Chapter 1-2

- **Software & Software Engineering**

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 7/e

by Roger S. Pressman

Slides copyright © 1996, 2001, 2005, 2009 by Roger S. Pressman

For non-profit educational use only

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 7/e.* Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

What is Software?

Software is (programs, data, and descriptive information)

(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.

What is Software?

- ***Software is developed or engineered, it is not manufactured in the classical sense.***
- ***Software doesn't "wear out."***
- ***Although the industry is moving toward component-based construction, most software continues to be custom-built.***

Hardware

- Relatively high failure rates early in its life
- Defects are corrected
- The failure rate drops to a steady-state level for some period of time
- As time passes, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, …
  ==> The hardware begins to **wear out**

Hardware: Failure Curve

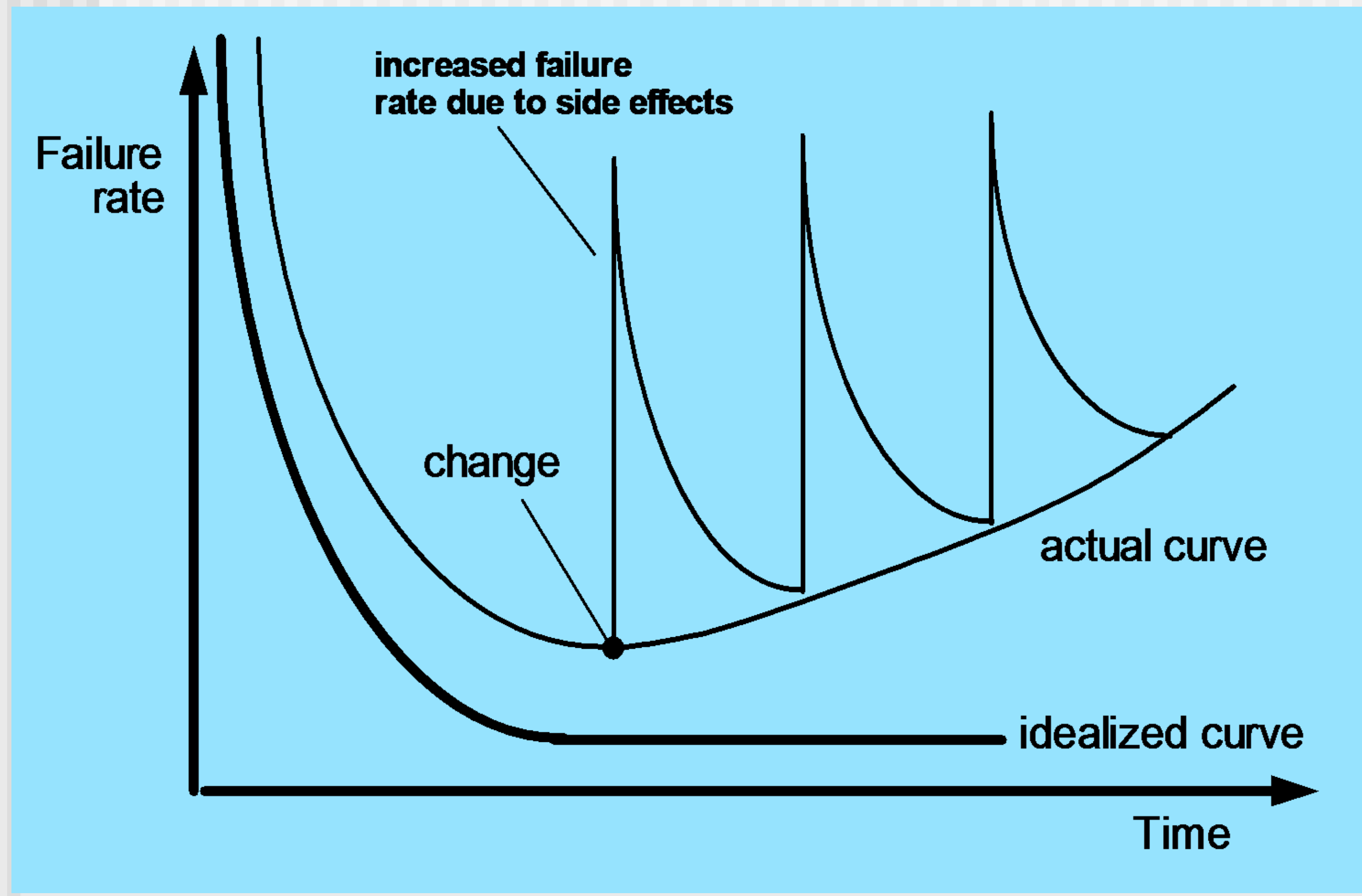
Bathtub curve

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

Software

- Not susceptible to the environmental maladies that cause hardware to wear out
- In theory, the failure rate curve for software should take the form of the “idealized curve”
  - Undiscovered defects will cause high failure rates early in the life of a program
  - These defects are corrected and the curve flattens - ➔ software doesn’t wear
- But, software deteriorates!
  - Software will undergo change during its life

Wear vs. Deterioration



Software Applications

- system software
  - Service other programs
  - *Determinate:* predicable inputs ➔ Compilers, editors, file management utilities
  - *Indeterminate:* unpredictable inputs ➔ Operating systems, network software
- application software
  - Stand-alone programs that solve a specific business need

Software Applications

- engineering/scientific software
  - A broad array of “number-crunching programs”
    - From astronomy to volcanology
    - From automotive stress analysis to orbital dynamics
    - From computer-aided design to molecular biology
    - From genetic analysis to meteorology
- embedded software
  - Resides within a product or system and is used to implement and control features and functions for the end user and for the system itself
- product-line software
  - Designed to provide a specific capability for use by many different customers
    - Inventory control products

Software Applications

- WebApps (Web applications)
  - Browser-based apps
  - Software that resides on mobile devices
- AI software
  - Makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis
  - Robotics
  - Expert systems
  - Pattern recognition (image, voice and text)
  - Artificial neural networks
  - Theorem proving
  - Game playing

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

Software—New Categories

- Open world computing—pervasive, distributed computing
- Ubiquitous computing—wireless networks
- Netsourcing—the Web as a computing engine
- Open source—"free" source code open to the computing community (a blessing, but also a potential curse!)
- Also … (see Chapter 31)
  - Data mining
  - Grid computing
  - Cognitive machines
  - Software for nanotechnologies

 Slides copyright 2009 by Roger Pressman.

Legacy Software

- Legacy software
  - Very old programs
  - Support core business functions and are indispensable to the business
  - Has been the focus of continuous attention and concern since the 1960s

- Characteristics
  - Longevity
  - Business criticality

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

Legacy Software: Negative characteristic

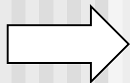
- **Poor Quality**
  - Inextensible designs
  - Convoluted code
  - Poor or nonexistent documentation
  - Test cases and results that were never archived
  - Poorly managed change history

- What do we do if we meet a legacy system that exhibits poor quality?
  - Do nothing, at least until the legacy system must undergo some significant change ➔ *Is it okay?*

Legacy Software: Often Evolves

Why must it change?

- software must be adapted to meet the needs of new computing environments or technology.
- software must be enhanced to implement new business requirements.
- software must be extended to make it interoperable with other more modern systems or databases.
- software must be re-architected to make it viable within a network environment.



If evolution occurs, a legacy system must be reengineered so that it remains viable into the future

Software: The Changing Nature

- Some categories of software are evolving to dominate the industry, but they are in their infancy little more than a decade ago

- 1) **WebApps**
- 2) **Mobile application**
- 3) **Cloud computing**
- 4) **Product line software**

WebApps

- In the early days of WWW, websites consists of little more than a set of linked hypertext files
- As time passed, the augmentation of HTML enabled Web engineers to provide computing capability along with informational content ➔ Web-based systems and applications (WebApps)
- Today, WebApps have evolved into sophisticated computing tools
  - Provide stand-alone function to the end user
  - Have been integrated with corporate databases and business applications

WebApps: Evolution

- Previously, WebApps involve a mixture b/w print publishing and software development, b/w marketing and computing, b/w internal communications and external relations, and b/w art and technology
- Today, WebApps provide full computing potential in most application categories
- Semantic Web Technology (Web 3.0)
  - Encompass “semantic databases that provide new functionality that requires Web linking, flexible representation, and external access APIs”
  - Sophisticated relational data structure ➔ entirely new WebApps in ways never before possible

WebApps: Characteristics

- **Network intensiveness.** A WebApp resides on a network and must serve the needs of a diverse community of clients.
- **Concurrency.** A large number of users may access the WebApp at one time.
- **Unpredictable load.** The number of users of the WebApp may vary by orders of magnitude from day to day.
- **Performance.** If a WebApp user must wait too long (for access, for server-side processing, for client-side formatting and display), he or she may decide to go elsewhere.
- **Availability.** Although expectation of 100 percent availability is unreasonable, users of popular WebApps often demand access on a "24/7/365" basis.

WebApps: Characteristics

- **Data driven.** The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end-user.
- **Content sensitive.** The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.
- **Continuous evolution.** Unlike conventional application software that evolves over a series of planned, chronologically-spaced releases, Web applications evolve continuously.
- **Immediacy.** Although *immediacy*—the compelling need to get software to market quickly—is a characteristic of many application domains, WebApps often exhibit a time to market that can be a matter of a few days or weeks.
- **Security.** Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end-users who may access the application.
- **Aesthetics.** An undeniable part of the appeal of a WebApp is its look and feel.

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

Mobile Applications

- *App*
  - Has evolved to software that has been specifically designed to reside on a mobile platform
- Mobile apps encompass
  - 1) a user interface that uses features provided by the mobile platform, 2) interoperability with Web-based resources, 3) local processing capabilities that collect, analyze, and format information, 4) persistent storage capabilities

Mobile Applications

- Mobile web application (Webapp)
  - Allows a mobile device to gain access to web-based content via a browser
- Mobile app
  - Can gain direct access to the hardware characteristics of the device
    - Accelerometer, GPS location
- The distinction b/w mobile WebApps and mobile apps will blur as mobile browers becomes more sophisticated

Cloud computing

  - an infrastructure or “ecosystem” that enables any user, anywhere to use a computing device to share computing resources on a broad scale
- Computing devices in cloud computing
  - Reside outside the cloud
  - Have access to a variety of resources within the cloud
  - The resources include
    - 1) applications, 2) platforms, 3) infrastructure
- In the simplest form
  - an external computing device accesses the cloud via a Web browser

Cloud computing: Implementation issue

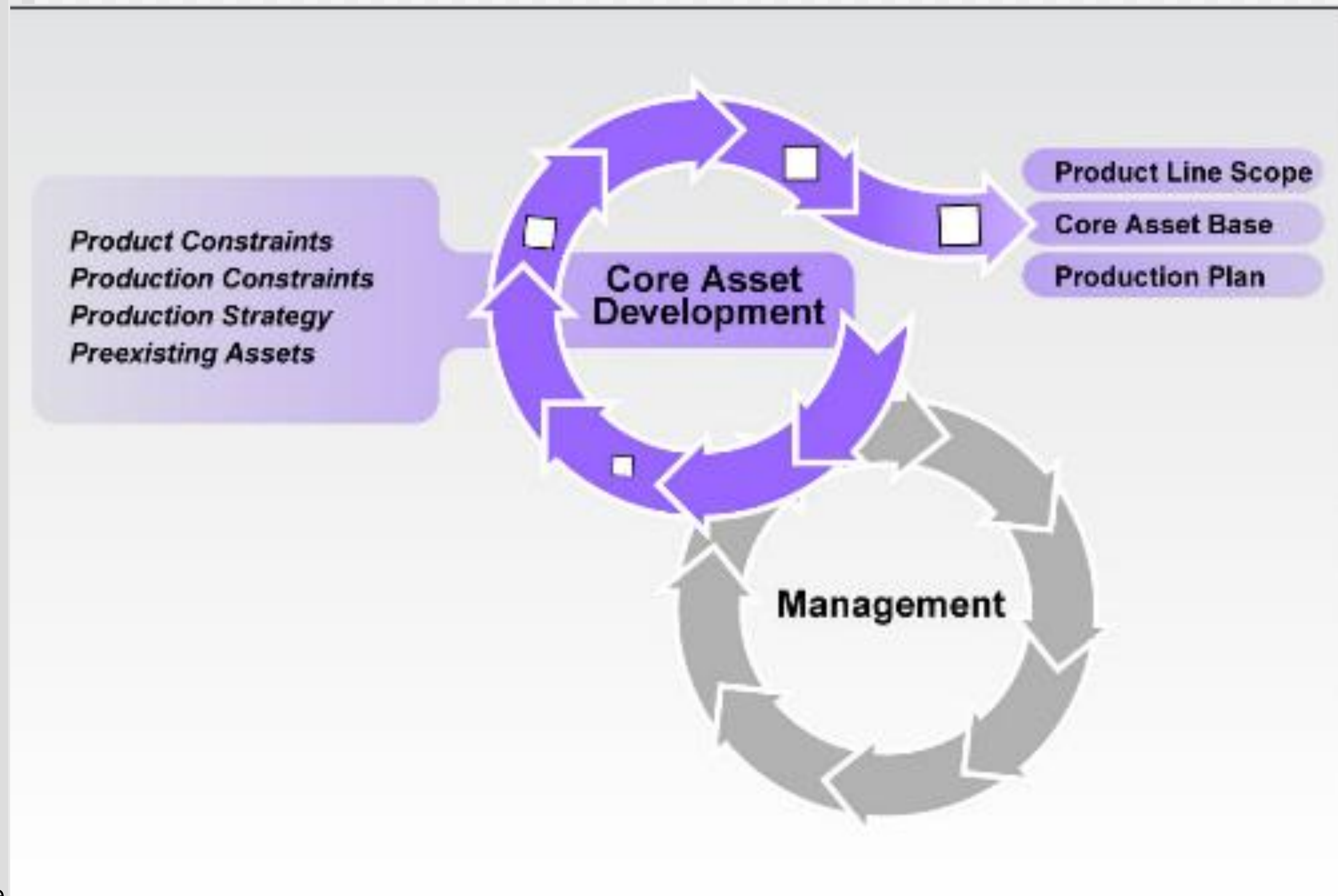
- Requires the development of front-end and back-end services
  - Similar to Client-server architecture
- Front-end
  - The client device
  - The application software (e.g. browser)
- Back-end
  - Servers
  - Related computing resources
  - Data storage systems (e.g. databases)
  - Server-resident applications
  - Administrative servers

Software product line

- Definition
  - A set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment and mission and that are developed from a common set of core assets in a prescribed way
- **Reuse all across the product line!**
  - All developed using the same underlying application and data architectures
  - All implemented using a set of reusable software components

  Leads to significant engineering leverages

Software product line



These
(McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

Chapter 1: Summary

- Software
  - The key element in the evolution of computer-based systems and products
  - One of the most important technologies on the world stage
  - Over the past 50 years, software has evolved from a specialized problem solving and information analysis tool to an industry in itself
- Yet we have trouble developing high-quality software on time and within budget

Chapter 1: Summary

  - Software (programs, data, descriptive information): addresses a wide array of technology and application areas
- Legacy software: continues to present special challenges to those who must maintain it
- The nature of software is changing
  - Web-based systems and applications have evolved
  - Mobile applications present new challenges as apps migrate to a wide array of platforms
  - Cloud computing will transform the way in which software is delivered and the environment in which it exists
  - Product line software offers potential efficiencies in the manner in which software is built

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

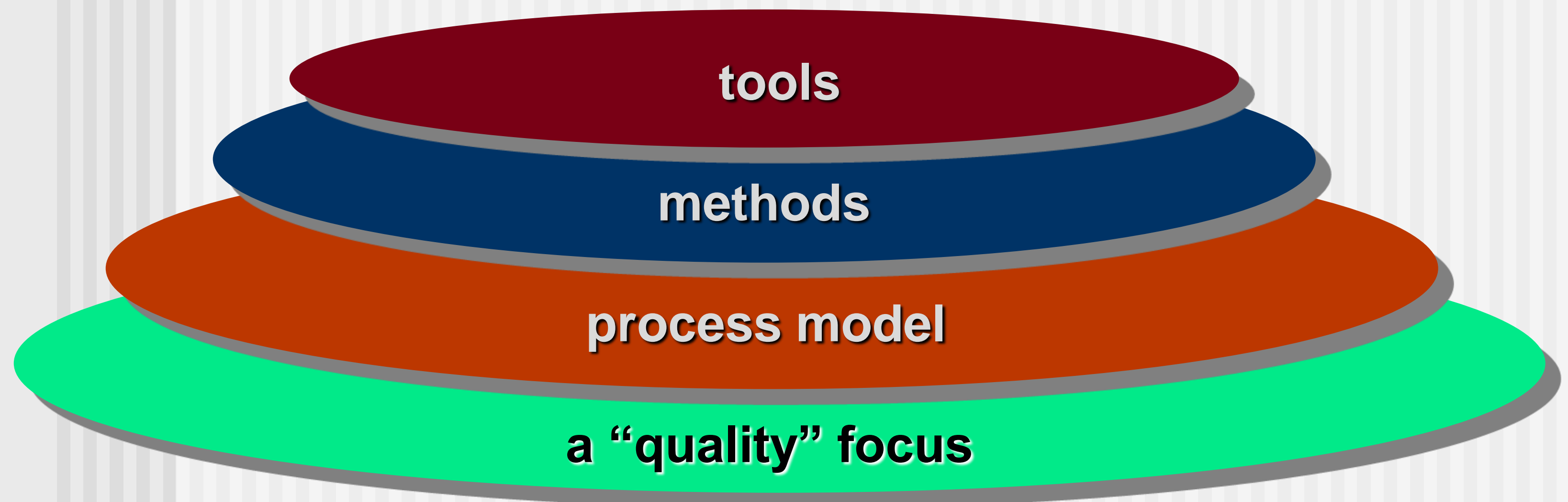
Software Engineering

- 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*
- The seminal definition:
  - *[Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.*

Software Engineering: Definition of Discipline

- 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).*
- And yet, a "systematic, disciplined and quantifiable" approach applied by one software team may be burdensome to another
- We need discipline, but also need adaptability and agility

Software Engineering: A Layered Technology



Software Engineering

Software Engineering: Quality focus

- Quality focus: Bedrock that supports software engineering
  - Any engineering approach (including software engineering) must rest on an organizational commitment to quality
  - Total quality management, Six Sigma, and similar philosophies foster a continuous process improvement culture
    - This ultimately leads to the development of increasingly more effective approaches to software engineering

Software Engineering
Process

- Process layer: The foundation of software engineering
- Process
  - Is the glue that holds the technology layers together
  - Enables rational and timely development of computer software
  - Defines a framework that must be established for effective delivery of software engineering technology
  - Forms the basis for management control of software projects
  - Establishes the context in which technical methods are applied, work products are produced, milestones are established, quality is ensured, and change is properly managed

Software Engineering
Methods

- Methods
  - Provide the technical how-to's for building software
  - Encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support
  - Rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques

Software Engineering Tools

- Tools
  - Provides automated or semi-automated support for the process and the methods
  - Computer-aided software engineering
    - Tools are integrated so that information created by one tool can be used by another

Software Process

- Process is a collection of activities, actions, and tasks that are performed when some work product is to be created
- Activity
  - Strives to achieve a broad objective (e.g. communication)
  - Is applied regardless of the application domain, size of the project, complexity of the effort
- Action (e.g. architectural design)
  - Encompasses a set of tasks that produce a major work product
- Task
  - Focuses on a small, but well-defined objective that produces a tangible outcome

Software Process

- Process is not a rigid prescription for how to build computer software
- But, it is an adaptable approach that enables the people doing the work to pick and choose the application set of work actions and tasks
- The intent is to deliver software in a timely manner and with sufficient quality

Process Framework

- Establishes the foundation for a complete software engineering process by identifying a small number of framework activities
  - That are applicable to all software projects, regardless of their size or complexity
- Encompasses a set of umbrella activities
  - That are applicable across the entire software process

A Process Framework

- **Process framework**
  - **Framework activities**
    - work tasks
    - work products
    - milestones & deliverables
    - QA checkpoints
  - **Umbrella Activities**

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

Framework Activities: General setting

- Communication
- Planning
- Modeling
  - Analysis of requirements
  - Design
- Construction
  - Code generation
  - Testing
- Deployment

Communication

- Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and other stakeholders)
- The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.

Planning

- Any complicated journey can be simplified if a map exists.
- A software project is a complicated journey, and the planning activity creates a “map” that helps guide the team as it makes the journey.
- The map—called a *software project plan*—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

Modeling

- Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day.
- You create a "sketch" of the thing so that you'll understand the big picture
- What it will look like architecturally, how the constituent parts fit together, and many other characteristics.
- If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it.

- A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.

Construction

- Code generation (either manual or automated) and the testing that is required to uncover errors in the code.

Deployment

- The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

Process Framework: Iteration

- **communication, planning, modeling, construction,** and **deployment** are applied repeatedly through a number of project iterations.
- Each project iteration produces a *software increment* that provides stakeholders with a subset of overall software features and functionality.
- For each increment, the software becomes more and more complete

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

Umbrella Activities: Typical setting

- Software project management
- Formal technical reviews
- Software quality assurance
- Software configuration management
- Work product preparation and production
- Reusability management
- Measurement
- Risk management

Umbrella Activities

- **Software project tracking and control**
  - Allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule
- **Risk management**
  - Accesses risks that may affect the outcome of the project or the quality of the product
- **Software quality assurance**
  - Defines and conducts the activities required to ensure software quality

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

Umbrella Activities

- **Technical reviews**
  - Access software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity
- **Measurement**
  - Defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs; can be used in conjunction with all other framework and umbrella activities
- **Software configuration and management**
  - Manages the effects of change throughout the software process

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

Umbrella Activities

- **Reusability management**
  - Defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components
- **Work product preparation and production**
  - Encompass the activities required to create work products such as models, documents, logs, forms, and lists

Process Adaptation

- software engineering process should be agile and adaptable (to the problem, to the project, to the team, and to the organizational culture).

- a process adopted for one project might be significantly different than a process adopted for another project

Process Adaptation: Differences

- the overall flow of activities, actions, and tasks and the interdependencies among them
- the degree to which actions and tasks are defined within each framework activity
- the degree to which work products are identified and required
- the manner which quality assurance activities are applied
- the manner in which project tracking and control activities are applied
- the overall degree of detail and rigor with which the process is described
- the degree to which the customer and other stakeholders are involved with the project
- the level of autonomy given to the software team
- the degree to which team organization and roles are prescribed

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

Software Engineering Practice

  - We have generic framework activities—**communication, planning, modeling, construction,** and **deployment**—and umbrella activities establish a skeleton architecture for software engineering work.
- How does the practice of software engineering fit in?
- We now gain a basic understanding of the generic concepts and principles that apply to framework activities

The Essence of Practice

- Polya suggests:
  1. *Understand the problem* (communication and analysis).
  2. *Plan a solution* (modeling and software design).
  3. *Carry out the plan* (code generation).
  4. *Examine the result for accuracy* (testing and quality assurance).

Understand the Problem

- *Who has a stake in the solution to the problem?* That is, who are the stakeholders?
- *What are the unknowns?* What data, functions, and features are required to properly solve the problem?
- *Can the problem be compartmentalized?* Is it possible to represent smaller problems that may be easier to understand?
- *Can the problem be represented graphically?* Can an analysis model be created?

Plan the Solution

- *Have you seen similar problems before?* Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- *Has a similar problem been solved?* If so, are elements of the solution reusable?
- *Can subproblems be defined?* If so, are solutions readily apparent for the subproblems?
- *Can you represent a solution in a manner that leads to effective implementation?* Can a design model be created?

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

Carry Out the Plan

- *Does the solution conform to the plan?* Is source code traceable to the design model?
- *Is each component part of the solution provably correct?* Has the design and code been reviewed, or better, have correctness proofs been applied to algorithm?

Examine the Result

- *Is it possible to test each component part of the solution?* Has a reasonable testing strategy been implemented?
- *Does the solution produce results that conform to the data, functions, and features that are required?* Has the software been validated against all stakeholder requirements?

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!*

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

- ***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 noting a piece of system functionality, 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.

- ***KISS (Keep It Simple, Stupid!)***
  - Software design is not a haphazard process.
  - There are many factors to consider in any design effort. *All design should be as simple as possible, but no simpler*.
  - The more elegant designs are usually the more simple ones

- ***Maintain the Vision***
  - *A clear vision is essential to the success of a software project.*
  - Without one, a project almost unfailingly ends up being “of two [or more] minds” about itself.
  - Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws.
  - Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.

- ***What You Produce, Others Will Consume***
  - In some way or other, 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.

- ***Be Open to the Future***
  - A system with a long lifetime has more value.
  - True "industrial-strength" software systems must endure far longer.
  - To do this successfully, these systems must be ready to adapt to these and other changes
  - *Never design yourself into a corner*.
  - 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

- ***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 has been proclaimed as a major benefit of using object-oriented technologies
  - *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated*.

- ***Think!***
  - *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.
  - A side effect of thinking is learning to recognize when you don't know something, at which point you can research the answer.
  - Applying the first six principles requires intense thought, for which the potential rewards are enormous.

Software Myths

- Affect managers, customers (and other non-technical stakeholders) 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

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

Management myths

- **Myth:** *We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?*
- **Reality:** The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it adaptable? Is it streamlined to improve time-to-delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

Management myths

- **Myth:** *If we get behind schedule, we can add more programmers and catch up (sometimes called the "Mongolian horde" concept).*
- **Reality:** Software development is not a mechanistic process like manufacturing. In the words of Brooks [Bro95]: "adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well coordinated manner.

Management myths

- **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 software projects.

Customer myths

- **Myth:** *A general statement of objectives is sufficient to begin writing programs—we can fill in the details later*

- **Reality:** Although a comprehensive and stable statement of requirements is not always possible, an ambiguous "statement of objectives" is a recipe for disaster. Unambiguous requirements (usually derived iteratively) are developed only through effective and continuous communication between customer and developer.

Customer myths

- **Myth:** *Software requirements continually change, but change can be easily accommodated because software is flexible.*

- **Reality:** It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small.[16] However, as time passes, the cost impact grows rapidly—resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification.

Practitioner’s myths

- **Myth:** *Once we write the program and get it to work, our job is done*
- **Reality:** Someone once said that “the sooner you begin ‘writing code,’ the longer it’ll take you to get done.” Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Practitioner's myths

- **Myth:** *Until I get the program "running" I have no way of assessing its quality.*

- **Reality:** One of the most effective software quality assurance mechanisms can be applied from the inception of a project—*the technical review.* Software reviews (described in Chapter 15) are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects

Practitioner’s myths

- **Myth:** *The only deliverable work product for a successful project is the working program.*
- **Reality:** A working program is only one part of a software configuration that includes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering and, more important, guidance for software support

Slides copyright 2009 by Roger Pressman.

Practitioner's myths

- **Myth:** *Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.*

- **Reality:** Software engineering is not about creating documents. It is about creating a quality product. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

How It all Starts

- *SafeHome:*
  - Every software project is precipitated by some business need—
    - the need to correct a defect in an existing application;
    - The need to adapt a ‘legacy system’ to a changing business environment;
    - the need to extend the functions and features of an existing application, or
    - the need to create a new product, service, or system.

Chapter 2: Summary

- Software engineering encompasses process, methods, and tools that enable complex computer-based systems to be built in a timely manner with quality.
- The software process incorporates five framework activities—communication, planning, modeling, construction, and deployment—that are applicable to all software projects.
- Software engineering practice is a problem solving activity that follows a set of core principles.
- A wide array of software myths continue to lead managers and practitioners astray, even as our collective knowledge of software and the technologies required to build it grows.