Chapter 1
Conventional Software Management

Introduction

- 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 more 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 <u>numerous</u> 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

Waterfall – Theory Historical Perspective and Update

- 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 <u>between</u> SRS generation and analysis.
  - Program designer looks at storage, timing, data. **Very high level…First glimpse. First concepts…**
  - During analysis: program designer must <u>then</u> impose storage, timing, and operational constraints to determine consequences.
  - Begin design process with <u>program designers</u>, 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, …..

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

- **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**!

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

- **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)

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

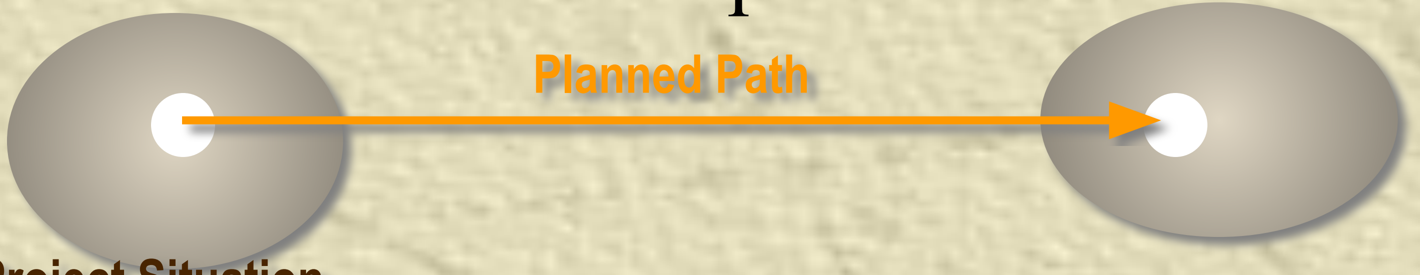
- **Point 5** – <u>Old</u>: Involve the Customer
- Old advice: involve customer in requirements definition, preliminary software review, preliminary program design (critical design review briefings…)
- Now: Involving the customer and all stakeholders **is <u>critical</u> to overall project success. Demonstrate 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



Initial Project Situation

- Reused or legacy assets
- Detailed plans, scope

Stakeholder Satisfaction Space

But: Less than 20% success rate

1.1.2 In Practice

- 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…

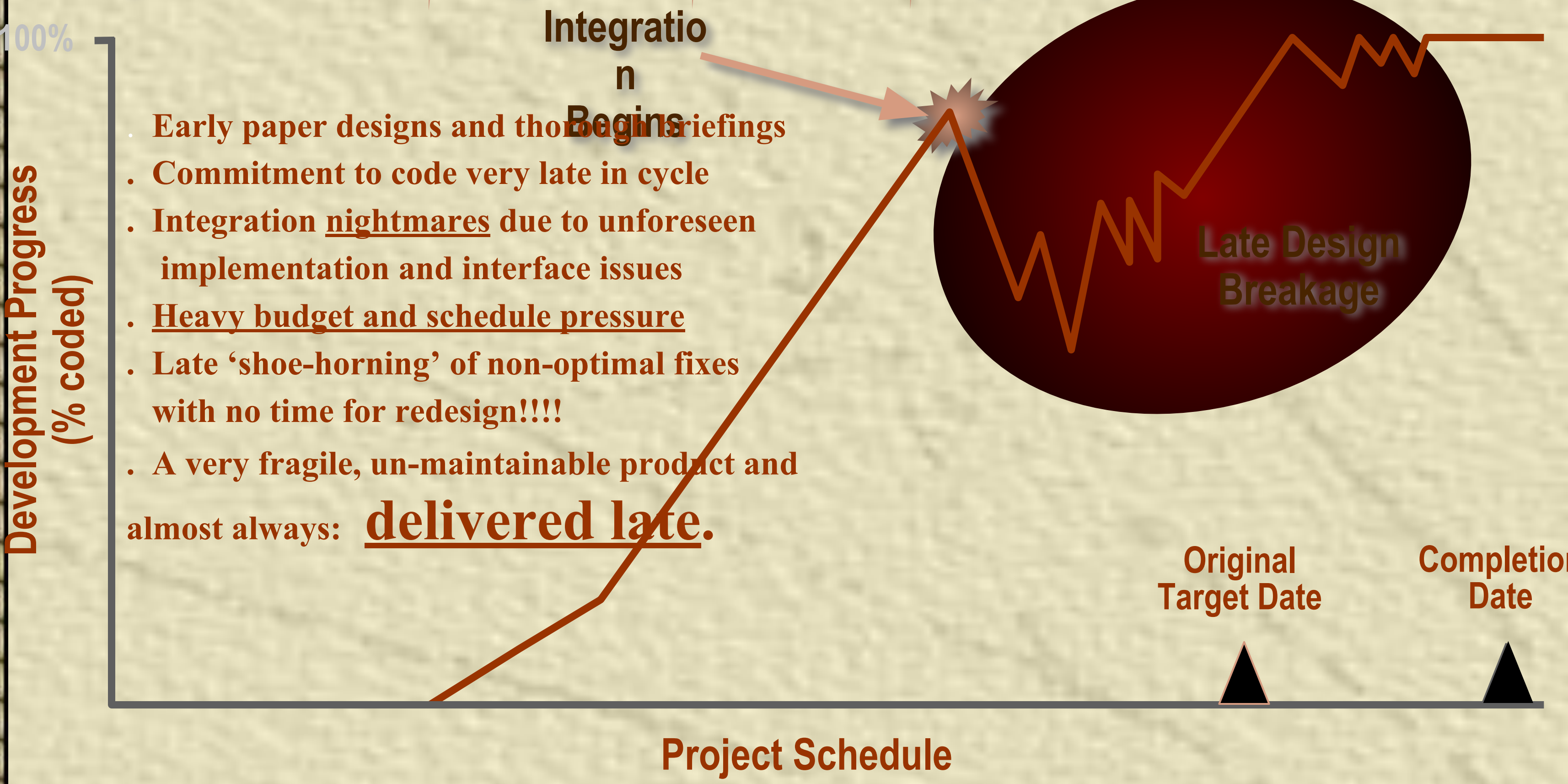
1. Protracted Integration and Late Design Breakage

Symptoms of conventional waterfall process

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

Sequential Activities:

Requirements → Design → Code → Integration →



. 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.**

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 <u>Relationships</u> (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
Common Occurrences:

- 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)
  - **Strained** contractor/customer relationships
  - Mutual distrust – basis for much litigation
  - Often, once approved, rendered obsolete later….(living document?)

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….