Conventional Software Management

Dr. P. Dileep Kumar Reddy
Professor-Dean-R&D,IPR, IIC
CSE Department
Narsimha Reddy Engineering College (Autonomous)
Secunderabad, Telangana State, India- 500100.
Ph.No: 09959845657

Unit 2
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.

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

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 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, …..

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 5** – Old: 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 critical 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

But: Less than 20% success rate

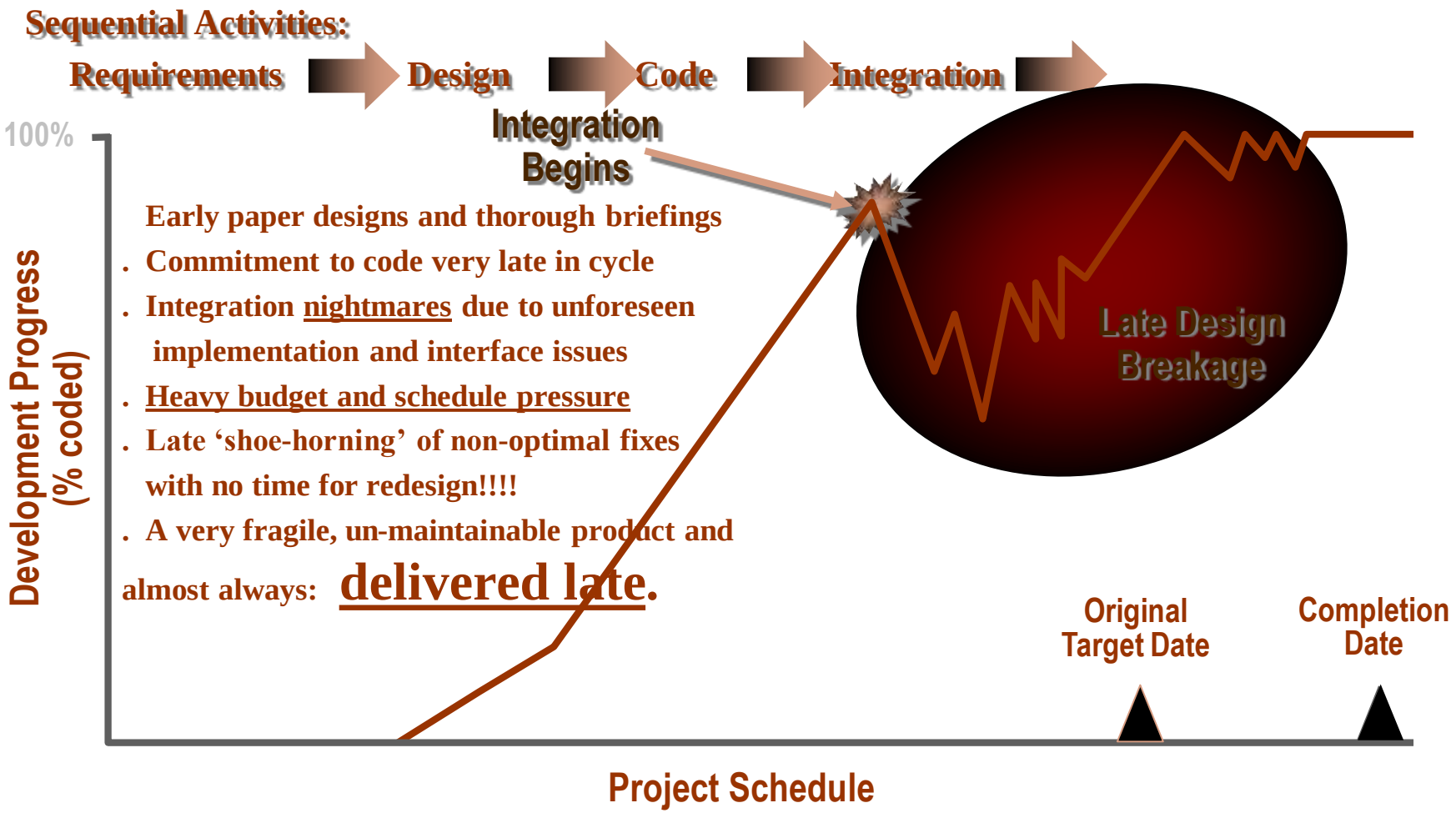
1.1.2 In Practice

- 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



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 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….

Thank You..

Evolution of Software Economics

Dr. P. Dileep Kumar Reddy
Professor-Dean-R&D,IPR, IIC
CSE Department
Narsimha Reddy Engineering College (Autonomous)
Secunderabad, Telangana State, India- 500100.
Ph.No: 09959845657

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!

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 – Parameter (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 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 architectures, processes, iterative processes, etc., all have **initial** high overhead;
  - But follow-on efforts result in economies of scale…and much better ROI. (See p. 25)
  - “All simple systems have been developed!”

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 architectures, processes, iterative processes, etc., all have **initial** high overhead;
  - But follow-on efforts result in economies of scale…and much better ROI. (See p. 25)
  - “All simple systems have been developed!”

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.

Cost Estimation Models

- 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."

Thank You..

IMPROVING SOFTWARE ECONOMICS

Dr. P. Dileep Kumar Reddy
Professor-Dean-R&D,IPR, IIC
CSE Department
Narsimha Reddy Engineering College (Autonomous)
Secunderabad, Telangana State, India- 500100.
Ph.No: 09959845657

The old way and the new

2.0 INTRODUCTION

Five basic parameters of the software cost model are

1. Reducing the size or complexity of what needs to be developed.
2. Improving the development process
3. Using more-skilled personnel and better teams (not necessarily the same thing)
4. Using better environments (tools to automate the process)
5. Trading off or backing off on quality thresholds

These parameters are given in priority order for most software domains. Table 2-1 lists some of the technology developments, process improvement efforts, and management approaches targeted at improving the economics of software development and integration.

Table 2-1: Important trends in improving software economics

COST MODEL PARAMETERS	TRENDS
Size Abstraction and component–based development technologies	Higher order languages (C++, Ada 95, Java, Visual Basic, etc.) Object-oriented (analysis, design, programming) Reuse Commercial components
Process Methods and techniques	Iterative development Process maturity models Architecture-first development Acquisition reform
Personnel People Factors	Training and personnel skill development Teamwork Win-win cultures
Environment Automation technologies and tools	Integrated tools (visual modeling, compiler, editor, debugger, change management, etc.). Open systems Hardware Platform performance Automation of coding, documents, testing, analyses
Quality Performance, reliability, accuracy	Hardware platform performance Demonstration-based assessment Statistical quality control

2.1 REDUCING SOFTWARE PRODUCT SIZE

The most significant way to improve affordability and return on investment (ROI) is usually to produce a product that achieves the design goals with the minimum amount of human-generated source material. *Component-based development* is introduced here as the general term for reducing the "source" language size necessary to achieve a software solution. Reuse, object oriented technology, automatic code production, and higher order programming languages are all focused on achieving a given system with fewer lines of human-specified source directives (statements). This size reduction is the primary motivation behind improvements in higher order languages (such as C++, Ada 95, Java, Visual Basic, and fourth-generation languages), automatic code generators (CASE tools, visual modeling tools, GUI builders), reuse of commercial components (operating systems, windowing environments, database management systems, middleware, networks), and object-oriented technologies (Unified Modeling Language, visual modeling tools, architecture frameworks). The reduction is defined in terms of human-generated source material. In general, when size-reducing technologies are used, they reduce the number of human-generated source lines.

2.1.1 LANGUAGES

- Universal function points (UFPs) are useful estimators for language-independent, early life-cycle estimates. The basic units of function points are external user inputs, external outputs, internal logical data groups, external data interfaces, and external inquiries. SLOC metrics are useful estimators for software after a candidate solution is formulated and an implementation language is known. Substantial data have been documented relating SLOC to function points. Some of these results are shown in Table 2-2.

Table 2-2: Language expressiveness of some of today's popular languages

Language	SLOC Per UFP
Assembly	320
C	128
Fortran 77	105
Cobol 85	91
Ada 83	71
C++	56
Ada 95	55
Java	55
Visual Basic	35

2.1.2 OBJECT-ORIENTED METHODS AND VISUAL MODELING

- There has been at widespread movement in the 1990s toward object-oriented technology. The advantages of object-oriented methods include improvement in software productivity and software quality. The fundamental impact of object-oriented technology is in reducing the overall size of what needs to be developed.

- These are interesting examples of the interrelationships among the dimensions of improving software economics.

1. An object-oriented model of the problem and its solution encourages a common vocabulary between the end users of a system and its developers, thus creating a shared understanding of the problem being solved.
2. The use of continuous integration creates opportunities to recognize risk early and make incremental corrections without destabilizing the entire development effort.
3. An object-oriented architecture provides a clear separation of concerns among disparate elements of a system, creating firewalls that prevent a change in one part of the system from rending the fabric of the entire architecture.

Booch also summarized five characteristics of a successful object-oriented project.

1. A ruthless focus on the development of a system that provides a well understood collection of essential minimal characteristics.
2. The existence of a culture that is centered on results, encourages communication, and yet is not afraid to fail.
3. The effective use of object-oriented modeling
4. The existence of a strong architectural vision
5. The application of a well-managed iterative and incremental development life cycle.

2.1.3 REUSE

- Reusing existing components and building reusable components have been natural software engineering activities since the earliest improvements in programming languages. Software design methods have always dealt implicitly with reuse in order to minimize development costs while achieving all the other required attributes of performance, feature set, and quality. Try to treat reuse as a mundane part of achieving a return on investment.

- Most truly reusable components of value are transitioned to commercial products supported by organizations with the following characteristics:

1. They have an economic motivation for continued support.
2. They take ownership of improving product quality, adding new features, and transitioning to new technologies.
3. They have a sufficiently broad customer base to be profitable.

- The cost of developing a reusable component is not trivial. Figure 3-1 examines the economic tradeoffs. The steep initial curve illustrates the economic obstacle to developing reusable components.
- Reuse is an important discipline that has an impact on the efficiency of all workflows and the quality of most artifacts.

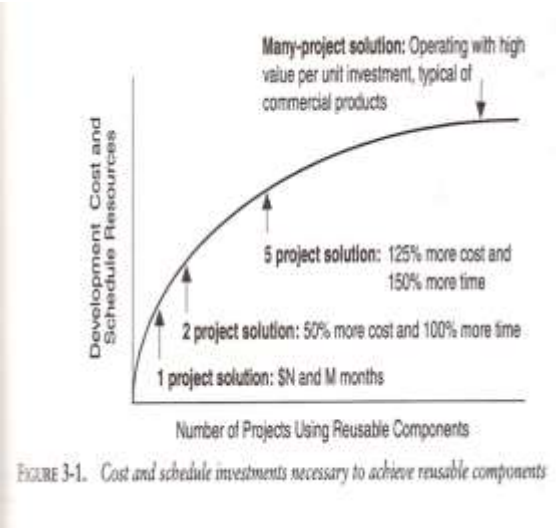


FIGURE 3-1. *Cost and schedule investments necessary to achieve reusable components*

2.1.4. COMMERCIAL COMPONENTS

- A common approach being pursued today in may domains is to maximize integration of commercial components and off-the-shelf products. While the use of commercial components is certainly desirable as a means of reducing custom development, it has not proven to be straight forward in practice. Table 2-3 identifies some of the advantages and disadvantages of using commercial components.

TABLE 2-3: *Advantages and disadvantage of commercial components versus custom software*

APPROACH	ADVANTAGES	DISADVANTAGES
Commercial components	1. Predictable license costs 2. Broadly used, mature technology Available now 3. Dedicated support organization 4. Hardware/ Software independence 5. Rich in functionality	1. Frequent upgrades 2. Up-front license fees 3. Recurring maintenance fees 4. Dependency on vendor 5. Run-time efficiency sacrifices 6. Functionality constraints. 7. Integration not always trivial 8. No control over upgrades and maintenance 9. Unnecessary features that consume extra resources 10. Often Inadequate reliability and Stability. 11. Multiple-vendor incompatibilities
Custom development	1. Complete change freedom 2. Smaller, often simpler implementations 3. Often better performance 4. Control of development and enhancement	1. Expensive, unpredictable development 2. Unpredictable availability date 3. Undefined maintenance model 4. Often immature and fragile 5. Single-platform dependency 6. Drain on expert resources

2.2 IMPROVING SOFTWARE PROCESSES

Process is an overloaded term. For software-oriented organizations, there are many processes and sub processes. Three distinct process perspectives are:

Metaprocess: an organization's policies, procedures, and practices for pursuing a software intensive line of business. The focus of this process is on organizational economics, long-term strategies, and software ROI.

Macroprocess: a project's policies, procedures, and practices for producing a complete software product within certain cost, schedule, and quality constraints. The focus of the macro process is on creating an adequate instance of the Meta process for a specific set of constraints.

Microprocess: a project team's policies, procedures, and practices for achieving an artifact of the software process. The focus of the micro process is on achieving an intermediate product baseline with adequate quality and adequate functionality as economically and rapidly as practical.

Although these three levels of process overlap somewhat, they have different objectives, audiences, metrics, concerns, and time scales as shown in Table 2-4

In a perfect software engineering world with an immaculate problem description, an obvious solution space, a development team of experienced geniuses, adequate resources, and stakeholders with common goals, we could execute a software development process one in iteration with almost no scrap and rework. Because we work in an imperfect world, however, we need to manage engineering activities so that scrap and rework profiles do not have an impact on the win conditions of any stakeholder. This should be the underlying premise for most process improvements.

- Table 2-4: three levels of process and their attributes

ATTRIBUTES	METAPROCESS	MACROPROCESS	MICROPROCESS
Subject	Line of Business	Project	Iteration
Objectives	Line-of-business profitability Competitiveness	Project Profitability Risk management Project Budget, Schedule, quality	Resource Management Risk resolution Milestone budge, schedule, quality
Audience	Acquisition authorities, customers, organizational management	Software project managers Software engineers	Subproject Managers Software Engineers
Metrics	Project predictability Revenue, market share	On budget, on schedule Major milestone success Project scrap and rework	On budget, on schedule major milestone progress/ iteration scrap and rework
Concerns	Bureaucracy Vs. Standardization	Quality Vs Financial Performance	Content Vs schedule
Time Scales	6 to 12 months	1 to many years	1 to 6 months

- Software project managers need many leadership qualities in order to enhance team effectiveness. The following are some crucial attributes of successful software project managers that deserve much more attention:

1. Hiring skills: Few decisions are as important as hiring decisions. Placing the right person in the right job seems obvious but is surprisingly hard to achieve.
2. Customer-interface skill: Avoiding adversarial relationships among stakeholders is a prerequisite for success.
3. Decision-Making skill: The jillion books written about management have failed to provide a clear definition of this attribute. We all know a good leader when we run into one, and decision-making skill seems obvious despite its intangible definition.
4. Team- building skill: Teamwork requires that a manager establish trust, motivate progress, exploit eccentric prima donnas, transition average people into top performers, eliminate misfits, and consolidate diverse opinions into a team direction.
5. Selling skill: Successful project managers must sell all stakeholders (including themselves) on decisions and priorities, sell candidates on job positions, sell changes to the status quo in the face of resistance, and sell achievements against objectives. In practice, selling requires continuous negotiation, compromise, and empathy

2.4 IMPROVING AUTOMATION THROUGH SOFTWARE ENVIRONMENTS

- The tools and environment used in the software process generally have a linear effect on the productivity of the process. Planning tools, requirements management tools, visual modeling tools, compilers, editors, debuggers, quality assurance analysis tools, test tools, and user interfaces provide crucial automation support far evolving the software engineering artifacts. Above all, configuration management environments provide the foundation for executing and instrument the process. At first order, the isolated impact of tools and automation generally allows improvements of 20% to 40% in effort. However, tools and environments must be viewed as the primary delivery vehicle for process automation and improvement, so their impact can be much higher.

- Automation of the design process provides payback in quality. The ability to estimate costs and schedules, and overall productivity using a smaller team. Integrated toolsets play an increasingly important role in incremental/iterative development by allowing the designers to traverse quickly among development artifacts and keep them up-to-date.

Round-trip Engineering is a term used to describe the key capability of environments that support iterative development. As we have moved into maintaining different information repositories for the engineering artifacts, we need automation support to ensure efficient and error-free transition of data from one artifact to another. *Forward Engineering* is the automation of one engineering artifact from another, more abstract representation. For example, compilers and linkers have provided automated transition of source code into executable code. *Reverse engineering* is the generation or modification of a more abstract representation from an existing artifact.

Economic improvements associated with tools and environments. It is common for tool vendors to make relatively accurate individual assessments of life-cycle activities to support claims about the potential economic impact of their tools. For example, it is easy to find statements such as the following from companies in a particular tool

Requirements analysis and evolution activities consume 40% of life-cycle costs.

Software design activities have an impact on more than 50% of the resources.

Coding and unit testing activities consume about 50% of software development effort and schedule.

Test activities can consume as much as 50% of a project's resources.

Configuration control and change management are critical activities that can consume as much as 25% of resources on a large-scale project.

Documentation activities can consume more than 30% of project Engineering resources.

Project management, business administration, and progress assessment can consume as much as 30% of project budgets.

2.5 ACHIEVING REQUIRED QUALITY

Software best practices are derived from the development process and technologies.

Key practices that improve overall software quality include the following:

Focusing on driving requirements and critical use cases early in the life cycle, focusing on requirements completeness and traceability late in the life cycle, and focusing throughout the life cycle on a balance between requirements evolution, design evolution, and plan evolution.

Using metrics and indicators to measure the progress and quality of an architecture as it evolves from a high-level prototype into a fully compliant product.

Providing integrated life-cycle environments that support early and continuous configuration control, change management, rigorous design methods, document automation, and regression test automation.

Using visual modeling and higher level languages that support architectural control, abstraction, reliable programming, reuse, and self-documentation

Early and continuous insight into performance issues through demonstration-based evaluations.

- Conventional development processes stressed early sizing and timing estimates of computer program resource utilization. However, the typical chronology of events in performance assessment was as follows:

  - **Project Inception:** The proposed design was asserted to be low risk with adequate performance origin.

  - **Initial design review:** Optimistic assessments of adequate design margin were based mostly on paper analysis or ought simulation of the critical threads. In most cases, the actual application algorithms and database sizes were fairly well understood.

  - **Mid-life-cycle design review:** The assessments started whittling away at the margin, as early benchmarks and initial tests began exposing the optimism inherent in earlier estimates.

  - **Integration and Test:** Serious performance problems were uncovered, necessitating fundamental changes in the architecture. The underlying infrastructure was usually the scapegoat, but the real culprit was immature use of the infrastructure, immature architectural solutions, or poorly understood early design trade-offs.

2.6 PEER INSPECTIONS: A PRAGMATIC VIEW

Peer inspections are frequently over hyped as the key aspect of a quality system. In my experience, peer reviews are valuable as secondary mechanisms, but they are rarely significant contributors to quality compared with the following primary quality mechanisms and indicators, which should be emphasized in the management process:

- Transitioning Engineering information from one artifact set to another, thereby assessing the consistency, feasibility, understandability, and technology constraints inherent in the engineering artifacts.
- Major milestone demonstrations that force the artifacts to be assessed against tangible criteria in the context of relevant use cases
- Environment tools (compilers, debuggers, analyzers, automated test suites) that ensure representation rigor, consistency, completeness, and change control
- Life-cycle testing for detailed insight into critical trade-offs, acceptance criteria and requirements compliance.
- Change management metrics for objective insight into multiple-perspective change trends and convergence or divergence from quality and progress goals.

Inspections are also a good vehicle for holding authors accountable for quality products. All authors of software and documentation should have their products scrutinized as a natural by product of the process. Therefore, the coverage of inspections should be across all authors rather than across all components.

2.7 THE PRINCIPLES OF CONVENTIONAL SOFTWARE ENGINEERING

1. **Make quality #1:** Quality must be quantified and mechanisms put into place to motivate its achievement.
2. **High-quality software is possible:** Techniques that have been demonstrated to increase quality include involving the customer, prototyping, simplifying design, conducting inspections, and hiring the best people.
3. **Give products to customers early:** No matter how hard you try to learn users needs during the requirements phase, the most effective way to determine real needs is to give users a product and let them play with it.
4. **Determine the problem before writing the requirements:** When faced with what they believe is a problem, most engineers rush to offer a solution. Before you try to solve a problem, be sure to explore all the alternatives and don't be blinded by the obvious solution.
5. **Evaluate Design Alternatives:** After the requirements are agreed upon, you must examine a variety of architectures and algorithms. You certainly do not want to use "architecture" simply because it was used in the requirements specification.
6. **Use an appropriate process model:** Each project must select a process that makes the most sense for that project on the basis of corporate culture, willingness to take risks, application area, volatility of requirements, and the extent to which requirements are well understood.
7. **Use different languages for different phases:** Our industry's eternal thirst for simple solutions to complex problems has driven many to declare that the best development method is one that uses the same notation throughout the life cycle.

8. **Minimize Intellectual Distance:** To minimize intellectual distance, the software's structure should be as close as possible to the real –world structure.
9. **Put techniques before tools:** An undisciplined software engineer with a tool becomes a dangerous, undisciplined software Engineer.
10. **Get it right before you make it faster:** It is far easier to make a working program run faster than it is to make a fast program work. Don't worry about optimization during initial coding.
11. **Inspect Code:** Inspecting the detailed design and code is a much better way to find errors than testing.
12. **Good Management is more important than good technology:** Good management motivates people to do their best, but there are no universal "right" styles of management.
13. **People are the key to success:** Highly skilled people with appropriate experience, talent, and training are key.
14. **Follow with Care:** Just because everybody is doing something does not make it right for you. It may be right, but you must carefully assess its applicability to your environment.
15. **Take responsibility:** When a bridge collapses we ask, "What did the engineers do wrong?" Even when software fails, we rarely ask this. The fact is that in any engineering discipline, the best methods can be used to produce awful designs, and the most antiquated methods to produce elegant designs

16. **Understand the customer's priorities:** It is possible the customer would tolerate 90% of the functionality delivered late if they could have 10% of it on time.
17. **The more thy see, the more they need:** The more functionality (or performance) you provide a user, the more functionality (or performance) the user wants.
18. **Plan to throw one away:** One of the most important critical success factors is whether or not a product is entirely new. Such brand-new applications, architectures, interfaces, or algorithms rarely work the first time.
19. **Design for Change:** The architectures, components and specification techniques you use must accommodate change.
20. **Design without documentation is not design:** I have often heard software engineers say, "I have finished the design. All that is left is the documentation."
21. **Use tools, but be realistic:** Software tools make their users more efficient.
22. **Avoid tricks:** Many programmers love to crate programs with tricks constructs that perform a function correctly, but in an obscure way. Show the world how smart you are by avoiding tricky code.
23. **Encapsulate:** Information-hiding is a simple, proven concept that results in software that is easier to test and much easier to maintain.
24. **Use coupling and cohesion:** Coupling and cohesion are the best ways to measure software's inherent maintainability and adaptability.

25 **Use the McCabe complexity measure:** Although there are many metrics available to report the inherent complexity of software, none is as intuitive and easy to use as Tota McCabe's.

26 **Don't test your own software:** Software developers should never be the primary testers of their own software.

27 **Analyze causes for errors:** It is far more cost-effective to reduce the effect of an error by preventing it than it is to find and fix it. One way to do this is to analyze the causes of errors as they are detected.

28 **Realize that software's entropy increases:** Any software system that undergoes continuous change will grow in complexity and will become more and more disorganized.

29 **People and time are not interchangeable:** Measuring a project solely by person-months makes little sense.

30 **Expect Excellence:** Your employees will do much better if you have high expectations for them.

2.8 THE PRINCIPLES OF MODERN' SOFTWARE MANAGEMENT

- Top 10 principles of modern software management are

1. **Base the process on an architecture-first approach:** This requires that a demonstrable balance be achieved among the driving requirements, the architecturally significant design decisions, and the life-cycle plans before the resources are committed for full-scale development.

2. **Establish an iterative life-cycle process that confronts risk early that confronts risk early:** With today's sophisticated software systems, it is not possible to define the entire problem, design the entire solution, build the software, then test the end product in sequence. Instead, an iterative process that refines the problem understanding, an effective solution, and an effective plan over several iterations encourages a balanced treatment of all stakeholder objectives. Major risks must be addressed early to increase predictability and avoid expensive downstream scrap and rework.

3. **Transition design methods to emphasize component-based development:** Moving from a line-of-code mentality to a component-based mentality is necessary to reduce the amount of human-generated source code and custom development.

4. **Establish a change Management Environment:** the dynamics of iterative development, including concurrent workflows by different teams working on shared artifacts, necessitates objectively controlled baselines.

5. **Enhance change freedom through tools that support round-trip Engineering:** Round trip engineering is the environment support necessary to automate and synchronize engineering information in different formats (such as requirements specifications, design models, source code, executable code, test cases).

6. **Capture design artifacts in rigorous, model-based notation:** A model based approach (such as UML) supports the evolution of semantically rich graphical and textural design notations.

7. **Instrument the process for objective quality control and progress assessment:** Life-cycle assessment of the progress and the quality of all intermediate products must be integrated into the process.

8. **Use a demonstration-based approach:** to assess intermediate artifacts.

9. **Plan intermediate releases in groups of usage scenarios with evolving levels or detail:** It is essential that the software management process drive toward early and continuous demonstrations within the operational context of the system, namely its use cases.

10. **Establish a configurable process that is economically scalable:** No single process suitable for all software developments.

2.9 TRANSITIONING TO AN ITERATIVE PROCESS

- Modern software development processes have moved away form the conventional waterfall model, in which each stage of the development process is dependent on completion of the previous stage.

- The economic benefits inherent in transitioning from the conventional waterfall model to an iterative development process are significant but difficult to quantify. As one benchmark of the expected economic impact of process improvement, consider the process exponent parameters of the COCOMO II mode. This exponent can range from 1.01 (virtually no diseconomy of scale) to 1.26 (significant diseconomy of scale). The parameters that govern the value of the process exponent are application precedentedness, process flexibility, architecture risk resolution, team cohesion and software process maturity.

- The following paragraphs map the process exponent parameters of CO COMO II to my top 10 principles of a modern process.

1. Application Precedentedness: domain experience is a critical factor in understanding how to plan and execute a software development project. For unprecedented systems, one of the key goals is to confront risks and establish early precedents, even if they are incomplete or experimental. This is process. Early iterations in the life cycle establish precedents from which the product, the process and the plans can be elaborated in evolving levels of detail.

2. **Process flexibility:** Development of modern software is characterized by such a broad solution space and so many interrelated concerns that there is a paramount need for continuous incorporation of changes. These changes may be inherent in the problem understanding, the solution space, or the plans. Project artifacts must be supported by efficient change management commensurate with project needs. A configurable process that allows a common framework to be adapted across a range of projects is necessary to achieve a software return on investment.

3. **Architecture Risk Resolution:** Architecture-first development is a crucial theme underlying a successful iterative development process. A project team develops and stabilizes architecture before developing all the components that make up the entire suite of applications components. An Architecture-first and component-based development approach forces tile infrastructure, common mechanisms, and control mechanisms to be elaborated early in the life cycle and drives all component make/buy decisions into the architecture process.

4. **Team Cohesion:** Successful teams are cohesive, and cohesive teams are successful. Successful teams and cohesive teams share common objectives and priorities. Advances in technology (such as programming languages, UML, and visual modeling) have enabled more rigorous and understandable notations for communicating software engineering information, particularly in the requirements and design artifacts that previously were ad hoc and based completely on paper exchange. These model-based formats have also enabled the round-trip engineering support needed to establish change freedom sufficient for evolving design representations.

5. **Software Process Maturity:** The Software Engineering Institute's Capability Maturity Model (CMM) is a well-accepted benchmark for software process assessment. One of key themes is that truly mature processes are enabled through an integrated environment that provides the appropriate level of automation to instrument the process for objection quality control.

MODEL QUESTIONS

1. Why does software not give as must returns on investment as other industries?
2. What are the ways of achieving better economics in software?
3. Explain how the use of good languages and object-oriented modeling can reduce the SLOC in software.
4. What are the issues in obtaining reusable components/ what kind of organizations should be chosen for buying COTS?
5. What are the relative advantages and disadvantage soft custom SW development and development using commercial components?
6. Explain the process of buy/build decision with following example. Given the projected costs and probability (in parenthesis). Shown in figure.
7. Discuss the use of CASE tools in the software process for better cost economics.
8. Compare three levels of process along with its attributes. Also discuss the three dimensions of schedule improvement.
9. How will an activity network help in deciding the skills and number of people require during different phases of the projects?
10. How can good teams be built? How can you continue to have a team that works effectively and efficiently?

11. How should the team be balanced in different dimensions of human skills?
12. What kind of special skills are required for a SW Project Manager? Explain with an example how these are useful or
    What are the skills required from a project manager? Explain with examples.
13. Write short notes on 'Improving automation through software environments'.
14. Distinguish between Round-trip engineering, Forward engineering and Reverse engineering.
15. What are the key practices that improve the overall software quality?
16. What are the different quality improvements a associated with a modern software process? Explain.
17. Discuss the role of peer inspections in improving software quality.
18. List and explain the principles of conventional software engineering.
19. What are the pitfalls of the conventional software engineering principles?
20. The modern software management is based on certain principles. Discuss them briefly.
21. What are the process exponent parameters of COCOMO II Model? Also, discuss how they are mapped to the principles of a modern process.

Thank You..

LIFE CYCLE PHASES AND ARTIFACTS OF THE PROCESS

Dr. P. Dileep Kumar Reddy

Professor-Dean-R&D,IPR, IIC

CSE Department

Narsimha Reddy Engineering College (Autonomous)

Secunderabad, Telangana State, India- 500100.

Ph.No: 09959845657

3. LIFE CYCLE PHASES AND ARTIFACTS OF THE PROCESS

3.0 INTRODUCTION

- Characteristic of a successful software development process is the well-defined separation between "research and development" activities and "production" activities. Most unsuccessful project:; exhibit one of the following characteristics:
  - An overemphasis on research and development
  - An overemphasis on production.
- Successful modern projects-and even successful projects developed under the conventional process-tend to have a very well-defined project milestone when there is a noticeable transition from a research attitude to a production attitude. Earlier phases focus on achieving functionality. Later phases revolve around achieving a product that can be shipped to a customer, with explicit attention to robustness, performance, and finish.
  - A modern software development process must be defined to support the following:
    1. Evolution of the plans, requirements, and architecture, together with well defined synchronization points
    2. Risk management and objective measures of progress and quality
    3. Evolution of system capabilities through demonstrations of increasing functionality

3.1 ENGINEERING AND PRODUCTION STAGES

- To achieve economies of scale and higher returns on investment, we must move toward a software manufacturing process driven by technological improvements in process automation and component based development. Two stages of the life cycle are:
  - The **Engineering stage,** driven by less predictable but smaller teams doing design and synthesis activities.
  - The **Production stage,** driven by more predictable but larger team~ doing construction, test, and deployment activities.

TABLE 5-1: *The two stages of the Life Cycle: Engineering and Production.*

LIFE-CYCLE ASPECT	ENGINEERING STAFF EMPHASIS	PRODUCTION STAGE EMPHASIS
Risk reduction	Schedule, technical feasibility	Cost
Products	Architecture baseline	Product release baselines
Activities	Analysis, design, planning	Implementation, testing
Assessment	Demonstration, inspection, analysis	Testing
Economics	Resolving diseconomies of scale	Exploiting economies of scale
Management	Planning	Operations

3.2 INCEPTION, ELEBORATION, CONSTRUCTION, TRANSITION PHASES

3.2.1 INCEPTION PHASE

- The overriding goal of the inception phase is to achieve concurrence among stakeholders on the lifecycle objectives for the project.

❑ Primary Objectives

- Establishing the project's software scope and boundary condition, including all operational concept, acceptance criteria, and a clear understanding of what is and is not intended to be in the product.
- Discriminating the critical use cases of the system and the primary scenarios of operation that will drive the major design trade-offs.
- Demonstrating at least one candidate architecture against some of the primary scenarios.
- Estimating the cost and schedule for the entire project (including detailed estimates for the elaboration phase).
- Estimating potential risks (sources of un predictability)

❑ Essential Activities

Formulating the scope of the project. The information repository should be sufficient to define the problem space and derive the acceptance criteria for the end product.

Synthesizing the architecture: An information repository is created that is sufficient to demonstrate the feasibility of at least one candidate architecture and an, initial baseline of make/buy decisions so that the cost, schedule, and resource estimates can be derived.

Planning and preparing a business case. Alternatives for risk management, staffing, iteration plans, and cost/schedule/profitability trade-offs are evaluated.

❑ Primary Evaluation Criteria

Do all stakeholders concur on the scope definition and cost and schedule estimates?

Are requirements understood, as evidenced by the fidelity of the critical use cases?

Are the cost and schedule estimates, priorities, risks, and development processes credible?

Do the depth and breadth of an architecture prototype demonstrate the preceding criteria? (The primary value of prototyping candidate architecture is to provide a vehicle for understanding the scope and assessing the credibility of the development group in solving the particular technical problem.)

Are actual resource expenditures versus planned expenditures acceptable

3.2.2 ELABORATION PHASE

- At the end of this phase, the “Engineering” is considered complete. The elaboration phase activities must ensure that the architecture, requirements, and plans are stable enough, and the risks sufficiently mitigated, that the cost and schedule for the completion of the development call be predicted within an acceptable range. During the elaboration phase, an executable architecture prototype is built in one or more iterations, depending on the scope, size and risk.
- **Primary Objectives**
- Base lining the architecture as rapidly as practical (establishing a configuration-managed snapshot in which all changes are rationalized, tracked, and maintained)
- Base lining the vision
- Base lining a high-fidelity plan for the construction phase
- Demonstrating that the baseline architecture will support the vision at a reasonable cost in a reasonable time
- **Essential Activities**
- Elaborating the vision.
- Elaborating the process and infrastructure.
- Elaborating the architecture and selecting components.
- **Primary Evaluation Criteria**
- Is the vision stable?
- Is the architecture stable?
- Does the executable demonstration show that the major risk elements have been addressed and credibly resolved?
- Is the construction phase plan of sufficient fidelity, and is it backed up with a credible basis of estimate?
- Do all stakeholders agree that the current vision can be met if the current plan is executed to develop the complete system in the context of the current architecture?
- Are actual resource expenditures versus planned expenditures acceptable?

3.2.3 CONSTRUCTION PHASE

- During the construction phase, all remaining components and application features are integrated into the application, and all features are thoroughly tested. Newly developed software is integrated where required. The construction phase represents a production process, in which emphasis is placed on managing resources and controlling operations to optimize costs, schedules and quality.
- **Primary Objectives**
- Minimizing development costs by optimizing resources and avoiding unnecessary scrap and rework
- Achieving adequate quality as rapidly as practical
- Achieving useful versions (alpha, beta and other test releases) as rapidly as practical
- **Essential Activities**
- Resource management, control and process optimization
- Complete component development and testing against evaluation criteria.
- Assessment of product releases against acceptance criteria of the vision.
- **Primary Evaluation Criteria**
- Is this product baseline mature enough to be deployed in the user community? (Existing defects are not obstacles to achieving the purpose of eth next release).
- Is this product baseline stable enough to be deployed in the user community? (Pending changes are not obstacles to achieving the purpose of the next release.)
- Are the stakeholders ready for transition to the user community?
- Are actual resource expenditures versus planned expenditures acceptable?

3.2.4 TRANSITION PHASE

- The transition phase is entered when a baseline is mature enough to be deployed in the end-user domain. This typically requires that a usable subset of the system has been achieved with acceptable quality levels and user documentation so that transition to the user will provide positive results. This phase could include any of the following activities:
- Beta testing to validate the new system against user expectations
- Beta testing and parallel operation relative to a legacy system it is replacing
- conversion of operational databases
- Training of user and maintainers
  - The transition phase concludes when the deployment baseline has achieved the complete vision.
- **Primary Objectives**
  - Achieving user self-supportability
  - Achieving stakeholder concurrence that deployment baselines are complete and consistent with the evaluation criteria of the vision
  - Achieving final produce baselines as rapidly and cost-effectively as practical.
- **Essential Activities**
  - Synchronization and integration of concurrent construction increments into consistent deployment baselines
  - Deployment-specific engineering (cutover, commercial packaging and production, sales rollout kit development, field personnel training).
  - Assessment of deployment baselines against the complete vision and acceptance criteria in the requirements set.
- **Primary Evaluation Criteria**
  - Is the user satisfied
  - Are actual resource expenditures versus planned expenditures acceptable

3.3 THE ARTIFACT SETS

- To make the development of a complete software system manageable, distinct collections of information are organized into artifact sets. Artifact represents cohesive information that typically is developed and reviewed as a single entity.
- Life-cycle software artifacts are organized into five distinct sets that are roughly partitioned by the underlying language of the set: management (ad hoc textual formats), requirements (organized text and models of the problem space, (design models of the solution space), implementation (human-readable programming, language and associated source files), and deployment (machine-process able languages and associate files). The artifact sets are shown in Figure 6-1.

Requirement Set	Design Set	Implementation Set	Deployment Set
1. Vision document 2. Requirements Model(s)	1. Design Model(s) 2. Test Model 3. Software architecture description	1. Source code baselines 2. Associated compile-time files 3. Component executables.	1. Integrated product executable baselines 2. Associated run-time files 3. User Manual
Management Set			
Planning Artifacts		**Operational Artifacts**	
1. Work breakdown structure 2. Business case 3. Release specifications 4. Software development plan		5. Release descriptions 6. Status assessment 7. Software change order database 8. Deployment documents 9. Environment	

3.3.1 THE MANAGEMENT SET

- The management set captures the artifacts associated with process planning and execution. These artifacts use ad hoc notations, including text, graphics, or whatever representation is required to capture the "contracts" among project personnel (project management, architects, developers, testers, marketers, administrators), among stakeholders (funding authority, user, software project manager, organization manager, regulatory agency), and between project personnel and stakeholders. Specific artifacts included in this set are the work breakdown structure (activity breakdown and financial tracking mechanism), the business case (cost, schedule, profit expectations), the release specifications (scope, plan, objectives for release baselines), the software development plan (project process instance), the release descriptions (results of release baselines), the status assessments (periodic snapshots of project progress), the software change orders (descriptions of discrete baseline changes), the deployment documents (cutover plan, training course, sales rollout kit), and the environment (hardware and software tools, process automation & documentation).
- Management set artifacts are evaluated, assessed, and measured through a combination of the following:
  - Relevant stakeholder review.
  - Analysis of changes between the current version of the artifact and previous versions.
  - Major milestone demonstrations of the balance among all artifacts and, in particular, the accuracy of the business case and vision artifacts.

3.3.2 THE ENGINEERING SETS

- The Engineering sets consist of the requirements set, the design set, the implementation set, and the deployment set.

❑ Requirements Set

- Requirements artifacts are evaluated, assessed, and measured through a combination of the following:
  - Analysis of consistency with the release specifications of the management set.
  - Analysis of consistency between the vision and the requirements models.
  - Mapping against the design, implementation, and deployment sets to evaluate the consistency and completeness and the semantic balance between information in the different sets.
  - Analysis of changes between the current version of requirements artifacts and previous versions (scrap, rework, and defect elimination trends).
  - Subjective review of other dimensions of quality.

Design Set

- UML notation is used to engineer the design models for the solution. The design set contains varying levels of abstraction that represent the components of the solution space (their identities, attributes, static relationships, dynamic interactions). The design set is evaluated, assessed and measured through a combination of the following:

  - Analysis of the internal consistency and quality of the design model
  - Analysis of consistency with the requirements models
  - Translation into implementation and deployment sets and notations (for example, traceability, source code generation, compilation, linking) to evaluate the consistency and completeness and the semantic balance between information in the sets.
  - Analysis of changes between the current version of the design model and previous versions (scrap, rework, and defect elimination trends).
  - Subjective review of other dimensions of quality.

❑Implementation Set

- The implementation set includes source code (programming language notations) that represents the tangible implementations of components (their form, interface, and dependency relationships).

- Implementation sets are human-readable formats that are evaluated, assessed, and measured through a combination of the following:

  - Analysis of consistency with the design models.
  - Translation into deployment set notations (for example, compilation and linking) to evaluate the consistency and completeness among artifact sets.
  - Assessment of component source or executable files against relevant evaluation criteria through inspection, analysis, demonstration, or testing
  - Execution of stand-alone component test cases that automatically compare expected results with actual results.
  - Analysis of changes between the current version of the implementation set and previous versions (scrap, rework, and defect elimination trends).
  - Subjective review of other dimensions of quality.

❑Deployment Set

- The deployment set includes user deliverables and machine language notations, executable software, and the build scripts, installation scripts, and executable target specific data necessary to use the product in its target environment.
- Deployment sets are evaluated, assessed, and measured through a combination of the following:
  - Testing against the usage scenarios and quality attributes defined in the requirements set to evaluate the consistency and completeness and the semantic balance between information in the two sets.
  - Testing the partitioning, replication, and allocation strategies in mapping components of the implementation set to physical resources of the deployment system (platform type, number, network topology).
  - Testing against the defined usage scenarios in the user manual such as installation, user - oriented dynamic reconfiguration, mainstream usage, and anomaly management
  - Analysis of changes between the current version of the deployment set and previous versions (defect elimination trends, performance changes).
  - Subjective review of other dimensions of quality.

- Each artifact set is the predominant development focus of one phase of the life cycle; the other sets take on check and balance roles. As illustrated in Figure 6-2, each phase has a predominant focus:

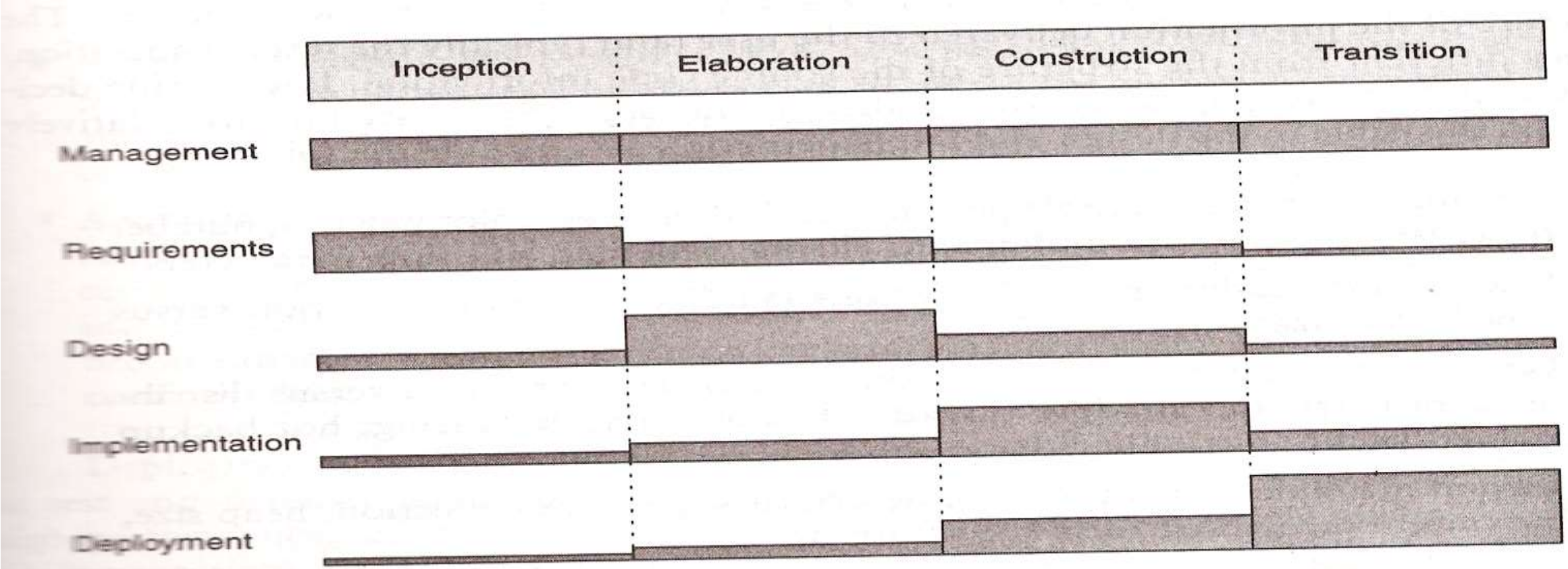


FIGURE 6-2. *Life-cycle focus on artifact sets*

- Requirements are the focus of the inception phase; design, the elaboration phase; implementation, the construction phase; and deployment, the transition phase. The management artifacts also evolve, but at a fairly constant level across the life cycle.

- Most of today's software development tools map closely to one of the five artifact sets.

1. **Management:** scheduling, workflow, defect tracking, change management, documentation, spreadsheet resource management, and presentation tools.
2. **Requirements:** requirements management tools.
3. **Design:** visual modeling tools.
4. **Implementation:** compiler/debugger tools, code analysis tools, test coverage analysis tools, and test management tools.
5. **Deployment:** test coverage and test automation tools, network management tools, commercial components (Operating Systems, GUIs, RDBMS, networks, middleware), and installation tools.

❑Implementation Set versus Deployment Set

- The separation of the implementation set (source code) from the deployment set (executable code) is important because there are very different concerns with each set. The structure of the information delivered to the user (and typically the test organization) is very different from the structure of the source code information. Engineering decisions that have an impact on the quality of the deployment set but are relatively incomprehensible in the design and implementation sets include the following:
  - Dynamically reconfigurable parameters (buffer sizes, color palettes, number of servers, number of simultaneous clients. data files, run-time parameters)
  - Effects or compiler/link optimizations (such as space optimization versus speed optimization)
  - Performance under certain allocation strategies (centralized versus distributed, primary and shadow threads, dynamic load balancing, hot backup versus checkpoint/rollback)
  - Virtual machine constraints (file descriptors, garbage collection, heap size, maximum record size, disk file rotations)
  - Process-level concurrency issues (deadlock and race conditions)
  - Platform-specific differences in performance or behavior.

3.3.3 ARTIFACT EVOLUTION OVER THE LIFE CYCLE

- Each state of development represents a certain amount of precision in the final system description. Early in the life cycle, precision is low and the representation is generally high. Eventually, the precision of representation is high and everything is specified in full detail. Each phase of development focuses on a particular artifact set. At the end of each phase, the overall system state will have progressed on all sets, as illustrated in Figure 6-3.

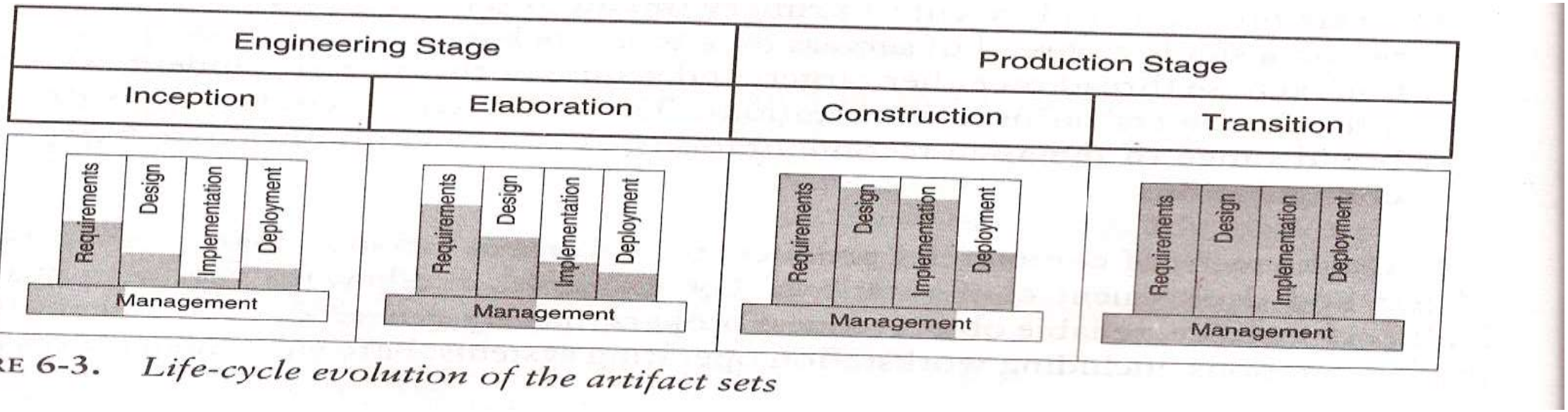


FIGURE 6-3. *Life-cycle evolution of the artifact sets*

- The inception phase focuses mainly on critical requirements usually with a secondary focus on an initial deployment view. During the elaboration phase, there is much greater depth in requirements, much more breadth in the design set, and further work on implementation and deployment issues. The main focus of the construction phase is design and implementation. The main focus of the transition phase is on achieving consistency and completeness of the deployment set in the context of the other sets.

3.3.4 TEST ARTIFACTS

- The test artifacts must be developed concurrently with the product from inception through deployment. Thus, testing is a full-life-cycle activity, not a late life-cycle activity.
- The test artifacts are communicated, engineered, and developed within the same artifact sets as the developed product.
- The test artifacts are implemented in programmable and repeatable formats (as software programs).
- The test artifacts are documented in the same way that the product is documented.
- Developers of the test artifacts use the same tools, techniques, and training as the software engineers developing the product.
- Test artifact subsets are highly project-specific, the following example clarifies the relationship between test artifacts and the other artifact sets. Consider a project to perform seismic data processing for the purpose of oil exploration. This system has three fundamental subsystems: (1) a sensor-subsystem that captures raw seismic data in real time and delivers these data to (2) a technical operations subsystem that converts raw data into an organized database and manages queries to this database from (3) a display subsystem that allows workstation operators to examine seismic data in human-readable form. Such a system would result in the following test artifacts:

- **Management Set:** The release specifications and release descriptions capture the objectives, evaluation criteria, and results of an intermediate milestone. These artifacts are the test plans and test results negotiated among internal project teams. The software change orders capture test results (defects, testability changes, requirements ambiguities, enhancements) and the closure criteria associated with making a discrete change to a baseline.
- **Requirements Set:** The system-level use cases capture the operational concept for the system and the acceptance test case descriptions, including the expected behavior of the system and its quality attributes. The entire requirement set is a test artifact because it is the basis of all assessment activities across the life cycle.
- **Design Set:** A test model for non deliverable components needed to test the product baselines is captured in the design set. These components include such design set artifacts as a seismic event simulation for creating realistic sensor data; a “virtual operator” that can support unattended, after-hours test cases; specific instrumentation suites for early demonstration of resource usage; transaction rates or response times; and use case test drivers and component stand-alone test drivers.
- **Implementation Set:** Self-documenting source code representations for test components and test drivers provide the equivalent of test procedures and test scripts. These source files may also include human-readable data files representing certain statically defined data sets that are explicit test source files. Output files from test drivers provide the equivalent of test reports.
- **Deployment Set:** Executable versions of test components, test drivers, and data files are provided.

3.4 MANAGEMENT ARTIFACTS

- The management set includes several artifacts that capture intermediate results and ancillary information necessary to document the product/process legacy, maintain the product, improve the product and improve the process.

❑ Business Case

- The business case artifact provides all the information necessary to determine whether the project is worth investing in. It details the expected revenue, expected cost, technical and management plans, and backup data necessary to demonstrate the risks and realism of the plans. The main purpose is to transform the vision into economic terms so that an organization can make an accurate ROI assessment. The financial forecasts are evolutionary, updated with more accurate forecasts as the life cycle progress. Figure 6-4 provides a default outline for a business case.

I. Context (domain, market, scope)
II. Technical approach
- A. Feature set achievement plan
- B. Quality achievement plan
- C. Engineering trade-offs and technical risks

III Management approach
- A. Schedule and schedule risk assessment
- B. Objective measures of success

IV Evolutionary appendixes
- A. Financial forecast
  - 1 Cost estimate
  - 2. Revenue estimate
  - 3. Bases of estimates

FIGURE 6-4: ***Typical business case outline***

Software Development Plan

- The software development plan (SDP) elaborates the process framework into a fully detailed plan. Two indications of a useful SDP are periodic updating (it is not stagnant shelf ware) and understanding and acceptance by managers and practitioners alike. Figure 6-5 provides a default outline for a software development plan.

I. Context (scope, objectives)
II. Software development process
A. Project Primitives
   1. Life-cycle phases
   2. Artifacts
   3. Workflows
   4. Checkpoints
B. Major milestone scope and content
C. Process improvement procedures
III. Software Engineering Environment
A. Process automation (hardware and software resource configuration)
B. Resource allocation procedures (sharing across organizations, security access).
IV. Software change management
   A. Configuration control board Plan and Procedures
   B. Software change order definitions and procedures.
   C. Configuration baseline definitions and procedures
V. Software Assessment
A. Metrics collection and reporting procedures
B. Risk Management procedures (risk identification, tracking and resolution
C. Status assessment plan
D. Acceptance test plan.
VI. Standards and Procedures
A. Standards and procedures for technical artifacts
VII. Evolutionary Appendixes
A. Minor milestone scope and content
B. Human resources (organization, staffing plan, training plan).

FIGURE 6-5: ***Typical software development plan outline.***

Software Development Plan

- The software development plan (SDP) elaborates the process framework into a fully detailed plan. Two indications of a useful SDP are periodic updating (it is not stagnant shelf ware) and understanding and acceptance by managers and practitioners alike. Figure 6-5 provides a default outline for a software development plan.

I. Context (scope, objectives)
II. Software development process
A. Project Primitives
   1. Life-cycle phases
   2. Artifacts
   3. Workflows
   4. Checkpoints
B. Major milestone scope and content
C. Process improvement procedures
III. Software Engineering Environment
A. Process automation (hardware and software resource configuration)
B. Resource allocation procedures (sharing across organizations, security access).
IV. Software change management
   A. Configuration control board Plan and Procedures
   B. Software change order definitions and procedures.
   C. Configuration baseline definitions and procedures
V. Software Assessment
A. Metrics collection and reporting procedures
B. Risk Management procedures (risk identification, tracking and resolution
C. Status assessment plan
D. Acceptance test plan.
VI. Standards and Procedures
A. Standards and procedures for technical artifacts
VII. Evolutionary Appendixes
A. Minor milestone scope and content
B. Human resources (organization, staffing plan, training plan).

FIGURE 6-5: ***Typical software development plan outline.***

❑ Work Breakdown Structure

- Work breakdown structure (WBS) is the vehicle for budgeting and collecting costs. To monitor and control a project's financial performance, the software project manager must have insight into project costs and how they are expended. The structure of cost accountability is a serious project planning constraint.

❑ Software Change Order Database

- Managing change is one of the fundamental primitives of an iterative development process. With greater change freedom, a project can iterate more productively. This flexibility increases the content, quality and number of iterations that a project can achieve within a given schedule. Change freedom has been achieved in practice through automation, and today's iterative development environments carry the burden of change management. Organizational processes that depend on manual change management techniques have encountered major inefficiencies.

Release Specifications

- The scope, plan, and objective evaluation criteria for each baseline release are derived from the vision statement as well as many other sources (make/buy analyses, risk management concerns, architectural considerations, shots in the dark, implementation constraints, quality thresholds). These artifacts are intended to evolve along with the process, achieving greater fidelity as the life cycle progresses and requirements understanding matures. Figure 6-6 provides a default outline for a release specification

I. Iteration Content

II. Measurable objectives

A. Evaluation criteria

B. Follow through approach

III. Demonstration Plan

A. Schedule of activities

B. Team responsibilities

IV. Operational scenarios (use cases demonstrated)

A. Demonstration Procedures

B. Traceability to vision and business case.

FIGURE 6-6: ***Typical release specification outline.***

Release descriptions

- Release description documents describe the results of each release, including performance against each of the evaluation criteria in the corresponding release specification. Release baselines should be accompanied by a release description document that describes the evaluation criteria for that configuration baseline and provides substantiation (through demonstration, testing, inspection, or analysis) that each criterion has been addressed in an acceptable manner. Figure 6-7 provides a default outline for a release description.

I. Context
A. Release baseline content
B. Release metrics
II. Release notes
A. Release-specific constraints or limitations.
III. Assessment Results
A. Substantiation of passed evaluation criteria
B. Follow-up plans for failed evaluation criteria
C. Recommendations for next release
IV. Outstanding issues
A. Action Items
B. Post-mortem summary of lessons learned.

FIGURE 6-7: ***Typical release description outline***

❑ Status Assessments

- Status assessments provide periodic snapshots of project health and status, including the software project manager's risk assessment, quality indicators, and management indicators. Typical status assessments should include a review of resources, personnel staffing, financial data (cost and revenue), top 10 risks, technical progress (metrics snapshots), major milestone plans and results, total project or product scope & action items.

❑ Environment

- An important emphasis of a modern approach is to define the development and maintenance environment as a first-class artifact of the process. A robust, integrated development environment must support automation of the development process. This environment should include requirements management, visual modeling, document automation, host and target programming tools, automated regression testing, and continuous and integrated change management, and feature and defect tracking.

❑ Deployment

- A deployment document can take many forms. Depending on the project, it could include several document subsets for transitioning the product into operational status. In big contractual efforts in which the system operations manuals, software installation manuals, plans and procedures for cutover (from a legacy system), site surveys, and so forth. For commercial software products, deployment artifacts may include marketing plans, sales rollout kits, and training courses.

❑ Management Artifact Sequences

- In each phase of the life cycle, new artifacts are produced and previously developed m1ifacts are updated to incorporate lessons learned and to capture further depth and breadth of the solution. Figure 6-8 identifies a typical sequence of artifacts across the life-cycle phases.

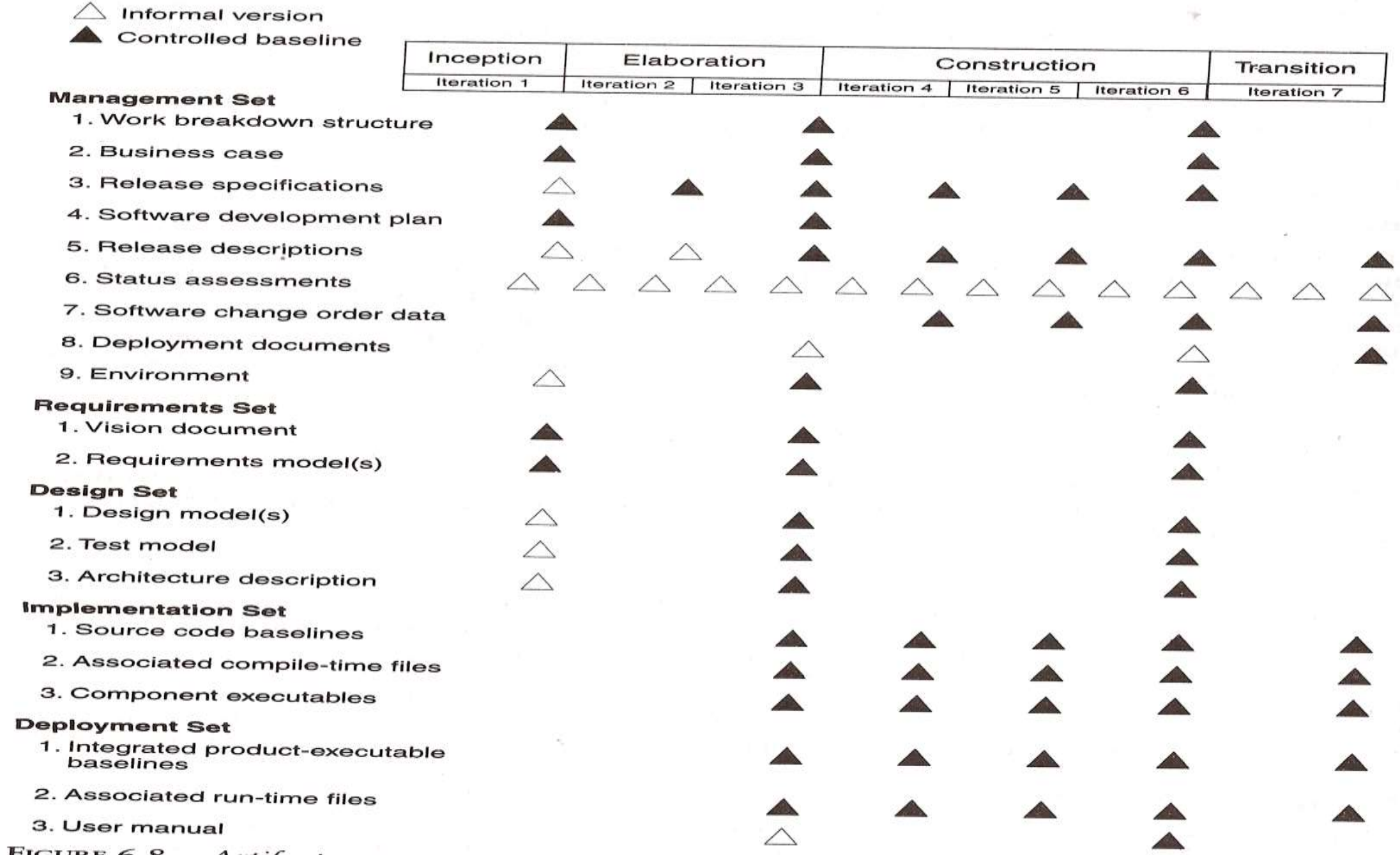


FIGURE 6-8. *Artifact sequences across a typical life cycle*

3.5 ENGINEERING ARTIFACTS

- Most of the engineering artifacts are captured in rigorous engineering notations such as UML, programming languages, or executable machine codes. Three engineering artifacts are explicitly intended for more general review, and they deserve further elaboration.

❑Vision document

- The vision document provides a complete vision for the software system under development and supports the contract between the funding authority and eth development organization. A project vision is meant to be changeable as understanding evolves of the requirements, architecture, plans and technology. A good visions document should change slowly. Figure 6-9 provides a default outline for a visions document

I. Feature set description
- A. Precedence and priority

II. Quality attributes and ranges

III. Required constraints
- A. External interfaces

IV. Evolutionary Appendixes
- A. Use cases
  1. Primary Scenarios
  2. Acceptance criteria and tolerances
- B. Desired freedoms (potential change scenarios).

FIGURE 6-9: Typical vision document outline

❑Architecture Description:

- The Architecture description provides an organized view of the software architecture under development. It is extracted largely from the design model and includes views of the design, implementation and deployment sets sufficient to understand how the operational concept of the requirements et will be achieved. The breadth of the architecture description will vary from project to project depending on many factors. Figure 6-10 provides a default outline fro an architecture description.

I. Architecture overview
- A. Objectives
- B. Constraints
- C. Freedoms

II. Architecture views
- A. Design view
- B. Process view
- C. Component view
- D. Deployment view

III. Architectural interactions
- A. Operational concept under primary scenarios
- B. Operational concept under secondary scenarios
- C. Operational concept under anomalous conditions

IV. Architecture performance

V. Rationale, trade-offs and other substantiation.

FIGURE 6-10: ***Typical architecture description outline***

❑Software Use Manual

- The software user manual provides the user with the reference documentation necessary to support delivered software. Although content is highly variable across application domains, the user manual should include installation procedures, usage procedures and guidance, operational constraints, and a user interface description at a minimum. For software products with a user interfaced, this manual should be developed early in the life cycle because it is a necessary mechanism for communication and stabilizing an important subset of requirements. The user manual should be written by members of the test team, who are more likely to understand the user's perspective than the development team.

3.6 PRAGMATIC ARTIFACTS

- **People want to review information but don't understand the language of the artifact.** Many interested reviewers of a particular artifact will resist having to learn the engineering language in which the artifact is written. It is not uncommon to find people (such as veteran software managers, veteran quality assurance specialists, or an auditing authority from a regulatory agency) who react as follows: "I'm not going to learn UML, but I want to review the design of this software, so give me a separate description such as some flowcharts and text that I can understand."
- **People want to review the information but don't have access to the tools.** It is not very common for the development organization to be fully tooled; it is extremely rare that the/other stakeholders have any capability to review the engineering artifacts on-line. Consequently, organization is forced to exchange paper documents. Standardized formats (such as UML, spreadsheets, Visual Basic, C++ and Ada 95), visualization tools, and the web are rapidly making it economically feasible for all stakeholders to exchange information electronically.
- **Human-readable engineering artifacts should use rigorous notations that are complete, consistent, and used in a self-documenting manner.** Properly spelled English words should be used for all identifiers and descriptions. Acronyms and abbreviations should be used only where they are well accepted jargon in the context of the component's usage. Readability should be emphasized and the use of proper English words should be required in all engineering artifacts. This practice enables understandable representations, browse able formats (paperless review), more-rigorous notations, and reduced error rates.
- **Useful documentation is self-defining:** It is documentation that gets used.
- **Paper is tangible; electronic artifacts are too easy to change.** On-line and Web-based artifacts can be changed easily and are viewed with more skepticism because of their inherent volatility.

MODEL QUESTIONS

1. Justify the dividing of the four phases of software life-cycle into engineering and production stages.
2. What are the primary objectives of the four phases of software life-cycle?
3. What are the essential activities in inception and elaboration phases?
4. What are the essential activities in construction and transition phases?
5. How do you evaluate the completion of each of the four phases in software life cycle?
6. Give an overview of the artifact sets. Also, explain the artifacts in management set.
7. Discuss briefly about the various sets that are included in the engineering set.
8. Write notes on the following.
    a. Artifact evolution over the software life-cycle.
    b. Test Artifacts.
9. Distinguish between implementation asset and deployment set.
10. Discuss the planning artifacts of a management set.
11. How an operational artifact of a management set differs from planning artifacts? Explain.
12. Draw and explain the artifact sequences across the software life-cycle.
13. Discuss in detail about the various engineering artifacts in software project management.
14. Explain pragmatic artifacts of software project management.

Thank You..

MODEL BASED SOFTWARE ARCHITECTURE

Dr. P. Dileep Kumar Reddy

Professor-Dean-R&D,IPR, IIC

CSE Department

Narsimha Reddy Engineering College (Autonomous)

Secunderabad, Telangana State, India- 500100.

Ph.No: 09959845657

ARCHITECTURE: A MANAGEMENT PERSPECTIVE

- The most critical technical product of a software project is its architecture: the infrastructure, control and date interfaces that permit software components to co-operate as a system and software designers to co-operate efficiently as a tem. When the communications media include multiple languages and inter group literacy varies, the communications problem can become extremely complex and even unsolvable. If a software development team is to be successful, the inter project communications, as captured in the software architecture, must be both accurate and precise.
- From a management perspective, there are three difference aspects of architecture.
  - *An architecture* (the intangible design concept) is the design of a software system this includes all engineering necessary to specify a complete bill of materials.
  - *An architecture baseline* (the tangible artifacts) is a slice of information across the engineering artifact sets sufficient to satisfy all stakeholders that the vision (function and quality) can be achieved within the parameters of the business case (cost, profit, time, technology and people).

- *An architecture description* (a human-readable representation of an architecture, which is one of the components of an architecture baseline) is an organized subset of information extracted form the design set model(s). The architecture description communicates how the intangible concept is realized in the tangible artifacts.

- The number of views and the level of detail in each view can vary widely.
- The importance of software architecture and its close linkage with modern software development processes can be summarized as follows:
  - **Achieving stable software architecture represents a significant project milestone at which the critical make/buy decisions should have been resolved.**
  - **Architecture representations provide a basis for balancing the trade-offs between the problem space (requirements and constraints) and the solution space (the operational product).**
  - **The architecture and process encapsulate many of the important (high-payoff or high-risk) communications among individuals, teams, organizations and stakeholders.**
  - **Poor architectures and immature processes are often given as reasons for project failures.**
  - **A mature process, an understanding of the primary requirements, and a demonstrable architecture are important prerequisites fro predictable planning.**
  - **Architecture development and process definition are the intellectual steps that map the problem to a solution without violating the constraints; they require human innovation and cannot be automated.**

ARCHITECTURE: A TECHNICAL PERSPECTIVE

- An architecture framework is defined in the terms of views that are abstractions of the UML models in the design set. The design model includes the full breadth and depth of information. An architecture view is an abstraction of the design model; it contains only the architecturally significant information. Most real-world systems require four views: design, process, component and deployment. The purposes of these views are as follows:
  - Design: Describes architecturally significant structures and functions of the design model.
  - Process: Describes concurrency and control thread relationship among the design, component and deployment views.
  - Component: Describes the structure of the implementation set.
  - Deployment: Describes the structures of the deployment set.
- Figure 7-1: Summarizes the artifacts of the design set, including the architecture views and architecture description.
- The requirements model addresses the behavior of the system as seen by its end users, analysts, and testers. This view is modeled statically using use case and class diagrams and dynamically using sequence, collaboration, state chart and activity diagrams.

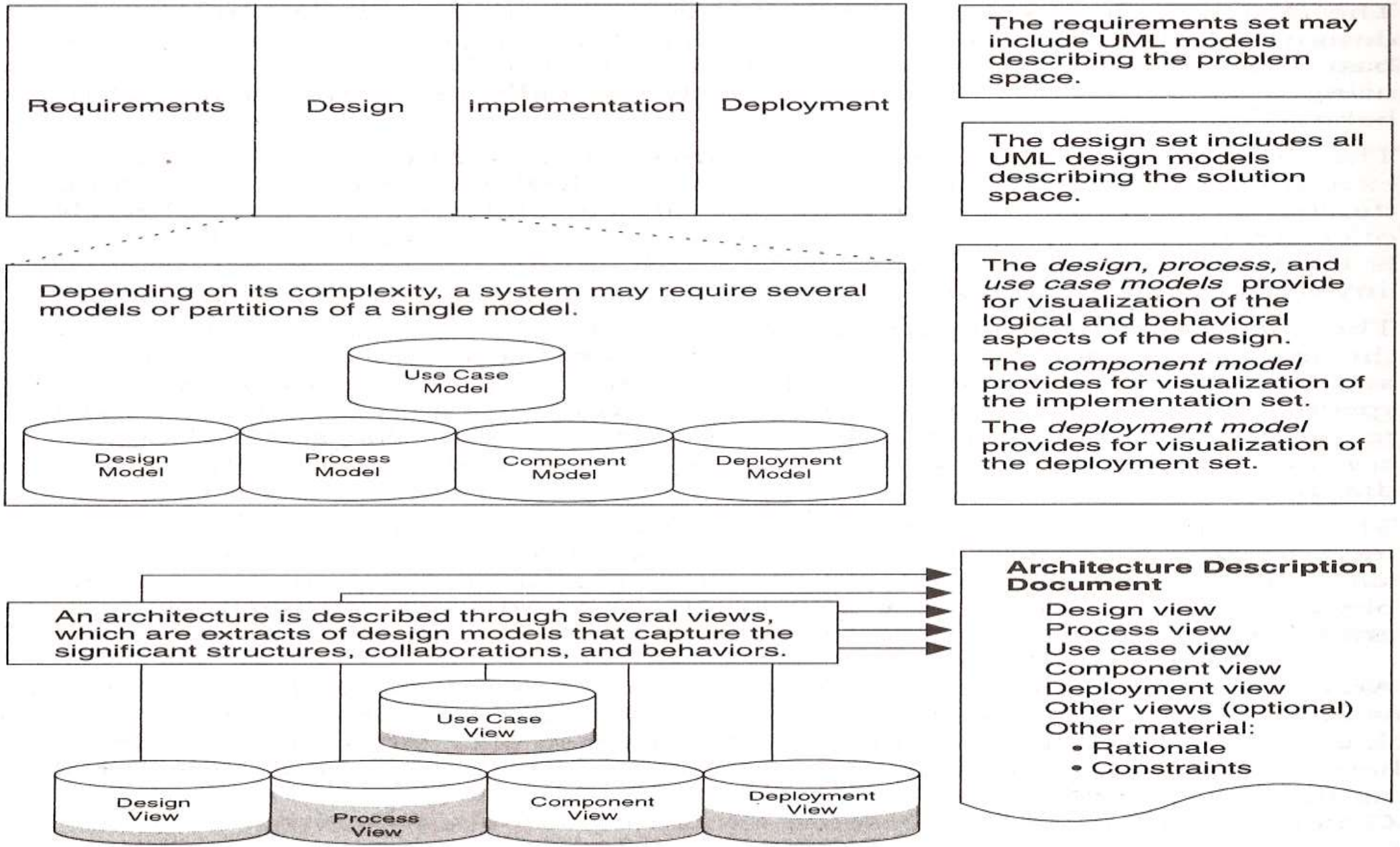


FIGURE 7-1. *Architecture, an organized and abstracted view into the design models*

- The use case view describes how the system's critical (architecturally significant) use cases are realized by elements of the design model. It is modeled statically using use case diagrams and dynamically using any of the UML behavioral diagrams.
- The design view describes the architecturally significant elements of the design model. This view, an abstraction of the design model, addresses the basic structure and functionality of the solution. It is modeled statically using calls and object diagrams and dynamically using any of the UML behavioral diagrams.
- The process view addresses the run-time collaboration issues involved in executing the architecture on a distributed deployment model, including the logical software network topology (allocation to process and threads of control), inter process communication and state management. This view is modeled statically using deployment diagrams and dynamically using any of the UML behavioral diagrams.
- The component view describes the architecturally significant elements of the implementation set. This view, an abstraction of the design model, addresses the software source code realization of the system from the perspective of the project's integrators and developers, especially with regard to releases and configuration management. It is modeled statically using component diagrams and dynamically using any of the UML behavioral diagrams.
- The deployment view addresses the executable realization of the system, including the allocation of logical processes in the distribution view (the logical software topology) to physical resources of the deployment network (the physical system topology). It is modeled statically using deployment diagrams and dynamically using any of the UML behavioral diagrams.

- Generally, an architecture baseline should including the following:

  - **Requirements:** critical use cases system-level quality objectives and priority relationships among features and qualities
  - **Design:** names, attributes, structures, behaviors, groupings and relationships of significant classes and components
  - **Implementation:** source component inventory and bill of materials (number, name, purpose, cost) of all primitive components
  - **Development:** executable components sufficient to demonstrate the critical us cases and the risk associated with achieving the system qualities.

MODEL QUESTIONS

1. Define the terms ‘model’ and ‘view’. What are the three different aspects of software architecture form management’s perspective?
2. Explain the significance of software architecture in modern software development process.
3. What does each of the views (design, process, component, deployment) address in the software architecture? Explain with an example.
4. What are the seven workflows in the life cycle?
5. What levels of activity takes place in these workflows during each of the four phases (inception, elaboration, construction and transition).
6. Define iteration. Discuss the sequence of activities in an iteration workflow.
7. Bring out the differences between iterations and increments along with suitable diagrams.

Thank You..