

---

# The Mythical Man-Month

---

Jiakang Lu

Qual Study Group

May 29<sup>th</sup>, 2007

---

# Overview

- Published 1975, Republished 1995
  - Experience managing the development of OS/360 in 1964-65
  - Central Argument
    - Large programming projects suffer management problems different in kind than small ones, due to division of labor.
    - Critical need is the preservation of the conceptual integrity of the product itself.
  - Central Conclusion
    - Integrity achieved through exceptional designer.
    - Implementation achieved through well-managed effort.
-

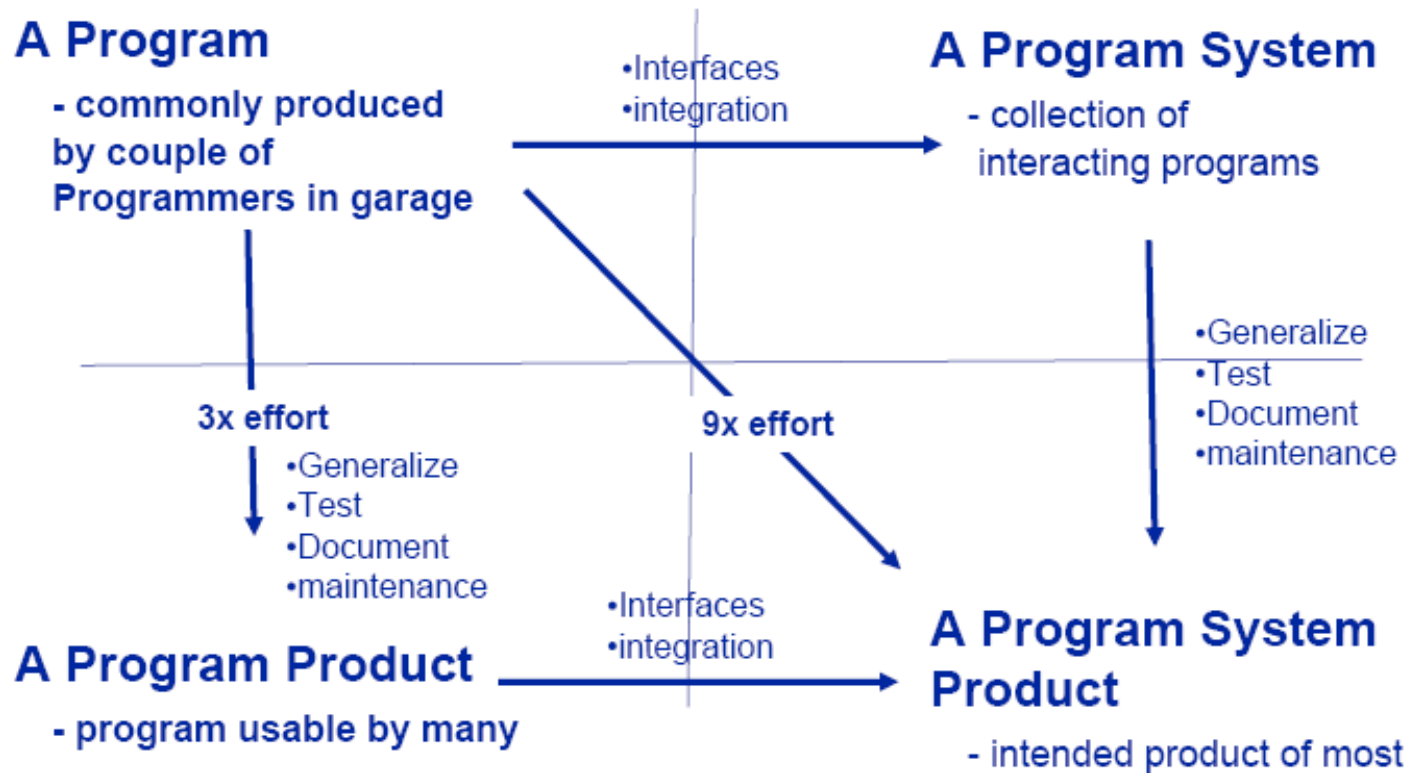
---

# Chapter 1: The Tar Pit

## ■ The Tar Pit

- Inability to finish projects on time, within budget
    - Software is like a tar pit: the more you fight it, the deeper you sink!
  - Causes:
    - Gradual growth of size and complexity of software projects
    - Lack of understanding SE issues
-

# The Tar Pit



Evolution of the programming systems product

---

# Chapter 2: The Mythical Man-Month

## ■ Myths

- 1. Optimism
  - 2. Man-Month Myth
  - 3. System Test
  - 4. Gutless Estimating
  - 5. Regenerative Schedule Disaster
-

---

# 1. Optimism

- Programmers nature seems to be optimistic
  - Medium of execution has no limitations
  - Ideas are faulty
  - Optimism is probabilistic
    - “All might go well” but combination of potential sources of failure mounts up quickly
-

---

## 2. Man-Month Myth

- Man-Month only applies for independent subtasks
  - Communication costs
    - $n(n-1)/2$
  - Notes
    - True: Project Cost is Proportional to Number of Personnel
    - False: Progress is Proportional to Number of Personnel
    - Fallacy is in an assumption of subtasks requiring no communication (independent).
-

---

# 3. System Test

- Time for testing depends on number of bugs
    - integration bugs generally anticipated to be zero
  - Suggested schedule guide
    - 1/3 planning
    - 1/6 coding
    - 1/4 unit testing
    - 1/4 system testing

} Usually on time for these

} Because no planning for these
-

---

# 4. Gutless Estimating

## ■ Cause

- Bidding to meet desire of customer
- Lack of data, guidance to support doing better

## ■ Solution

- Develop & Publicize data
    - Productivity
    - Bug incidence and source
    - Estimation rules
-

---

# 5. Regenerative Schedule Disaster

- Brooks's Law
    - “Add manpower to a late software project makes it later”
  - Adding people to a software project:
    - Increases communication requirement
    - Requires education
    - Requires repartitioning of work
-

---

# Chapter 3: The Surgical Team

## – Mill's Suggestion

- A small sharp team is best
    - as few minds as possible
    - but too slow for really big systems
  - *A chief-programmer, surgical-team* organization offers a way to get the product integrity of few minds and the total productivity of many helpers, with radically reduced communication
-

---

# The Surgical Team -- Notes

- Based on 10:1 ratio
  - Scales up through hierarchical division of problems
  - Single surgeon on problem (subproblem) maintains *conceptual integrity* of design
  - Requires good communication among surgeons
-

---

# Chapter 4: Aristocracy, Democracy and System Design

- Conceptual integrity system has same look and feel throughout
  - Problem: How do we keep things consistent same look and feel when different teams work on different parts?
  - Solution:
    - Hire an architect
    - Record the vision
    - Enforce constraints
-

---

# Conceptual Integrity

- Conceptual Integrity = consistency and accuracy of model
  - Conceptual Integrity implies ease of use
  - Achieved more easily
    - with fewer designers (a surgeon/architect-based approach)
    - with fewer functions
  - Ratio of productivity gain to cost [of system and training] usually decreases with increased functionality
-

---

# Chapter 5: The Second-System Effect

- The first system is minimal and solid great ideas are saved for the second one
  - ALL the great ideas are stuffed into the second one
  - Result:
    - Small percentage increase in functionality and quality
    - Large percentage increase in size and complexity
-

---

# Suggested Remedies

- Solution: Disciplined architecting
    - Consider cost/benefit tradeoff for each innovation
    - Listen to implementers
      - Suggest don't dictate
      - Keep an open mind
      - Keep discussions quiet
      - Keep ego out of it
-

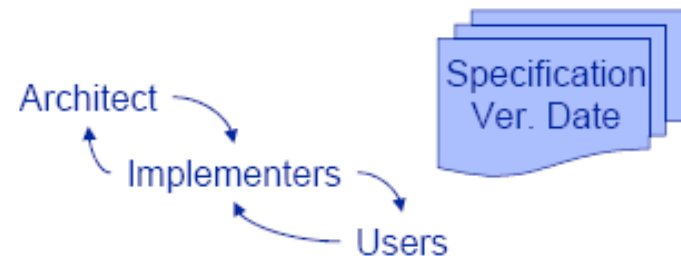
---

# Chapter 6: Passing the Word

- Communication is the root of most problems
  - “How is a team of architects to make sure that the horde of implementers knows just what to do?”
  - **A MANUAL!**
-

# The Manual

- The Written Specification is the chief product of the Architect
  - Example: ECS (EOSDIS Core System)
  - Prescription: Documenting Software Architectures: Views and Beyond
    - Clements et al., 2002
  - Must describe:
    - What user sees
    - NOT what they don't



---

# The Manual (2)

- Must read as a whole
  - Must have conceptual integrity
  - Have many architects but only one or two writers



---

# Formal Notations

- Natural language – imprecise and allows ambiguity
  - Formal languages and notations exist because they support precise description
    - Good as base for discussion
    - Shows gaps so tend to be complete
  - But
    - They lack richness of prose
    - They are hard to both write and to read
-

---

# Formal Specs vs Implementation

- Complete formal specification defines the implementation
  - But a truly complete spec is the implementation
  - Implementation can also define the spec
    - Reengineering
  - Testing can be used to define the behavior
    - Problems: testing is limited, all tests must be valid, extra stuff might be captured
    - Good Points: All questions can be answered
-

---

# Meetings

- Two levels of meetings
    - Hold weekly meetings of the high-level stakeholders
    - Hold periodically meetings of ALL stakeholders
-

---

# Chapter 7: Tower of Babel

- The first engineering fiasco
  - Why didn't they finish?
    - The Tower of Babel project failed because of lack of *communication* and of its consequent, *organization*.
-

---

# Enhancing Team Communication

- Informally

- Lots of telephone calls

- These help create common understandings

- Formally

- Team meetings

- These should be technical exchanges, not managerial meetings
- They are for uncovering misunderstandings

- Project workbook

- This is the record of the project – everyone can consult it.
-

---

# Team Structure

- Managerial structure tree-like
    - Hierarchical authority and responsibility
    - Necessary parts
      - Mission
      - Producer (project manager)
      - Technical director (architect)
      - Schedule
      - Division of labor
      - Architectural description
  - Communication structure network
-

---

# Chapter 8: Calling the Shot

## -- Developer productivity

- Not all working hours are devoted to a project
    - Interruptions include meetings, high-frequency, unrelated tasks.
  - Productivity is constant in the units.
  - Higher-level tools imply higher productivity
-

---

# Chapter 9: Optimization

- Discuss variety of issues related to size meeting memory size requirements
    - Calls for higher level concern about such issues
    - Calls for reasoning carefully about such issues and recording the rationale
      - Inlined code takes less space but that is good for readability
  - Today: Early focus on quality attributes
-

---

# Chapter 10: The Documentary Hypothesis

- Seems to be core set of documents required of any project type
    - Objectives
    - Product specification
    - Schedule
    - Budget
    - Space allocation
    - Organization chart
      - This is inextricably intertwined with the interface specification
  - Why write it down
    - Precision, communication, support progress checks
-

---

# Summary (Chapter 6 -10)

- Communication must happen
    - The left hand and right hands must know what each other are doing.
  - Meetings support communication and allow joint decisions to be made
  - Creating documentation encourages communication
  - Having documentation supports communication
-

---

# Chapter 11: Plan to Throw One Away

## -- Pilot Systems

- *Plan to throw one away; you will, anyhow.*
    - Save your customers frustration
    - It will save you embarrassment
  - Accept the fact, change is unavoidable
    - Programmer's job is to satisfy users' desires and desires evolve with programmer products
  - Use of configuration management is required
  - Reducing redundancy and automated support for documentation helps
-

---

# On the other hand...

- Too much acceptance of change is poor management
  - Threatening organizational structure encourages lack of documentation
  - Flexibility in assignments is a must
    - Have the best person on the job at all times
    - Keep management and technical people changeable  
“Member of the Technical Staff”
    - Suggest overcompensating for move from managerial to technical position to overcome perceived hierarchy
-

---

# Calculate Change

- **Uncalculated change**
    - As opposed to Hardware, Software Maintenance is usually unplanned change
      - No cleaning, no lubrication
      - Usually means fixing mistakes and adding functionality
    - Change in architecture means change visible to user
-

---

# Survival of the Fittest

- What Brooks is referring to is what we now call *Software Evolution*
    - It is inevitable that people desires grow with their knowledge of what is possible
    - Their ability to find bugs grows with their willingness to experiment with features
    - System designers constantly work to keep the users happy
    - Systems evolve as they are used
-

---

# Problem with Fixing Things

- Fixing a bug usually breeds more bugs
    - Most changes (bug fixes) have far-reaching effects
      - Lack of documentation
      - Lack of modularity
      - Lack of cohesion on programming team
      - Chief programmer got hit by a bus
  - Call for *Regression Testing* techniques
    - Costly!
  - Programs become “brittle” with time – they “age”
    - Design gets lost
    - Modularity decreases
    - Software systems do die
-

---

# Chapter 12: Sharp Tools

- Hording of tools is natural but unproductive
    - Communication is #1
    - Call for central IT staff
  - Developing on machines other than “target” machines leads to unexpected bugs – memory size, CPU speed, bus speed
    - Broader implications for “target environment”
-

---

# Chapter 13: The Whole and the Parts

## -- System Integration

- Create piece, test, release for integration
  - Integrate, test, ...
  - **Control is the key**
    - Who has what
    - Who knows who has what
-

---

# Glimpse of the Future (1/2)

- High-Level Languages and Interactive Programming
    - High-level languages
      - Productivity
      - Increased debugging speeds (increased readability)
      - Optimizing compilers make readability good for humans and efficient code available to machines
-

---

# Glimpse of the Future (2/2)

- Batch vs. Interactive
    - Batch – all data input, result is output
    - Interactive – a conversation (remember, he'd never seen a mouse or a “window”)
  - Interactive Programming
    - Required OS support (protections, interleaving, etc.)
    - Interactive tools require high-level language
    - Debugging is naturally interactive
-

---

# Designing the Bugs Out (1/2)

- Bug-proofing the Definition a.k.a. Conceptual Integrity
  - Testing the Specification
  - *Top-down Design* (Wirth)
    - Clear idea of modularity makes requirements easier to define
    - Partitioning and independence avoids bugs
  - *Structured Programming* (Dijkstra)
    - Accepting constraints allows analyzability
  - Component Debugging and Reuse
  - Interactive Debugging
  - Use Debugging Scaffolding
    - as much as 50%
-

---

# Designing the Bugs Out (2/2)

- Control Changes
    - Versioning
  - Add One Component at a Time
  - Quantize Updates
    - Large and Infrequent (recommended)
    - Small and Frequent
-

---

# Chapter 14: Hatching a Catastrophe

“How does a project get to be a year late?  
One day at a time.”

- Milestones, Milestones, Milestones
    - Can't meet a schedule without having one
    - Milestones must be:
      - Concrete
      - Specific
      - Measurable events
      - Precisely defined
    - As opposed to most programming related tasks meeting milestones is all-or-nothing
    - “Fuzzy” milestones are hard to live with for everyone
-

---

# Chapter 15: The Other Face

## -- Documentation

- End Users
    - To use a program
  - Acceptance Cases
    - To believe a program
  - Modification [Flowchart]
    - To modify a program
  - Self-Documenting Programs
    - Greatest use and power in high-level languages used with on-line systems
-

---

# Summary (Chapter 11 – 15)

- Prototypes are good but they must be close to the real thing
  - Tools are a must but can't take too long to learn to use
  - System integration is REALLY hard
  - Project management is a MUST
  - Documentation needs to be Automatic
-

---

# Chapter 16: No Silver Bullet

- “There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.” -- Fred Brooks, 1986
-

---

# Why? Essence and Accidents

- Brooks divides the problems facing software engineering into two categories
    - **essence**
      - difficulties inherent in the nature of software
    - **accidents**
      - difficulties related to the production of software
  - Brooks argues that most techniques attack the accidents of software engineering
-

---

# An Order of Magnitude

- In order to improve the development process by a factor of 10
    - the accidents of software engineering would have to account for 9/10ths of the overall effort
    - tools would have to reduce accidents to zero
  - Brooks
    - doesn't believe the former is true and
    - the latter is highly unlikely, even if it was true
-

---

# The Essence

- Brooks divides the essence into four subcategories
    - complexity
    - conformity
    - changeability
    - invisibility
-

---

# Complexity (1 / 3)

- Software entities are amazingly complex
    - No two parts (above statements) are alike
      - Contrast with materials in other domains
    - They have a huge number of states
      - Brooks claims they have an order of magnitude more states than computers (e.g. hardware) do
    - As the size of the system increases, its parts increase exponentially
-

---

# Complexity (2/3)

## ■ Problem

- You can't abstract away the complexity
    - Physics models work because they abstract away complex details that are not concerned with the essence of the domain; with software the complexity is part of the essence!
  - The complexity comes from the tight interrelationships between heterogeneous artifacts: specs, docs, code, test cases, etc.
-

---

# Complexity (3/3)

- Problems resulting from complexity
    - difficult team communication
    - product flaws
    - cost overruns
    - schedule delays
    - personnel turnover (loss of knowledge)
    - unenumerated states (lots of them)
    - lack of extensibility (complexity of structure)
    - unanticipated states (security loopholes)
    - project overview is difficult (impedes conceptual integrity)
-

---

# Conformity (1/2)

- A significant portion of the complexity facing software engineers is arbitrary
    - Consider a system designed to support a particular business process
    - New VP arrives and changes the process
    - System must now conform to the (from our perspective) arbitrary changes imposed by the VP
-

---

# Conformity (2/2)

- Other instances of conformity
    - Non-standard module or user interfaces
      - Arbitrary since each created by different people
        - not because a domain demanded a particular interface
    - Adapting to a pre-existing environment
      - May be difficult to change the environment
      - However if the environment changes, the software system is expected to adapt!
  - It is difficult to plan for arbitrary change!
-

---

# Changeability

- Software is constantly asked to change
    - Other things are too, however
      - manufactured things are rarely changed
        - the changes appear in later models
        - automobiles are recalled infrequently
        - buildings are expensive to remodel
  - With software, the pressures are greater
    - software = functionality (plus its malleable)
      - functionality is what often needs to be changed!
-

---

# Invisibility

- Software is invisible and unvisualizable
    - In contrast to things like blueprints
      - here geometry helps to identify problems and optimizations of space
    - Its hard to diagram software
      - We find that one diagram may consist of many overlapping graphs rather than just one
        - flow of control, flow of data, patterns of dependency, etc.
  - This lack of visualization deprives the engineer from using the brain s powerful visual skills
-

---

# What about accidents?

- Brooks argues that past breakthroughs solve accidental difficulties
    - High-level languages
    - Time-Sharing
    - Programming Environments
  - New hopefuls
    - Ada, OO Programming, AI, expert systems, automatic programming, etc.
-

---

# Summary (NSB)

- *Essence*—the difficulties inherent in the nature of the software [complexity]
  - *Accidents*—those difficulties that today attend its production but that are not inherent [production]
  - Solution: Grow Great Designers
-