Reverse Engineering of Legacy Code Exposed
Bruce W. Weide, Wayne D. Heym, and Joseph E. Hollingsworth
ICSE '95. Proceedings of the 17th International Conference on Software Engineering, pages 327-331

Abstract
- Reverse engineering of large legacy systems generally cannot meet its objectives because it cannot be cost-effective. [great - another pessimist!]
  - Costly to "understand" the old code
  - Reengineering cannot produce new code than can be reverse engineered in the future.

Introduction
- Large systems deteriorate due to "unanticipated long-range 'weird interactions'"
- Options for system replacement: (1) build new system from scratch; (2) reengineer it.
- Goal: to understand the whats, hows, and whys of the legacy system as a while that its code can be re-engineered to meet new requirements on behavior, performance, structure, system dependencies, etc.

Reverse Engineering of Legacy Code is Intractable.
- "There seems to be a general agreement that, in practice, reverse engineering of legacy code is at least quite expensive." [Parnas!] [like I said - a pessimist]
- Effort required to reverse engineer a system is related to the effort required to formally verify its functional correctness.

Forward Engineering is Not a Solved Problem
- Forward engineering techniques must be perfected that effectively prevent "software rot"
- "Software engineers do not know how to design and build truly modular systems when starting from scratch, let alone when starting from legacy code." [Obviously devoid of any real-world work on large commercial systems. This guy likes to quite himself a lot]

Observations and Implications
- There is a really pessimistic belief that the central business of IT is to upgrade poorly constructed systems.
- "for all large legacy systems, program verification/reverse engineering is prohibitively expensive."
• Posits the finding of "trouble spots" in legacy code as a primary area for automation.

**The Nature of the Reverse Engineering Task**

• Successful reverse engineering of legacy systems entail:
  - Identifying functional components and the roles they play
  - Creating a valid explanation of how and why the behavior of the higher-level systems arises from these components
  - "Components" are not necessarily the structural components, but functional components. [transactions are not necessarily modules, for instance - why we need slicing]

**Testing vs. Proving**

• People make hypotheses about the code/changes, but do not check them in a decisive way.
• Approach can only disprove hypotheses, not prove them.
• Approximation is not sufficient to achieve the ultimate objective of reverse engineering.

**Substantive Hypotheses**

• Reverse engineering hypothesis H for system S should be substantive:
  - Effective
  - Comprehensive
  - Concise
  - Independent
  - Systemic

**The Intractability Result**

• The required explanation (requirement 2 of reverse engineering) is what is intractable.

**Source Code is a Compact Representation of Behavior**

• One intended result of modularity is the ability to reason modularly about a program. [This assumes low coupling and high cohesion.]
• Even a small program can have arbitrarily long execution sequences due to looping and recursion.
• Must be possible to reason about the effect of all code without resorting to execution or pseudo-execution.
Problems Result from Failed Attempts at Modularity

- Coupling through side effects and aliased variables, arrays, pointers, dynamic storage management.
- Non-Modularity Premise: Every legacy system is hard to maintain because, in some crucial places, it has been designed or coded so that modularity is not achieved.

Conclusion

- Given real legacy code, the time required to show the validity of a proposed explanation for why it exhibits any significant system-level behavior is at least exponential to the size of the source code.