sdc – 05/07/2019

Code Review

NOT ONE SIZE FITS ALL!

- team size
  - 2 vs 10
- product type
  - agency vs in-house vs open source

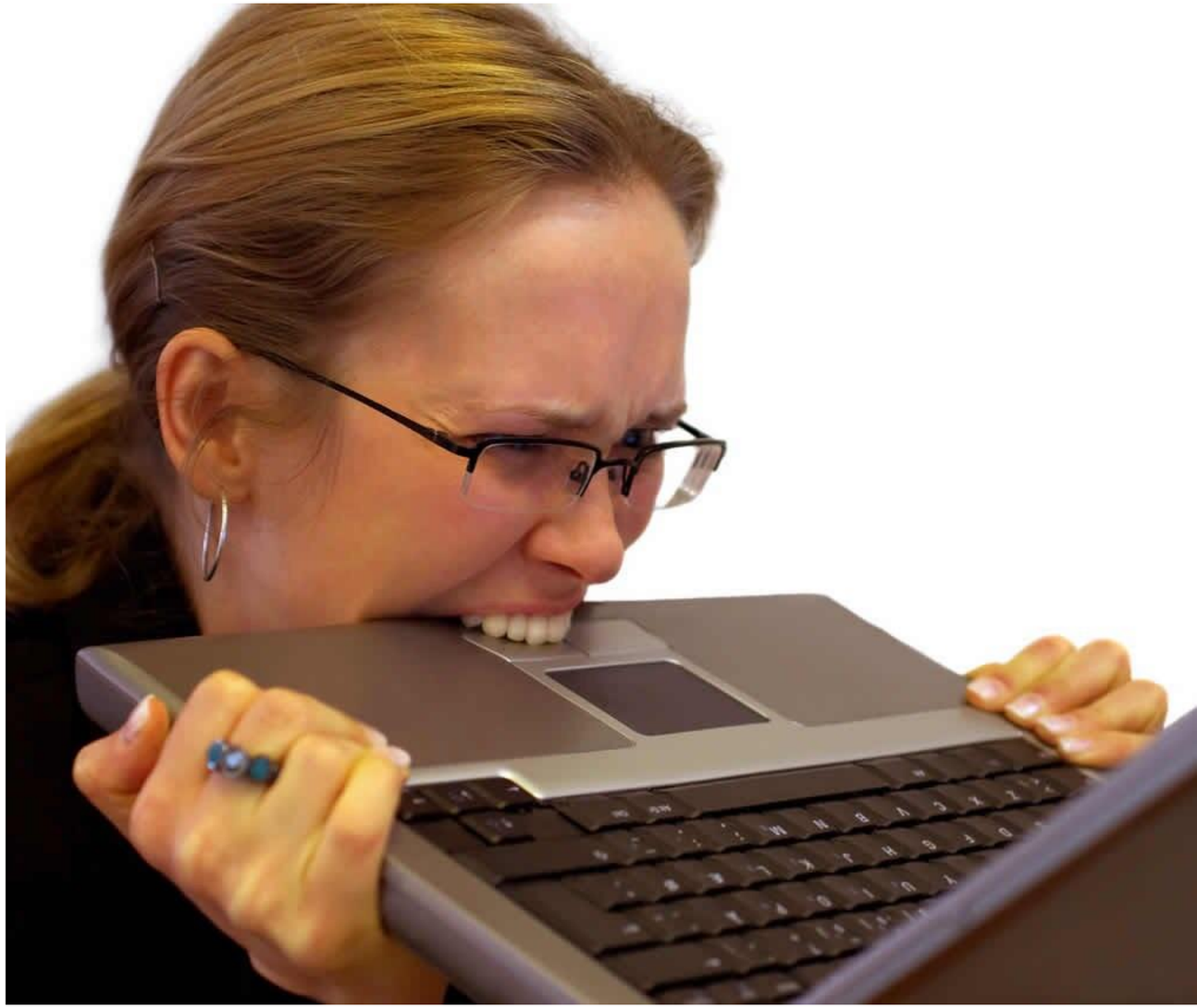
NOT ONE SIZE FITS ALL!

> defect tolerance

> jet engines vs mobile games

WHY CODE REVIEW?

CODE REVIEWS CAN BE FRUSTRATING



@nnja

APPARENT CODE REVIEW FRUSTRATIONS

> Adds time demand
> Adds process
> Can bring up team tensions
> “smart” devs think they don’t need it 

CODE REVIEW BENEFITS

FIND BUGS & DESIGN FLAWS

> Design flaws & bugs can be identified and remedied before the code is complete

> Case Studies on Review[5]:

> ⬇ bug rate by 80%

> ⬆ productivity by 15%

[5] blog.codinghorror.com/code-reviews-just-do-it/

THE GOAL IS TO FIND BUGS BEFORE YOUR CUSTOMERS DO

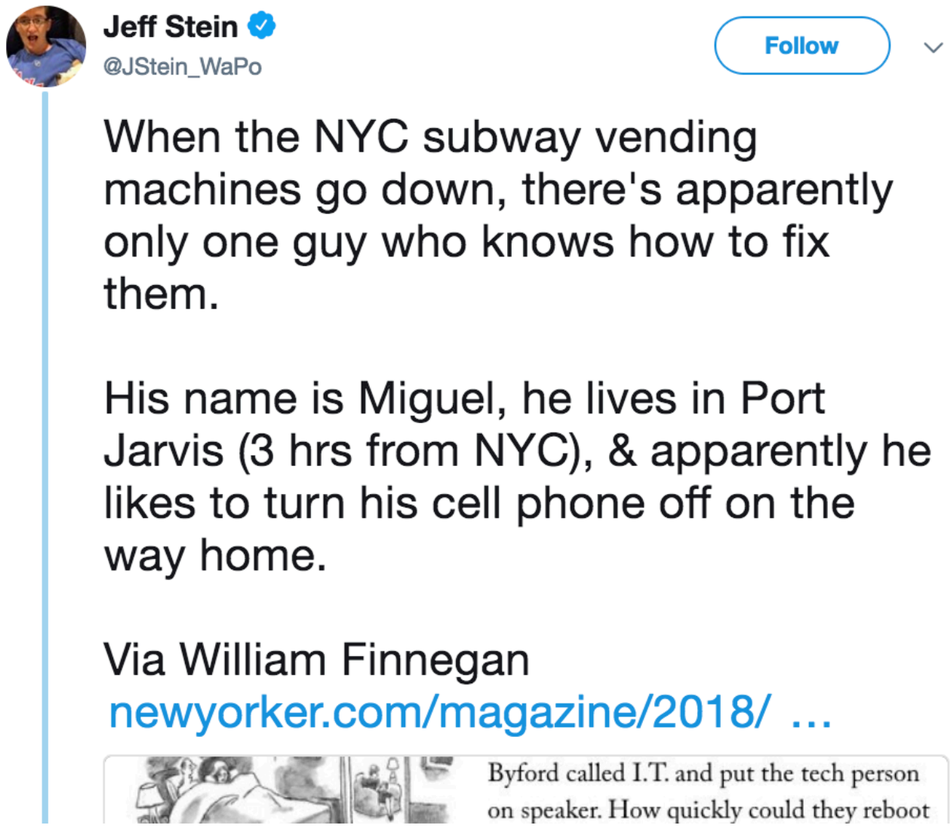
 @nnja

SHARED OWNERSHIP & KNOWLEDGE

> We’re in this together

> No developer is the only expert

LOTTERY FACTOR



New Yorker: Can Andy Byford Save the Subways?

CODE REVIEW BENEFITS?

> Find Bugs
> Shared Ownership
> Shared Knowledge
> Reduce "Lottery Factor"

@nnja

HOW?

@nnja

CONSISTENT CODE

> Your code isn't yours, it belongs to your company

> Code should fit your company's expectations and style **(not your own)**

> Reviews should encourage consistency for code longevity

CODE REVIEWS NEED TO BE UNIVERSAL & FOLLOW GUIDELINES

> Doesn't matter how senior / junior you are

> Only senior devs reviewing == bottleneck

> Inequality breeds dissatisfaction

@nnja

CONSISTENT CODE IS EASIER TO MAINTAIN BY A TEAM

@nnja

CODE REVIEW IS DONE BY YOUR PEERS & NOT MANAGEMENT

@nnja

DON'T POINT FINGERS!

@nnja

WHEN CODE REVIEWS ARE POSITIVE.
DEVELOPERS
DON'T EXPECT THEIR
CHANGES TO BE REVIEWED.
THEY WANT THEIR
CHANGES TO BE REVIEWED.

@nnja

The Zen of Python

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Key principles

- Reusability
- Readability
- Modularity
- Maintainability
  - Extensibility
  - Reworking
- Try to keep one responsibility per component used

What does “good” code look like?

- Is it readable?
- Is it easy to understand?
- Does it flow?
- Does the implementation reflect the design
- Is it modular?
- Is the code reusable?
- Are functions clearly defined? - cohesion
- Are interactions clear with minimal dependencies?
  - Loosely coupled
- Is it maintainable?
- Is there any undefined behaviour?

What might “bad” code look like?

- Cyclic dependencies
- Components with multiple functions - violates single responsibility principle
- A single component that does everything
- Function spread over multiple components
- Ambiguity
- Component meshing – components heavily coupled
- Components that only call other components

Readability

- Use a style guide: PEP8, PEP257 (see Medical Physics QMS)
- (Try to) be consistent in naming:
  - CamelCase for classes
  - lowercase_with_underscores_for_definitions
- Use a development environment that will nag you to obey the style guide e.g. PyCharm

If we’re being picky…

- Fully defined conditional flows
- Premature optimizations
- Use of inheritance vs composition
- Choice of data structures
- Use of design patterns