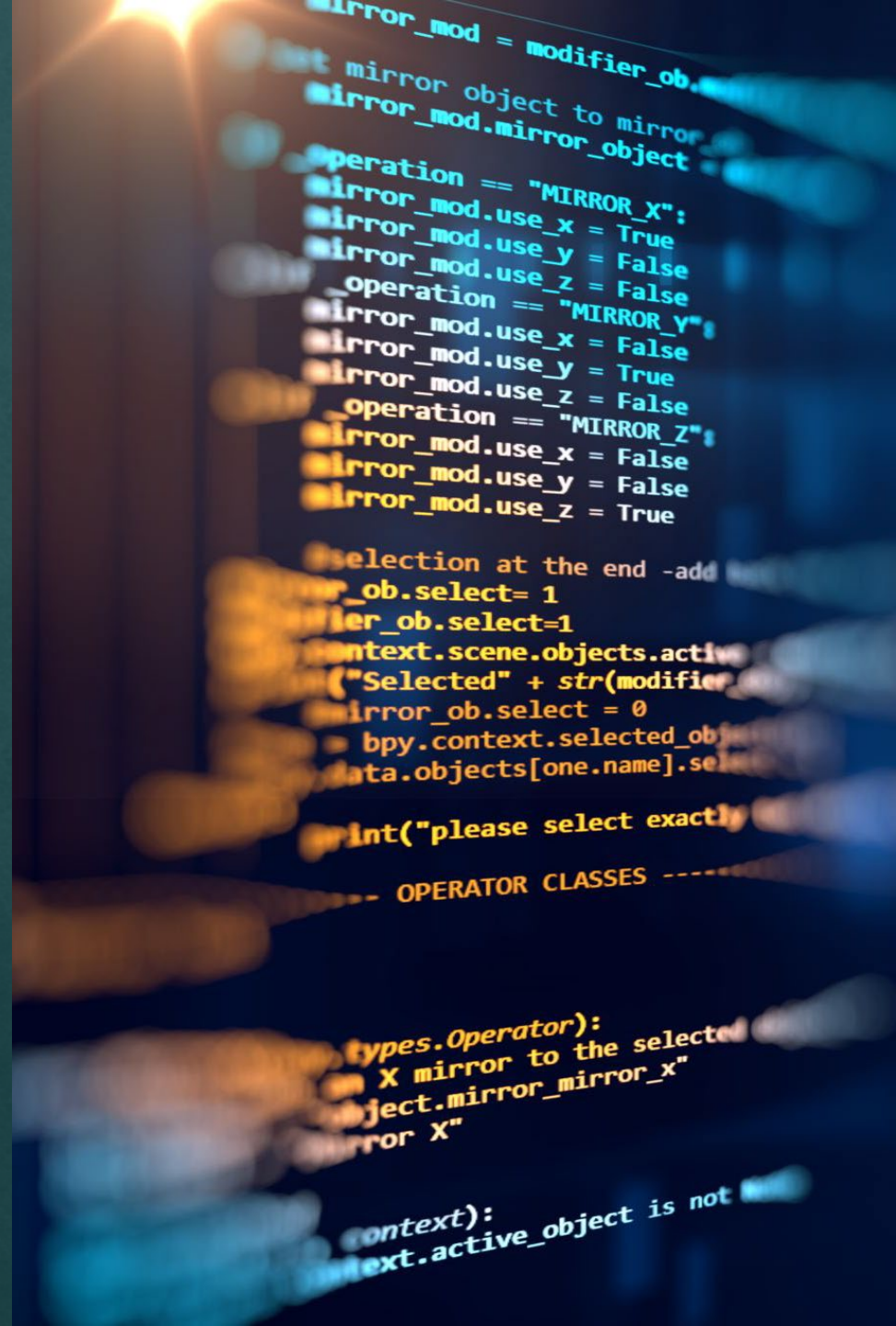
CS 428
Facts & Fallacies of Software Engineering (chapter 2)

WINTER 2023 – BRUCE F. WEBSTER



Chapter 2: About the Life Cycle Section 1: Requirements

- #23: One of the two most common causes of runaway projects is unstable requirements.
  - Most projects that spin out of control were never in control in the first place.
  - Customers/users aren't really sure of what problem they want solved or how specifically to solve it; when they saw a solution, they didn't like it.
    - "We want it to be just like the old system, but better."
  - Original focus was on strict requirements tracing and management; still popular today (in theory) but actual practice requires feedback and flexibility.
  - Agile/XP grew out of the effort to work with the customers continuously. Sometimes it works, sometimes it doesn't.

Chapter 2, Section 1: Requirements

- #24: Requirement errors are the most expensive to fix when found during production but the cheapest to fix early in development.
  - What could have been corrected for almost nothing early on becomes far more difficult to correct later. (Pages & rotating headlines)
  - Again, what is needed is proactive requirements reviews, preferably with actual/representative customers and users, early in the life cycle
  - Everyone acknowledges the problem, but there is little agreement on the best solution

Chapter 2, Section 1: Requirements

- #25: Missing requirements are the hardest requirement errors to correct.
  - Failure to elicit requirement through analysis and customer/user interviews.
  - Without knowing the full scope of the desired solution, you may make architecture and design decisions that block or hinder the missing requirements, thus requiring more extensive rework to accommodate them.
  - Plus, remember the 25%/100% rule: 25% increase in problem complexity => 100% increase in solution complexity.
  - The most persistent software errors – those that escape the testing process and persist into the production version of the software – are errors of omitted logic. Missing requirements result in omitted logic. It is difficult to test for something that isn't there.

Chapter 2, Section 2: Design

- #26: When moving from requirements to design, there is an explosion of "derived requirements" (the requirements for a particular design solution) caused by the complexity of the solution process. The list of these design requirements is often 50x [not 50% but 50 *times]* longer than the list of original requirements.
  - These are 'requirements' tied to our intended solution – in other words, in order to implement requirement X, we have a long list of specific implementation requirements.
  - This is a major factor behind the 25% problem => 100% solution complexity issue.
  - This requirements explosion is part of the reason that it is difficult to implement 'requirements traceability', even though everyone agrees that it is desirable to do so.

Chapter 2, Section 2: Design

- #27: There is seldom one best design solution to a software problem.
  - Key words are "one" and "best" – there are usually multiple solutions, and a subset of them may be equally desirable.
    - "In a room full of top software designers, if any two of them agree, that's a majority." – Bill Curtis
    - Studies to attempt to build a 'design toolkit' found instead that the design process is 'opportunistic' – i.e., discovering/inventing a novel solution.
  - This is also an inherent limitation on component/automated software creation – they may provide a functioning solution, but it's almost never going to be the only one, and it's unlikely to be the best (or even acceptable).

Chapter 2, Section 2: Design

- #28: Design is a complex, iterative process. The initial design solution will likely be wrong and certainly not optimal.
  - Solving the 'hard parts first' will help to going back and revising the overall design approach.
  - Design work tends to be heuristic ["seeking to discover"], trial-and-error.
  - Design work tends to be mental – sometimes without even writing/typing/drawing anything – because of the rate at which we consider, evaluate, and then reject or modify possible design approaches.
  - "For problems of substance, the XP/AD approaches harbor serious dangers."

Chapter 2, Section 3: CODING

- #29: Programmers shift from design to coding when the problem is decomposed to a level of "primitives" that the coder has mastered. If the coder is not the same person as the designer, the designer's primitives are unlikely to match the coder's primitives, and trouble will result.
  - It's easy to implement your own design. It's a lot harder to (correctly) implement someone else's design. This can be true whether the design is too high-level or too low-level.
  - This, by the way, is one of the major pitfalls in separation of designers and coders (cf. outsourced/offshore development).

Chapter 2, Section 3: CODING

- #30: COBOL is a very bad language, but all the others (for business data processing) are so much worse.
  - Most of you will never write a single line of COBOL (or FORTRAN or PL/1 or APL).
    - Yet there are 20 billion lines of COBOL in production, with over 1 billion new lines of COBOL written each year.
  - At some point, most of you will find yourselves working in a language that you consider bad, obsolete, even archaic, at some point or another.
  - The fact that C is among the top 5 most widely used programming languages speaks volumes as to how some languages come and go and others live on forever.
  - Be prepared to use languages that you don't want to or never thought you would.

Chapter 2, Section 4: Error Removal

- #31: Error removal is the most time-consuming phase of the life cycle.
  - Brooks: 50% of the total schedule, whether you plan for that or not.
  - Glass: 40% of the total schedule (and cites others who say 30%).
  - BUT: management never allows even 30% (much less 50%) of the schedule for error removal. They tend to see it on the order of 10% to 20%, if that much.
  - Hence, so many projects are over schedule, and many never stabilize at all.

Chapter 2, Section 5: Testing

- #32: Software that a typical programmers believes to be thoroughly tested has often had only about 55%-60% of its logic paths executed. Automated support can raise that to 85%-90%. It is nearly impossible to test 100% of the logic paths.
  - Testing is an act of compromise, and it is vital to make the proper compromise choices.
  - Most significant software products are released with errors [both known and unknown] remaining in them.
  - Too many organizations are unwilling to invest the time, money, and staffing in doing even minimal structural testing.

Chapter 2, Section 5: Testing

- #33: Even 100% test coverage would not be adequate. About 35% of software defects come from missing logic paths and another 40% from execution of a unique combination of logic paths. These will not be caught by 100% coverage.
  - Errors of omission: the logic to perform a required task is not in the program.
  - Errors of combination: the error appears only when a specific combinations of logic paths (a specific program state) occurs
  - Problem: program 'state space' can be astronomical (billions of unique states, etc.)

Chapter 2, Section 5: Testing

- #34: It is nearly impossible to do a good job of error removal without tools. Debuggers are commonly used, but others, such as coverage analyzers, are not.
  - Near-universal lack of respect for SQA in organizations
  - Front-end processes (analysis, architecture, design) are more visible and technically easy to understand
  - Back-end work (including testing) is "grubby" and technically hard to understand
  - Thus management tends to focus resources and prestige on front-end processes and to short-change back-end work
  - Most testing is (wrongly) done at the end of the life cycle, when the schedule pressures are most intense, and therefore time (and tools) to do things right are not permitted.

Chapter 2, Section 5: Testing

- #35: Test automation rarely is. That is, certain testing processes can and should be automated. But there is a lot of the testing activity that cannot be automated.
  - Again, test automation was seen for years as another 'silver bullet'
  - Test automation tools are necessary but not sufficient
  - Testing aspects that require human effort and judgment include:
    - What should be tested, and how (processes, tools, etc.)
    - Creation of test plans/cases and analysis of their collective results
    - Evaluation of reported defects, including prioritization, assignment, deferral
    - Making changes to the entire testing process (and personnel) based on results to date

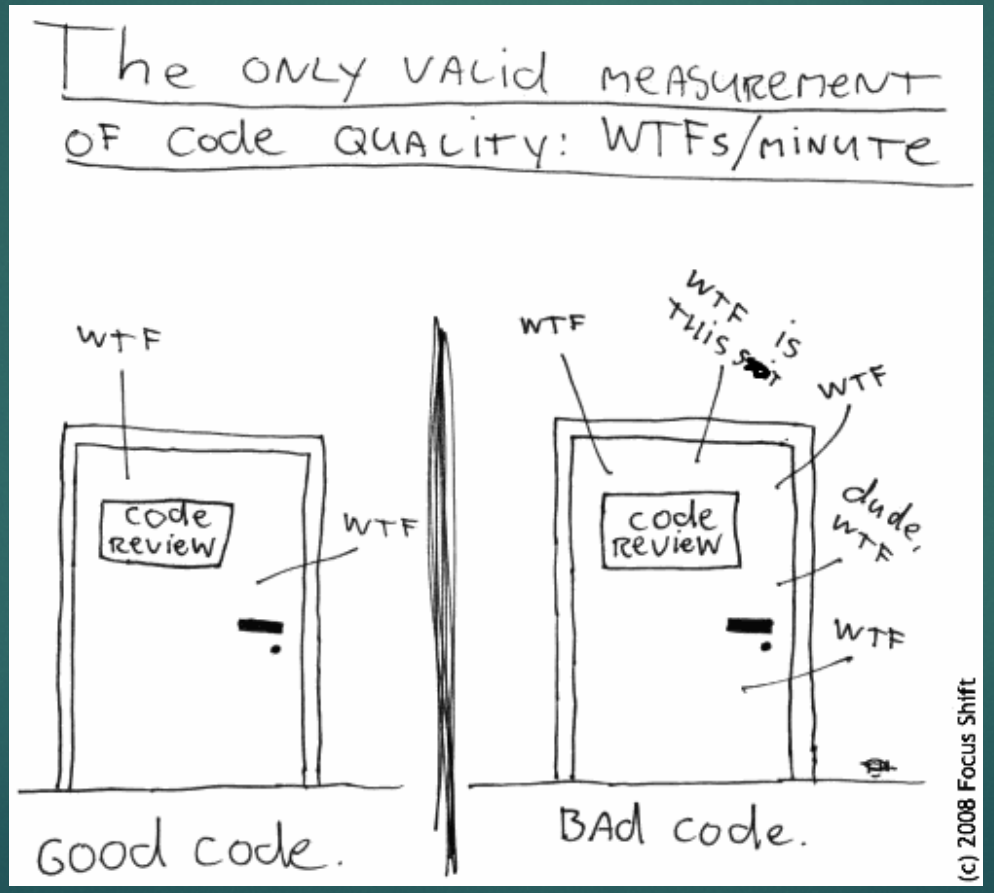
Chapter 2, Section 5: Testing

- #36: Programmer-created built-in debug code is an important supplement to testing tools.
  - Classic `#IF _DEBUG_` bracked code that prints incoming and outgoing parameters, dumps intermediate values, traces flow of control, etc.
  - Still one of the best and fastest ways of screening for and tracking down errors
  - Some IDEs/debuggers make this not strictly necessary, yet I often find it faster than setting breakpoints, stepping through code, etc.

Chapter 2, Section 6: Reviews and Inspections

- #37: Rigorous inspection can remove up to 90% of errors from a software product before the first test case is run.
  - Why don't we do this more often?
    - It's tedious
    - It can be embarrassing
    - It ties up multiple people at a time when you may face serious schedule pressures
    - Management doesn't understand its value and so discourages it
  - Simple version (via Tom Affinito): at the end of the day, grab another developer and do a 15-minute walkthrough of all the code changes you made that day
  - Also: one of the best debugging techniques is to drag someone else over and start explaining the bug and what you've looked at so far.
- #38: In spite of the benefits of rigorous inspections, they cannot and should not replace testing.

Chapter 2, Section 6: Reviews and Inspections



Chapter 2, Section 6: Reviews and Inspections

- #39: Post-delivery reviews ("retrospectives" or "post-mortems") are important. But most organizations don't do them.
  - Why? Good question.
    - To protect the guilty.
    - To avoid embarrassment.
    - Because everyone is being rushed onto a new project.
    - Management doesn't want to learn what it needs to change in its approach.
  - "The wisdom of the software field is not increasing."
    - Actually, I think we have lots of captured wisdom in books and articles.
    - The problem is: nobody reads them, and when they do, they don't want to believe them.

Chapter 2, Section 6: Reviews and Inspections

- #40: Peer [design/code] reviews are both technical and sociological. Paying attention to one without the other is a recipe for disaster.
  - Face it: it's hard to expose our work to others for criticism.
  - Also, programmers are known for having healthy egos. Sometimes, too healthy.
  - The pain of having our code reviewed – and critiqued – by our peers is a significant discouragement to frequent and thorough code reviews.
  - A few suggestions:
    - Don't have (non-technical) managers or unprepared engineers attend.
    - Have the review leader be someone other than the code's author.

Assignments for this week

- By midnight on Saturday (03/18)
  - Team status report #7 is due
- By start of next class period (03/20):
  - Read *Facts & Fallacies of Software Engineering*, chapters 3
  - Keep reading Webster #6
- In-progress demos and midterm review on 03/27 (two weeks)
- Midterm on 04/03 (three weeks)
- Final demos on 04/17 (five weeks)