Coding Style

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.” – Martin Fowler

Coding Style: Understandability

- Anti-Patterns:
  - "Style doesn't matter; it passes all the tests"
  - Code that is clever instead of clear

> "There are two ways of constructing a software design: one way is to make it so simple that there are **obviously no deficiencies** and the other way is to make it so complicated that there are **no obvious deficiencies**."
>
> — C.A.R. (Tony) Hoare, 1980 Turing Award Talk

- Other people must understand your code
  - Peer reviews won't work if nobody can read your code
    - Write code so that others can tell it is obviously correct
  - If others can't understand it, they will inject bugs
  - If it's not obviously correct, then it's wrong.

Make Code Easy To Read

- Consistent formatting
  - Consistent indentation, braces
  - Templated headers for files and functions
  - Spaces and “()” to avoid precedence confusion
  - Use space instead of tab
- Comments
  - Explain what & why, not just code paraphrase
  - Comments are not a design
- Naming
  - Descriptive, consistent naming conventions
    - E.g., variables are nouns; functions are verbs
- Avoid magic numbers (use const)
  - Avoid macros (use inline)

Obfuscated C Winner: Flight Simulator

```
#include                                    <math.h>
#include                                    <sys/time.h>
#include                                    <X11/Xlib.h>
#include                                    <X11/keysym.h>
                                            double L ,o ,P
                                           ,_=dt,T,Z,D=1,d,
                                           s[999],E,h= 8,I,
                                           J,K,w[999],M,m,O
                                           ,n[999],j=33e-3,i=
                                           1E3,r,t, u,v ,W,S=
                                           74.5,l=221,X=7.26,
                                           a,B,A=32.2,c, F,H;
                                           int N,q, C, y,p,U;
                                          Window z; char f[52]
                                        ; GC k; main(){ Display*e=
 XOpenDisplay( 0); z=RootWindow(e,0); for (XSetForeground(e,k=XCreateGC (e,z,0,0),BlackPixel(e,0))
 ; scanf("%lf%lf%lf",y +n,w+y, y+s)+1; y ++); XSelectInput(e,z= XCreateSimpleWindow(e,z,0,0,400,400,
 0,0,WhitePixel(e,0) ),KeyPressMask); for(XMapWindow(e,z); ; T=sin(O)){ struct timeval G={ 0,dt*1e6}
 ; K= cos(j); N=1e4; M+= H*_; Z=D*K; F+=_*P; r=E*K; W=cos( O); m=K*W; H=K*T; O+=D*_*F/ K+d/K*E*_; B=
 sin(j); a=B*T*D-E*W; XClearWindow(e,z); t=T*E+ D*B*W; j+=d*_*D-_*F*E; P=W*E*B-T*D; for (o+=(I=D*W+E
 *T*B,E*d/K *B+v+B/K*F*D)*_; p<y; ){ T=p[s]+i; E=c-p[w]; D=n[p]-L; K=D*m-B*T-H*E; if(p [n]+w[ p]+p[s
 ]== 0|K <fabs(W=T*r-I*E +D*P) |fabs(D=t *D+Z *T-a *E)> K)N=1e4; else{ q=W/K *4E2+2e2; C= 2E2+4e2/ K
  *D; N-1E4&& XDrawLine(e ,z,k,N ,U,q,C); N=q; U=C; } ++p; } L+=_* (X*t +P*M+m*l); T=X*X+ l*l+M *M;
   XDrawString(e,z,k ,20,380,f,17); D=v/l*15; i+=(B *l-M*r -X*Z)*_; for(; XPending(e); u *=CS!=N){
                                          XEvent z; XNextEvent(e ,&z);
                                              ++*((N=XLookupKeysym
                                                (&z.xkey,0))-IT?
                                                N-LT? UP-N?& E:&
                                                J:& u: &h); --*(
                                                DN -N? N-DT ?N==
                                                RT?&u: & W:&h:&J
                                                 ); } m=15*F/l;
                                                 c+=(I=M/ l,l*H
                                                 +I*M+a*X)*_; H
                                                 =A*r+v*X-F*l+(
                                                 E=.1+X*4.9/l,t
                                                 =T*m/32-I*T/24
                                                  )/S; K=F*M+(
                                                  h* 1e4/l-(T+
                                                  E*5*T*E)/3e2
                                                  )/S-X*d-B*A;
                                                  a=2.63 /l*d;
                                                  X+=( d*l-T/S
                                                   *(.19*E +a
                                                   *.64+J/1e3
                                                   )-M* v +A*
                                                   Z)*_; l +=
                                                   K *_; W=d;
                                                   sprintf(f,
                                                   "%5d  %3d"
                                                   "%7d",p =l
                                                   /1.7,(C=9E3+
 O*57.3)%0550,(int)i); d+=T*(.45-14/l*
X-a*130-J* .14)*_/125e2+F*_*v; P=(T*(47
*I-m* 52+E*94 *D-t*.38+u*.21*E) /1e2+W*
179*v)/2312; select(p=0,0,0,0,&G); v-=(
 W*F-T*(.63*m-I*.086+m*E*19-D*25-.11*u
  )/107e2)*_; D=cos(o); E=sin(o); } }
```

http://blog.aerojockey.com/post/iocccsim

Good Code Hygiene

Modularity

- Many smaller .c/.cpp files (one per class)
- Externally visible declarations into .h file

Conditional Statements

- Boolean conditional expression results; no assignments
- All switch statements have a default (usually error trap)
- Limited nesting (see also cyclomatic complexity)

Variables

- Descriptive names that differ significantly
- Smallest practicable scope for variables; initialize at point of definition
- Use typedefs to define narrow types (also use uint32_t, use enum, etc.)
- Range checks & bounds checks (e.g., buffer overflow)

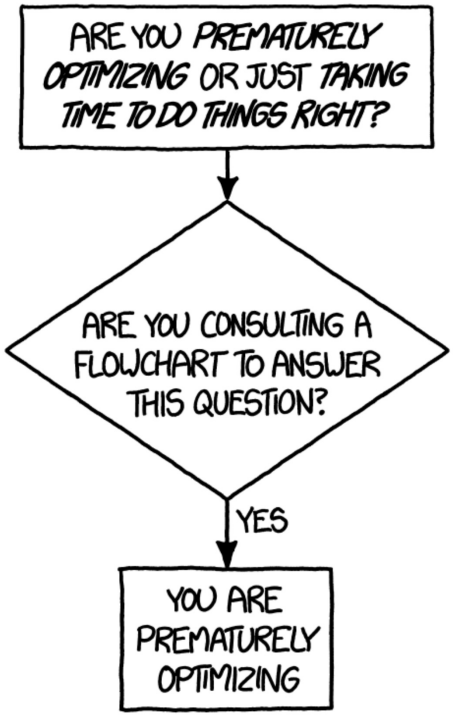
Handle errors returned by called functions

Optimization

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%"

- Donald Knuth (December 1974). "Structured Programming with go to Statements". ACM Journal Computing Surveys 6 (4): 268.

- Don't optimize unless you have performance data
  - Most code doesn't matter for speed
  - Use little or no assembly language. Get a better compiler.
- Optimization makes it hard to know your code is right
  - Do you want correct code or tricky code?
    - (Pick one. Which one is safer?)
  - Buy a bigger CPU if you have to

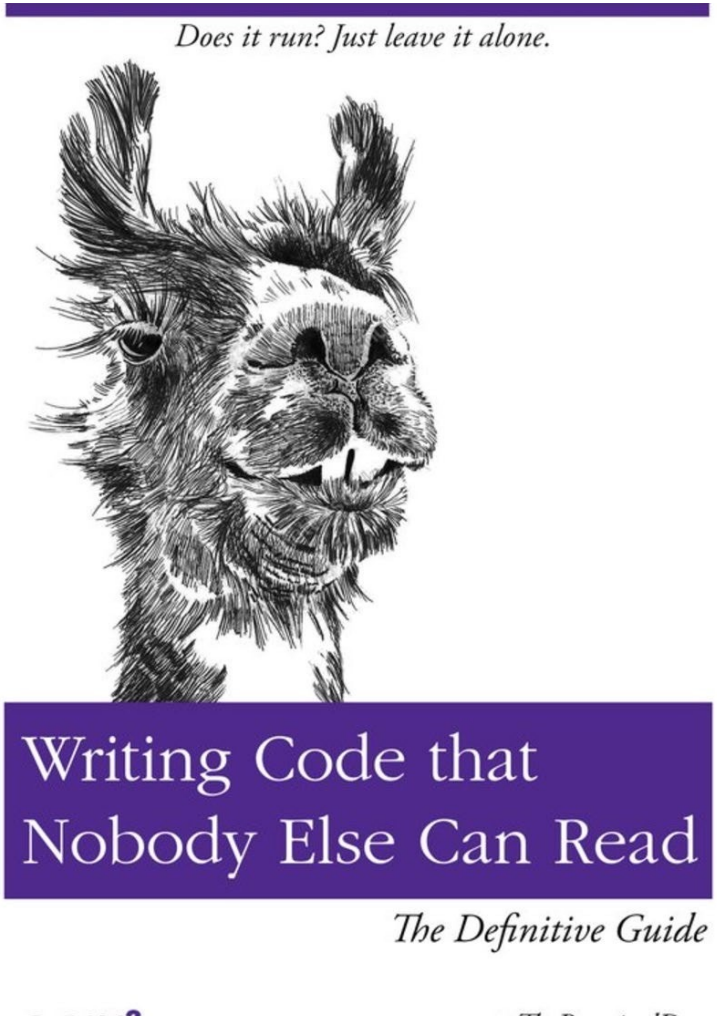


https://xkcd.com/1691/

Coding Understandability Best Practices

- Pick a coding style and follow it
  - Use tool support for language formatting
  - Evaluate naming as part of peer review
  - Comments are there to explain implementation
- The point of good style is to avoid bugs
  - Make it hard for a reviewer to miss a problem
    - Even better, make it easy for a tool to find problem
  - Also need to have a good technical style
- Coding style pitfalls:
  - Optimizing for the author instead of the reviewer
  - Making it too easy to deviate from style rules

Great style depends upon point of view.

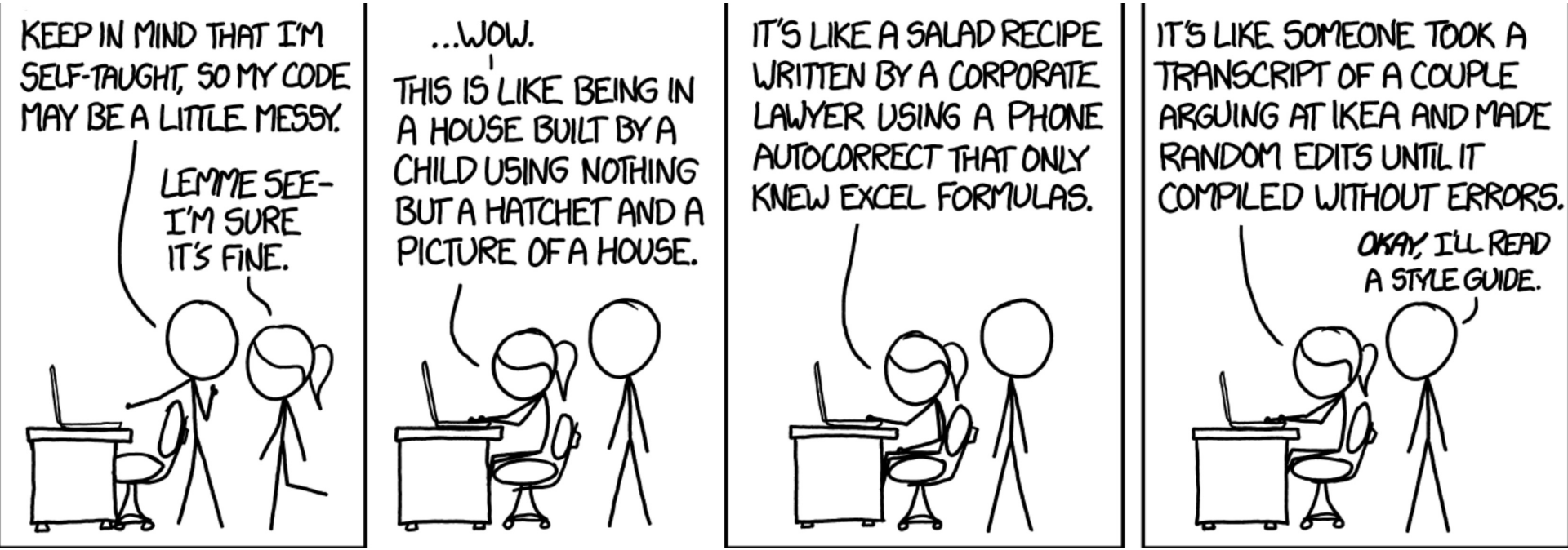


https://goo.gl/pvDMHX CC BY-NC 2.0

“Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.

Code for readability.”

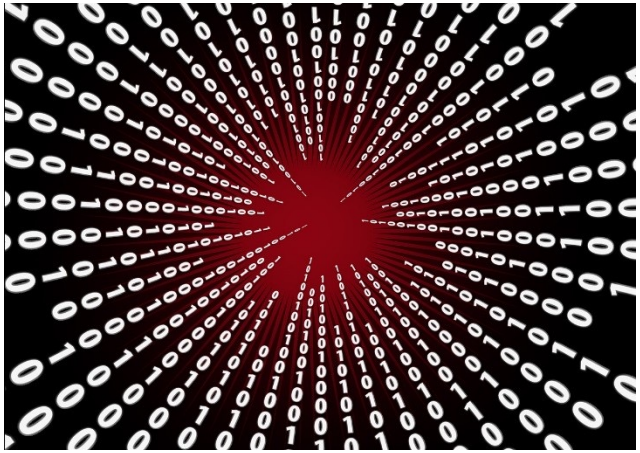
(Author unclear)



https://xkcd.com/1513/

Coding Style: Language Use

- Anti-Patterns:
  - Code compiles with warnings
  - Warnings are turned off or over-ridden
  - Insufficient warning level set
  - Language safety features over-ridden



- Make sure the compiler understands what you meant
  - A warning means the compiler might not do what you think
    - Your particular language use might be "undefined"
  - A warning might mean you're doing something that's likely a bug
    - It might be valid C code, but should be avoided
  - Don't over-ride features designed for safe language use

The C Language Doesn't Always Play Nice

- Defined, but potentially dangerous
  - `if (a = b) { … }  // a is modified`
  - `while (x > 0);  {x = x-1;}  // infinite loop`

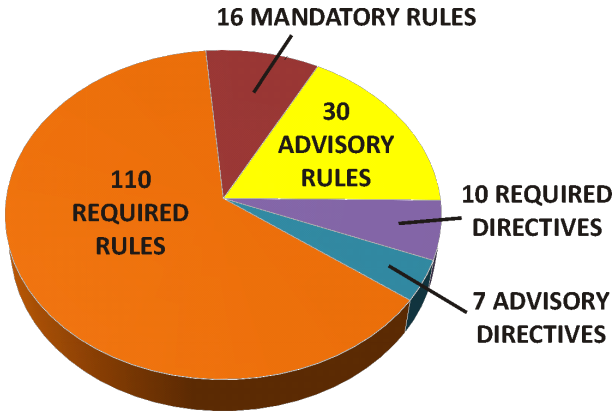
- Undefined or unspecified ➔ dangerous
  - You might think you know what these do …

…but it varies from system to system

  - `int *p = NULL;  x = *p;  // null pointer dereference`
  - `int b;  c = b;  // uninitialized variable`
  - `int x[10]; …  b = x[10];  // access past end of array`
  - `x = (i++) + a[i];  // when is i incremented?`

Language Use Guidelines & Tools

- MISRA C, C++
  - Guidelines for critical systems in C (e.g., no malloc)
  - Portability, avoiding high risk features, best practices
- CERT Secure C, C++, Java
  - Rules to reduce security risks (e.g., buffer overflows)
  - Includes list of which tools check which rules



MISRA C:2012 with Security

- Static analysis tools
  - More than compiler warnings (e.g., strong type warnings)
  - Many tools, both commercial and free. Start by going far past "–Wall" on gcc
- Dynamic Analysis tools
  - Executes the program with checks (e.g., memory array bounds)
  - Again, many tools. Start by looking at Valgrind tool suite

MISRA C 2012 Example

Rule 13.4 The result of an assignment operator should not be *used*

C90 [Unspecified 7, 8; Undefined 18], C99 [Unspecified 15, 18; Undefined 32] [Koenig 6]

Category Advisory

Analysis Decidable, Single Translation Unit

Amplification

This rule applies even if the expression containing the assignment operator is not evaluated.

Rationale

The use of assignment operators, simple or compound, in combination with other arithmetic operators is not recommended because:

- It can significantly impair the readability of the code;
- It introduces additional *side effects* into a statement making it more difficult to avoid the undefined behaviour covered by Rule 13.2.

Example

```
x = y;                          /* Compliant                                */
a[ x ] = a[ x = y ];            /* Non-compliant - the value of x = y
                                 * is used                                 */

/*
 * Non-compliant - value of bool_var = false is used but
 * bool_var == false was probably intended
 */
if ( bool_var = false )
{
}
```

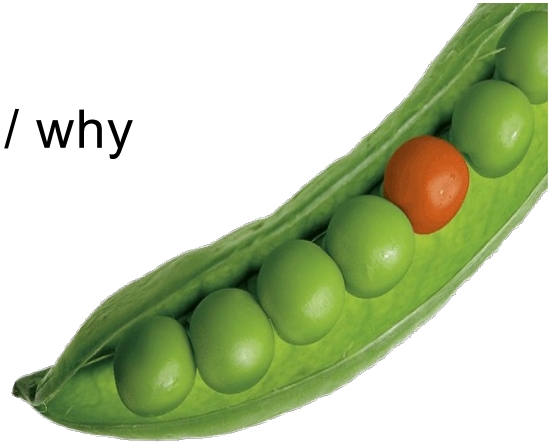
[MISRA C-2012 Guidelines; FairUse]

Let the Language Help!

- Use enum instead of int
  - `enum color {black, white, red}; // avoids bad values`
- Use const instead of #define
  - `const uint64_t x = 1; // helps with type checking`
    `uint64_t y = x << 40; // avoids 32-bit overflow bug`
- Use inline instead of #define
  - If it's too big to inline, the call overhead doesn't matter
  - Many compilers inline automatically even without keyword
- Use typedef with static analysis
  - `typedef uint32_t feet; typedef uint32_t meters;`
    `feet   x = 15;`
    `meters y = x;  // feet to meters assignment error`
- Use stdint.h for portable types
  - int32_t is 32-bit integer, uint16_t is 16-bit unsigned, etc.

Deviations & Legacy Code

- Use deviations from rules with care
  - Use “pragma” deviations sparingly; comment what / why



- What about legacy code that generates lots of warnings?
  - Strategy 1: fix one module at a time
    - Useful if you are refactoring/re-engineering the code
    - Sometimes might need to keep warnings off for 3rd party headers
  - Strategy 2: turn on one warning at a time
    - Useful if you have to keep a large codebase more or less in synch
  - Strategy 3: start over from scratch
    - If the code is bad enough this is more efficient … if business conditions permit

Or – You Can Use A Better Language!

- Desirable language capabilities:
  - Type safety and strong typing (e.g., pointers aren't ints)
  - Memory safety (e.g., bounds on arrays)
  - Robust static analysis (language & tool support)
  - In general, no surprises

```
procedure Increment (X : in out Counter_Type)
  with Global  => null,
       Depends => (X => X),
       Pre     => X < Counter_Type'Last,
       Post    => X = X'Old + 1;
```

- Spark Ada as a safety critical language
  - Formally defined language; verifiable programs
    - The language doesn't have ambiguities or undefined behaviors
  - You can prove that a program is correct
    - E.g., can prove absence of: array index out of range, division by zero
    - (In practice, this makes you clean up your code until proof succeeds)
  - Key idea: design by contract
    - Preconditions, post-conditions, side effects are defined

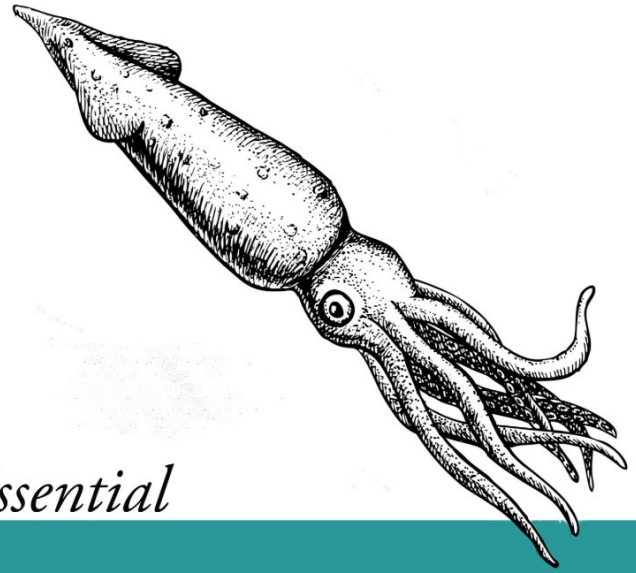
Wikipedia
https://goo.gl/3w6RF6

Spark Ada is a subset of the Ada programming language.

Language Style Best Practices

- Adopt a safe coding style (or a safe language)
  - MISRA C & CERT C are good starting points
  - Specify a static analysis tool and config settings
    - To degree practical, let machines find the style problems
  - When static analysis is set up, add dynamic analysis
- The point of good style is to avoid bugs
  - Let the compiler find many bugs automatically
  - Reduce chance of compiler mistaking your intention
- Coding style pitfalls:
  - “The code passes tests, so warnings don’t matter”
  - Real bugs lost in a huge mass of warnings
  - Making it too easy to deviate from style rules

It's only a clever hack if you're the one who wrote it



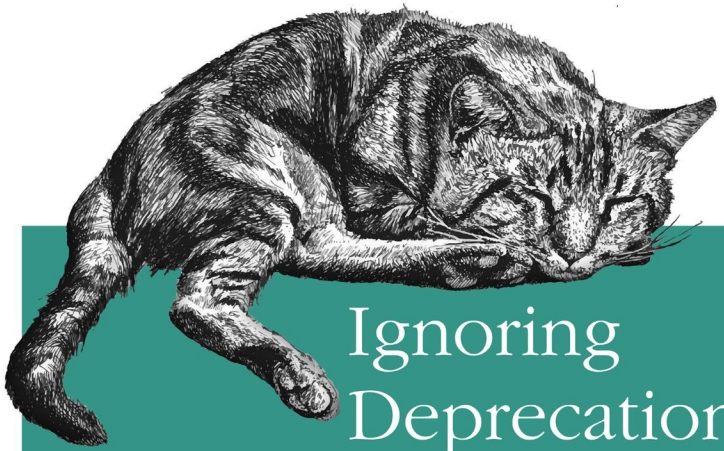
Essential

Hating Other People's Code

O RLY?

@ThePracticalDev

Maybe they'll just go away on their own.

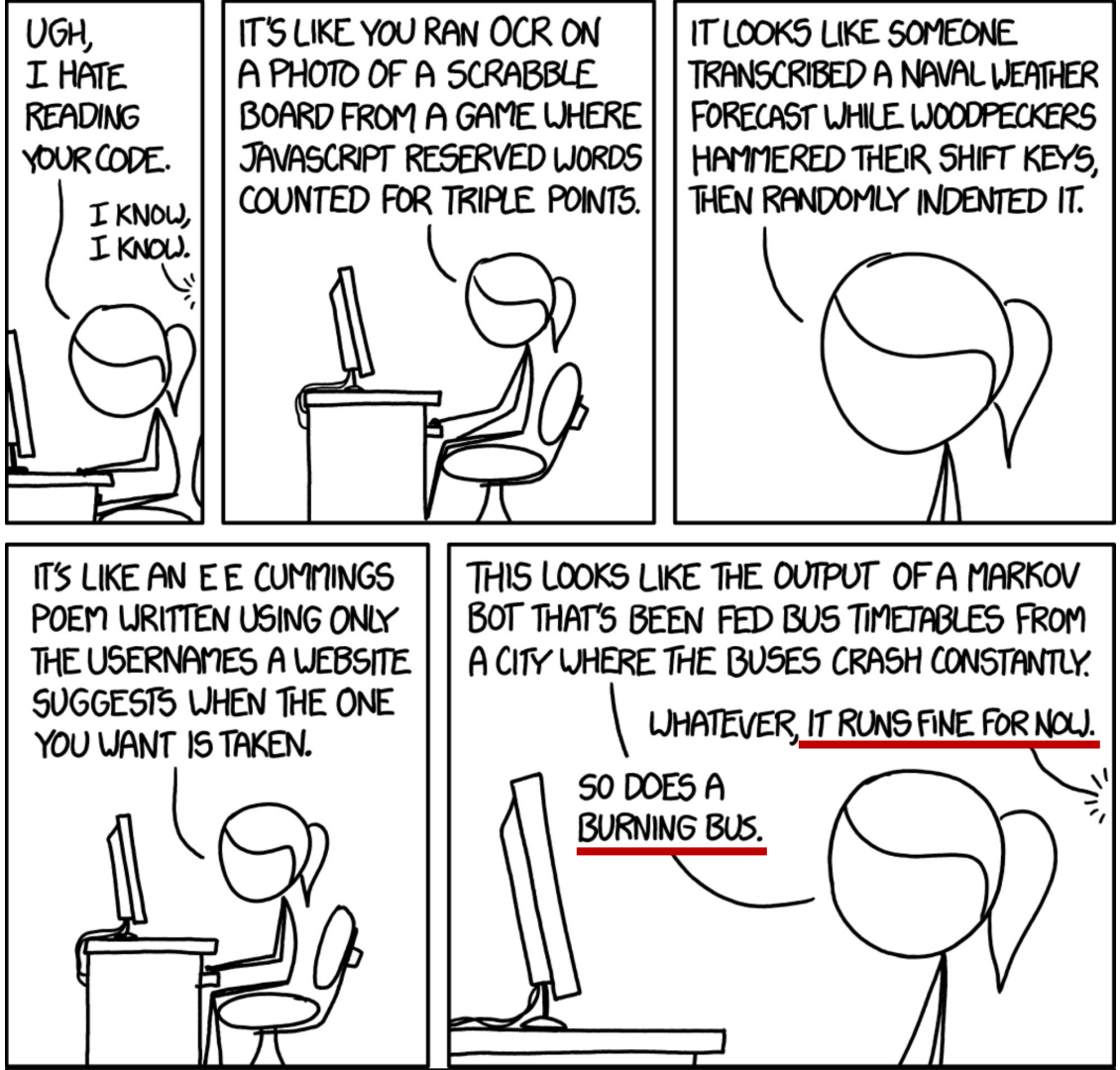


Ignoring Deprecation Warnings

A Practical Guide

O RLY?

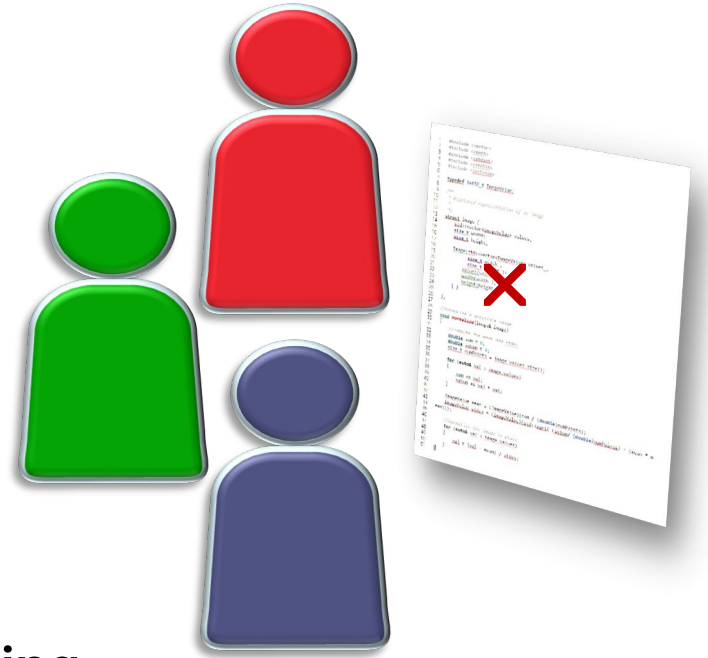
@ThePracticalDev



https://xkcd.com/1695/

Peer Reviews

- Anti-Patterns:
  - No peer reviews
  - Reviews too informal/too fast
  - Reviews find <50% of all bugs



- Fresh eyes find defects
  - Code and other document benefit from a second (and third) set of eyes
  - Peer reviews find more bugs/$ than testing
    - And, they find them earlier when bugs are cheaper to fix
  - Everything written down can benefit from a review

Most Effective Quality Practices

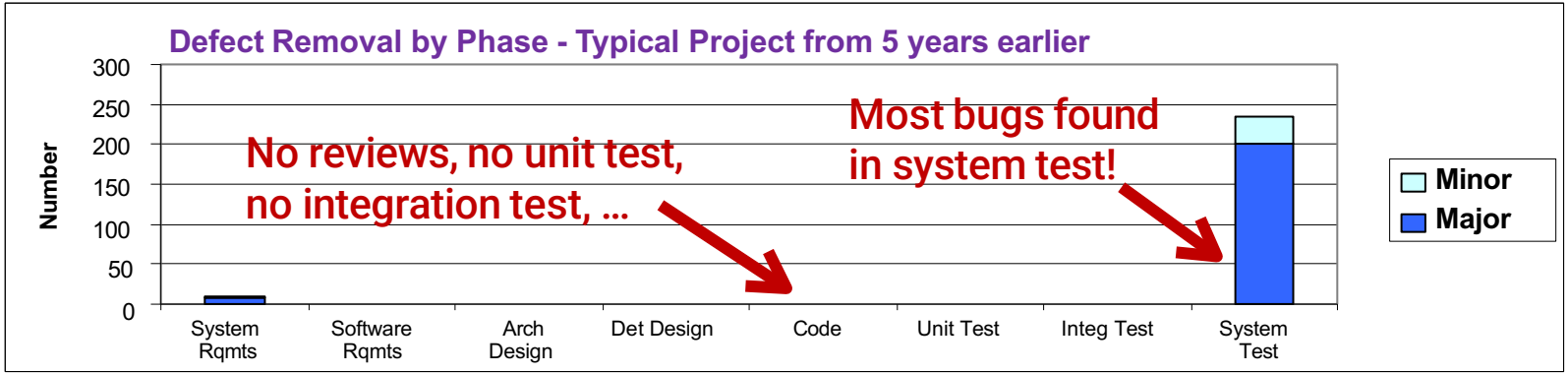
Ebert & Jones, "Embedded Software: Facts, Figures, and Future," IEEE Computer, April 2009, pp. 42-52

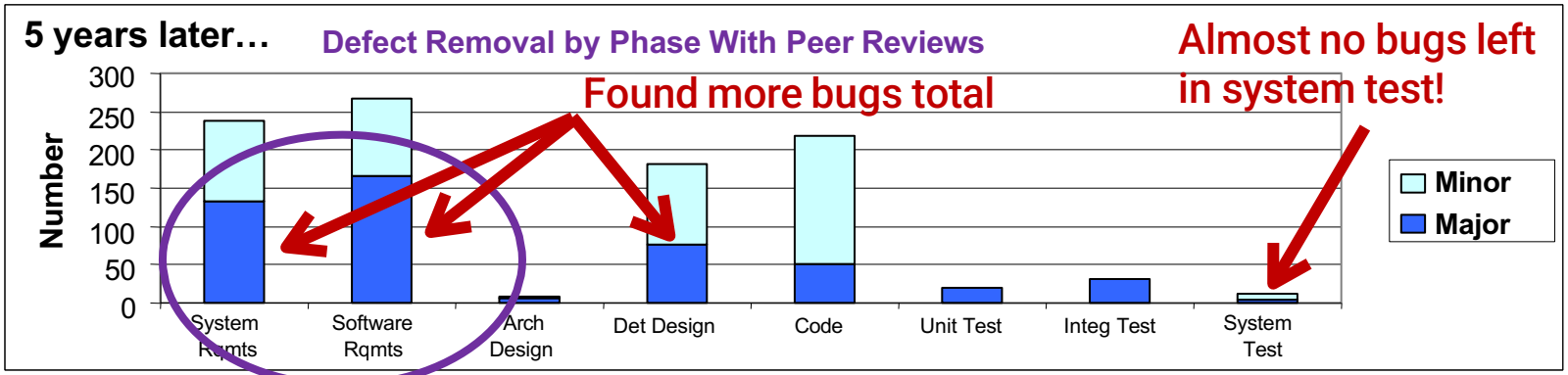
Ranked by defect removal effectiveness in percent defects detectable at that stage that are removed.

"*" means exceptionally productive technique (more than 750+ function points/month)

- * 87% static code analysis ("lint" tools, compiler warnings)
- 85% design inspection
- 85% code inspection
- 82% Quality Function Deployment (requirements analysis)
- 80% test plan inspection
- 78% test script inspection
- * 77% document review (other documents)
- 75% pair programming (informal on-the-fly review)
- 70% bug repair inspection
- * 65% usability testing
- 50% subroutine testing (unit test)
- * 45% SQA (Software Quality Assurance) review
- * 40% acceptance testing

Peer Reviews Are Effective + Efficient





Found many bugs up front, where fixes are cheaper

[Source: Roger G., Aug. 2005]

Gold Standard: Fagan Style Inspections

- Methodical, in-person review meetings
  - Pre-meeting familiarity with project
  - Producer explains item then leaves
  - Moderator keeps things moving
  - Reader (not author) summarizes as you go
  - Reviewers go over every line, using checklists (perspective-based)
  - Recorder takes written notes
  - Result: written list of defects. The Producer fixes code off-line
  - Re-inspection if the defect rate was too high
- Methodical reviews are the most cost effective
  - Important to measure bug discovery rate to ensure review quality

Rules for Successful Peer Reviews

- Inspect the item, not the author
  - Don't attack the author.
- Don't get defensive
  - Nobody writes perfect code. Get over it.
- Find but don't fix problems
  - Don't try to fix them; just identify them.
- Limit meetings to two hours
  - People are less productive after that point.
- Keep a reasonable pace
  - About 150 lines of code (or equivalent) per hour. Too fast and too slow are both bad.
- Avoid "religious" debates on style
  - Enforce conformance to your style guide. No debates on whether style guide is correct.
- Inspect, early, often, and as formally as you can
  - Keep records to document value (might take a while to mature).

Example Light-Weight Review Report

	Peer Review Template for Project X	
Date:	4/17/2011	
Artifact:	Xyzzy.cpp Functions: Foo(), Bar(), Baz()	
Reviewers:	Stella K., Joe B., Sam Q., Trish R.	
Size:	357	SLOC
Time Spent:	112	Minutes
# Issues:	3	
Outcome:	Re-Review of Bug Fixes Required	

 # issues found is the most important item!

Issue#	Issue Description	Status
1	Issue 1…..	Fixed
2	Issue 2….	Bugzilla
3	Issue 3….	Bugzilla
4	Issue 4….	Not a Bug
5		
6		
7		
8		

Free form text issue description

Status Key:	Fixed (trivial fix by author; no need to enter in defect list)
	Bugzilla (entered into project defect system)
	Not a Bug (false alarm)

Just enter "fixed" if fixed within 24 hours

Perspective-Based Peer Reviews

- Perspective-based Peer Reviews are 35% more effective

[https://www.cs.umd.edu/projects/SoftEng/ESEG/papers/82.78.pdf]

- Mechanics of a Perspective-based review
  - Divide a peer review checklist into three sections
  - Assign each participant a different section of the checklist
    - OK to notice other things, but primary responsibility is that section
    - Multiple sets of eyes + perspective breadth

- Example perspectives for a review:
  - Control flow issues
  - Data handling issues
  - Style issues

Peer Review Checklist Template

Peer Review Checklist: Embedded C Code

Before Review:

0	_____	Code compiles clean with extensive warning checks (e.g. MISRA C rules)

Reviewer #1:

1	_____	
2	_____	
3	_____	
4	_____	
5	_____	
6	_____	
7	_____	

Reviewer #2:

8	_____	Single point of exit
9	_____	Loop entry and exi
10	_____	Conditionals shoul
11	_____	All functions can b
12	_____	Use const and inlir
13	_____	Avoid use of magic
14	_____	Strong typing (incl
15	_____	All variables have

Reviewer #3:

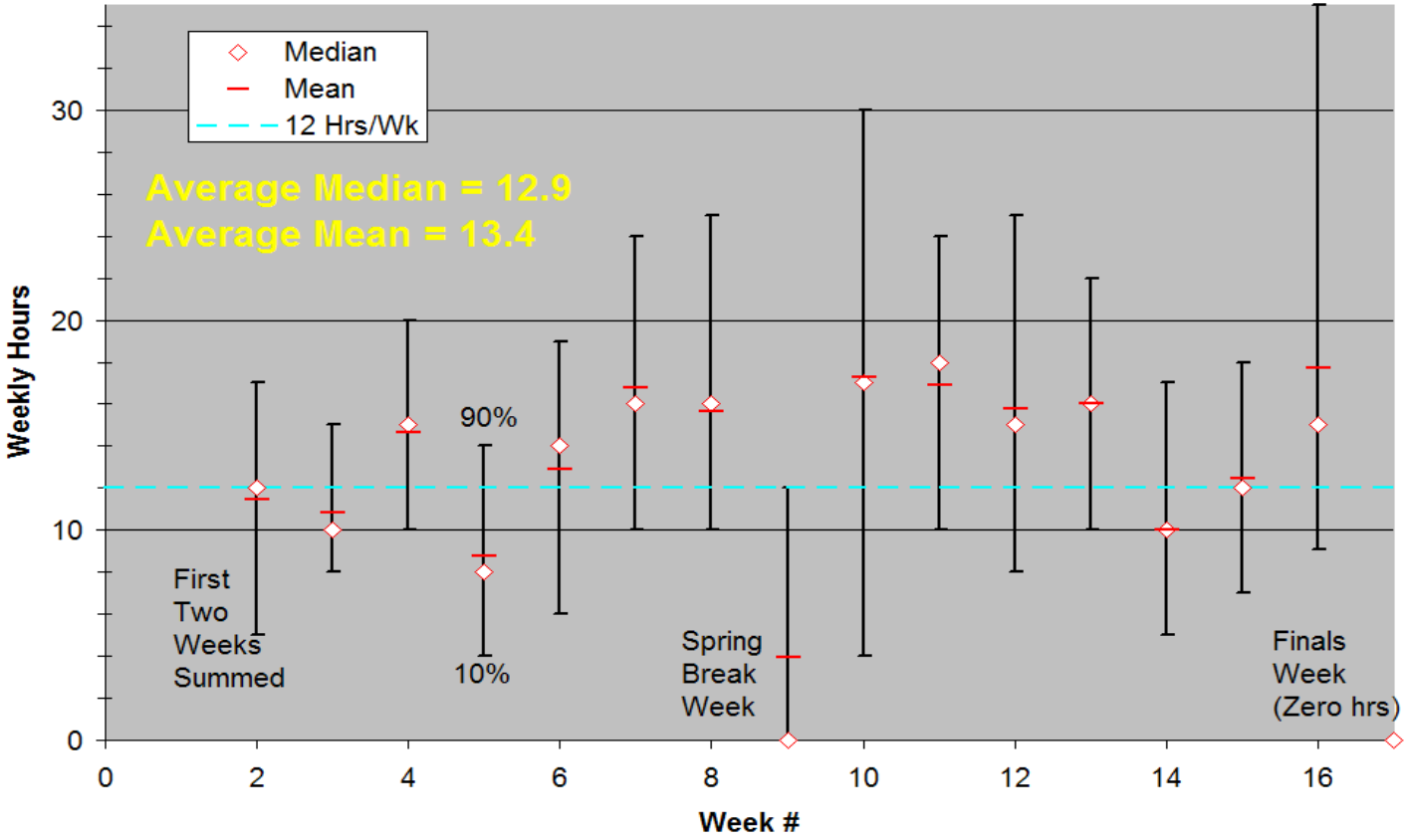
16	_____	Minimum scope for all functions and variables; essentially no globals
17	_____	Concurrency (locking, volatile keyword, minimize blocking time)
18	_____	Input parameter checking (style, completeness)
19	_____	Error handling for function returns
20	_____	Handle null pointers, division by zero, null strings, boundary conditions
21	_____	Floating point issues (equality, NaN, INF, roundoff); use of fixed point
22	_____	Buffer overflow safety (bound checking, avoid unsafe string operations)

All Reviewers

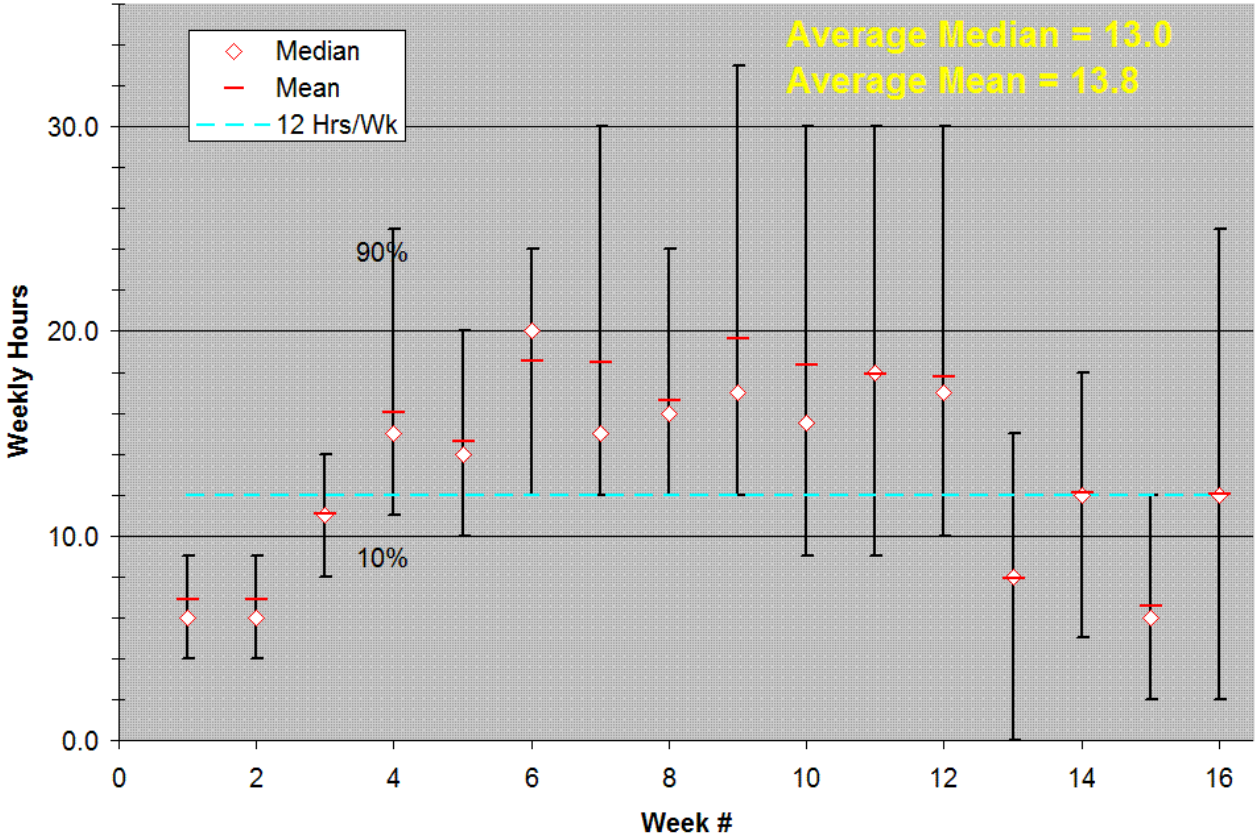
23	_____	Does the code match the detailed design (correct functionality)?
24	_____	Is the code as simple, obvious, and easy to review as possible?
		For TWO Reviewers assign items: Reviewer#1: 1-11; 23-24 Reviewer#2: 12-24

- **Customize as needed**

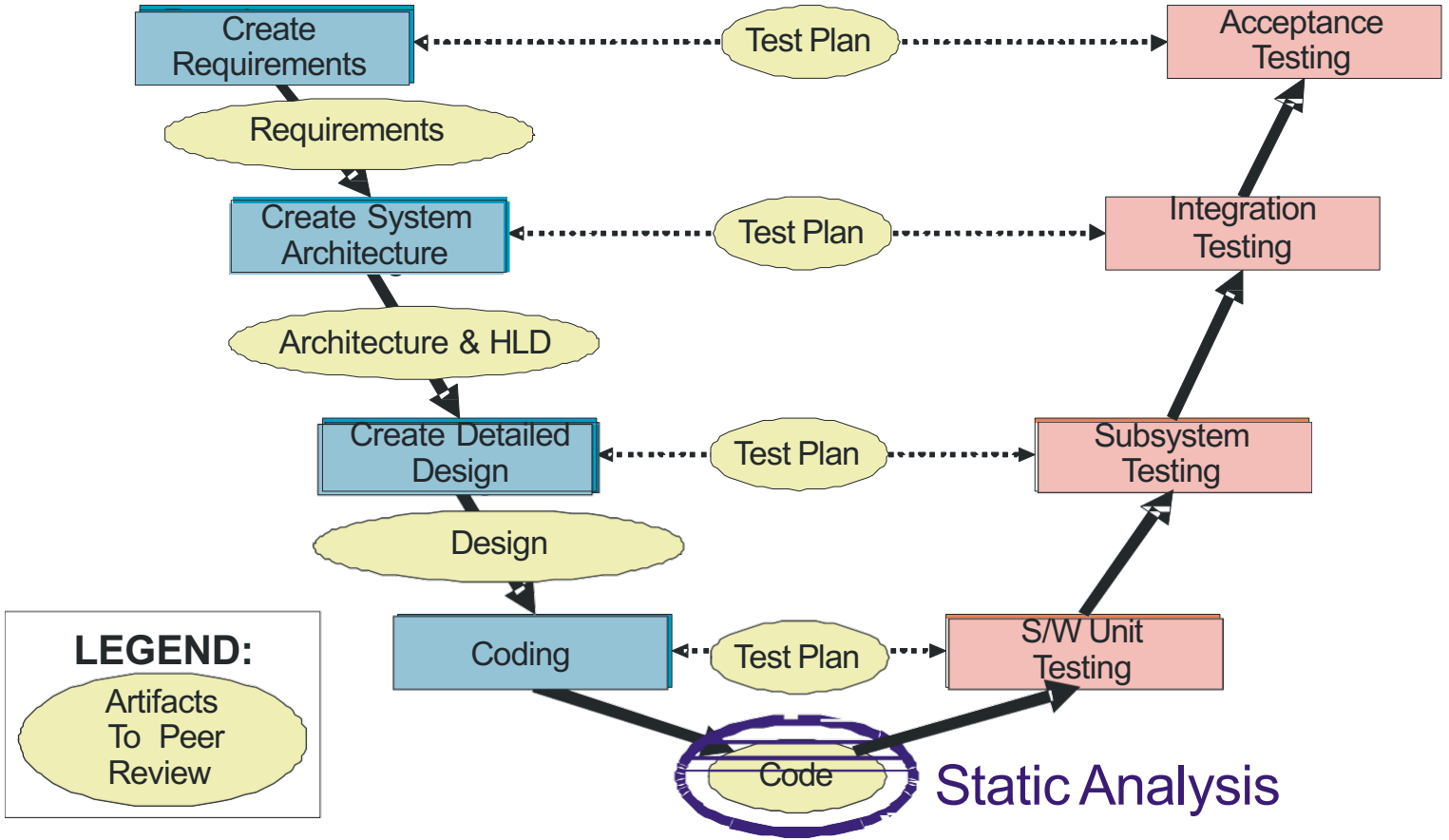
Before (Ineffective Reviews)



With Weekly Defect Reporting



Review More Than Just The Code



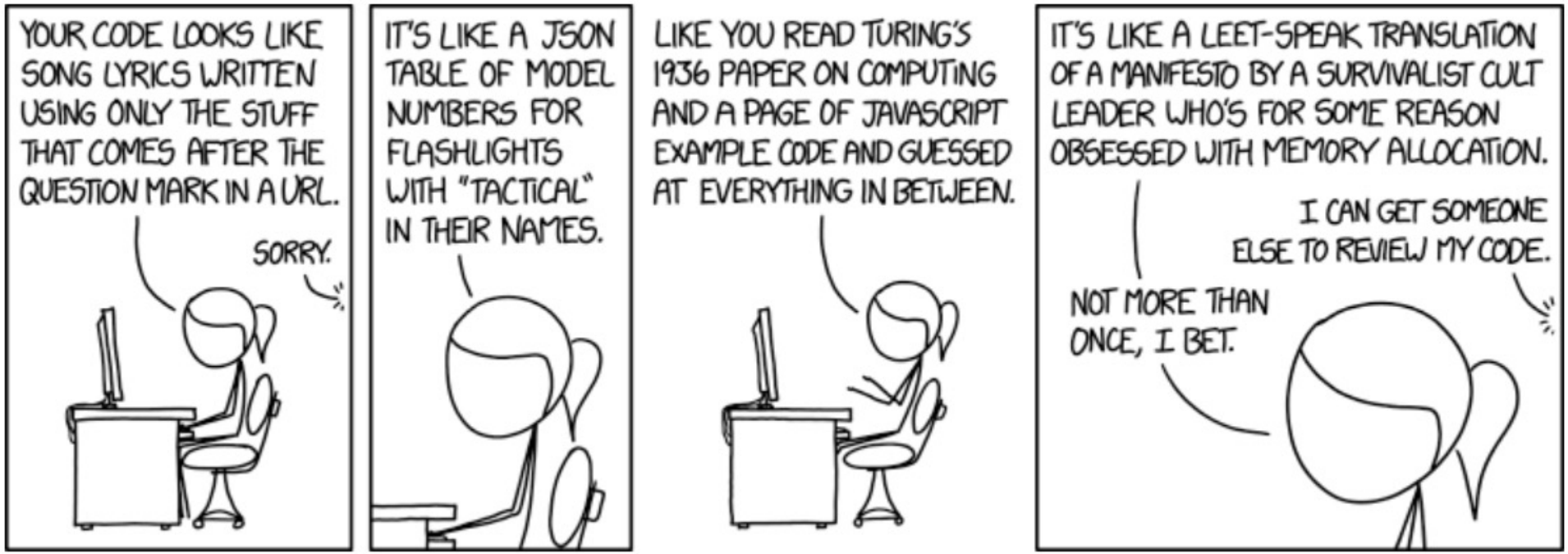
Economics Of Peer Review

- Peer reviews provide more eyeballs to find bugs in an affordable way
  - Good embedded coding rate is 1-2 lines of code/person-hr
    - (Across entire project, including reqts, test, etc.)
  - A person can review 50-100 times faster than they can write code
    - If you have 4 people reviewing, that is still >10x faster than writing!
  - How much does peer review cost?
    - 4 people * 100-200 lines of code reviewed per hour
    - E.g., 300 lines; 4 people; 2 hrs review+1 hr prep = 25 LOC/person-hr
  - Reviews are only about 5%-10% of your project cost
- Good peer reviews find at least half the bugs!
  - And they find them early, so total project cost can be reduced
- Why is it folks say they don't have time to do peer reviews?

Peer Review Best Practices

- Formal reviews (inspections) optimize bugs/$
  - Target 10% of project effort to find 50% of bugs
    - You can review 100x faster than write code; it's cheap
  - Review everything written down, not just code
  - Use a perspective-based checklist to find more bugs

- Review pitfalls
  - If your reviews find <50% of defects, they are BROKEN
    - The 80/20 rule does NOT apply to review formality! Formal reviews are best.
    - You can't review at end; need to review throughout project
- Review tools
  - On-line review tools are OK, but not a substitute for in-person meeting
  - Static analysis tools are great – but not a review!



https://www.xkcd.com/1833/

Disclaimer

This lecture contains materials from:

- Philip Koopman - CMU