The Nature of Software & Software Engineering

- ## Chapters 1 & 2

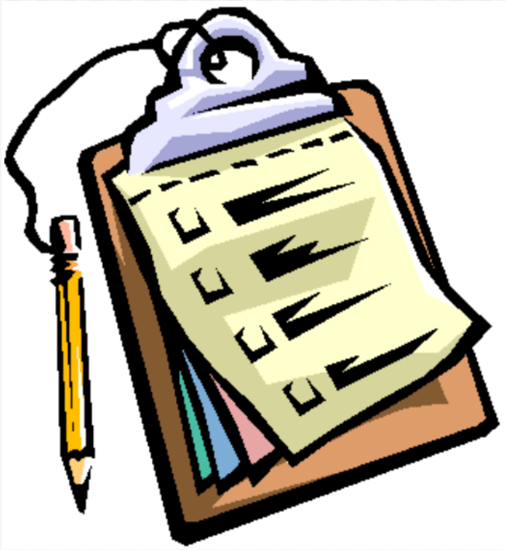
Slide Set to accompany

Software Engineering: A Practitioner's Approach, 8/e

by Roger S. Pressman and Bruce R. Maxim

Agenda

- The Nature of Software
- Software Engineering



The Nature of Software

What is Software?

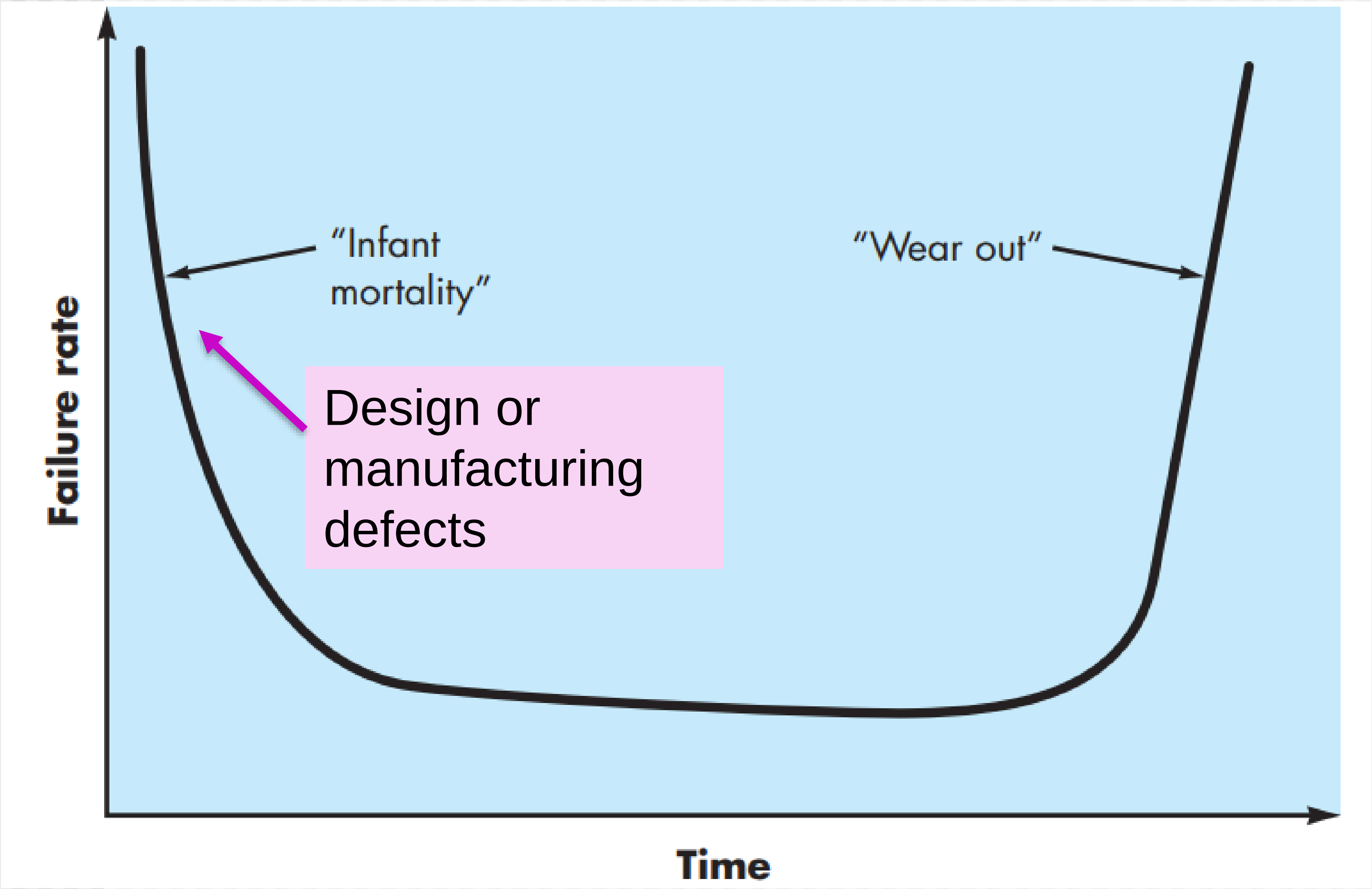
Software is:
(1) ***instructions*** *(computer programs) that when executed provide desired features, function, and performance;*
(2) ***data structures*** *that enable the programs to adequately manipulate information*
and (3) ***documentation*** *that describes the operation and use of the programs.*

The Nature of Software

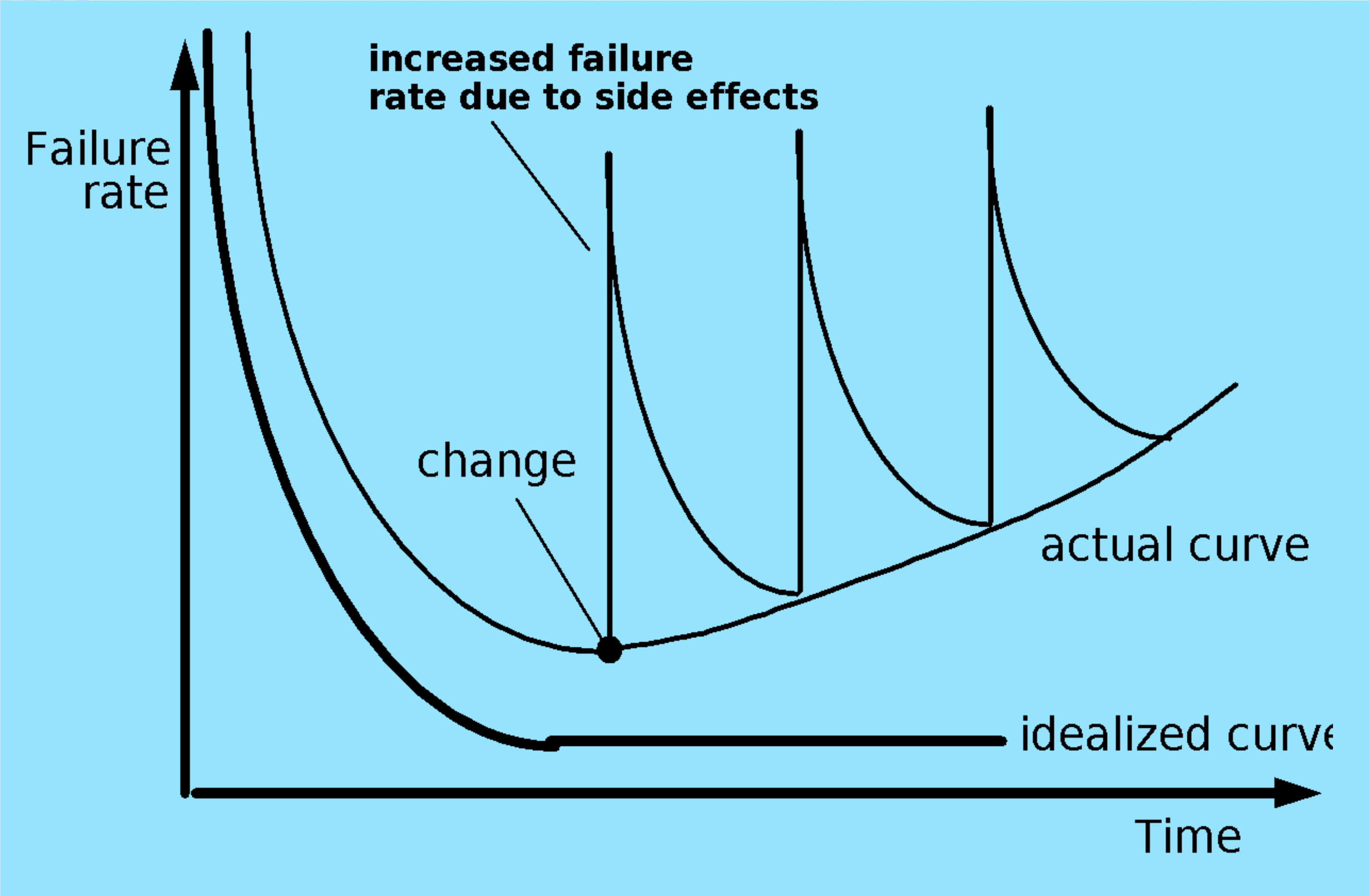
- Software is developed or engineered, it is not manufactured in the classical sense.
- Software is a logical rather than a physical element.
- Software doesn't "wear out."
  - wear out:

- Software deteriorates
- Deteriorate = become progressively worse

Failure Curve for Hardware



Wear vs. Deterioration



Software Changes

- What may change in a software, and why?
  - Requirements
    - Technical requirements
    - Business requirements
  - Analysis & Design
  - Implementation

Software Application Domains

- System software
  - Programs that service other programs (OS, drivers, compilers)
- Application software
- Engineering/Scientific software
- Embedded software
- Product-line software
- Web/Mobile applications
- AI software (robotics, neural nets, game playing)

Hybrid Categories

Legacy Software

- Those old programs
- Developed decades ago
- Have been continually modified to meet changes
- Headaches for large organization:
  - They are costly to maintain and risky to evolve
- But they are critical for the business
  - Many legacy systems remain supportive to core business functions and are ‘**indispensable**’ to the business
- Often with poor quality
  - Inextensible designs, convoluted code
  - Lack of documentation, test cases and results
  - Poorly managed change history

Legacy Software (cont.)

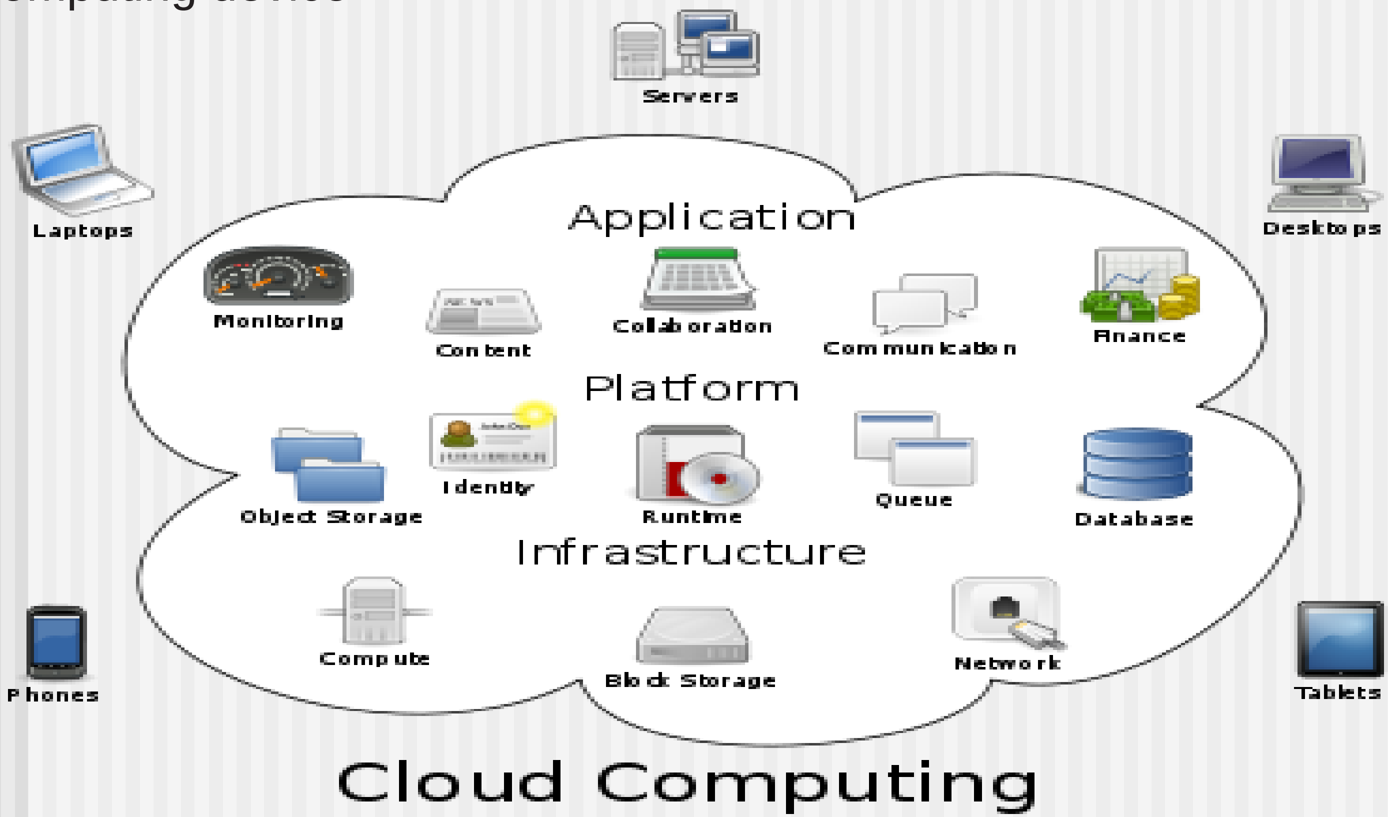
- What to do?
  - The only reasonable answer is:
  - Do nothing,
  - At least until the legacy system must undergo some **significant** change
    - Software must be re-engineered to make it viable within a evolving environment

New Trends

- WebApps
  - Internet-based, Unpredictable load, Performance, Availability, …
- Mobile Apps
- Cloud Computing
- Product Line Software

Cloud Computing

- *Cloud computing* encompasses an infrastructure or “ecosystem” that enables any user, anywhere, to share computing resources using a computing device



Cloud Computing (cont.)

- Provides distributed storage and processing resources to different computing devices
- Computing devices reside outside the cloud and have access to a variety of resources inside the cloud
  - Applications, platforms, and infrastructure
- In its simplest form, an external computing device accesses the cloud via a Web browser or analogous software
- The cloud provides access to data that resides with databases and other data structures
- In addition, devices can access executable applications that can be used in lieu of apps that reside on the computing device
- Requires developing an architecture containing both frontend and backend services

Cloud Computing (cont.)

- Frontend services include the client (user) devices and application software (e.g., a browser) that allows the back-end to be accessed
- Backend services include servers, data storage (e.g., databases), and server-resident applications
- Cloud architectures can be segmented to provide access at a variety of different levels
  - From full public access to private cloud architectures accessible only to those with authorization

Software Engineering

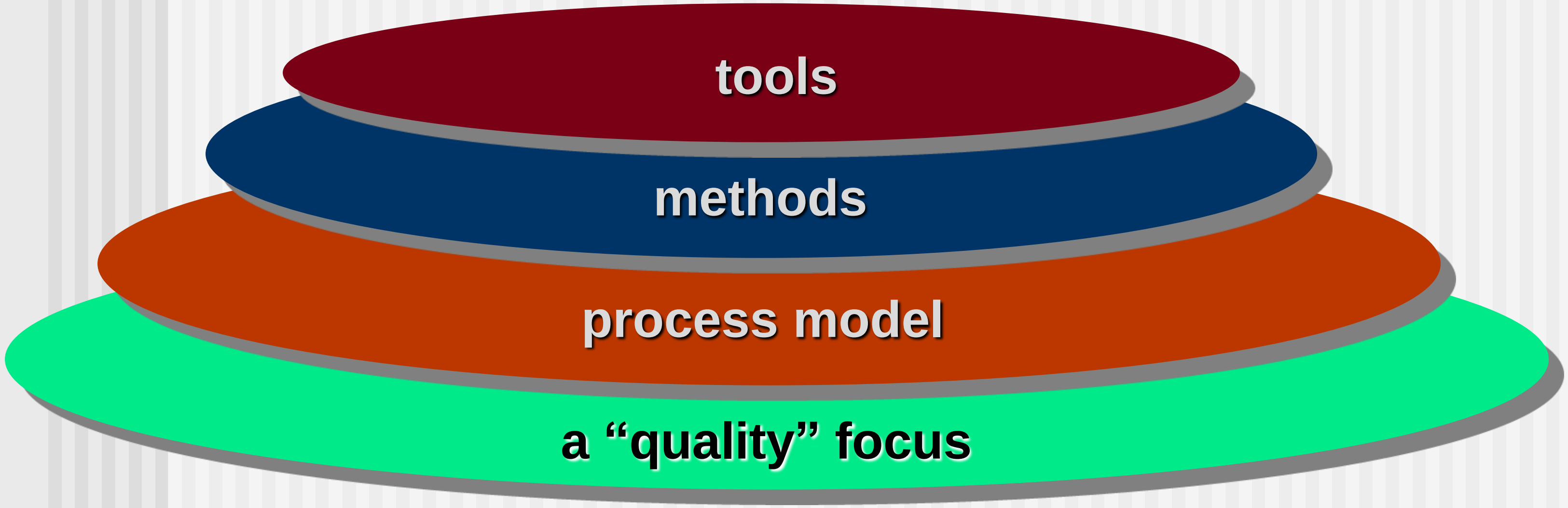
Why Software Engineering?

- Some realities:
  - *A concerted effort should be made to understand the problem before a software solution is developed*
  - *Design becomes a pivotal activity*
  - *Software should exhibit high quality*
  - *Software should be maintainable*
- Conclusion:
  - Software in all of its forms and across all of its application domains should be engineered

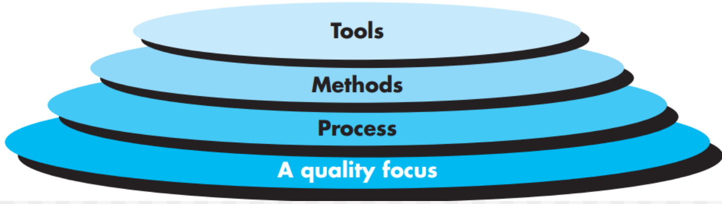
What is Software Engineering?

- The IEEE definition:
  - *Software Engineering:*
  - *(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.*
  - *(2) The study of approaches as in (1)*

Software Engineering: A Layered Technology

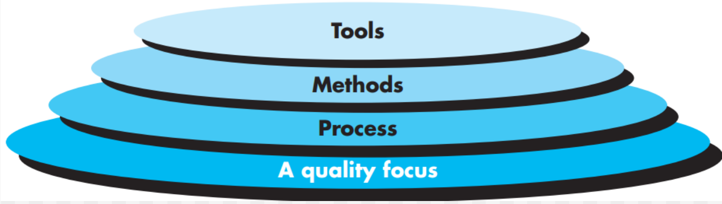


Software Engineering



Layered Technology

- **Quality** Layer
  - An organizational commitment to quality
    - A continuous improvement culture
- **Process** Layer
  - The foundation for software engineering
  - Defines a framework
    that must be established for effective delivery of software
  - It forms the basis for:
    - project management,producing work products, achieving milestones, quality assurance and change management
  - Example?



Layered Technology (cont.)

- **Methods** Layer
  - Provides the technical how-to's for building software
  - Methods include techniques of:
    - communication, requirements analysis, design, program construction, testing, and support
- **Tools** Layer
  - Provides automated or semi-automated support for the process and methods
  - Computer-Aided Software Engineering (CASE)
  - Example?
    - Management, Documentation, Design, Versioning, Test,…

Hooker's General Principles

- 1: *The Reason It All Exists*
- 2: *KISS (Keep It Simple, Stupid!)*
- 3: *Maintain the Vision*
- 4: *What You Produce, Others Will Consume*
- 5: *Be Open to the Future*
- 6: *Plan Ahead for Reuse*
- 7*: Think!*

If every software engineer and every software team simply followed Hooker's seven principles, many of the difficulties we experience in building complex computer-based systems would be eliminated

1) The Reason It All Exists

- A software system exists for **one reason**: to provide value to its users
- All decisions should be made with this in mind
- Before specifying a system requirement, before determining the hardware platforms or development processes,
  - Ask yourself questions such as:
    - "Does this add real VALUE to the system?"
  - If the answer is
    - “No"
  - **Don't** do it
- All other principles support this one

2) KISS (Keep It Simple, Stupid!)

- All design should be as simple as possible, **but no simpler**
- The more elegant designs are usually the more simple ones
- This facilitates having a more easily understood, and easily maintained system
- This is not to say that features should be discarded in the name of simplicity
- Simple also does not mean "quick and dirty"
- In fact, it often takes a lot of thought and work over multiple iterations to simplify
- The payoff is software that is more maintainable and less error-prone

3) Maintain the Vision

- A clear vision is essential to the success of a software project
- Without one, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws
- Having a clean internal structure is essential to constructing a system that is understandable, extendible, maintainable and testable
- Having an empowered Architect who can hold the vision and enforce compliance helps ensure a very successful software project

4) What You Produce, Others Will Consume

- Someone else will use, maintain, document, or otherwise depend on being able to understand your system
- So, always specify, design, and implement knowing someone else will have to understand what you are doing
- Specify with an eye to the users
- Design, keeping the implementers in mind
- Code with concern for those that must maintain and extend the system
  - Someone may have to debug the code you write, and that makes them a user of your code
- Making their job easier **adds value to the system**

5) Be Open to the Future

- A system with a long lifetime has more value
- Today Software lifetimes are typically measured in months instead of years
- True “industrial-strength" software systems must endure far longer
  - These systems must be ready to **adapt** to changes
  - Systems that do this successfully are those that have been designed this way from the start
  - Always ask "what if ", and prepare for all possible answers by creating systems that solve the general problem, not just the specific one
  - This could very possibly lead to the reuse of an entire system

6) Plan Ahead for Reuse

- Reuse saves time and effort
- Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system
- The reuse of code and designs is a major benefit of using **object-oriented** technologies
- However, the return on this investment requires forethought and planning

7) Think!

- This last Principle is probably the most overlooked
- Placing clear, complete thought before action almost always produces better results
- When you think about something, you are more likely to do it right
- You also gain knowledge about how to do it right again
- If you do think about something and still do it wrong, it becomes valuable experience
- Applying the first six Principles requires intense thought

Software Myths

Recognition of software realities is the first step toward formulation of practical solutions for software engineering

- Erroneous beliefs about software and the process
- Misleading attitudes that affect managers, customers and practitioners
- Are believable because they often have elements of truth,

but …

- Invariably lead to bad decisions,

therefore …

- Insist on reality as you navigate your way through software engineering

Management myths

- ***Myth:*** *If we get behind schedule, we can add more programmers and catch up*
- **Reality:**
  - Software development is not a mechanistic process like manufacturing
  - Adding people to a late software project makes it later
- ***Myth:*** *If I decide to outsource the software project to a third party, I can just relax and let that firm build it*
- **Reality:** If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources

Customer myths

- In many cases, the customer believes myths about software because
  - Software managers and practitioners **do little** to correct misinformation.
- Myths lead to false expectations (by the customer)
  - Ultimately, dissatisfaction with the developer

Customer myths (1)

- ***Myth:*** *A general statement of objectives is sufficient to begin writing programs—we can fill in the details later*
- **Reality:**
  - A comprehensive and stable statement of requirements is not always possible
  - However, an ambiguous “statement of objectives” is a disaster
  - Unambiguous requirements (usually derived iteratively) are developed only through:
    - Effective and continuous communication between customer and developer

Customer myths (2)

- ***Myth:*** *Software requirements continually change, but change can be easily accommodated because software is flexible*
- **Reality:**
  - The impact of change varies with the time at which it is introduced
  - When requirements changes are requested early (before design or code) the cost impact is relatively small
  - As time passes, the cost impact grows rapidly
    - Additional resources and major design modification

Practitioner's myths (1)

- ***Myth:*** *Once we write the program and get it to work, our job is done*
- **Reality:**
  - "The sooner you begin 'writing code,' the longer it'll take you to get done."
  - Between 60 and 80 percent of all effort expended on software will be expended after its first delivery to customers

Practitioner's myths (2)

- ***Myth:*** *Until I get the program "running" I have no way of assessing its quality*
- **Reality:**
  - Technical review
    - One of the most effective software quality assurance mechanisms
    - It can be applied from the inception of a project
    - It is more effective than testing for finding certain classes of software defects

Practitioner's myths (3)

- ***Myth:*** *The only deliverable work product for a successful project is the working program*
- **Reality:**
  - A working program is only <u>one part</u> of a software configuration
  - A variety of work products (e.g., models, documents, plans)
    - Foundation for successful engineering
    - Guidance for software support

Practitioner’s myths (4)

- ***Myth:*** *Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down*
  - We have no time!!!
- **Reality:**
  - Software engineering is not about creating documents
  - It is about creating a quality product
  - Better quality leads to **reduced rework**
    - Reduced rework results in faster delivery times

Further Reading

- Chapters 1 & 2 of Pressman

The End