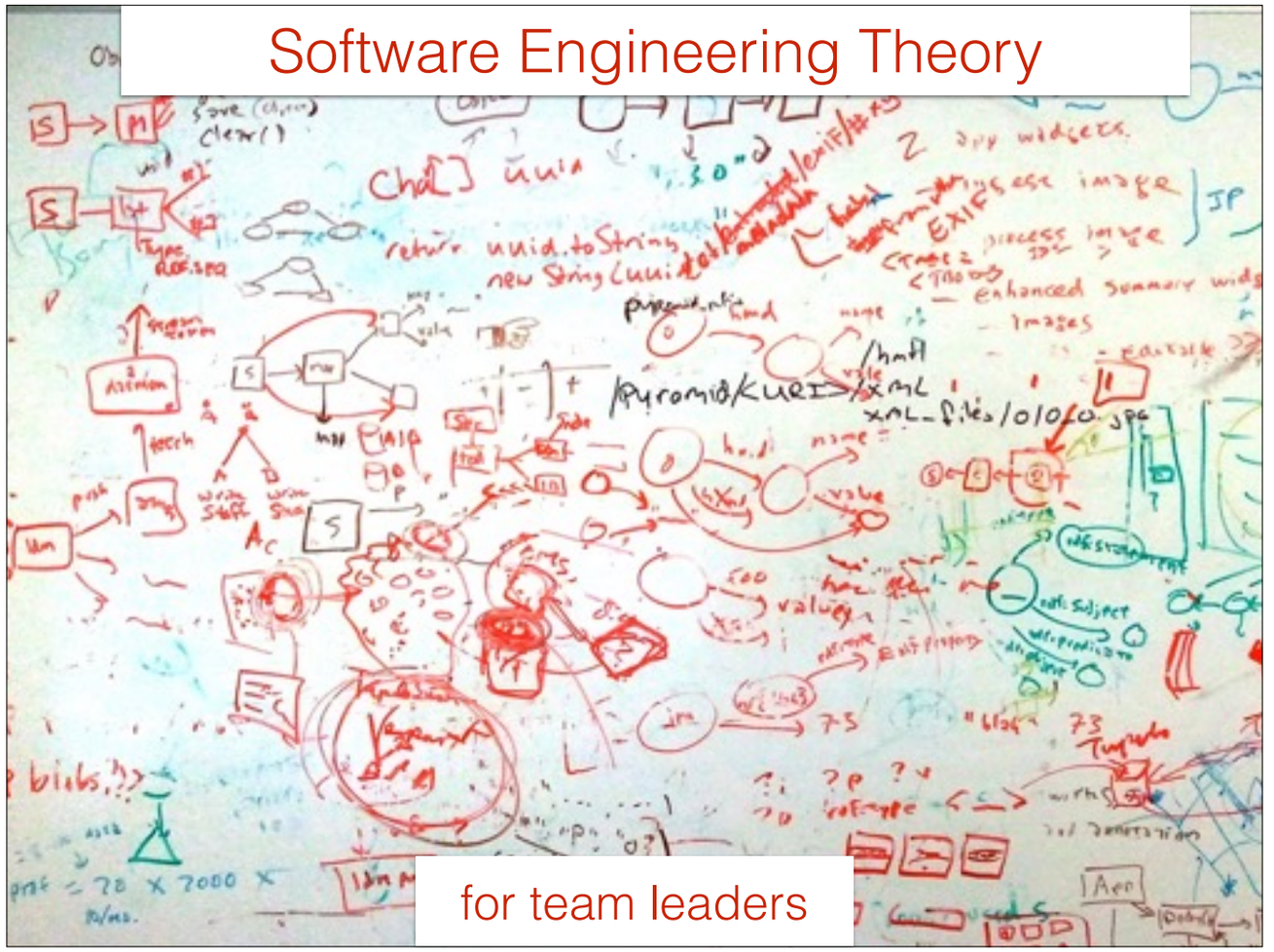
Software Engineering Theory



for team leaders

"I believe that failure is less frequently attributable to either insufficiency of means or impatience of labour than to a **confused understanding of the thing actually to be done**."

–John Ruskin

Three Pioneers

- Fred Brooks (1960s-90s)
- Manny Lehman (1970s-2002)
- Bob Glass (1970s-2000s)

All three began their careers in industry,
and all three ended as academics.

Meir Lehman

Fred Brooks

- The Mythical Man Month (1975)
  - Brooks' Law: "Adding manpower to a late software project makes it later."
- No Silver Bullet (1987)
  - "There is no single development… which promises even one order of magnitude improvement… in productivity"
  - There is no Moore's Law equivalent for software.

Software is **complex** because programmers continue to define and use new levels of abstraction. This is the reverse procedure from the physical sciences, where practitioners have spent generations by doing the exact opposite. Software becomes more complex as it is forced to **conform** to other systems, some of which are arbitrary (and arbitrarily complex) such as human social systems. Software **evolves**, it is changeable throughout its life cycle. Finally, the geometric abstractions of software are mostly difficult or impossible to visualize. They are, to use Brooks' term, **invisible**.

Fred Brooks

Software has four “inherent properties”:

- complexity
- conformity
- changeability
- invisibility

Software is **complex** because programmers continue to define and use new levels of abstraction. This is the reverse procedure from the physical sciences, where practitioners have spent generations by doing the exact opposite. Software becomes more complex as it is forced to **conform** to other systems, some of which are arbitrary (and arbitrarily complex) such as human social systems. Software **evolves**, it is changeable throughout its life cycle. Finally, the geometric abstractions of software are mostly difficult or impossible to visualize. They are, to use Brooks’ term, **invisible**.

Manny Lehman

Lehman defined eight laws relating to evolving systems. Arguably the two most important echo Brooks:

- The **Law of Continuing Change** notes that a software project evolves continuously (Brooks' changeability)
- the **Law of Increasing Complexity** suggests that a software project becomes less structured, or more complex, with time (Brooks' complexity and conformity).

Meir M. Lehman (1980). "Programs, Life Cycles, and Laws of Software Evolution". Proc. IEEE, vol. 68, no. 9, pp. 1060-1076.

One may also consider Lehman's **Law of Declining Quality**, which states that a project's quality decreases during its lifecycle unless specific efforts are made to address it, although that would appear to be a consequence of increasing complexity.

Bob Glass

- Authored more than 200 papers and 25 books (Robert L. Glass).
- Focused on **measuring software quality**, and recorded many **case studies** of software failures.

Bob Glass

55 “facts” in 4 themes:

- complexity
- poor estimation coupled with schedule pressure
- a disconnect between software managers and technologists
- the delusion of hype

Robert L. Glass. Facts and Fallacies of Software Engineering. Pearson Education, Boston, MA, 2003.

“I would suggest that practitioners considering some tool, technique, method or methodology that is at odds with one or more of these facts should beware of serious pitfalls in what they are about to embark on.”

–Bob Glass

Lehman's Laws

Lehman noted three types of software systems:

- Software systems defined entirely by their **specification** (S-type systems);

- Software systems defined entirely by their **procedures** (P-type systems); and

- Software systems written to perform some real-world activity, which must then **evolve** over time to match that activity (E-type systems).

Unsurprisingly, E-type systems form the vast bulk of software systems currently extant or planned. ***Lehman's Laws of software evolution apply to E-type systems.***

Of the three, it is Lehman that has long been recognized as the hallmark of the field.

Perhaps the reason for this is simply that Lehman's laws are **more detailed than Brooks'**, but **easier to understand than the many facts put forth by Glass**. That they all point in the same direction is comforting.

Lehman's Laws

1. **Continuing Change**: An E-type system must be continually adapted or it be- comes progressively less satisfactory.

2. **Increasing Complexity**: As an E-type system evolves, its complexity increases unless work is done to maintain or reduce it.

3. **Self Regulation**: E-type system evolution processes are self-regulating with the distribution of product and process measures close to normal.

4. **Conservation of Organizational Stability** (invariant work rate): The average effective global activity rate in an evolving E-type system is invariant over the product's lifetime.

Lehman's Laws

5. **Conservation of Familiarity**: As an E-type system evolves, all associated with it, developers, sales personnel and users, for example, must maintain mastery of its content and behaviour to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.

6. **Continuing Growth**: The functional content of an E-type system must be continually increased to maintain user satisfaction over its lifetime.

7. **Declining Quality**: The quality of an E-type system will appear to be declining unless it is rigorously maintained and adapted to operational environment changes.

8. **Feedback System**: E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

The Building Metaphor

“Software is largely a service industry operating under the persistent but unfounded delusion that it is a manufacturing industry.”

–Eric Raymond

Fred Brooks first complained about the building metaphor in 1958!

> “The building metaphor has outlived its usefulness. It is time to change again. If, as I believe, the conceptual structures we construct today are too complicated to be accurately specified in advance, and complex to be built flawlessly, then we must take a radically different approach.”
>
> –Fred Brooks, 1987

The approach that Brooks intended to take was to ***grow software, rather than build it***. He specifically used evolutionary metaphors.

Note how this **presaged Scrum and Agile** development.

Software Maintenance

“Computer Science is the only discipline in which we view adding a new wing to a building as being maintenance.”

–Jim Horning

Software Maintenance

- Software has long been one of the most complex structures created by humans.

  Brooks, F.P. (1987). No Silver Bullet: Essence and Accidents of Software Engineering, Computer, IEEE Computer Society Press, vol. 20, no. 4, pp. 10-19.

- and often the most expensive portion of engineered systems that include it, as foreseen in the 1970s by Barry Boehm.

  Boehm, B. (1973). Software and its Impact: A Quantitative Assessment. Datamation, 19, pp. 48- 59.

- Software maintenance is, often by far, the largest portion of the software lifecycle in terms of both cost and time.

  Glass, R. L. (2003). Facts and Fallacies of Software Engineering. Pearson Education, Boston, MA.

Software Maintenance

- Maintenance failure is exacerbated by the **rapid divergence of software systems and information about them**. This divergence is typically a consequence of the loss of coupling between software components and system metadata.

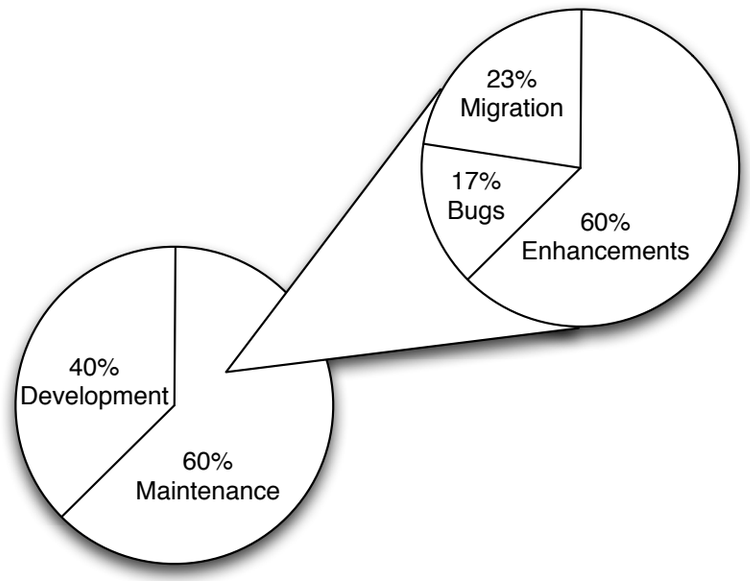
Van Doren, E. (1997). Maintenance of operational systems - an overview. Carnegie Mellon Software Engineering Institute, http://www.sei.cmu.edu/str/descriptions/mos_body.html, accessed July 29, 2007.

A common euphemism for a programmer is
“developer”, not “maintainer”.

“Maintenance is… the normal state”

–Kent Beck

60/60 Rule

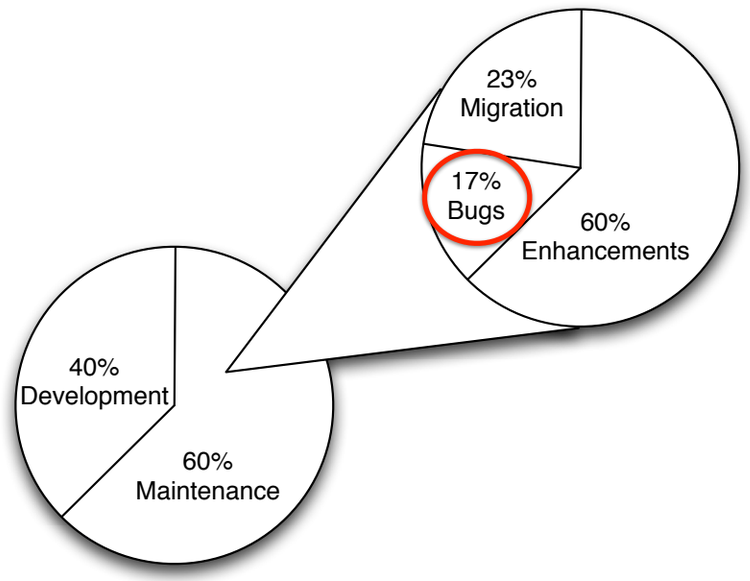


Glass, R. L. (2003). Facts and Fallacies of Software Engineering. Pearson Education, Boston, MA, pp. 115-118.

One of the few proposed laws of software engineering.

Understanding changes to be made is a major activity during maintenance. 30% of total maintenance time spent on understanding the existing product. [Glass 2003, pp. 115-124 and Shere 1988, pp. 61]. Martin [Martin 1983, pp.4] found that up to 80% of maintenance activities relate to changing requirements.

60/60 Rule



Glass, R. L. (2003). Facts and Fallacies of Software Engineering. Pearson Education, Boston, MA, pp. 115-118.

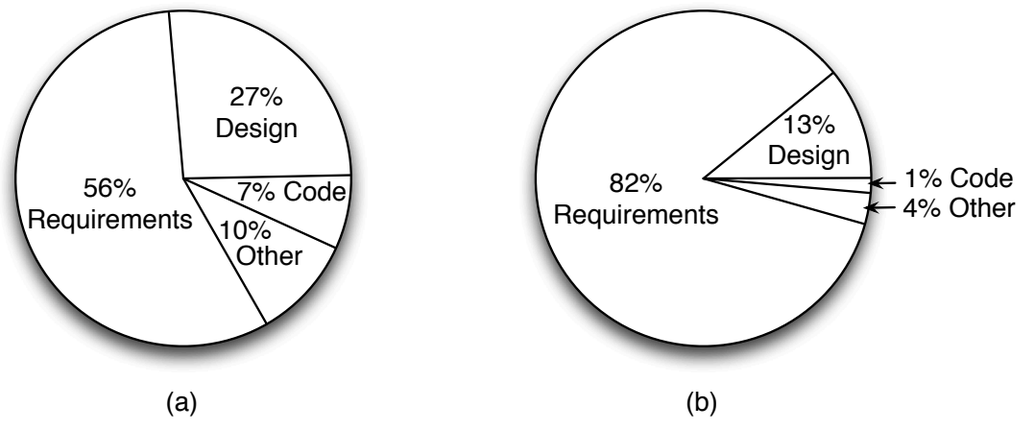
So, should team leaders focus on squashing bugs? Or focus on making code easy to change?

Typical Developer Time

Task	Development	Maintenance
Understanding requirements	20%	15%
Design/undesign	20%	30%
Implementing	20%	20%
Testing & debugging	40%	30%
Documentation		5%

Cross-industry, many sizes/languages/etc.

Bugs and Costs



(a) The greatest number of bugs occur in requirements specification and design, (b) The greatest cost is incurred in correcting requirements bugs
(After Martin, J. and McClure, C. (1983). Software Maintenance, The Problem and Its Solutions. Prentice Hall, Englewood Cliffs, NJ., Figure 12-3).

Wood's Law

"A Lehman E-type software system is defined to be in maintenance failure when knowledge of its design and implementation is lost. Avoidance of maintenance failure therefore entails constant actions toward knowledge retention or knowledge re-acquisition."

Wood's Law

Maintenance failure is caused by the loss of knowledge of a software product.

Solution: DON'T DO THAT :)

A software project dies when it cannot recover from "maintenance failure", so focusing effort on maintenance activities that prolong the life of a project makes the most economic sense.