Lecture Notes
November 21, 2017

Second Iteration Development and Code Inspection due November 30
(includes preliminary demo for second iteration)

Legacy Codebases

Nearly every software engineer's first job starts by debugging someone else's code - but that's not truly "legacy" in most cases: the codebase is under active development by more experienced engineers who are available to answer questions

Legacy 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"

...ABOUT THAT BUG IN THE CUSTOMER ACCOUNTS SYSTEM...
YOU NEED TO FIND SOMEONE WHO UNDERSTANDS THAT LEGACY SYSTEM. IT'S WAY BEFORE MY TIME!
ANY SUGGESTIONS AS TO WHO?
SOMEONE WHO KNOWS WHAT THE HECK COBOL IS!!!

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

Example tools: jGRASP, Class Visualizer, MoDisco

Isolate business logic from plumbing code

Dependencies on outdated libraries, frameworks or APIs

Or, worse, homegrown plumbing

Write *characterization tests*: Accept at face value what the code does and document those behaviors with tests

Compare any changes against the original tests

Isolate sources of *technical debt* (arises in active codebases as well as legacy)

“Quick and dirty” design and/or implementation that needs to be redone later (the principal) and, in the meantime, makes other changes harder (the interest)

Can involve architecture, structure, duplication, test coverage, comments and documentation, potential bugs, complexity, code smells, coding practices and style

Example: copy/paste within same codebase

The principal: should instead refactor to share common code (this would take longer up front)

The interest: any bug found in copy/pasted code has to be fixed in all the copies - but how do you find all the copies, which may have diverged so the bug has to be found, fixed and tested separately for each (this takes longer later)

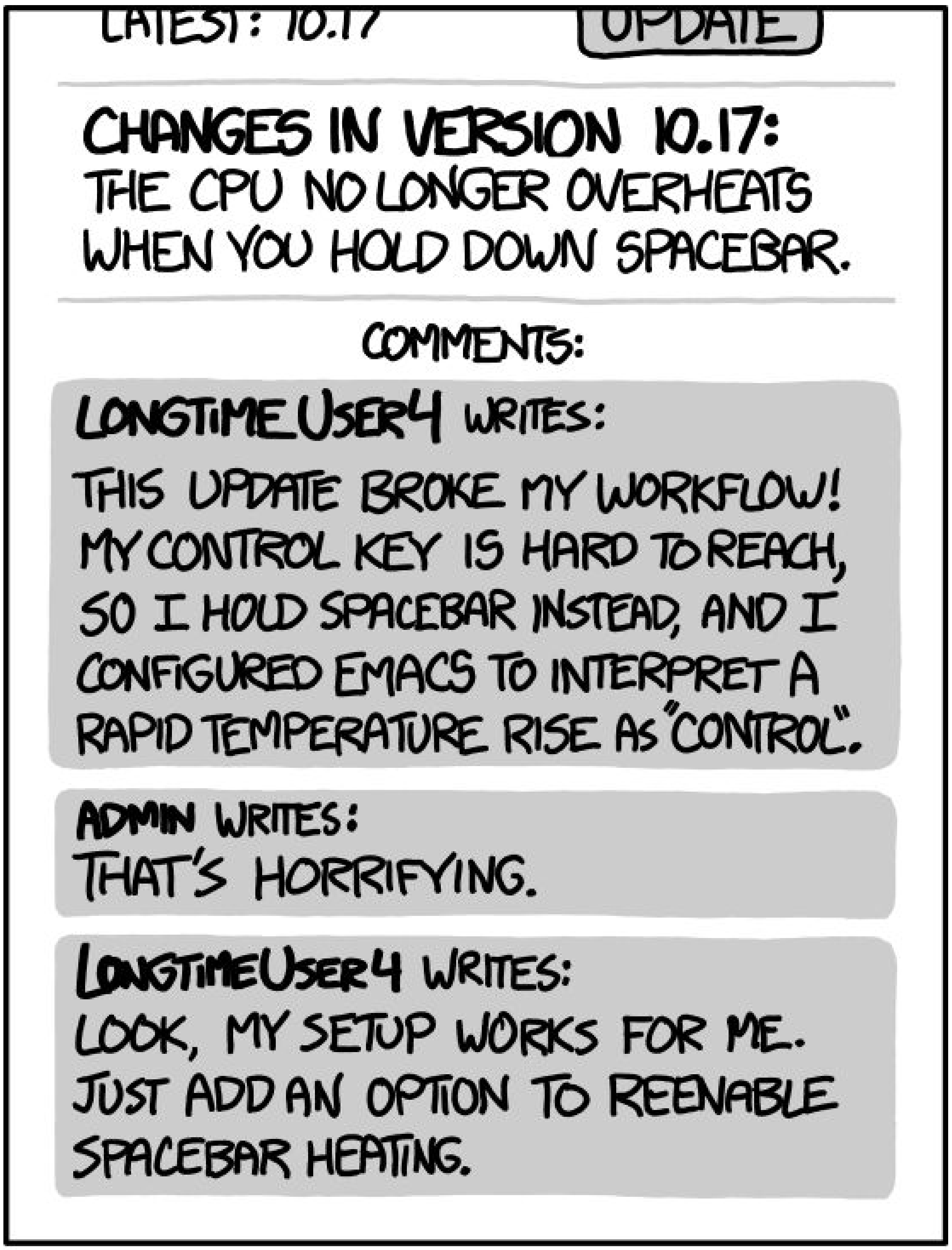
More interest: refactoring after the code has diverged (this takes longer later)

Changes to legacy codebases often defined by “ticket” and very closely scoped

Branch to make small change, unit test and code review, regression tests, 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


LATEST: 10.17
UPDATE
CHANGES IN VERSION 10.17:
THE CPU NO LONGER OVERHEATS WHEN YOU HOLD DOWN SPACEBAR.
COMMENTS:
LONGTIMEUSER4 WRITES:
THIS UPDATE BROKE MY WORKFLOW! MY CONTROL KEY IS HARD TO REACH, SO I HOLD SPACEBAR INSTEAD, AND I CONFIGURED EMACS TO INTERPRET A RAPID TEMPERATURE RISE AS "CONTROL".
ADMIN WRITES:
THAT'S HORRIFYING.
LONGTIMEUSER4 WRITES:
LOOK, MY SETUP WORKS FOR ME. JUST ADD AN OPTION TO REENABLE SPACEBAR HEATING.
EVERY CHANGE BREAKS SOMEONE'S WORKFLOW.

Program Understanding

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? (This applies to active as well as legacy.)

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

Strategies followed - overall approach, steps, and activities performed to reach goal

- 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 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 test 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)

- Take [transient] notes to reflect mental model and record knowledge - function names or signatures, control or data flow charts (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 as in user stores and use cases 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

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

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

stackoverflow is one of numerous sites