Foundations of Software Engineering

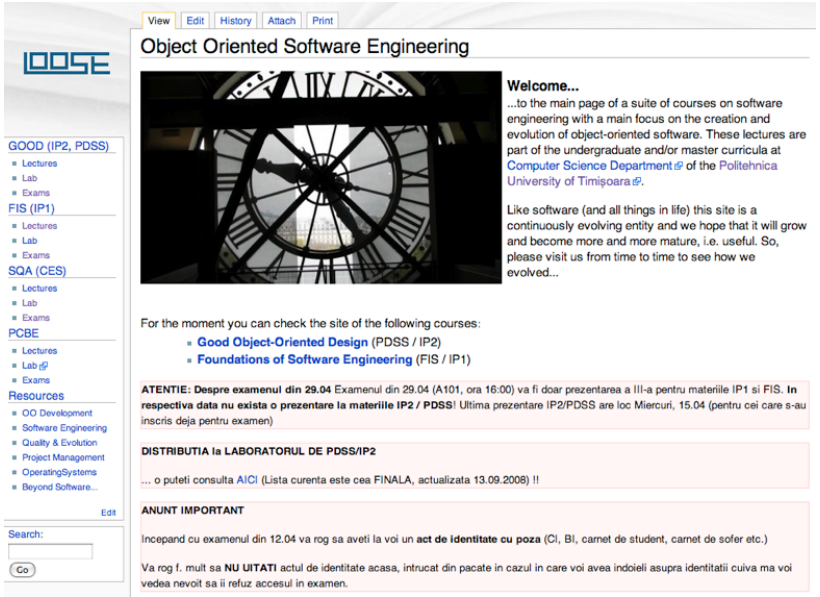
radu.marinescu@cs.upt.ro
0256-40.40.58
ASPC, P11

Examen

NU reproducere mecanica

Surse multiple de informare

Cititi ca sa va formati un punct de vedere



loose.upt.ro/~oose

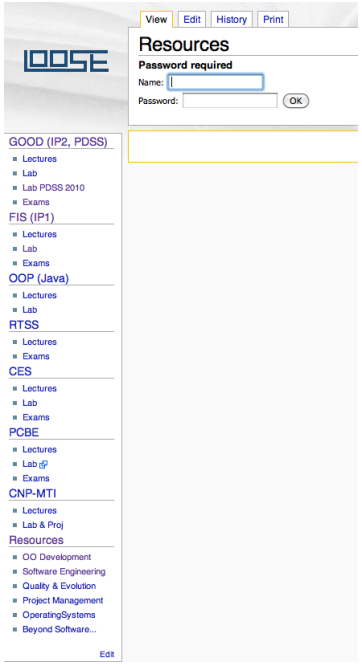
Teorie & Exercitii

50% - 50%

Examenul se promovează
doar integral

Despre măriri...

Examen suplimentar oral...
doar pentru mariri de la
8 - 9 la 10 (nota finale)



Name:

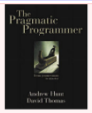
Password: oose07

Software Engineering

Title: Software Engineering 8
Author: Ian Sommerville
Publisher: Addison-Wesley, 2006
Reccomendation: TBW

Title: Code Complete: A Practical Handbook of Software Construction, 2nd Edition
Author: Steve McConnell
Publisher: Microsoft Press, 2004
Reccomendation: TBW

Title: Software Engineering: A Practitioner's Approach, 5th Edition
Author: Roger S. Pressman
Publisher: McGraw-Hill, 2004
Reccomendation: TBW



Title: The Pragmatic Programmer: From Journeyman to Master
Author: Andrew Hunt, David Thomas
Publisher: Addison-Wesley, 1999
Reccomendation: TBW

Title: The Mythical Man-Month: Essays on Software Engineering, 2nd Edition
Author: Frederic P. Brooks
Publisher: Addison-Wesley, 1995
Reccomendation: TBW

Goals, Roles and Myths of Software Engineering

Joys of Programming [Brooks,1995]

- Joy of creating things
  - and pleasure of making useful things for the others
- Fascination of fashioning complex puzzle-like objects
- Delight of working in a highly tractable (flexible) medium
  - Slightly removed from the realm of "pure ideas"
  - Easy to polish and rework
  - It's fun because it gratifies creative longings
- Joy of always learning

Woes of Programming [Brooks,1995]

- We must perform perfectly
- Dependency upon others
  - Including dependencies on other people's imperfections
  - Study and fix things that in an ideal world would be perfect
- Creativity is not the whole story
  - Designing grand concepts is fun, finding little bugs is just work :-(
- Products are oftentimes obsolete upon (or before) completion
  - Laws of Software Evolution

Software Crisis in the 60's

- Rapid increase of computer power and complexity of problems to be solved
- Difficulty to write programs that are:
  - Correct
  - Understandable
  - Verifiable

Root of Software Crisis

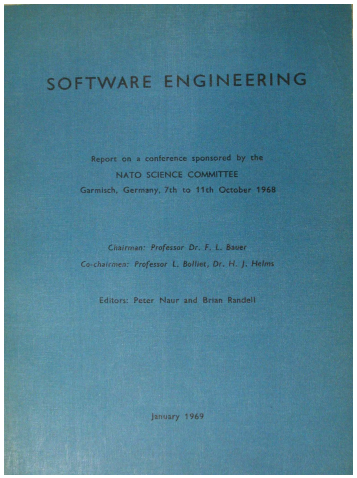
- **Complexity**
  - Orders of magnitude larger and more complex than previously developed software
- **Expectations**
  - Hardware prices were going down, prices for software were increasing dramatically
- **Change**
  - Need to maintain and evolve programs

In other words…

"The major cause of the software crisis is that
the machines have become several orders of magnitude more powerful!
To put it quite bluntly:
as long as there were no machines,
programming was no problem at all;
when we had a few weak computers,
programming became a mild problem,
and now we have gigantic computers,
thus programming has become an equally gigantic problem."

E.W.Dijkstra, 1972
(The Humble Programmger, ACM Turing Award Lecture)

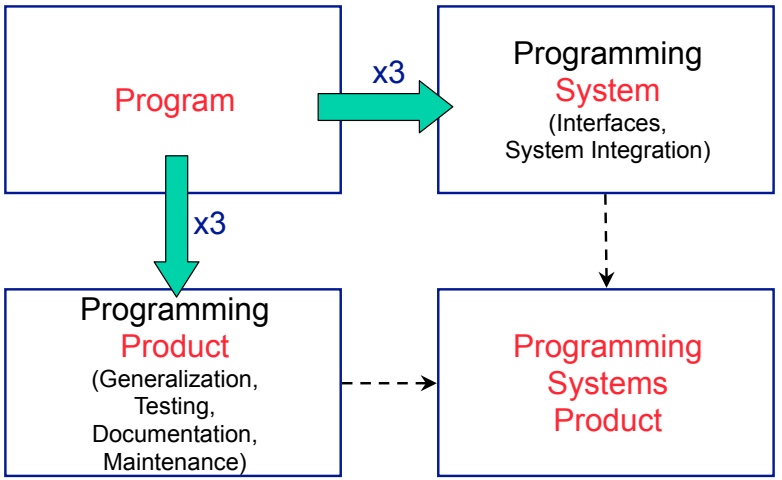
1st Conference on Software Engineering (1968)



Small vs. Large Projects

- Small projects can be understood and implemented by one person
  - For 1 person, by 1 person
- Large projects require division of labor, integration, management
  - For many people, by many people
- "Real" systems are more than just code…
- Let's see…

Program vs. Software [Brooks,1995]



The Tar Pit of Software [Brooks,1995]

Large-system programming has [...] been such a tar-pit,
and many great and powerful beasts have thrashed violently in it.

Most have emerged with running systems --
few have met goals, schedules, and budgets ....

No one thing seems to cause the difficulty,
any particular paw can be pulled away.

But the accumulation of simultaneous and interacting factors
brings slower and slower motion.

F.P.Brooks, 1995

*"The challenge and the mission are to find **real** solutions to **real** problems, on **actual** schedules with **available** resources."*

F.P.Brooks, 1995

What is software engineering?

- Software engineering is an engineering discipline that is concerned with all aspects of software production.
- Adequate usage of appropriate tools and techniques depending on
  - the problem to be solved,
  - development constraints
  - resources available.
- Software engineering is concerned with methods, techniques and tools for professional software development.

What do we need to learn ?

- **Processes**
  - how to manage long-term development/deployment of software
    - what are the phase and what's the flow?
- **Techniques**
  - how should we approach each phase of the process?
- **Tools**
  - how to automate as much as possible of the process?

What is CASE? (Computer-Aided Software Engineering)

- Software systems that are intended to provide automated support for software process activities.
  - often used for method support.
- Upper-CASE
  - Tools to support the early process activities of requirements and design;
- Lower-CASE
  - Tools to support later activities such as programming, debugging and testing.

3 Major questions about software engineering

1. What is software?
2. What are the costs?
3. What is good software?

Q1: What is Software?

What is software?

- Computer programs and associated documentation
  - e.g. requirements, design models and user manuals.
- Software products may be
  - Generic - developed to be sold to a range of different customers e.g. PC software such as Excel or Word.
  - Custom - developed for a single customer according to their specification.
- New software can be created by
  - developing new programs,
  - configuring generic software systems or
  - reusing existing software.

Characteristics of Software

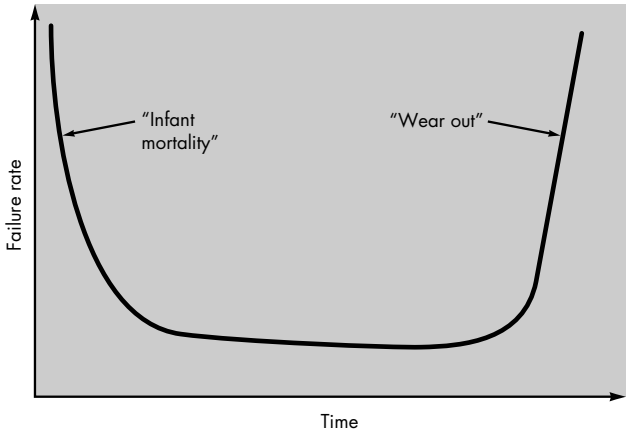
1. Software is engineered, not manufactured
2. Software does not “wear out”... but it deteriorates
3. Most software continues to be custom built

1. Software is engineered, not manufactured

- Engineered = a human creative process
  - analysis, design, construction, testing
- Hardware after being engineered is translated into a physical form (manufacturing)
  - e.g. chips, circuit boards etc.
- Software is only about engineering
  - the manufacturing phase does not introduce quality problems
  - ...or easily correctable problems

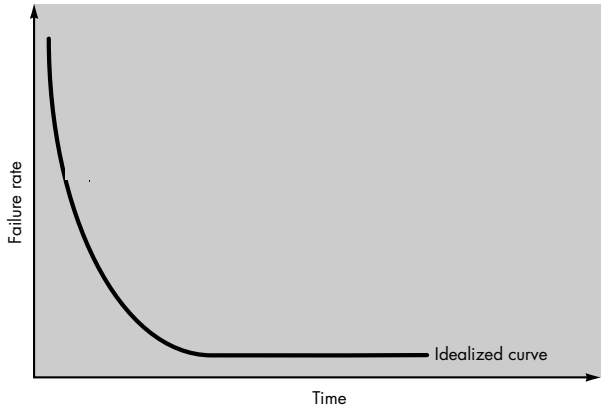
Costs of software are strictly in engineering.
Software projects can't be managed as manufacturing projects

2. Hardware "Wears Out". Software Doesn't!



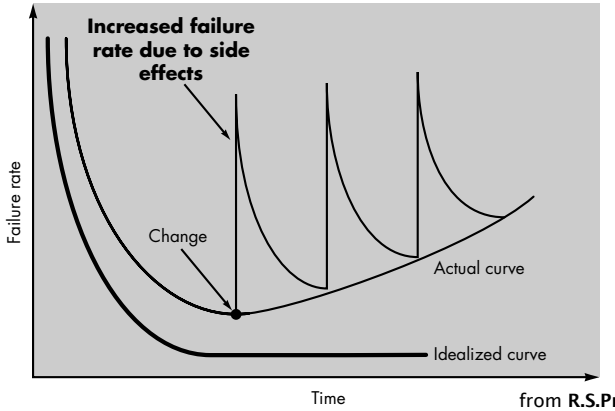
from **R.S.Pressman, 2005**

Ideal Failure Curve for Software



from **R.S.Pressman, 2005**

Actual Failure Curve for Software



from **R.S.Pressman, 2005**

Software does not "wear out"! But it does deteriorate!
The main cause for this phenomenon is **repeated change**

Why must it change?

- **Enhance**
  - to implement **new business requirements**.
- **Extend & Adapt**
  - make it **interoperable** with other more modern systems
  - to meet the needs of **new computing environments**

Laws of Software Evolution

- **The Law of Continuing Change** (1974): Systems must be continually adapted else they become progressively less satisfactory.
- **The Law of Continuing Growth** (1980): The functional content of systems must be continually increased to maintain user satisfaction over their lifetime.
- **The Law of Increasing Complexity** (1974): As a system evolves its complexity increases unless work is done to maintain or reduce it.
- **The Law of Declining Quality** (1996): The quality of systems will appear to be declining, unless they are rigorously maintained and **adapted** to operational environment changes.

Source: Lehman, M., et al, "Metrics and Laws of Software Evolution—The Nineties View," *Proceedings of the 4th International Software Metrics Symposium (METRICS '97)*, IEEE, 1997

Software has no "sparse parts"

- When hardware components wear out, they can be replaced by sparse parts.
- ... but there are no sparse parts for software
  - failures indicate design errors or in the process of translating design into machine executable code.

Software maintenance involves considerably more complexity than hardware maintenance!

3. Software is still custom- **not** component-built

- In each engineering discipline standard design components are created
  - e.g. in hardware: integrated circuits
- In software we need more than algorithmic libraries
  - reuse both algorithms and data structures
  - e.g. GUIs
- Example of reusing in large: **Frameworks**
  - inversion of control: "Hollywood Principle"

Q2: What are the Costs?

Costs of software engineering

- Software costs often dominate computer system costs.
- Software costs more to **maintain** than it does to develop.
  - For systems with a long life, maintenance costs may be **several times** development costs.
- Software engineering is concerned with cost-effective software development.

Main Cost is **not** coding!

- System Tests are the strongest sequential constraint

Brook's Scheduling Rule:

1/3 Planning
1/6 Coding
1/4 Component Test
1/4 System Test

- Fraction devoted to planning is larger
- Half of schedule goes to testing

Not enough testing times is disastrous because bad news come late!

"In examining conventionally scheduled projects,
I have found that few allowed
one-half of the projected schedule for testing,
but that most did indeed
spend half of the actual schedule for that purpose."

F.P.Brooks

Software Myths

Software Myths [Pressman, 2005]

Myths are not heroic legends :) ... they are propagated misinformation!

- Myths are dangerous because:
  1. they appear reasonable and intuitive
  2. they are hard to change!
- Each involved site has its myths:
  - Management myths
  - Customer myths
  - Developer myths

Management Myths

I. We have books, standards and procedures that guarantee good software

✓ Are they up-to-date? Are they used? Are they complete?

II. We buy high-end computers and most expensive CASE tools

✓ Are these used efficiently? Are these customized for specific needs?

III. Outsourcing will release us from software engineering problems

✓ Managing a project externally is harder than doing it internally

IV. If project gets late we will add more programmers and catch up

✓ F.P. Brooks calls this the "mythical man-month": *Adding people to a late project, makes the project even later*

Adding more programmers to a late project... [Brooks 95]

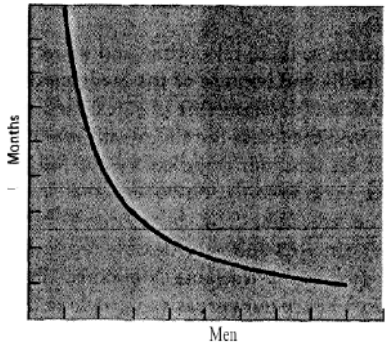
- Project is Planned for 3 people x 4 months
- If there is no time (e.g. we have to finish it in 2 months)…

What do we do??

- **ADD MANPOWER**
  - E.g. employ 3 more people
  - Believing that 3 people x 4 months = 6 people x 2 months

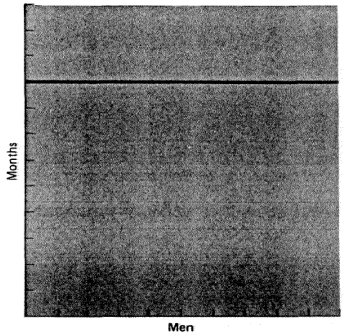
WRONG!

Types of Tasks



Perfectly Partitionable Task

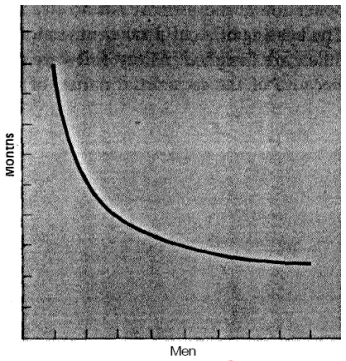
- No inter-communication required
- E.g. reaping wheat



Totally Unpartitionable Task

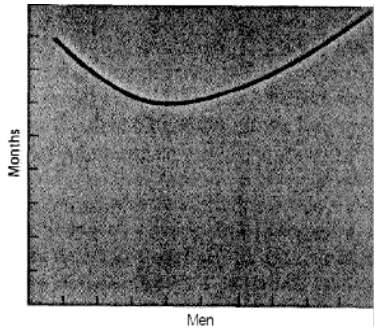
- Sequential constraints
- More effort no effect on schedule
- E.g. bearing a child

Types of Tasks (2)



Task Requiring Communication

- Effort of communication must be added to the amount of work to be done
- Poorer than even trade of men for months



Task with Complex Interrelationships

- Communication may fully counteract the division of the original task

Communication

- Training
  - Technological, goals of the effort, overall strategy plan of work
- Intercommunication
  - For N people involved: N*(N-1)/2
  - E.g. It is **6 x** in a team of 4 people compared to a team of 2 people

Brook's Law

Adding manpower to a late software project makes it later

Corollaries

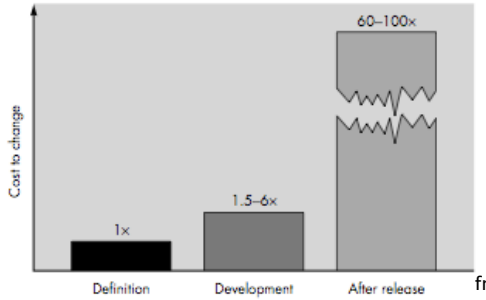
Nr. of months of a project depends on its **sequential constraints**

Maximum nr. of people depends on the nr. of **independent subtasks**

Customer Myths

I. Just sketching the requirements is enough to start building software

II. Requirements can permanently change at a low cost because software is flexible



from **R.S.Pressman, 2005**

Developer Myths

I. Once we write a program and get it to work, our job is done

- ✓ "*the sooner you begin 'writing code' the longer it'll take to get done*"

II. The only deliverable work product is the working program

- ✓ program is only a part of a working configuration, which also includes documentation

III. Software engineering will make use create voluminous and unnecessary documentation, which slows us down

- ✓ Pressman: "*SWE is not about creating documents. It's about creating quality. Better quality leads to reduced work.*"