CS 428
Facts & Fallacies of Software Engineering (chapters 2-3)

WINTER 2021

Bruce F. Webster

Chapter 2, Section 7: Maintenance

- #41: Maintenance typically consumes 40% to 80% (60% average) of software costs. Therefore, it is probably the most important life cycle phase of software.
  - Also, large organizations can spent 50% to 80% of their entire IT budget on maintenance
  - Yet it is often given low priority and attention
  - “Old hardware becomes obsolete; old software goes into production every night.”

Chapter 2, Section 7: Maintenance

- #42: Enhancement is responsible for roughly 60% of software maintenance costs. Error correction is roughly 17%. Therefore, software maintenance is largely about adding new features to old software, not fixing it.
  - "The 60/60 rule: 60% of software's dollar is spent on maintenance, and 60% of that maintenance is enhancement. Enhancing old software is, therefore, a big deal."
    - And it represents roughly 1/3rd (36%) of software's total budget.
  - 17% on fixings bugs
  - 18% on adaptive maintenance (getting software to work as environment changes)
  - 5% on preventive maintenance/refactor (paying down technical debt)

Chapter 2, Section 7: Maintenance

- #43: Maintenance is a solution, not a problem
  - Must like SQA, maintenance is sort of a “second-class” domain
  - Yet with proper investment of personnel, time, and tools, maintenance often becomes a far less expensive and less risky option than wholesale replacement
  - Maintenance of existing systems can also avoid the “I hate change” roadblock
  - But upper management often succumbs to the lure of the new car smell

Chapter 2, Section 7: Maintenance

- #44: In examining the tasks of software development vs software maintenance, most of the [lifecycle] tasks are the same – except for the additional maintenance task of “understanding the existing product.” This task consumes roughly 30% of of the total maintenance time and is the dominant maintenance activity. Thus it is possible to claim that maintenance is a more difficult task than development.
  - Key challenge: design a solution within the context of the existing product’s design.
  - Key challenge: figuring out what exploration and tradeoffs led to the current design.
  - Key challenge: finding out that the current design can’t support the proposed design.
  - Cf. Webster, “Controlling IT Costs: Using a Maintenance Architect” [link]

Chapter 2, Section 7: Maintenance

- #45: Better software engineering leads to more maintenance, not less.
  - The better a given system is designed and built, the longer it will stay in production.
  - The better a given system is designed and built, the easier it is to modify.
  - Therefore, a well-designed and well-built system will require more ‘maintenance’ – in terms of enhancements and lifespan – than a bad one.

Chapter 3 Quality
Section 1: Quality

- #46: Quality is a selection of attributes
  - For Glass: portability, reliability, efficiency, human engineering, testability, understandability, modifiability
  - Mine: reliability, performance, functionality, compatibility, lifespan, deployment, support, cost
  - Priorities and acceptable levels vary based on the project (and, frankly, your definition of those terms)
- #47: Quality is not user satisfaction, meeting requirements, meeting cost and schedule targets, or reliability[!]
  - Glass acknowledges he's contradicting himself a bit here on reliability – frankly, I think he's a bit incoherent
  - My definition of *reliability*: the system must carry out its functions without causing unacceptable errors or having an unacceptable downtime.

Chapter 3 Section 2: Reliability

- #48: There are errors that most programmers tend to make.
  - These range from the “if (a = b)” syntactic typos to omitting what glass calls “deep design details”
  - This is one reason why the open-source assertion “given enough eyes, all bugs are shallow” has turned out not to be (sufficiently) true
- #49: Errors tend to cluster
  - Various studies show that the majority of errors tend to occur in a small portion of the code
  - Glass doesn’t give a reason; my personal opinion is that this is a symptom of the “deferring hard problems” pitfall

Chapter 3 Section 2: Reliability

- #50: There is no single best approach to error removal.
  - Finding, tracking down, and repairing software defects is a very intensely intellectual and time-consuming exercise. There is no ‘silver bullet’.
- #51: Residual errors will always persist. The goal should be to minimize or eliminate severe errors.
  - Hence my definition of ‘reliability’: the system must carry out its functions without causing unacceptable errors or having an unacceptable downtime.
  - You also need to distinguish between the severity of the error and the likelihood of it occurring (and thus having an impact)

Chapter 3 Section 2: Efficiency

- #52: Efficiency stems more from good design than from good coding.
  - Think through likely bottlenecks beforehand; minimize your dependency upon them.
  - Understand that the memory/code tradeoff (pre-computed vs procedurally-derived, cached vs fetch on demand) is very real.
  - “Premature [code] optimization is the root of all evil.” – Don Knuth
    - Make sure you understand how to correctly solve the problem before trying to optimize.
- #53: High-order languages, with appropriate compiler optimization, can be about 90% as efficient as comparable assembler code.
  - Similar debate about bytecode/interpreted languages vs compiled languages.

Chapter 3 Section 2: Efficiency

- #52: Efficiency stems more from good design than from good coding.
  - Think through likely bottlenecks beforehand; minimize your dependency upon them.
  - “Premature [code] optimization is the root of all evil.” – Don Knuth
    - Make sure you understand how to correctly solve the problem before trying to optimize.
- #53: High-order languages, with appropriate compiler optimization, can be about 90% as efficient as comparable assembler code.
  - Similar debate about bytecode/interpreted languages vs compiled languages.
- #54: There are tradeoffs between size and time optimization. Often, improving one degrades the other.
  - Understand that the memory/code tradeoff (pre-computed vs procedurally-derived, cached vs fetch on demand) is very real.
  - On the other hand, excessive use of memory can sometimes lead to poor speed performance as well.