity; BuildBot; Electric Commander
  - schedule and initiate build on all required machines
  - collect output; filter errors, warnings
  - report (web, email) versus code revision

Functionality Tests

- “Does my software actually work?”
- Science by itself; ingredients:
  - unit tests; regression tests; integration tests
  - rules when to write a test
  - testing libraries: cppunit / Google’s 5 or so / CTest
- Needs automation!

Topical Tests

- Memory error checkers - use after free / before initialization
  - e.g. valgrind
- Thread error checkers
  - e.g. hellgrind

Static Analysis

```
#include <iostream>
int func(char* buf) {
 strcat(buf, “<default>”);
 if (!buf) return 1;
 int pos;
 std::cout << “Number between 0 and 8:\n”;
 std::cin >> pos;
 buf[pos] = 0;
 if (!buf) return 12;
 return 0;
}
```

- What’s wrong? (I see 4 errors.)

Static Analysis

- Analyzes source code without running it; creating branch tree to follow possible if etc combinations
- Finds use after delete; impossible if conditions; memory errors etc
- **Cannot** be replaced by test suite: it tests the things that “never happen”

Static Analysis

- Several tools out there, for instance
  - basic checker: compiler warnings!
  - clang static analysis
  - Coverity
- Differ in set of bugs checked; tracing capabilities (through function calls etc); user interface; **false positive rate**

Performance Test

- Changes can deteriorate performance:
  - takes more CPU cycles to get an answer
  - takes more RAM
  - takes more I/O operations
  - takes more disk space
- Criteria vary depending on product

Performance Test

- Usually part of release baking
- Better yet: automate
- Problem: which changes are intentional?
- Tools vary with criteria; e.g. cgroups; massif; CDash


ATOMIC
Rooster

100%

Current Challenges

- Massive multi-threading
- Data-oriented programming
- C++11 and up
- Move every tool into the FOSS world

Conclusion

- Good software development is an art by itself
  - complex; many aspects; need to juggle many tools and often conflicting goals
- Using tools pays off:
  - 1 hour more work for one dev means 10 minutes saved for 10k users *each*
  - users will trust your software more