LECTURE NOTES

ON

SOFTWARE PROJECT MANAGEMENT

2018 – 2019

IV B. Tech I Semester (JNTUA-R15)

Mr. P. Suresh, Assistant Professor

CHADALAWADA RAMANAMMA ENGINEERING COLLEGE

(AUTONOMOUS)

Chadalawada Nagar, Renigunta Road, Tirupati – 517 506

Department of Computer Science and Engineering
SYLLABUS

Unit I

Unit II
Improving Software Economics: Reducing Software product size, improving software processes, improving team effectiveness, improving automation, Achieving required quality, peer inspections. The old way and the new: The principles of conventional software engineering, principles of modern software management, transitioning to an iterative process

Unit III
Life cycle phases: Engineering and production stages, inception, Elaboration, construction, transition phases. Artifacts of the process: The artifact sets, Management artifacts, Engineering artifacts, programmatic artifacts. Model based software architectures: A Management perspective and technical perspective

Unit IV

Unit V

TEXT BOOKS:
1. Software Project Management, Walker Royce, Pearson Education.

REFERENCE BOOKS
2. Head First PMP, Jennifer Greene & Andrew Stellman, O”Reilly,2007
3. Agile Project Management, Jim Highsmith, Pearson education, 2004
UNIT - I


1. CONVENTIONAL SOFTWARE MANAGEMENT

Conventional software management practices are sound in theory, but practice is still tied to archaic (outdated) technology and techniques.

Conventional software economics provides a benchmark of performance for conventional software management principles.

The best thing about software is its flexibility: It can be programmed to do almost anything.

The worst thing about software is also its flexibility: The "almost anything" characteristic has made it difficult to plan, monitor, and control software development.

Three important analyses of the state of the software engineering industry are

1. Software development is still highly unpredictable. Only about 10% of software projects are delivered successfully within initial budget and schedule estimates.
2. Management discipline is more of a discriminator in success or failure than are technology advances.
3. The level of software scrap and rework is indicative of an immature process.

All three analyses reached the same general conclusion: The success rate for software projects is very low. The three analyses provide a good introduction to the magnitude of the software problem and the current norms for conventional software management performance.

1.1 THE WATERFALL MODEL

Most software engineering texts present the waterfall model as the source of the "conventional" software process.

1.1.1 IN THEORY
It provides an insightful and concise summary of conventional software management. In 1970 Winston Royce, presented a paper titled “Managing the Development of Large Scale Software Systems”.

Three main primary points are
1. There are two essential steps common to the development of computer programs: analysis and coding.

Waterfall Model Part 1: The two basic steps to building a program

Analysis

Coding

Analysis and coding both involve creative work that directly contributes to the usefulness of the end product.
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. Below Figure illustrates the resulting project profile and the basic steps in developing a large-scale program.

**Waterfall Model part 2: The large scale system approach**

![Waterfall Model](image)

3. The basic framework described in the waterfall model is risky and invites failure. The testing phase 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.

**Waterfall Model part 3: Five necessary improvements for this approach to work**

1. **Program design comes first.** Insert a preliminary program design phase between the software requirements generation phase and the analysis phase. **By this technique, the program designer assures that the software will not fail because of storage, timing, and data flux (continuous change).** As analysis proceeds in the succeeding phase, the program designer must impose on the analyst the storage, timing, and operational constraints in such a way that he senses the consequences. If the total resources to be applied are insufficient or if the embryonic(in an early stage of development) operational design is wrong, it will be recognized at this early stage and the iteration with requirements and preliminary design can be redone before final design, coding, and test commences. How is this program design procedure implemented?
The following steps are required:

- Begin the design process with program designers, not analysts or programmers.
- Design, define, and allocate the data processing modes even at the risk of being wrong.
- Allocate processing functions, design the database, allocate execution time, define interfaces and processing modes with the operating system, describe input and output processing, and define preliminary operating procedures.
- Write an overview document that is understandable, informative, and current so that every worker on the project can gain an elemental understanding of the system.

2. **Document the design.** The amount of documentation required on most software programs is quite a lot, certainly much more than most programmers, analysts, or program designers are willing to do if left to their own devices. Why do we need so much documentation?

   - Each designer must communicate with interfacing designers, managers, and possibly customers.
   - During early phases, the documentation is the design.
   - The real economic value of documentation is to support later changes by a separate test team, a separate maintenance team, and operations personnel who are not software literate.

3. **Do it twice.** If a computer program is being developed for the first time, arrange matters so that the version finally delivered to the customer for operational deployment is actually the second version insofar as critical design/operations are concerned. Note that this is simply the entire process done in a time scale that is relatively small with respect to the overall effort. In the first version, the team must have a special broad competence where they can quickly sense trouble spots in the design, model them, model alternatives, forget the straightforward aspects of the design that aren't worth studying at this early point, and, finally, arrive at an error-free program.

4. **Plan, control, and monitor testing.** The biggest user of project resources—manpower, computer time, and/or management judgment—is the test phase. This is the phase of greatest risk in terms of cost and schedule. It occurs at the latest point in the schedule, when backup alternatives are least available, if at all. The previous three recommendations were all aimed at uncovering and solving problems before entering the test phase. However, even after doing these things, there is still a test phase and there are still important things to be done, including:

   1. Employ a team of test specialists who were not responsible for the original design;
   2. Employ visual inspections to spot the obvious errors like dropped minus signs, missing factors of two, jumps to wrong addresses (do not use the computer to detect this kind of thing, it is too expensive);
   3. Test every logic path;
   4. Employ the final checkout on the target computer.

5. **Involve the customer.** It is important to involve the customer in a formal way so that he has committed himself at earlier points before final delivery. There are three points following requirements definition where the insight, judgment, and commitment of the customer can bolster the development effort. These include a "preliminary software review" following the preliminary program design step, a sequence of "critical software design reviews" during program design, and a "final software acceptance review".
1.1.2 IN PRACTICE

Some software projects still practice the conventional software management approach.

It is useful to summarize the characteristics of the conventional process as it has typically been applied, which is not necessarily as it was intended. Projects destined for trouble frequently exhibit the following symptoms:

1. Protracted integration and late design breakage.
2. Late risk resolution.
4. Adversarial (conflict or opposition) stakeholder relationships.
5. Focus on documents and review meetings.

Protracted Integration and Late Design Breakage

For a typical development project that used a waterfall model management process, Figure 1-2 illustrates development progress versus time. Progress is defined as percent coded, that is, demonstrable in its target form.

The following sequence was common:

1. Early success via paper designs and thorough (often too thorough) briefings.
2. Commitment to code late in the life cycle.
3. Integration nightmares (unpleasant experience) due to unforeseen implementation issues and interface ambiguities.
4. Heavy budget and schedule pressure to get the system working.
5. Late shoe-homing of no optimal fixes, with no time for redesign.
6. A very fragile, unmentionable product delivered late.

<table>
<thead>
<tr>
<th>Format</th>
<th>Ad hoc text</th>
<th>Flowcharts</th>
<th>Source code</th>
<th>Configuration baselines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirements analysis</td>
<td>Program design</td>
<td>Coding and unit testing</td>
<td>Protracted integration and testing</td>
<td></td>
</tr>
<tr>
<td>Product</td>
<td></td>
<td></td>
<td></td>
<td>Fragile baselines</td>
</tr>
<tr>
<td>Documents</td>
<td>Documents</td>
<td>Coded units</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sequential activities: requirements — design — coding — integration — testing

Figure 1-2. Progress profile of a conventional software project
In the conventional model, the entire system was designed on paper, then implemented all at once, then integrated. Table 1-1 provides a typical profile of cost expenditures across the spectrum of software activities.

<table>
<thead>
<tr>
<th>ACTIVITY</th>
<th>COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Management</td>
<td>5%</td>
</tr>
<tr>
<td>Requirements</td>
<td>5%</td>
</tr>
<tr>
<td>Design</td>
<td>10%</td>
</tr>
<tr>
<td>Code and unit testing</td>
<td>30%</td>
</tr>
<tr>
<td>Integration and test</td>
<td>40%</td>
</tr>
<tr>
<td>Deployment</td>
<td>5%</td>
</tr>
<tr>
<td>Environment</td>
<td>5%</td>
</tr>
<tr>
<td>Total</td>
<td>100%</td>
</tr>
</tbody>
</table>

**Late risk resolution** A serious issue associated with the waterfall lifecycle was the lack of early risk resolution. Figure 1.3 illustrates a typical risk profile for conventional waterfall model projects. It includes four distinct periods of risk exposure, where risk is defined as the probability of missing a cost, schedule, feature, or quality goal. Early in the life cycle, as the requirements were being specified, the actual risk exposure was highly unpredictable.

**Requirements-Driven Functional Decomposition:** This approach depends on specifying requirements completely and unambiguously before other development activities begin. It naïvely treats all requirements as equally important, and depends on those requirements remaining constant over the software development life cycle. These conditions rarely occur in the real world. Specification of requirements is a difficult and important part of the software development process.

Another property of the conventional approach is that the requirements were typically specified in a functional manner. Built into the classic waterfall process was the fundamental
assumption that the software itself was decomposed into functions; requirements were then allocated to the resulting components. This decomposition was often very different from a decomposition based on object-oriented design and the use of existing components. Figure 1-4 illustrates the result of requirements-driven approaches: a software structure that is organized around the requirements specification structure.

![Diagram of software component organization](image)

**Figure 1-4.** Suboptimal software component organization resulting from a requirements-driven approach

**Adversarial Stakeholder Relationships:**

The conventional process tended to result in adversarial stakeholder relationships, in large part because of the difficulties of requirements specification and the exchange of information solely through paper documents that captured engineering information in ad hoc formats.

The following sequence of events was typical for most contractual software efforts:

1. The contractor prepared a draft contract-deliverable document that captured an intermediate artifact and delivered it to the customer for approval.
2. The customer was expected to provide comments (typically within 15 to 30 days).
3. The contractor incorporated these comments and submitted (typically within 15 to 30 days) a final version for approval.

This one-shot review process encouraged high levels of sensitivity on the part of customers and contractors.

**Focus on Documents and Review Meetings:**

The conventional process focused on producing various documents that attempted to describe the software product, with insufficient focus on producing tangible increments of the products themselves. Contractors were driven to produce literally tons of paper to meet milestones and demonstrate progress to stakeholders, rather than spend their energy on tasks that would reduce risk and produce quality software. Typically, presenters and the audience reviewed the simple things that they understood rather than the complex and important issues. Most design reviews therefore resulted in low engineering value and high cost in terms of the effort and schedule involved in their preparation and conduct. They presented merely a facade of progress.
Table 1-2 summarizes the results of a typical design review.

<table>
<thead>
<tr>
<th>APPARENT RESULTS</th>
<th>REAL RESULTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big briefing to a diverse audience</td>
<td>Only a small percentage of the audience understands the software.</td>
</tr>
<tr>
<td></td>
<td>Briefings and documents expose few of the important assets and risks of complex software systems.</td>
</tr>
<tr>
<td>A design that appears to be compliant</td>
<td>There is no tangible evidence of compliance.</td>
</tr>
<tr>
<td></td>
<td>Compliance with ambiguous requirements is of little value.</td>
</tr>
<tr>
<td>Coverage of requirements (typically hundreds)</td>
<td>Few (tens) are design drivers.</td>
</tr>
<tr>
<td></td>
<td>Dealing with all requirements dilutes the focus on the critical drivers.</td>
</tr>
<tr>
<td>A design considered “innocent until proven guilty”</td>
<td>The design is always guilty.</td>
</tr>
<tr>
<td></td>
<td>Design flaws are exposed later in the life cycle.</td>
</tr>
</tbody>
</table>

1.2 CONVENTIONAL SOFTWARE MANAGEMENT PERFORMANCE

Barry Boehm's "Industrial Software Metrics Top 10 List" is a good, objective characterization of the state of software development.

1. “Finding and fixing a software problem after delivery costs 100 times more than finding and fixing the problem in early design phases.”

   This metric is not unique to software development. When one of the big automobile companies implements a recall for post-delivery defect, cost of repair can be many orders of magnitude greater than cost fixing the defect during the engineering or production stage.

2. “You can compress software development schedules 25% of nominal, but no more.”

   One reason for this is that N% reduction in schedule would require an M% increase in personnel resources (assuming that other parameters remain fixed). Any increase in the number of people requires more management overhead.

3. “For every $1 you spend on development, you will spend $2 on maintenance.”

   Boehm calls this the “iron law of software development”. Whether you build long-lived product that undergoes commercial version upgrades twice a year or build a one-of-a-kind custom software system, twice as much money will probably be spent over the maintenance life cycle than spent in the development life cycle. Most of the software in operation is considered to be difficult maintain.

4. “Software development and maintenance costs are primarily a function of the number of source lines of code.”

   This metric is primarily the result of the majority of custom software development, lack of commercial componentry, and lack of reuse inherent in the era of the conventional process.
5. “Variations among people account for the **biggest** differences in software productivity.”

This is key piece of conventional wisdom: Hire good people. This metric is both overhype and underhype. When you don’t know objectively why you succeed or failed, the noticeable problem is the quality of the people. This judgment is subjective and difficult to challenge.

6. “The overall ratio of software to hardware costs is still growing. In 1955 it was 15:85; in 1985, 85:15.”

The fact that the software costs 85% of the cost of most systems doesn’t denote software productivity as it is about the level of functionality being allocated to software in system solutions. The need for software, its breadth of applications, and its complexity continue to grow almost without limits.

7. “Only about 15% of software development effort is devoted to programming.”

This is an important indicator of the need for balance. Many activities besides coding are necessary for software project success. Requirements management, design, testing, project control, planning, change management are equally important considerations that consume 85% of the resources.

8. “Software systems and products typically cost 3 times as much per SLOC as individual software programs. Software-system products (i.e., system of systems) cost 9 times as much.”

This exponential relationship is the essence of diseconomy of scale. Unlike other commodities, the more software you build, the more expensive it is per source line.

9. “Walkthroughs catch 60% of the errors.”

This may be true. In general, walkthroughs and other forms of human inspection are good at catching surface problems and style issues.

10. “80% of the contribution comes from 20% of the contributors.”

This is a primary statement that is true across almost any engineering discipline. 80% of the engineering is consumed by 20% of the requirements. 80% of the software cost is consumed by 20% of the components. 80% of the errors are caused by 20% of the components. 80% of the resources are consumed by 20% of the components. 80% of the progress is made by 20% of the people.
2. EVOLUTION OF SOFTWARE ECONOMICS

2.1 Software Economics

Most software cost models can be abstracted into a function of five basic parameters: size, process, personnel, environment, and required quality.

1. The size of the end product (in human-generated components), which is typically quantified in terms of the number of source instructions or the number of function points required to develop the required functionality.

2. The process used to produce the end product, in particular the ability of the process to avoid non-value-adding activities (rework, bureaucratic delays, communications overhead).

3. The capabilities of software engineering personnel, and particularly their experience with the computer science issues and the applications domain issues of the project.

4. The environment, which is made up of the tools and techniques available to support efficient software development and to automate the process.

5. The required quality of the product, including its features, performance, reliability, and adaptability.

The relationships among these parameters and the estimated cost can be written as follows:

\[ \text{Effort} = (\text{Personnel}) (\text{Environment}) (\text{Quality}) (\text{Size}^{\text{process}}) \]

One important aspect of software economics (as represented within today's software cost models) is that the relationship between effort and size exhibits a diseconomy of scale. The diseconomy of scale of software development is a result of the process exponent being greater than 1.0. Contrary to most manufacturing processes, the more software you build, the more expensive it is per unit item.

Figure 2-1 shows three generations of basic technology advancement in tools, components, and processes. The required levels of quality and personnel are assumed to be constant. The ordinate of the graph refers to software unit costs (pick your favorite: per SLOC, per function point, per component) realized by an organization.

The three generations of software development are defined as follows:

1. **Conventional:** 1960s and 1970s, craftsmanship. Organizations used custom tools, custom processes, and virtually all custom components built in primitive languages. Project performance was highly predictable in that cost, schedule, and quality objectives were almost always underachieved.

2. **Transition:** 1980s and 1990s, software engineering. Organizations used more-repeatable processes and off-the-shelf tools, and mostly (>70%) custom components built in higher level languages. Some of the components (<30%) were available as commercial products, including the operating system, database management system, networking, and graphical user interface.
3. **Modern practices**: 2000 and later, software production. This book's philosophy is rooted in the use of managed and measured processes, integrated automation environments, and mostly (70%) off-the-shelf components. Perhaps as few as 30% of the components need to be custom built. Technologies for environment automation, size reduction, and process improvement are not independent of one another. In each new era, the key is complementary growth in all technologies. For example, the process advances could not be used successfully without new component technologies and increased tool automation.

**Target objective: improved ROI**

- 1960s–1970s
  - Waterfall model
  - Functional design
  - Diseconomy of scale

- 1980s–1990s
  - Process improvement
  - Encapsulation-based
  - Diseconomy of scale

- 2000 and on
  - Iterative development
  - Component-based
  - Return on investment

**Corresponding environment, size, and process technologies**

<table>
<thead>
<tr>
<th>Conventional</th>
<th>Transition</th>
<th>Modern Practices</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Environments/tools:</strong></td>
<td>Environment/tools:</td>
<td>Environment/tools:</td>
</tr>
<tr>
<td>Custom</td>
<td>Off-the-shelf, separate</td>
<td>Off-the-shelf, integrated</td>
</tr>
<tr>
<td><strong>Size:</strong></td>
<td>Size:</td>
<td>Size:</td>
</tr>
<tr>
<td>100% custom</td>
<td>30% component-based</td>
<td>70% component-based</td>
</tr>
<tr>
<td><strong>Process:</strong></td>
<td>Process:</td>
<td>Process:</td>
</tr>
<tr>
<td>Ad hoc</td>
<td>Repeatable</td>
<td>Managed/measured</td>
</tr>
</tbody>
</table>

**Typical project performance**

- **Predictably bad**
  - Always:
    - Over budget
    - Over schedule

- **Unpredictable**
  - Infrequently:
    - On budget
    - On schedule

- **Predictable**
  - Usually:
    - On budget
    - On schedule

**Figure 2-1. Three generations of software economics leading to the target objective**

Organizations are achieving better economies of scale in successive technology eras—with very large projects (systems of systems), long-lived products, and lines of business comprising multiple similar projects. Figure 2-2 provides an overview of how a return on investment (ROI) profile can be achieved in subsequent efforts across life cycles of various domains.
2.2 PRAGMATIC SOFTWARE COST ESTIMATION

One critical problem in software cost estimation is a lack of well-documented case studies of projects that used an iterative development approach. Software industry has inconsistently defined metrics or atomic units of measure, the data from actual projects are highly suspect in terms of consistency and comparability. It is hard enough to collect a homogeneous set of project data within one organization; it is extremely difficult to homogenize data across different organizations with different processes, languages, domains, and so on.

There have been many debates among developers and vendors of software cost estimation models and tools.
Three topics of these debates are of particular interest here:
1. Which cost estimation model to use?
2. Whether to measure software size in source lines of code or function points.
3. What constitutes a good estimate?

There are several popular cost estimation models (such as COCOMO, CHECKPOINT, ESTIMACS, KnowledgePlan, Price-S, ProQMS, SEER, SLIM, SOFTCOST, and SPQR/20), COCOMO is also one of the most open and well-documented cost estimation models. The general accuracy of conventional cost models (such as COCOMO) has been described as "within 20% of actuals, 70% of the time."

Most real-world use of cost models is bottom-up (substantiating a target cost) rather than top-down (estimating the "should" cost). Figure 2-3 illustrates the predominant practice: The software project manager defines the target cost of the software, and then manipulates the parameters and sizing until the target cost can be justified. The basis for the target cost maybe to win a proposal, to solicit customer funding, to attain internal corporate funding, or to achieve some other goal.

The process described in Figure 2-3 is not all bad. In fact, it is absolutely necessary to analyze the cost risks and understand the sensitivities and trade-offs objectively. It forces the software project manager to examine the risks associated with achieving the target costs and to discuss this information with other stakeholders.

A good software cost estimate has the following attributes:
1. It is conceived and supported by the project manager, architecture team, development team, and test team accountable for performing the work.
2. It is accepted by all stakeholders as ambitious but realizable.
3. It is based on a well-defined software cost model with a credible basis.
4. It is based on a database of relevant project experience that includes similar processes, similar technologies, similar environments, similar quality requirements, and similar people.
5. It is defined in enough detail so that its key risk areas are understood and the probability of success is objectively assessed.

Extrapolating from a good estimate, 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.
UNIT - II


The old way and the new: The principles of conventional software Engineering, principles of modern software management, transitioning to an iterative process.

Improving Software Economics

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 3-1 lists some of the technology developments, process improvement efforts, and management approaches targeted at improving the economics of software development and integration.

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

Table 3-1. Important trends in improving software economics
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 as the general term for reducing the "source" language size 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).

Size reduction is the primary motivation behind improvements in higher order languages (such as C++, Ada 95, Java, Visual Basic), 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.

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. Function point metrics provide a standardized method for measuring the various functions of a software application.

The basic units of function points are external user inputs, external outputs, internal logical data groups, external data interfaces, and external inquiries.

<table>
<thead>
<tr>
<th>TABLE 3-2. Language expressiveness of some of today’s popular languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>LANGUAGE</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>Assembly</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>FORTRAN 77</td>
</tr>
<tr>
<td>COBOL 85</td>
</tr>
<tr>
<td>Ada 83</td>
</tr>
<tr>
<td>C++</td>
</tr>
<tr>
<td>Ada 95</td>
</tr>
<tr>
<td>Java</td>
</tr>
<tr>
<td>Visual Basic</td>
</tr>
</tbody>
</table>
1.2 OBJECT-ORIENTED METHODS AND VISUAL MODELING

Object-oriented technology is not germane to most of the software management topics discussed here, and books on object-oriented technology abound. Object-oriented programming languages appear to benefit both software productivity and software quality. The fundamental impact of object-oriented technology is in reducing the overall size of what needs to be developed. People like drawing pictures to explain something to others or to themselves. When they do it for software system design, they call these pictures diagrams or diagrammatic models and the very notation for them a modeling language.

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.

1.3 REUSE

Reusing existing components and building reusable components have been natural software engineering activities since the earliest improvements in programming languages. 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:

- They have an economic motivation for continued support.
- They take ownership of improving product quality, adding new features, and transitioning to new technologies.
- 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 trade-offs. 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.

**1.4 COMMERCIAL COMPONENTS**

A common approach being pursued today in many 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 straightforward in practice. Table 3-3 identifies some of the advantages and disadvantages of using commercial components.

<table>
<thead>
<tr>
<th>APPROACH</th>
<th>ADVANTAGES</th>
<th>DISADVANTAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commercial components</td>
<td>Predictable license costs</td>
<td>Frequent upgrades</td>
</tr>
<tr>
<td></td>
<td>Broadly used, mature technology</td>
<td>Up-front license fees</td>
</tr>
<tr>
<td></td>
<td>Available now</td>
<td>Recurring maintenance fees</td>
</tr>
<tr>
<td></td>
<td>Dedicated support organization</td>
<td>Dependency on vendor</td>
</tr>
<tr>
<td></td>
<td>Hardware/software independence</td>
<td>Run-time efficiency sacrifices</td>
</tr>
<tr>
<td></td>
<td>Rich in functionality</td>
<td>Functionality constraints</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Integration not always trivial</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No control over upgrades and maintenance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unnecessary features that consume extra resources</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Often inadequate reliability and stability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Multiple-vendor incompatibilities</td>
</tr>
<tr>
<td>Custom development</td>
<td>Complete change freedom</td>
<td>Expensive, unpredictable development</td>
</tr>
<tr>
<td></td>
<td>Smaller, often simpler implementations</td>
<td>Unpredictable availability date</td>
</tr>
<tr>
<td></td>
<td>Often better performance</td>
<td>Undefined maintenance model</td>
</tr>
<tr>
<td></td>
<td>Control of development and enhancement</td>
<td>Often immature and fragile</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Single-platform dependency</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Drain on expert resources</td>
</tr>
</tbody>
</table>
2. IMPROVING SOFTWARE PROCESSES

Process is an overloaded term. 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 3-4

<table>
<thead>
<tr>
<th>TABLE 3-4. Three levels of process and their attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ATTRIBUTES</strong></td>
</tr>
<tr>
<td>Subject</td>
</tr>
<tr>
<td>Objectives</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Audience</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Metrics</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Concerns</td>
</tr>
<tr>
<td>Time scales</td>
</tr>
</tbody>
</table>

Schedule improvement has at least three dimensions.

1. We could take an N-step process and improve the efficiency of each step.
2. We could take an N-step process and eliminate some steps so that it is now only an M-step process.
3. We could take an N-step process and use more concurrency in the activities being performed or the resources being applied.
3. IMPROVING TEAM EFFECTIVENESS

Teamwork is much more important than the sum of the individuals. With software teams, a project manager needs to configure a balance of solid talent with highly skilled people in the leverage positions. Some maxims of team management include the following:

- A well-managed project can succeed with a nominal engineering team.
- A mismanaged project will almost never succeed, even with an expert team of engineers.
- A well-architected system can be built by a nominal team of software builders.
- A poorly architected system will flounder even with an expert team of builders.

Boehm five staffing principles are:

1. The principle of top talent: Use better and fewer people
   
   There is a ‘natural’ team size for most jobs, and being grossly over or under this size is bad for team dynamics because it results in too little or too much pressure on individuals to perform.

2. The principle of job matching: Fit the tasks to the skills and motivation of the people available.

3. The principle of career progression: An organization does best in the long run by helping its people to self-actualize.
   
   Good performers usually self-actualize in any environment. Organizations can help and hinder employee self-actualization, but organizational energy will benefit average and below-average performers the most. Organizational training programs are typically strategic undertakings which educational value. Project training programs are purely tactical, intended to be useful and applied the day after training ends.

4. The principle of team balance: Select people who will complement and harmonize with one another.
   
   Software team balance has many dimensions, and when a team is unbalanced in any one of them, a project becomes seriously at risk. These dimensions include:
   
   Raw Skills: intelligence, objectivity, creativity, organization, analytical thinking
   
   Psychological makeup: leaders and followers, risk takers and conservatives etc.,
   
   Objectives: financial, feature set, quality, timeliness

5. The principle of phase-out: Keeping a misfit on the team doesn't benefit anyone.
   
   A misfit gives you a reason to find a better person or to live with fewer people. A misfit demotivates other team members, will not self-actualize, and disrupts the team balance in some dimension.

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.

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 for 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.

**Round-trip engineering** 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 (for example, creating a .visual design model from a source code representation).

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.
5. ACHIEVING REQUIRED QUALITY

Software best practices are derived from the development process and technologies. Table 3-5 summarizes some dimensions of quality improvement.

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.

### Table 3-5. General quality improvements with a modern process

<table>
<thead>
<tr>
<th>QUALITY DRIVER</th>
<th>CONVENTIONAL PROCESS</th>
<th>MODERN ITERATIVE PROCESSES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements misunderstanding</td>
<td>Discovered late</td>
<td>Resolved early</td>
</tr>
<tr>
<td>Development risk</td>
<td>Unknown until late</td>
<td>Understood and resolved early</td>
</tr>
<tr>
<td>Commercial components</td>
<td>Mostly unavailable</td>
<td>Still a quality driver, but trade-offs must be resolved early in the life cycle</td>
</tr>
<tr>
<td>Change management</td>
<td>Late in the life cycle, chaotic and malignant</td>
<td>Early in the life cycle, straightforward and benign</td>
</tr>
<tr>
<td>Design errors</td>
<td>Discovered late</td>
<td>Resolved early</td>
</tr>
<tr>
<td>Automation</td>
<td>Mostly error-prone manual procedures</td>
<td>Mostly automated, error-free evolution of artifacts</td>
</tr>
<tr>
<td>Resource adequacy</td>
<td>Unpredictable</td>
<td>Predictable</td>
</tr>
<tr>
<td>Schedules</td>
<td>Overconstrained</td>
<td>Tunable to quality, performance, and technology</td>
</tr>
<tr>
<td>Target performance</td>
<td>Paper-based analysis or separate simulation</td>
<td>Executing prototypes, early performance feedback, quantitative understanding</td>
</tr>
<tr>
<td>Software process rigor</td>
<td>Document-based</td>
<td>Managed, measured, and tool-supported</td>
</tr>
</tbody>
</table>

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 margin.
- Initial design review. Optimistic assessments of adequate design margin were based...
mostly on paper analysis or rough 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.

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.

7. THE PRINCIPLES OF CONVENTIONAL SOFTWARE ENGINEERING

Davis’s top 30 principles of conventional software engineering are as follows:

1. **Make quality #1.** Quality must be quantified and mechanism 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.
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 an “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 desire 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 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 important. The right people with insufficient tools, languages, and process will succeed.

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 poor designs, and the oldest 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 they see, the more they need.** The more functionality 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 create programs with tricks—constructs that perform a function correctly, but in an unclear 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 Tom 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.

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

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

30. **Expert excellence.** Your employees will do much better if you have high expectations for them.

### 8. THE PRINCIPLES OF MODERN SOFTWARE MANAGEMENT

Top 10 principles of modern software management by Walker Royce 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.** 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 textual 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.** Transitioning the current state of the product artifacts into an executable demonstration of relevant scenarios stimulates earlier convergence on integration, a more tangible understanding of design trade-offs, and earlier elimination of architectural defects.

9. **Plan intermediate releases in groups of usage scenarios with evolving levels of 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 is suitable for all software developments.

**Figure 4-1.** The top five principles of a modern process

**TABLE 4-1.** Modern process approaches for solving conventional problems

<table>
<thead>
<tr>
<th>CONVENTIONAL PROCESS: TOP 10 RISKS</th>
<th>MODERN PROCESS: INHERENT RISK RESOLUTION FEATURES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Late breakeage and excessive scrap/rework</td>
<td>Quality, cost, schedule</td>
</tr>
<tr>
<td>2. Attraction of key personnel</td>
<td>Quality, cost, schedule</td>
</tr>
<tr>
<td>3. Inadequate development resources</td>
<td>Cost, schedule</td>
</tr>
<tr>
<td>4. Adversarial stakeholders</td>
<td>Cost, schedule</td>
</tr>
<tr>
<td>5. Necessary technology insertion</td>
<td>Cost, schedule</td>
</tr>
<tr>
<td>6. Requirements creep</td>
<td>Cost, schedule</td>
</tr>
<tr>
<td>7. Analysis paralysis</td>
<td>Schedule</td>
</tr>
<tr>
<td>8. Inadequate performance</td>
<td>Quality</td>
</tr>
<tr>
<td>9. Overemphasis on artifacts</td>
<td>Schedule</td>
</tr>
<tr>
<td>10. Inadequate function</td>
<td>Quality</td>
</tr>
</tbody>
</table>

*Dept. of CSE, CREC*  
*P. Suresh, Asst. Professor*
9. TRANSITIONING TO AN ITERATIVE PROCESS

Modern software development processes have moved away from 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 model. 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 COCOMO II to above top 10 principles of a modern process.

- **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 one of the primary reasons that the software industry has moved to an **iterative life-cycle 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.

- **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.

- **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 the 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.

- **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 completely based 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.

- **Software process maturity.** The Software Engineering Institute's 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 **objective quality control.**
UNIT - III

Life cycle phases: Engineering and production stages, inception, Elaboration, construction, transition phases.

Artifacts of the process: The artifact sets, Management artifacts, Engineering artifacts, programmatic artifacts.

Model based software architectures: A Management perspective and technical perspective.

******************************

1. LIFE CYCLE PHASES

Characteristic of a successful software development process is the well-defined separation between "research and development" activities and "production" activities.

Most unsuccessful projects exhibit one of the following characteristics:
1. An overemphasis on research and development
2. An overemphasis on production.

Successful modern projects 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.

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:
1. The engineering stage – driven by less predictable but smaller teams doing design and synthesis activities.
2. The production stage – driven by more predictable but larger teams doing construction, test, and deployment activities.

<table>
<thead>
<tr>
<th>TABLE 5-1. The two stages of the life cycle: engineering and production</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIFE-CYCLE ASPECT</td>
</tr>
<tr>
<td>Risk reduction</td>
</tr>
<tr>
<td>Products</td>
</tr>
<tr>
<td>Activities</td>
</tr>
<tr>
<td>Assessment</td>
</tr>
<tr>
<td>Economics</td>
</tr>
<tr>
<td>Management</td>
</tr>
</tbody>
</table>
The transition between engineering and production is a crucial event for the various stakeholders. The production plan has been agreed upon, and there is a good enough understanding of the problem and the solution that all stakeholders can make a firm commitment to go ahead with production.

Engineering stage is decomposed into two distinct phases, inception and elaboration, and the production stage into construction and transition.

These four phases of the life-cycle process are loosely mapped to the conceptual framework of the spiral model as shown in Figure

![Figure 5.1: The phases of the life-cycle process](image)

### 1.1 INCEPTION PHASE

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

**Primary Objectives**

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

**Essential Activities**

1. 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.
2. 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.
3. Planning and preparing a business case. Alternatives for risk management, staffing, iteration plans, and cost/schedule/profitability trade-offs are evaluated.

Primary Evaluation Criteria

1. Do all stakeholders concur on the scope definition and cost and schedule estimates?
2. Are requirements understood, as evidenced by the fidelity of the critical use cases?
3. Are the cost and schedule estimates, priorities, risks, and development processes credible?
4. Do the depth and breadth of an architecture prototype demonstrate the preceding criteria?
5. Are actual resource expenditures versus planned expenditures acceptable?

1.2 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 can 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, & risk.

Primary Objectives

1. Baselining the architecture as rapidly as practical (establishing a configuration-managed snapshot in which all changes are rationalized, tracked, and maintained)
2. Baselining the vision
3. Baselining a high-fidelity plan for the construction phase
4. Demonstrating that the baseline architecture will support the vision at a reasonable cost in a reasonable time

Essential Activities

1. Elaborating the vision.
2. Elaborating the process and infrastructure.
3. Elaborating the architecture and selecting components.

Primary Evaluation Criteria

1. Is the vision stable?
2. Is the architecture stable?
3. Does the executable demonstration show that the major risk elements have been addressed and credibly resolved?
4. Is the construction phase plan of sufficient fidelity, and is it backed up with a credible basis of estimate?
5. 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?
6. Are actual resource expenditures versus planned expenditures acceptable?
1.3 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

1. Minimizing development costs by optimizing resources and avoiding unnecessary scrap and rework
2. Achieving adequate quality as rapidly as practical
3. Achieving useful versions (alpha, beta, and other test releases) as rapidly as practical

Essential Activities

1. Resource management, control, and process optimization
2. Complete component development and testing against evaluation criteria
3. Assessment of product releases against acceptance criteria of the vision

Primary Evaluation Criteria

1. Is this product baseline mature enough to be deployed in the user community? (Existing defects are not obstacles to achieving the purpose of the next release.)
2. 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.)
3. Are the stakeholders ready for transition to the user community?
4. Are actual resource expenditures versus planned expenditures acceptable?

1.4 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:

1. Beta testing to validate the new system against user expectations
2. Beta testing and parallel operation relative to a legacy system it is replacing
3. Conversion of operational databases
4. Training of users and maintainers

The transition phase concludes when the deployment baseline has achieved the complete vision.

Primary Objectives

1. Achieving user self-supportability
2. Achieving stakeholder concurrence that deployment baselines are complete and consistent with the evaluation criteria of the vision
3. Achieving final product baselines as rapidly and cost-effectively as practical
Essential Activities

1. Synchronization and integration of concurrent construction increments into consistent deployment baselines
2. Deployment-specific engineering
3. Assessment of deployment baselines against the complete vision and acceptance criteria in the requirements set

Primary Evaluation Criteria

1. Is the user satisfied?
2. Are actual resource expenditures versus planned expenditures acceptable?

Are actual resource expenditures versus planned expenditures acceptable?

2. ARTIFACTS OF THE PROCESS

2.1 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-processable languages and associated files). The artifact sets are shown in Figure 6-1.

![Figure 6-1. Overview of the artifact sets](image-url)