Chapter 1
SOFTWARE PROCESS MATURITY

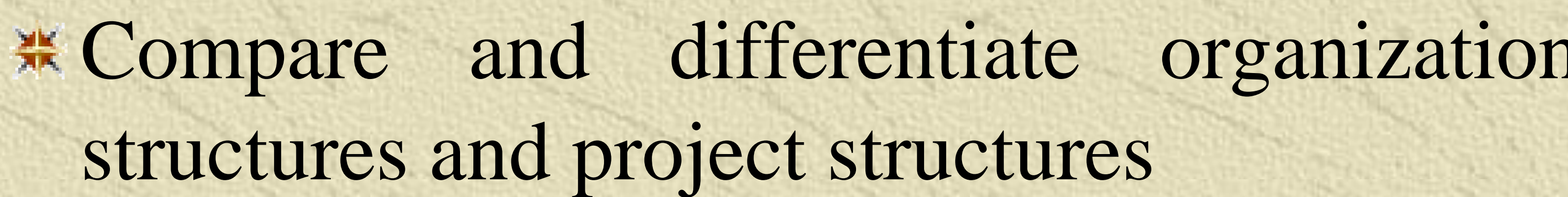
Course Objectives:

The main goal of software development projects is to create a software system with a predetermined functionality and quality in a given time frame and with given costs. For achieving this goal, models are required for determining target values and for continuously controlling these values.

This course focuses on principles, techniques, methods & tools for model-based management of software projects, assurance of product quality and process adherence (quality assurance), as well as experience-based creation & improvement of models (process management).

- The goals of the course can be characterized as follows:
- Understanding the specific roles within a software organization as related to project and process management
- Describe the principles, techniques, methods & tools for model-based management of software projects, assurance of product quality and process adherence (quality assurance), as well as experience-based creation & improvement of models (process management).

- Understanding the basic infrastructure competences (e.g., process modeling and measurement)
- Understanding the basic steps of project planning, project management, quality assurance, and process management and their relationships
- **Course Outcomes:**
- Describe and determine the purpose and importance of project management from the perspectives of planning, tracking and completion of project.



- Compare and differentiate organization structures and project structures
- Implement a project to manage project schedule, expenses and resources with the application of suitable project management tools

Introduction:

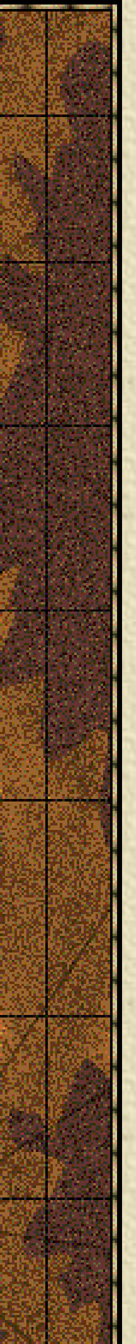
- 1. **IMPORTANT QUOTES:**
  If you don't know where you are going, any road will do." Chinese Proverb
- "If you don’t know where you are, a map won't help." Watts Humphrey
- "If you don't know where you are going, a map won't get you there any faster." Anonymous
  "You can't expect to be a functional employee in a dysfunctional environment" Watts Humphrey

- **WHY SHOULD WE MANAGE THE SOFTWARE PROCESS?**
  Individuals, Teams, and Armies:
  History of software is one of increasing scale
- Initially a few people could craft small programs
- Today large projects require the coordinated work of many teams
- The increase in scale requires a more structured approach to software process management
- People and the Software Process
- Talented people are the most important element in a software organization
- Successful organizations provide a structured and disciplined environment to do cooperative work

MYTH OF THE SUPER PROGRAMMERS:

- Common view: First-class people intuitively know how to do first-class work
  - Implication: No orderly process framework is needed
  - Conclusion: Organizations with the best people should not suffer from software quality and productivity problems
- However, studies show that companies with top graduates from leading universities are still plagued with the same problems

- **MYTH OF TOOLS AND TECHNOLOGY:**
- Common View: Some technically advanced tool or method will provide a magic answer to the software crisis
- Reality: Technology is vital, but unthinking reliance on an undefined "silver bullet" will divert attention from the need for better process management
- **MAJOR CONCERNS OF SOFTWARE PROFESSIONALS:**
- Open-ended requirements
- Uncontrolled change
- Arbitrary schedules
- Insufficient test time
- Inadequate training
- Unmanaged system standards



- **LIMITING FACTORS IN USING SOFTWARE TECHNOLOGY:**
- Poorly-defined process
- Inconsistent implementation
- Poor process management
- **FOCUSING ON SOFTWARE PROCESS MANAGEMENT:**
- Software process: the set of actions required to efficiently transform a user's need into an effective software solution
- Many software organizations have trouble defining and controlling this process
  - "

- **A SOFTWARE MATURITY FRAMEWORK:**

- Software maturity Framework: Fundamentally, software development must be predictable. The software process is the set of tools, methods, and practices we use to produce a software product. The objectives of software process management are to produce products according to plan while simultaneously improving the organization's capability to produce better products. The basic principles are those of statistical process control

- •Effective change requires great knowledge of the current process
- •Change is continuous
- •Software process changes will not be retained without conscious effort and periodic reinforcement
- •Software process improvement requires investment.
- Continuous Change:
- •Reactive changes generally make things worse
- •Every defect is an improvement opportunity
- •Crisis prevention is more important than crisis recovery **SOFTWARE PROCESS ASSESSMENT**
- Process assessments help software organizations improve themselves by identifying their
- crucial problems and establishing improvement priorities. The basic assessment objectives

- **ASSESSMENT OVERVIEW:**
- A software assessment is not an audit. Audits are conducted for senior managers who suspect problems and send in experts to uncover them. A software process assessment is a review of a software organization to advise its management and professionals on how they can improve their operation.
- The phases of assessment are:
- •Preparation - Senior management agrees to participate in the process and to take actions on the resulting recommendations or explain why not. Concludes with a training program for the

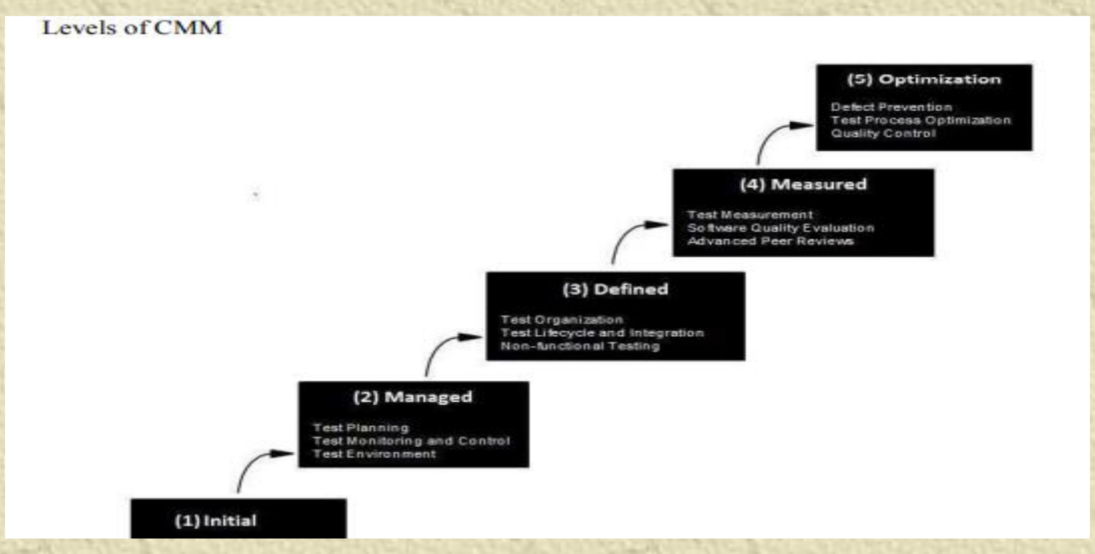
- **THE INITIAL PROCESS(LEVEL1)**
- Usually ad hoc and chaotic - Organization operates without formalized procedures, cost estimates, and project plans. Tools are neither well integrated with the process nor uniformly applied. Change control is lax, and there is little senior management exposure or understanding of the problems and issues.
- **THE REPEATABLE PROCESS (LEVEL 2)**
- This level provides control over the way the organization establishes plans and commitments. This control provides such an improvement over Level 1 that the people in the organization tend

- **THE DEFINED PROCESS (LEVEL 3)**
- The organization has the foundation for major and continuing change. When faced with a crisis, the software teams will continue to use the same process that has been defined.
- However, the process is still only qualitative; there is little data to indicate how much is accomplished or how effective the process is. There is considerable debate about the value of software process measurements and the best one to use.

- **THE MANAGED PROCESS (LEVEL 4)**
- Largest problem at Level 4 is the cost of gathering data. There are many sources of potentially valuable measure of the software process, but such data are expensive to collect and maintain.
- Productivity data are meaningless unless explicitly defined. For example, the simple measure of lines of source code per expended development month can vary by 100 times or more,

- **THE OPTIMIZING PROCESS (LEVEL 5)**
- To this point software development managers have largely focused on their products and will typically gather and analyze only data that directly relates to product improvement. In the Optimizing Process, the data are available to tune the process itself.

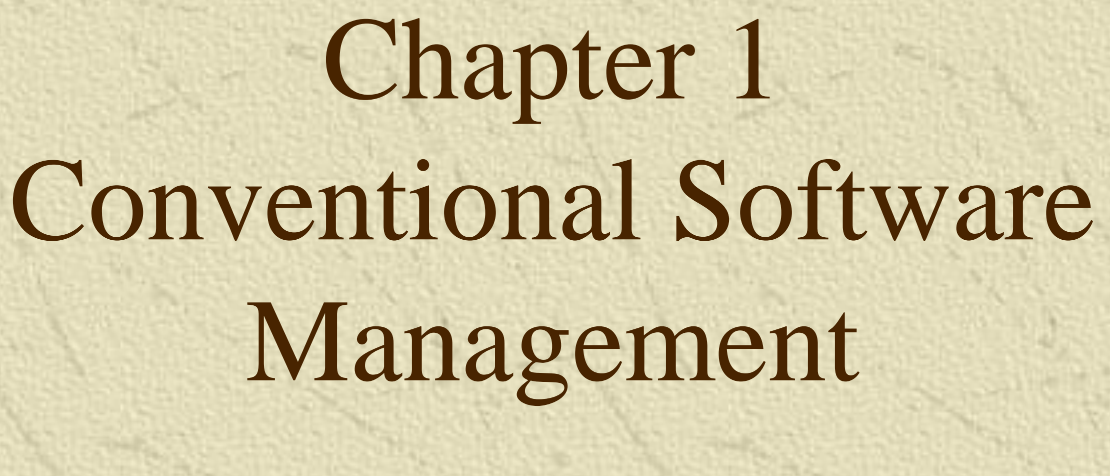
CAPABILITY MATURITY MODEL (CMM):



- **WHAT IS CMMI ?**
- CMM Integration project was formed to sort out the problem of using multiple CMMs.
- CMMI Product Team's mission was to combine three Source Models into a single
- improvement framework to be used by the organizations pursuing enterprise-wide process
- improvement. These three Source Models are :
  - Capability Maturity Model for Software (SW-CMM) - v2.0 Draft C
  - Electronic Industries Alliance Interim Standard (EIA/IS) - 731 Systems Engineering
  - Integrated Product Development Capability Maturity

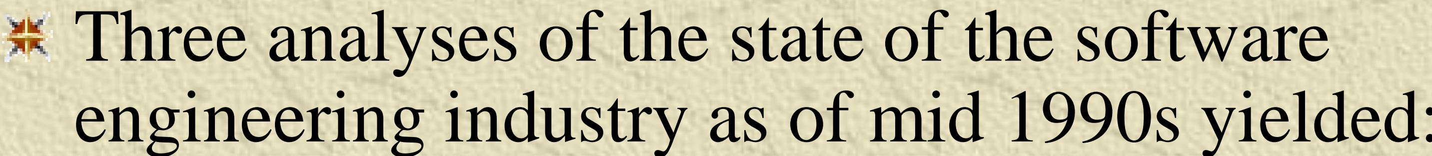
- **PSP**
- The Personal Software Process (PSP) is a structured software development process that is designed to help software engineers better understand and improve their performance by bringing discipline to the way they develop software and tracking their predicted and actual development of the code.

- **TSP**
- The team software process (TSP) provides a defined operational process framework that is designed to help teams of managers and engineers organize projects and produce software the principles products that range in size from small projects of several thousand lines of code (KLOC) to very large projects greater than half a million lines of code


Chapter 1
Conventional Software
Management

Introduction:

- 1. The best thing about software is its flexibility:
- - It can be programmed to do almost anything.
- 2. The worst thing about software is its flexibility:
- - The “almost anything” characteristic has made it difficult to plan, monitor, and control software
- development.
- 3. In the mid-1990s, three important analyses were performed on the software engineering industry

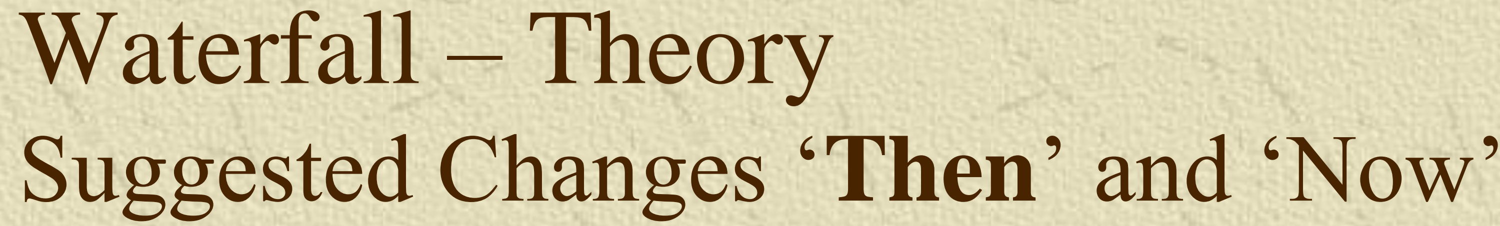


- Three analyses of the state of the software engineering industry as of mid 1990s yielded:
  - Software Development is still highly unpredictable
    - Only about 10% of software projects are delivered successfully on time, within initial budget, and meets user requirements
    - ➔The management discipline is <u>more</u> of a discriminator in success or failure than are technology advances
    - The level of software scrap and rework is indicative of an immature process.
- Behold the magnitude of the software problem and current norms!
- But is the ‘theory’ bad? “Practice bad?” Both?
- Let’s consider….

I. The Waterfall Model

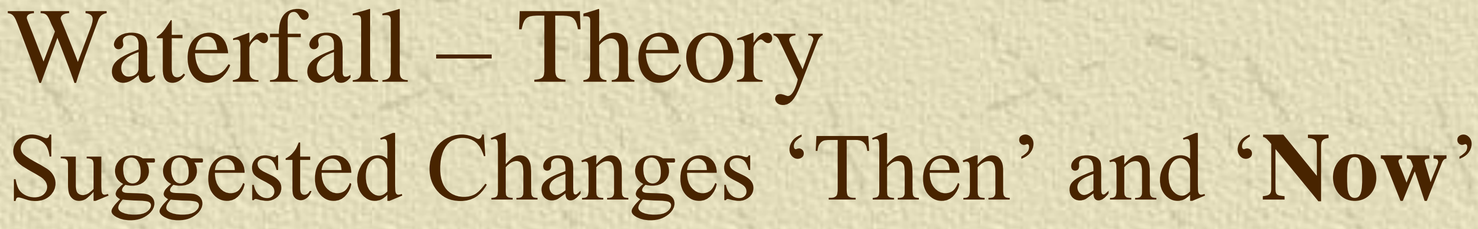
- Recognize that there are numerous variations of the ‘waterfall model.’
  - Tailored to many diverse environments
- The ‘theory’ behind the waterfall model – good
  - Oftentimes ignored in the ‘**practice**’
- The ‘practice’ – some good; some poor

- Circa 1970: lessons learned and observations
  - Point 1: There are two essential steps common to the development of computer programs: **analysis and coding** - More later on this one.
  - Point 2: In order to manage and control all of the intellectual freedom associated with software development, one must introduce several other 'overhead' steps, including system requirements definition, software requirements definition, program design, and testing. These steps supplement the analysis and coding steps." (See Fig 1-1, text, p. 7, which model basic programming steps and large-scale approach)
  - Point 3: The basic framework … is risky and invites failure. The testing phases that occurs at the end of the development cycle is the first event for which timing, storage, input/output transfers, etc. are experienced as distinguished from analyzed. The resulting design changes are likely to be so disruptive that the software requirements upon which the design is based are likely violated. Either the **requirements must be modified or a substantial design change is warranted. ➔ Discuss.**



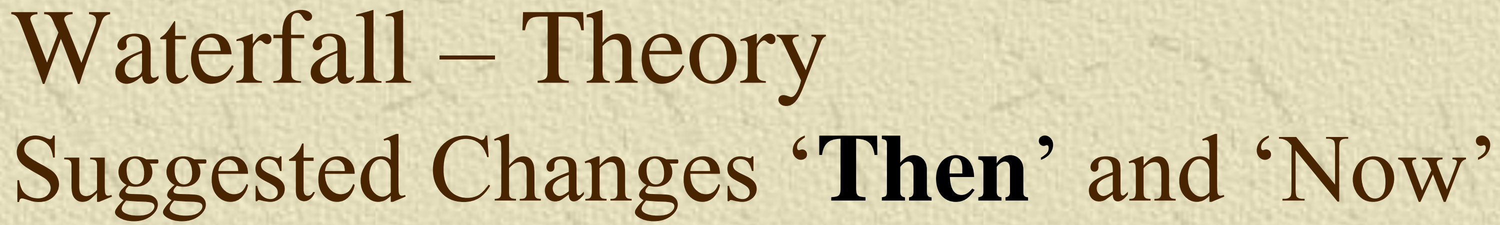
Waterfall – Theory
Suggested Changes '**Then**' and 'Now'

- 1. **"Program design" comes first**.
  - Occurs between SRS generation and analysis.
  - ➔ Program designer looks at storage, timing, data. **Very high level…First glimpse. First concepts…**
  - During analysis: program designer must then impose storage, timing, and operational constraints to determine consequences.
  - Begin design process with program designers, not analysts and programmers
  - Design, define, and allocate data processing modes even if wrong. (allocate functions, database design, interfacing, processing modes, i/o processing, operating procedures…. Even if wrong!!)
  - ➔ Build an overview document – to gain a basic understanding of system for all stakeholders.



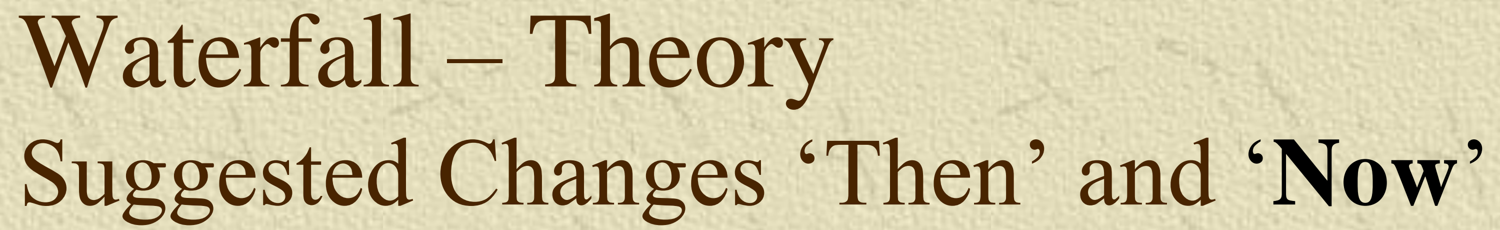
Waterfall – Theory
Suggested Changes 'Then' and '**Now**'

- **Point 1: Update: We use the term '<u>architecture first</u>' development rather than program design.**
  - Elaborate: distribution, layered architectures, components
- Nowadays, the basic architecture MUST come first.
- **Recall the RUP: use-case driven, architecture-centric, iterative development process……**
- Architecture comes **first; then** it is designed and developed in **parallel** with planning and requirements definition.
  - Recall RUP Workflow diagrams….

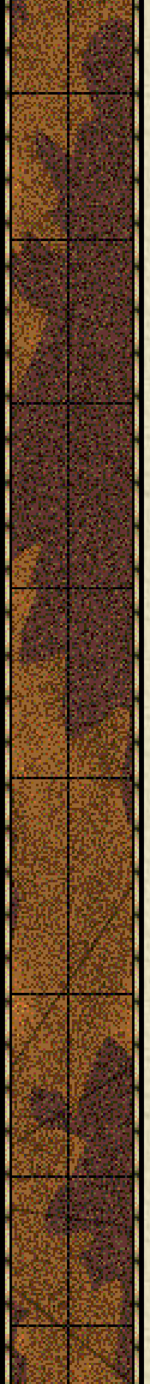


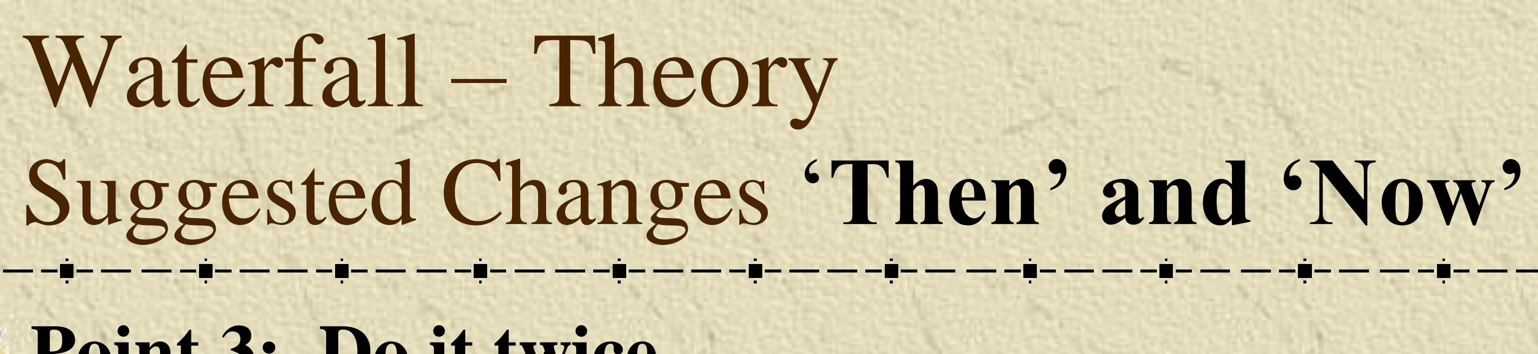
Waterfall – Theory
Suggested Changes '**Then**' and 'Now'

- **Point 2: Document the Design**
  - Development efforts required **huge amounts** of documentation – manuals for everything
    - User manuals; operation manuals, program maintenance manuals, staff user manuals, test manuals…
    - Most of us would like to 'ignore' documentation. ☺
  - Each designer MUST communicate with various stakeholders: interface designers, managers, customers, testers, developers, …..



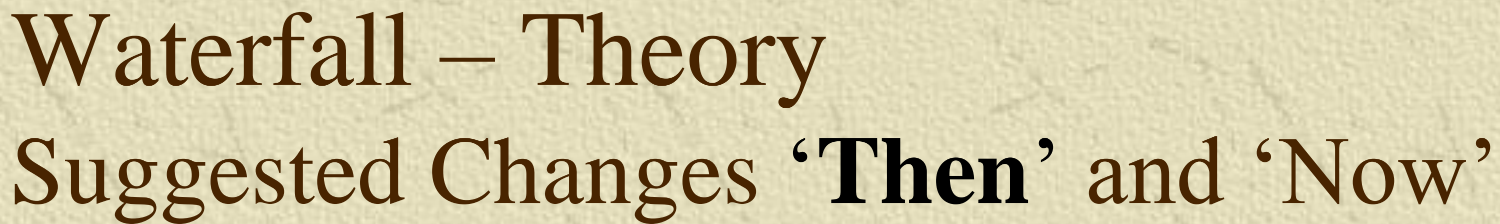
- **Point 2: Update: Document the Design**
  - Now, we concentrate primarily on '**artifacts**' – those models produced as a result of developing an architecture, performing analysis, capturing requirements, and deriving a design solution
    - Include Use Cases, static models (class diagrams, state diagrams, activity diagrams), dynamic models (sequence and collaboration diagrams), domain models, glossaries, supplementary specifications (constraints, operational environmental constraints, distribution, ….)
    - Modern tools / notations, and methods produce **self-documenting artifacts from development activities.**
    - **Visual modeling provides considerable documentation**





Waterfall – Theory
Suggested Changes **‘Then’ and ‘Now’**

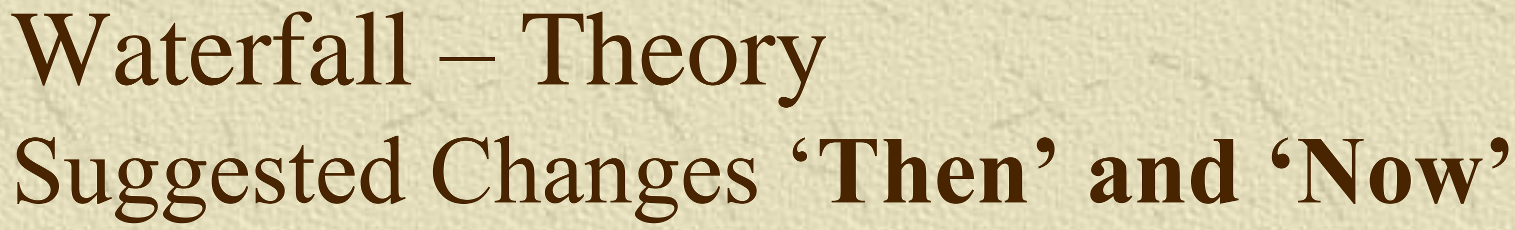
- **Point 3: Do it twice**.
  - History argues that the delivered version is really version #2. Microcosm of software development.
  - Version 1, major problems and alternatives are addressed – the ‘big cookies’ such as communications, interfacing, data modeling, platforms, operational constraints, other constraints. Plan to throw first version away sometimes…
  - Version 2, is a refinement of version 1 where the major requirements are implemented.
  - Version 1 often austere; Version 2 addressed shortcomings!
- **Point 3: Update**.
  - This approach is a precursor to architecture-first development (see RUP). Initial engineering is done. Forms the basis for **iterative development** and addressing **risk**!



- **Point 4: <u>Then</u>: Plan, Control, and Monitor Testing.**
  - Largest consumer of project resources (manpower, computing time, …) is the test phase.
    - ➔ Phase of greatest risk – in terms of cost and schedule. **(EST 1…)**
    - Occurs last, when alternatives are least available, and expenses are at a maximum.
    - Typically that phase that is **shortchanged** the most
  - To do:
    - 1. Employ a non-vested team of test specialists – not responsible for original design.
    - 2. Employ visual inspections to spot obvious errors (code reviews, other technical reviews and interfaces)
    - 3. Test every logic path
    - 4. Employ final checkout on target computer…..

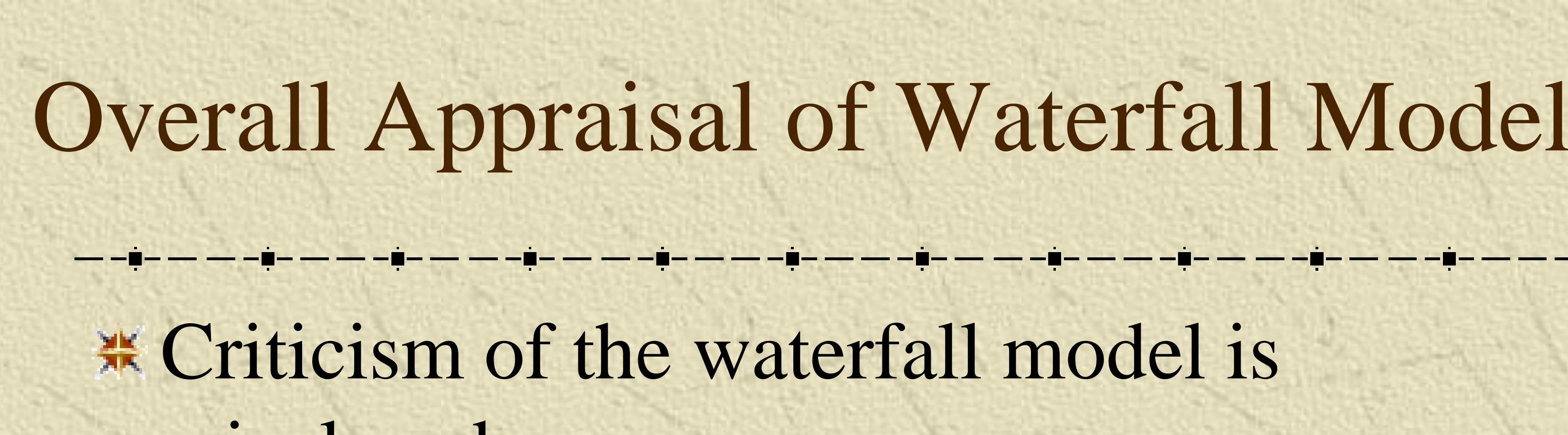
Waterfall – Theory Suggested Changes 'Then' and '**Now**'

- **Point 4: <u>Now</u>: Plan, Control, and Monitor Testing.**
  - Items 1 and 4 are still valid.
    - 1) Use a test team not involved in the development of the system – at least for testing other than 'unit testing…'
    - 4) Employ final checkout on target computer….
  - Item 2 (software inspections) – <u>good</u> <u>years</u> <u>ago</u>, but modern development environments obviate this need. Many code analyzers, optimizing compilers, static and dynamic analyzers are available to automatically assist…
    - May still yield good results – but not for significant problems! Stylistic!
  - Item 3 (testing every path) is <u>impossible</u>. Very difficult with distributed systems, reusable components (necessary?), and other factors…. (aspects)



- **Point 5** – <u>Old</u>: Involve the Customer
- Old advice: involve customer in requirements definition, preliminary software review, preliminary program design (critical design review briefings…)
- Now: Involving the customer and all stakeholders **is <u>critical</u> to overall project success. Demonstrate increments; solicit feedback; embrace change; cyclic and iterative and evolving software. Address risk early…..**

Overall Appraisal of Waterfall Model



- Criticism of the waterfall model is misplaced.
- Theory is fine.
- Practice is what was poor!

The Software Development Plan: *Old Version*

- Define precise requirements
- Define precise plan to deliver system
  - Constrained by specified time and budget
- Execute and track to plan

Planned Path

- Reused or legacy assets
- Detailed plans, scope

Stakeholder Satisfaction Space

But: Less than 20% success rate

- Characteristics of Conventional Process – as it has been applied (in general)
- Projects not delivered on-time, not within initial budget, and rarely met user requirements
- Projects frequently had:
  - 1. Protracted integration and late design breakage
  - 2. Late risk resolution
  - 3. Requirements-driven functional decomposition
  - 4. Adversarial stakeholder relationships
  - 5. Focus on documents and review meetings
- Let’s look at these five major problems…

1. Protracted Integration and Late Design Breakage

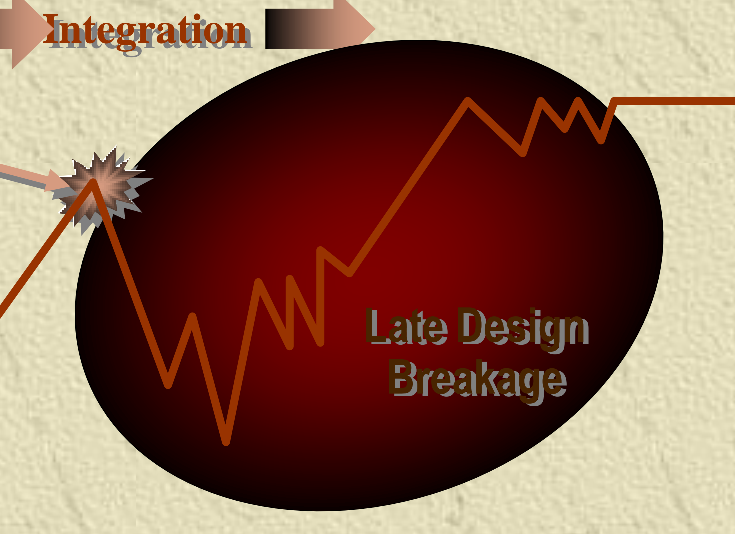
Symptoms of conventional waterfall process

- Late design breakage
- 40% effort on integration & test

Sequential Activities:

Requirements → Design → Code → Integration →

Integration Begins



100%

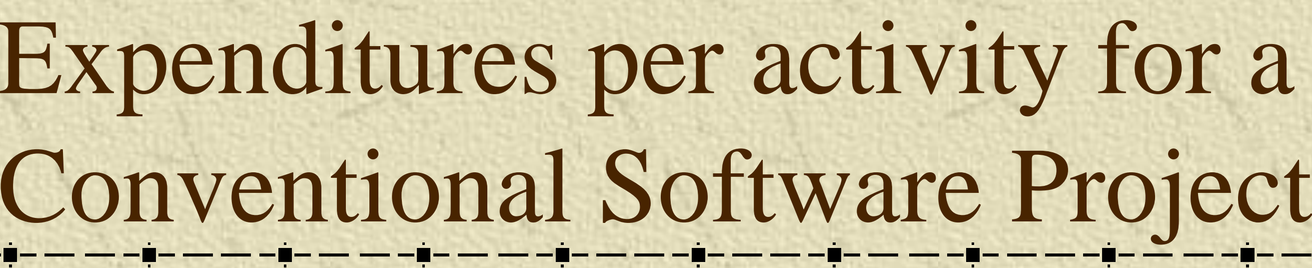
Development Progress (% coded)

. Early paper designs and thorough briefings
. Commitment to code very late in cycle
. Integration nightmares due to unforeseen implementation and interface issues
. Heavy budget and schedule pressure
. Late 'shoe-horning' of non-optimal fixes with no time for redesign!!!!
. A very fragile, un-maintainable product and almost always: **delivered late.**

Original Target Date

Completion Date

Project Schedule



Expenditures per activity for a Conventional Software Project

Activity	Cost
Management	5%
Requirements	5%
Design	10%
Code and unit test	30%
Integration and Test	40%
Deployment	5%
Environment	5%
Total	100%

➔ Lots of time spent on ‘**perfecting the software design**’ prior to commitment to code.

➔ Typically had: requirements in English, design in flowcharts, detailed design in pdl, and implementations in Fortran, Cobol, or C

Waterfall model ➔ late integration and **performance showstoppers**.

Could only perform testing ‘at the end’ (other than unit testing)

Testing ‘should have’ required 40% of life-cycle resources: often didn’t!!

2. Late Risk Resolution

- Problem here: ➔ **focused on early paper artifacts**.
- Real issues – still unknown and hard to grasp.
  - Difficult to resolve risk during requirements when many key items still not fully understood.
  - Even in design, when requirements better understood, still difficult to get objective assessment.
    - Risks were at a very high level
  - During coding, some risks resolved, BUT during
  - ➔ **Integration**, many risks were quite clear and changes to many artifacts and retrenchment often **had** to occur

While much 'retrenchment '**did**' occur, it often caused **missed dates, delayed requirement compliance, or, at a minimum, sacrificed quality (extensibility, maintainability, loss of original design integrity, and more).**

Quick fixes, often without documentation occurred a lot!

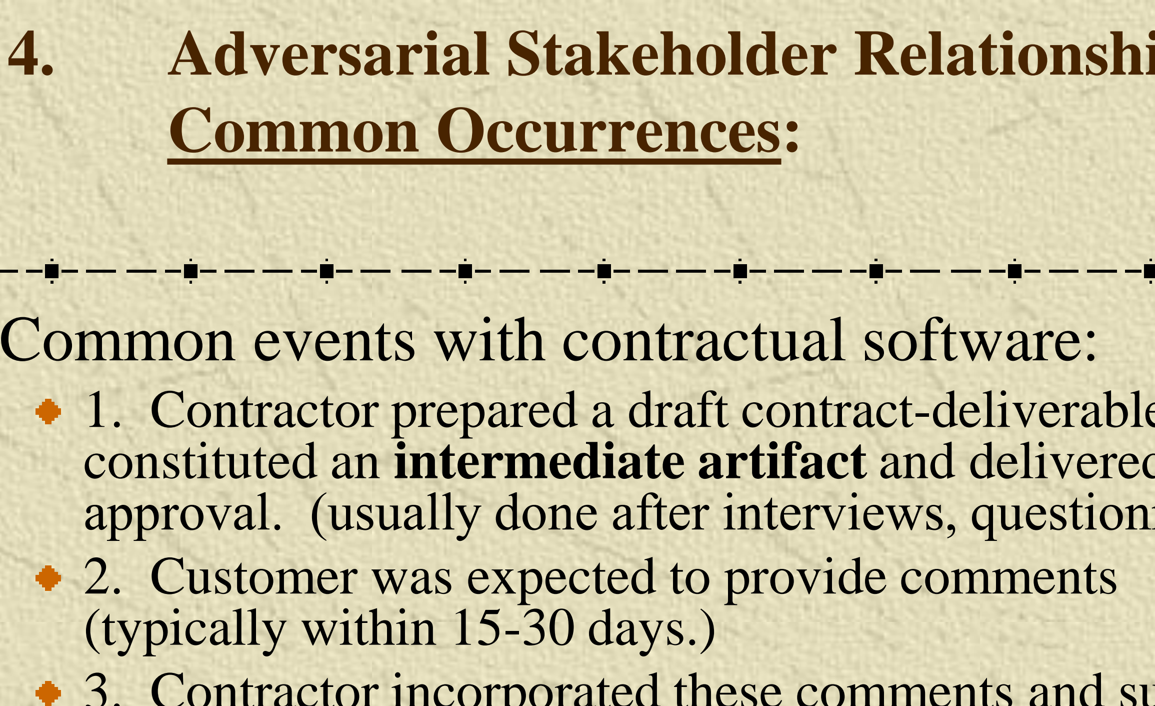
3. Requirements-Driven Functional Decomposition

- Traditionally, software development processes have been **requirements-driven.**
  - Developers: assumed requirement specs: complete, clear, necessary, feasible, and remaining constant! This is **RARELY** the case!!!!
  - All too often, too much time spent on **equally** treating ‘all’ requirements rather than on critical ones.
  - Much time spent on **documentation** on topics (traceability, testability, etc.) that was later made **obsolete** as ‘DRIVING REQUIREMENTS AND SUBSEQUENT DESIGN UNDERSTANDING **EVOLVE**.’ We do not KNOW all we’d like to know ‘up front.’
  - Too much time addressing all of the scripted requirements
    - normally listed in tables, decision-logic tables, flowcharts, and **plain, old text**.
    - Much brainpower wasted on the ‘**lesser**’ requirements.
  - Also, assumption that all requirements could be captured as ‘**functions**’ and resulting **decomposition** of these **functions**.
  - Functions, sub-functions, etc. became the basis for contracts and work apportionment, while ignoring **major architectural-driven approaches and requirements** that are ‘threaded’ throughout functions and that transcend individual functions…... (security; authentication; persistency; performance…)
  - **Fallacy**: all requirements can be completely specified ‘up front’ and (and decomposed) via functions.

4. Adversarial Stakeholder Relationships (1 of 2)

- Who are stakeholders? **Discuss**….Quite a diverse group!
- Adversarial relationships OFTEN true!
- Misunderstanding of documentation usually written in English and with business jargon.
- Paper transmission of requirements – only method used….
- No real modeling, universally-agreed-to languages with common notations; (no GUIs, network components already available; Most systems were 'custom.')
- Subjective reviews / opinions. Generally without value!
- …more→
- Management Reviews; Technical Reviews!

4. Adversarial Stakeholder Relationships <u>Common Occurrences</u>:



- Common events with contractual software:
  - 1. Contractor prepared a draft contract-deliverable document that constituted an **intermediate artifact** and delivered it to the customer for approval. (usually done after interviews, questionnaires, meetings…)
  - 2. Customer was expected to provide comments (typically within 15-30 days.)
  - 3. Contractor incorporated these comments and submitted (typically 15-30 days) a final version for approval.
- Evaluation:
  - Overhead of paper was huge and ‘intolerable.’ Volumes of paper! (often under-read)
  - **<u>Strained</u>** contractor/customer relationships
  - Mutual distrust – basis for much litigation
  - Often, once approved, rendered obsolete later….(<u>living document</u>?)

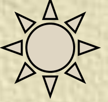
5. Focus on Documents and Review Meetings

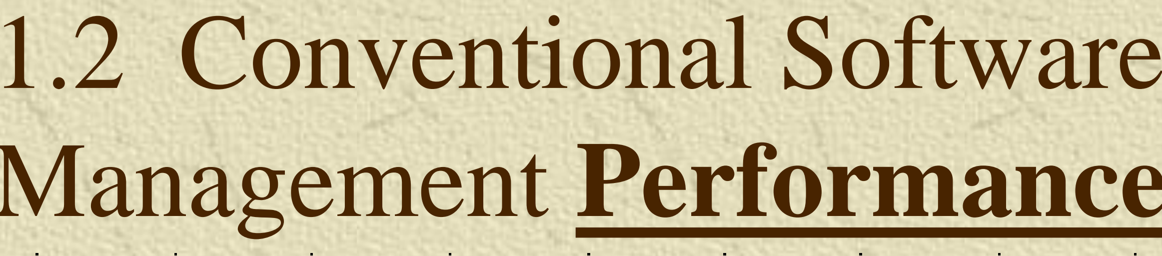
- **A very documentation-intensive approach.**
- Insufficient attention on producing credible '**increments**' of the desired products.
  - **Big bang approach – all FDs delivered at once;**
  - **All Design Specs 'ok'd' at once and 'briefed'…**
- Milestones 'commemorated' via **review meetings – technical, managerial, ….. Everyone nodding and smiling often…**
- Incredible energies expended on producing paper documentation to show **progress** versus efforts to address **real risk issues and integration issues.**
  - Stakeholders often did not go through design…
  - Very VERY low value in meetings and high costs
    - Travel, accommodations…..
- Many issues could have been averted early during development – during **early** life-cycle phases rather than encountered **huge** problems **late**….but…

Continuing….

Typical Software product design **Reviews**….

- 1. Big briefing to a diverse audience
  - Results: only a small percentage of the audience understands the software
  - Briefings and documents expose **few** of the important assets and risks of complex software.
- 2. A design that **appears** to be compliant
  - There is no tangible evidence of compliance
  - Compliance with ambiguous requirements is of little value.
- 3. Coverage of requirements (typically hundreds….)
  - Few (tens) are in reality the **real** design drivers, but many **presented**
  - Dealing with **all** requirements dilutes the focus on **critical drivers**.
- 4. A design considered ‘innocent until proven guilty’
  - **The design is always guilty**
  - **Design flaws are exposed later in the life cycle.**





1.2 Conventional Software Management **Performance**

- Very few changes from **Barry Boehm's** "industrial software metrics" from 1987.
- Most still generally describe some of the fundamental economic relationships that are derived from years of practice:
- What follows is Barry's top ten (and your author's (and my) comments.

Basic Software Economics…

- 1. Finding and fixing a software problem after delivery costs 100 times more than fining and fixing the problem in early design phases.
  - Flat true.
- 2. You can compress software development schedules 25% of nominal, but no more.
  - **Addition** of people requires **more management overhead and training of people.**
  - Still a good heuristic. Some compression is sometimes possible! Be **careful! Oftentimes it is a killer to add people….(Discuss later)**
- 3. For every dollar you spend on development, you will spend two dollars on maintenance. **We HOPE this is true**!
  - Hope so. Long life cycles mean revenue…Still, hard to tell
  - **Product's success in market place is driver**.
  - Successful products will have much higher ratios of "maintenance to development"…..
  - One of a kind development will most likely NOT spend this kind of money on maintenance .
    - Examples: implementation / conversion subsystems…..
    - Conversion software….

Basic Software Economics (cont)

- 4. Software development and maintenance costs are primarily a function of the number of **source lines of code**.
  - Generally true. **Component-based development** may dilute this as might **reuse -** but not in common use in the past.
- 5. **Variations among people** account for the biggest differences in software productivity.
  - **Always try to hire good people**. But we cannot always to that. **Balance is critical**. Don't want all team members trying to self-actualize and become heroes. Build the 'team concept.' While there is no "I" in 'team", there is an implicit "we."
- 6. Overall ratio of software to hardware costs is still growing. In 1955 it was 15:85; In 1985, it was 85:15. Now? I don't know.
  - While true, **impacting these figures is the ever-increasing demand for functionality and attendant complexity. They appear w/o bound.**

Basic Software Economics (cont)

- 7. ➔ Only about 15% of software development effort is devoted to programming. **(Sorry! But this is the way it is!)**
  - Approximately true. This figure has been used for years – and is shattering to a lot of programmers – especially 'new' ones. And, this 15% is only for the development! It does not include, hopefully, some 65% - 70% of the overall total life cycle expenses based on maintenance!!
- 8. Software systems and products typically cost three times as much per SLOC as individual software programs. Software-system products, that is system of systems, cost nine times as much.
  - **A real fact: the more software you build, the more expensive it is per source line. Why do you think? Discuss!**

Basic Software Economics (cont)

- 9. Walkthroughs catch 60% of the errors.
  - Usually good for catching stylistic things; sometimes errors, but usually do not represent / require the **deep analysis necessary to catch significant shortcomings.**
  - **Major problems, such as performance, resource contention, … are not caught.**
- 10. 80% of the contribution comes from 20% of the contributors.
  - 80/20 rule applies to many things: see text. But pretty correct!
    - See text for a number of these – which are 'generally' true….

Evolution of Software Economics

1.3 Software Economics

- Five fundamental parameters that can be abstracted from software costing models:
  - Size
  - Process
  - Personnel
  - Environment
  - Required Quality
- Overviewed in Chapter 2
- Much more detail in Chapter 3.

Software Economics – Parameters (1 of 4)

- Size: Usually measured in SLOC or number of Function Points required to realize the desired capabilities.
  - Function Points – a better metric earlier in project
  - LOC (SLOC, KLOC…) a better metric later in project
  - These are not new metrics for measuring size, effort, personnel needs,…

- Process – used to guide all activities.
  - Workers (roles), artifacts, activities…
  - Support heading toward target and eliminate non-essential / less important activities
  - Process **critical** in determining software economics
    - Component-based development; application domain…iterative approach, use-case driven…
  - Movement toward '**lean**' … everything!

Process

Software Economics – Parameters (2 of 4)

- Personnel – capabilities of the personnel in general and in the application domain in particular
  - Motherhood: get the right people; good people; Can't always do this.
  - Much specialization nowadays. Some are terribly expensive.
  - Emphasize 'team' and team responsibilities…Ability to work in a team;
    - Several newer light-weight methodologies are totally built around a team or very small group of individuals…

Software Economics – Parameters (3 of 4)

- Environment – the tools / techniques / automated procedures used to support the development effort.
  - Integrated tools; automated tools for modeling, testing, configuration, managing change, defect tracking, etc…

- Required Quality – the functionality provided; performance, reliability, maintainability, scalability, portability, user interface utility; usability…

Software Economics – Parameters (4 of 4)

Effort = (personnel)(environment)(quality)(size $_{Process}$)

(Note: effort is exponentially related to size….)

What this means is that a 10,000 line application will cost less per line than a 100,000 line application.

- These figures – surprising to the uninformed – are true.
- Fred Brooks – Mythical Man Month – cites over and over that the additional communications incurred when adding individuals to a project is very significant.
  - Tend to have more reviews, meetings, training, biases, getting people up to speed, personal issues…
- Let's look at some of the trends:

Notice the Process Trends….for three generations of software economics

- Conventional development (60s and 70s)
  - Application – custom; Size – 100% custom
  - Process – ad hoc …(discuss) – laissez faire;
  - 70s - SDLC; customization of process to domain / mission, structured analysis, structured design, code.
- Transition (80s and 90s)
  - Environmental/tools – some off the shelf.
    - Tools: separate, that is, often not integrated esp. in 70s…
  - Size: 30% component-based; 70% custom
  - → Process: repeatable
- Modern Practices (2000 and later)
  - Environment/tools: off-the-shelf; integrated
  - Size: 70% component-based; 30% custom
  - Process: managed; measured (refer to CMM)

Notice Performance Trends….for three generations of software economics

- Conventional: Predictably bad: (60s/70s)
  - usually always over budget and schedule; missed requirements
    - All custom components; symbolic languages (assembler); some third generation languages (COBOL, Fortran, PL/1)
    - Performance, quality almost always less than great.
- Transition: Unpredictable (80s/90s)
  - Infrequently on budget or on schedule
  - Enter software engineering; 'repeatable process;' **project management**
  - Some commercial products available – databases, networking, GUIs; But with huge growth in complexity, (especially to distributed systems) existing languages and technologies not enough for desired business performance
- Modern Practices: Predictable (>2000s)
  - Usually on budget; on schedule. Managed, measured **process management**. Integrated environments; 70% off-the-shelf components. Component-based applications RAD; iterative development; stakeholder emphasis.

All Advances Interrelated…

- Improved ‘process’ requires ‘improved tools’ (environmental support…)
- Better ‘economies of scale’ because
  - → Applications live for years;
  - Similarly-developed applications – common.
  - First efforts in common <u>architectures</u>, <u>processes</u>, <u>iterative processes</u>, etc., all have <u>**initial**</u> <u>high overhead;</u>
  - But follow-on efforts result in economies of scale…and much better ROI. (See p. 25)
  - “All simple systems have been developed!”

2.2 "Pragmatic" Software Cost Estimation

- Little available on estimating cost for projects using iterative development.
  - Difficult to hold all the controls constant
    - Application domain; project size; criticality; etc. Very 'artsy.'
    - Metrics (SLOC, function points, etc.) NOT consistently applied EVEN in the same application domain!
    - Definitions of SLOC and function points are not even consistent!
  - Much of this is due to the nature of development. There is no magic date when design is 'done;' or magic date when testing 'begins' …
  - Consider some of the issues:

Three Issues in Software Cost Estimation:

- 1. Which cost estimation model should be used?
- 2. Should software size be measured using SLOC or Function Points? (there are others too…)
- 3. What are the determinants of a good estimate? (How do we know our estimate is good??)

So very much is dependent upon estimates!!!!

Cost Estimation Models

- Many available.
- Many organization-specific models too based on their own histories, experiences…
  - Oftentimes, these are super if ‘other’ parameters held constant, such as process, tools, etc. etc.
- COCOMO, developed by Barry Boehm, is the most popular cost estimation model.
- Two primary approaches:
  - Source lines of code (SLOC) and
  - Function Points (FP)
- Let’s look at this – overview.

Source Lines of Code (SLOC)

- Many feel comfortable with 'notion' of LOC
- SLOC has great value – especially where applications are custom-built.
  - Easy to measure & instrument – have tools.
  - Nice when we have a history of development with applications and their existing lines of code and associated costs.
- Today – with use of components, source-code generation tools, and objects have rendered SLOC somewhat ambiguous.
  - We often don't know the SLOC – but do we care? How do we factor this in? →

Source Lines of Code (SLOC)

- Generally more useful and precise basis than FPs
- Appendix D – an extensive case study.
  - Addresses how to count SLOC where we have reuse, different languages, etc.
  - Read this appendix (five pages)

- We will address LOC in much more detail later.

- Appendix provides hint at the complexity of using LOC for software sizing particularly with the new technologies using automatic code generation, components, development of new code, and more.

Function Points

- Use of Function Points - many proponents.
  - International Function Point User's Group – 1984 – "is the dominant software measurement association in the industry."
  - Check out their web site (www.IFPUG.com ??)
  - Tremendous amounts of information / references
  - Attempts to create industry standards….

- ➔ Major advantage: Measuring with function points is independent of the technology (programming language, tools …) used and is thus better for comparisons among projects. →

Function Points

- Function Points measure numbers of
  - external user inputs,
  - external outputs,
  - internal data groups,
  - external data interfaces,
  - external inquiries, etc.
- ➔ Major disadvantage: Difficult to measure these things.
  - Definitions are primitive and inconsistent
  - Metrics difficult to assess especially since normally done earlier in the development effort using more abstractions.
- Yet, no project will be started without estimates!!!!

But:

- Cost estimation is a real necessity!!! Necessary to 'fund' project!
- All projects require estimation in the beginning (inception) and adjustments…
  - These must stabilize; They are rechecked…
  - Must be **reusable** for additional cycles
  - Can create organization's own methods of measurement on how to 'count' these metrics...
- No project is arbitrarily started without cost / schedule / budget / manpower / resource estimates (among other things)
- → SO critical to budgets, resource allocation, and to a host of stakeholders

So, How Good are the Models?

- COCOMO is said to be 'within 20%' of actual costs '70% of the time.' (COCOMO has been revised over the years…)

- Cost estimating is still **disconcerting** when one realizes that there are already a plethora of missed dates, poor deliverables, and significant cost overruns that characterize traditional development.

- Yet, all non-trivial software development efforts require costing; It is a basic management activity.

- RFPs on contracts force contractors to estimate the project costs for their survival.
- So, let's look at top down and bottom up estimating.

Top Down versus Bottom Up Substantiating the Cost…

- Most estimators perform bottom up costing - **substantiating** a target cost - rather than approaching it a top down, which would yield a 'should cost.'
- Many project managers create a 'target cost' and then play with parameters and sizing until the target cost can be justified…
  - Work backwards!
  - Attempts to win proposals, convince people, …
- Any approach should force the project manager to assess risk and discuss things with stakeholders…

Top Down versus Bottom Up

- Bottom up … substantiating? Good?
  - If well done, it requires considerable analysis and expertise based on much experience and knowledge;
    Development of similar systems a great help; similar technologies…
  - If not well done, causes team members to go ***crazy***! (This is not uncommon)
- Independent cost estimators (consultants…) not reliable.

Author suggests:

- Likely best cost estimate is undertaken by an **experienced project manager**, software architect, developers, and test managers – and this process can be quite iterative!
- **Previous experience is essential**. Risks identifiable, assessed, and factored in.
- When created, the **team must live with** the cost/schedule **estimate**.
- More later in course. But for now→ (Heuristics from our text:)

A Good Project Estimate:

- ➔Is conceived and supported by the project manager, architecture team, development team, and test team accountable for performing the work.
- ➔Is accepted by all stakeholders as ambitious but doable
- Is based on a well-defined software cost model with a credible basis
- ➔ Is based on a database of relevant project experience that includes similar processes, similar technologies, similar environments, similar quality requirements, and similar people, and
- ➔ Is defined in enough detail so that its key risk areas are understood and the probability of success is objectively assessed.

A Good Project Estimate

- Quoting: "An 'ideal estimate' would be derived from a mature cost model with an experience base that reflects multiple similar projects done by the same team with the same mature processes and tools.

- "Although this situation rarely exists when a project team embarks on a new project, good estimates can be achieved in a straightforward manner in later life-cycle phases of a mature project using a mature process."

23