Lecture Notes
December 4, 2018

Second Iteration Demo due tonight

Legacy code means code that no one (in your company or open source community) is actively developing - although there may be other engineers actively maintaining

OR

Code that someone was actively developing until recently, but that someone is now "gone"

Don't curse the people who wrote the code or take screenshots to send to
The Daily WTF (how-not-to guide for developing software)

Don't assume any documentation that came with the codebase is accurate/up-to-date

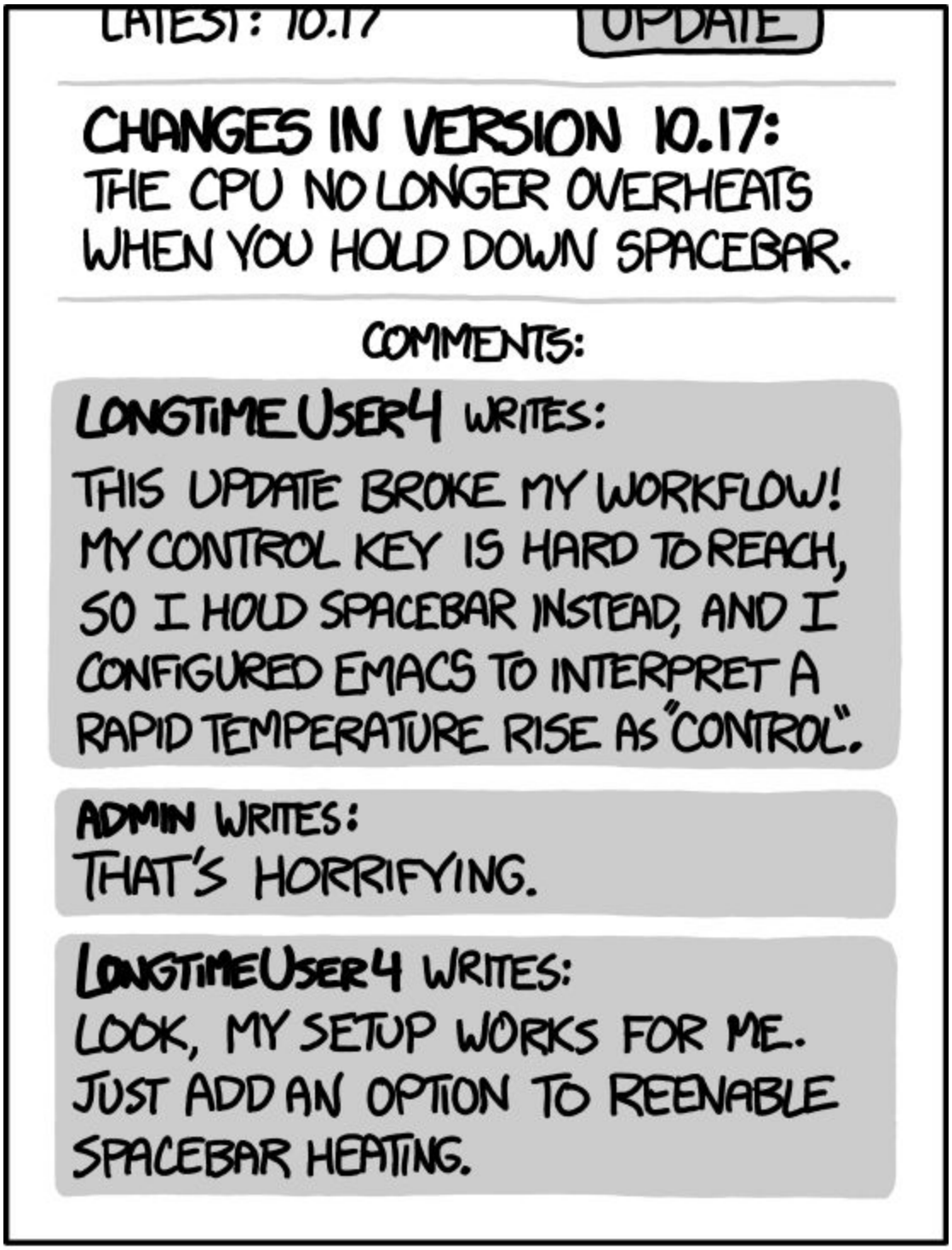
Generate a *visualization* of the code architecture/design and dependencies, e..g., jGRASP, MoDisco

Write *characterization tests*: Accept at face value what the code does and document those behaviors with tests. Compare any changes your team makes against the original tests

Isolate business logic from plumbing code, whether third-party or home grown, before any other refactoring - and run the characterization tests to make sure nothing broke

Limit other changes - Branch for even a small change then merge, Don't touch anything else

Don't be too quick to "improve" the codebase in ways that change the behaviors visible to external users



EVERY CHANGE BREAKS SOMEONE'S WORKFLOW.

Program Understanding (or program comprehension) is a concern whenever a software engineer is tasked to work with code that he/she did not write (or did not write recently), whether legacy or written yesterday by someone else

Even for non-legacy codebases, the original author(s) are not always available to help, or may not have enough time to help as much as needed - they usually have their own jobs to do <something else>

Industry studies, using observations, interviews and surveys, show that about **half** of professional developers' time is spent trying to comprehend code written by someone else (includes time searching for information)

How well do you need to understand previously unfamiliar code in order to perform code review, fix a bug, add a feature, port from one platform to another? Studies found that most developers seek to *minimize* understanding

Strategies seen during the industry studies:

- Employ a recurring, structured comprehension strategy depending on context - same developer usually used same approach, e.g., start by reading source code and find specific portion of code, start by reading code documentation or requirements

- Follow a problem-solution-test work pattern - find code to change, make the change, test the change, test that change didn't break other code

- Identify starting point for comprehension and filter irrelevant code based on experience - minimize code to read (experts much better at this than novices)

- Interact with UI to determine functionality or expected program behavior - e.g., which code is triggered by a button click, check whether data entered into GUI form stored in database, "what is program supposed to do?", "does this component provide that functionality?"

- Debug application to elicit runtime information - use interactive debugger to inspect running state of application

- Establish and test hypotheses - e.g., where are certain values set or used, make assumption then check it

- Clone to avoid comprehension and minimize effort - copy existing code instead of refactoring to share that code (avoids breaking the existing code), but this adds to "technical debt"

- Take notes to reflect mental model and record knowledge - function names or signatures, control or data flow charts, but notes rarely saved beyond current task to consult on later tasks

Tools used - applications or IDE features used during comprehension tasks

- Nearly all developers use IDEs to read/edit code and most use interactive debuggers to inspect application state - but dedicated program comprehension tools rarely used (developers may not know about them)

- Standalone generic tools such as grep and text editors are used side by side with IDE even when IDE provides equivalent feature - e.g., full text search, comparing two versions

- Developers often do not know IDE features, that a specific feature exists, what it is good for, how to use it effectively, or in which situations can the feature help

- Compiler is used to elicit structural information - e.g., change name of constant or comment out method definitions and look at locations of consequent compiler errors (there are much better tools to do this!)

Knowledge needed - useful information required to understand the software or otherwise generated during the comprehension tasks

- Source code and inline comments is more trusted than documentation (non-existent, outdated, out of context) - software development organizations need required documentation styles and reviews analogous to coding styles and reviews

- Personal communication is preferred over written documentation - but colleagues must be knowledgeable, available, approachable, and sometimes finding out "who knows what" is itself challenging in large development organizations

- Standards facilitate comprehension - e.g., naming conventions, consistent architecture such as imposed by development framework

- Rationale and intended usage is important, but rare information - "why was the code implemented this way", "what was the developer's intention when writing this code" forgotten even by developer who wrote original code because not documented

- Real usage scenarios are useful but rare - how end users really use application, or intend to use application, often missing for later versions

Channels used

- Work colleagues - in-person when co-located, via emails and internal mailing lists otherwise, effective but not always efficient - knowledge lost when experts leave organization, experts frequently interrupted from their own tasks and may need to provide same information over and over again rather than writing documentation once

- Internet - web search engine, public documentation, public forums and mailing lists (more frequent with open source code than closed source)

- Project artifacts - API description, comments in source code, issue and bug reports, commit messages (often hard to search)

- Personal artifacts - work item/task descriptions (to-dos), personal emails, personal notes

- Knowledge Management Systems - project or organization wiki, intranet, experience database rarely used

- Information is often scattered across various tools and channels

These studies demonstrate the problem but do not propose a solution

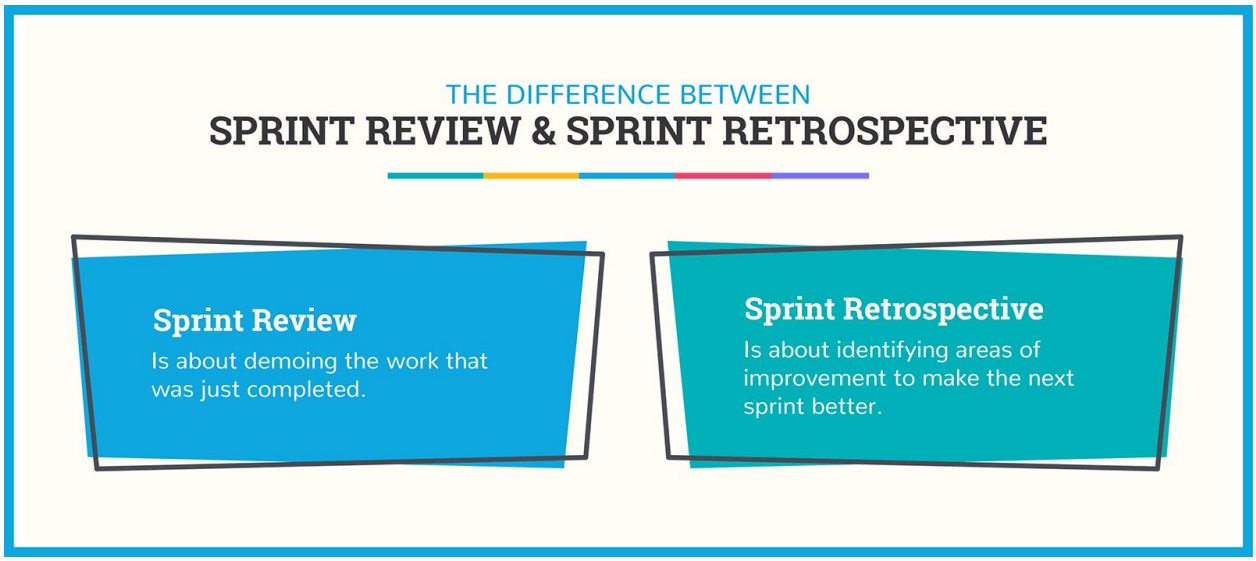
Does everyone know about the stackexchange network of knowledge sharing websites?

https://stackexchange.com/sites?view=list#traffic

stackoverflow is only one of numerous sites

Process Improvement can and should be part of any process, whether based on waterfall, agile, or "transitioning" between process models, but is most often associated with agile

Agile Retrospective - Continuous improvement: ideally an hour at end of every iteration. Not the same thing as *Post Mortem* when entire project ends



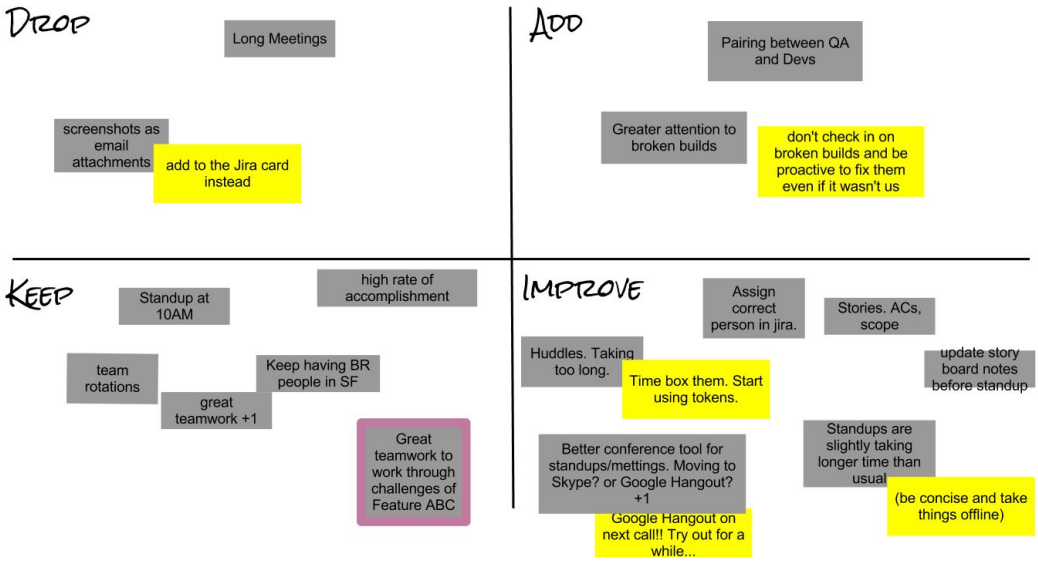
"Regardless of what we discover, we understand and truly believe that everyone did the best job they could, given what they knew at the time, their skills and abilities, the resources available, and the situation at hand."

- What process goals did we achieve in the past iteration
- What would we like to celebrate/remember, share appreciations of what other team members did
- What areas need improvement
- What do we have the time and energy to improve during the upcoming iteration

Set **SMART** goals = Specific, Measurable, Attainable, Realistic/Relevant, Timely

SaMoLo feedback - same as (reinforce), more of (encourage or increase), less of (discourage or reduce)

Art of the Possible - don't discuss anything that "could never work here", No buts - no "we could do xxx, but …"



Second Individual Assessment available later today, due next Tuesday December 11, 11:59pm. Will go over questions in class on Thursday

Final Iteration due Friday, December 14, 11:59pm

Final Demo due on "demo day" - Monday, December 17, 10am-3pm in 6LE1 CEPSR. This is the only time period in which Prof. Kaiser will be available to see demos (CVN will be handled separately). You need to give an elevator-pitch description of your project and a very brief demo, for about five (5) minutes. NO SLIDES! Any team that insists on showing slides or equivalent will receive 0 on this assignment. Your entire team does not need to be present.

Your team should also present a final demo to your IA mentor, for about 15 minutes. In this case the whole team should be present. This can be before, during, or after the "demo day" period.

After **both** demos have been completed, submit text entry stating the days/times and who attended for each of the two demos. The deadline is Monday December 17, 11:59pm