UNIT-II
PART - A

Improving Software Economics: Reducing Software product size, improving software processes, improving team effectiveness, improving automation, Achieving required quality, peer inspections.

Introduction

- Improvements in the economics of software development have been not only difficult to achieve but also difficult to measure and substantiate (prove).
- If an organization focuses too much on improving only one aspect of its software development process, it will not realize any significant economic improvement even though it improves this one aspect spectacularly.

Introduction

- The key to substantial improvement is a balanced attack across several inter-related dimensions.
- The presentation of the important dimensions around the **five** basic parameters of the software cost model presented earlier.

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

Introduction

- These parameters are given in priority order for most software domains.
- Below table lists some of the technology developments, process improvement efforts, and management approaches targeted at improving the economics of software development and integration.
- Most software experts would also stress the significant dependencies among these trends.
- **For example, tools enable size reduction and process improvements, size-reduction approaches lead to process changes, and process improvements drive tool requirements.**

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

Introduction

- **Two decades ago, teams developing a user interface would spend extensive time analyzing operations, human factors, screen layout, and screen dynamics.**
- **All this would be done on paper because it was extremely expensive to commit designs, even informal prototypes, to executable code.**
- Therefore, the process emphasized a fairly heavyweight set of early paper artifacts and user concurrence so that these “requirements” could be frozen and the high construction costs could be minimized.

Introduction

- **Graphical user interface (GUI) technology is a good example of tools enabling a new and different process.**
- As GUI technology matured, the conventional user interface process became obsolete.
- **GUI builder tools permitted engineering teams to construct an executable user interface faster and at less cost.**

Introduction

- Another important factor that has influenced software technology improvements across the board is the ever-increasing advances in hardware performance.
- The availability of more cycles, more memory, and more bandwidth has eliminated many sources of software implementation complexity.
- Simpler, brute-force solutions are now possible, and hardware improvements are probably the enabling advance behind most software technology improvements of substance.

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.

Reducing Software product size

- 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).

Reducing Software product size

- By choosing the type of the language
- By using Object-Oriented methods and visual modeling
- By reusing the existing components and building reusable components
- By using commercial components, we can reduce the product size of a software

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 below table.

Languages

Table: 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

Object Oriented Methods And Visual Modeling

- There has been a widespread movements in the 1990s toward Object-Oriented technology.
- Some studies concluded that Object-Oriented programming languages appear to benefit both software productivity and software quality.
- One of such Object-Oriented method is UML-Unified Modeling Language.

Object Oriented Methods And Visual Modeling

- **Booch described the following three reasons for the success of the projects that are using Object-Oriented concepts:**

1. An Object-Oriented model of the problem and its solution encourages a common vocabulary between the end user 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 weaken the entire development effort.
3. An Object-Oriented architecture provides a clear separation among different elements of a system, creating firewalls that prevent a change in one part of the system from the entire architecture.

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.
- Beware of “open” reuse libraries sponsored by nonprofit organizations. They lack economic motivation, trustworthiness, and accountability for quality, support, improvement, and usability.

Reuse

- **Organizations that translates reusable components into commercial products has 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. Below figure examines the economic trade-offs.
- The steep initial curve illustrates the economic obstacle to developing reusable components.

Reuse

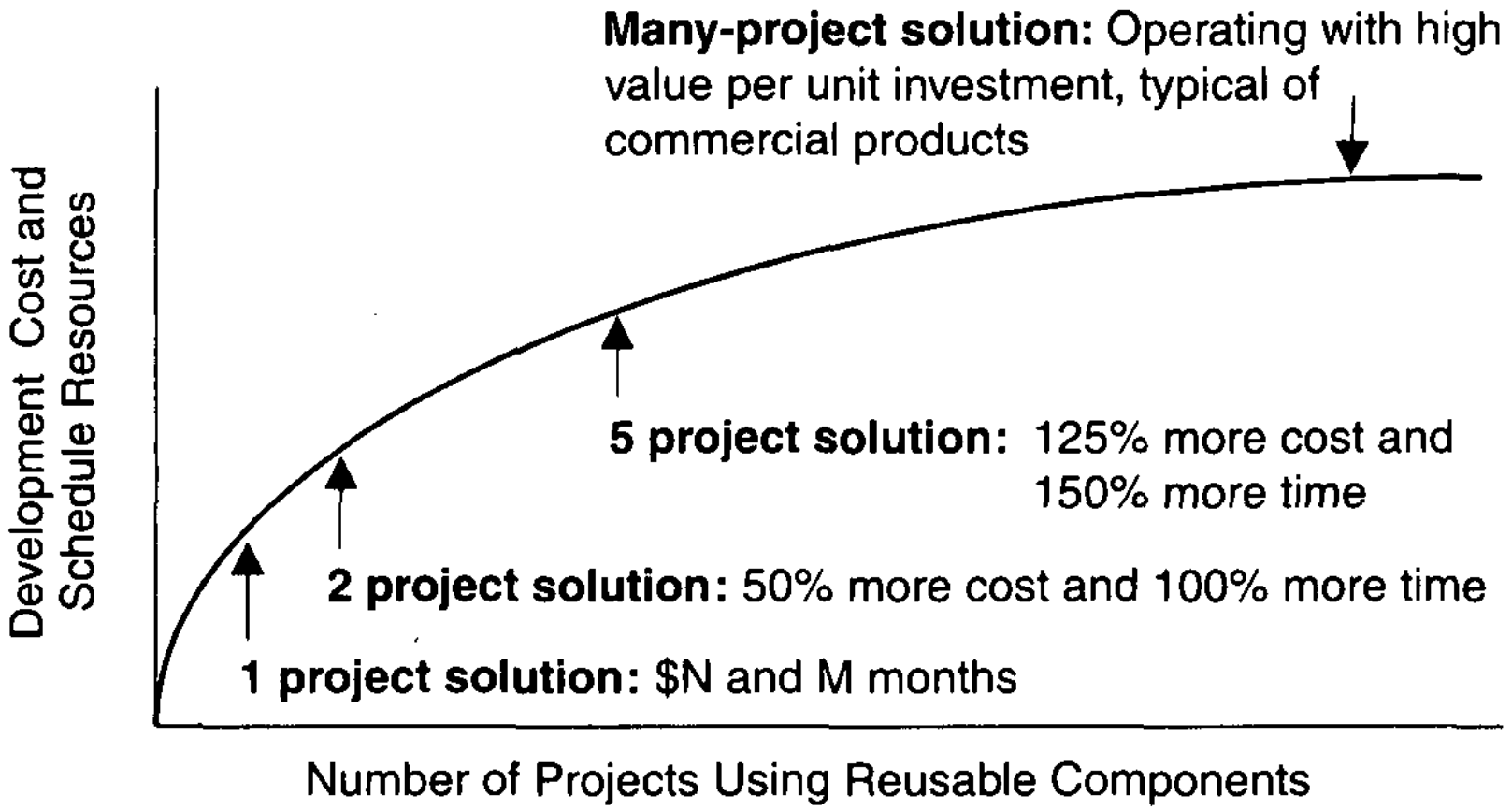

Figure: Cost and schedule investments necessary to achieve reusable components

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 is not proven straight forward in practice.
- Below table identifies some of the advantages and disadvantages of using commercial components desirable as a means of reducing custom development.

Commercial Components

TABLE 3-3. *Advantages and disadvantages of commercial components versus custom software*

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

Improving Software Processes

- Process is overloaded term for software oriented organizations, there are many processes and sub processes. It use three distinct process perspectives.

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

Improving Software Processes

2. Macroprocess

- A project's policies, and practices for producing a complete software product within certain cost, schedule, and quality constraints.
- The focus of the macroprocess is on creating an sufficient instance of the metaprocess for a specific set of constraints.

Improving Software Processes

3. **Microprocess**
   - A projects team's policies, procedures, and practices for achieving an artifact of a software process.
   - The focus of the microprocess is on achieving an intermediate product baseline with sufficient 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 below table.

Improving Software Processes

Table: 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 budget, 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 Release/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

Improving Software Processes

- To achieve success, most software projects require an incredibly complex web of sequential and parallel steps.
- As the scale of a project increases, more overhead steps must be included just to manage the complexity of this web.
- All project processes consist of **productive activities and overhead activities**.
- **Productive activities** result in tangible progress toward the end product.
- For software efforts, these activities include prototyping, modeling, coding, debugging, and user documentation.

Improving Software Processes

- **Overhead activities** that have an intangible impact on the end product are required in plan preparation, documentation, progress monitoring, risk assessment, financial assessment, configuration control, quality assessment, integration, testing, late scrap and rework, management, personnel training, business administration, and other tasks.
- The objective of process improvement is to maximize the allocation of resources to productive activities and minimize the impact of overhead activities on resources such as personnel, computers, and schedule.

Improving Software Processes

- 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.

Improving Software Processes

- 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 in one 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.

Improving Team Effectiveness

- Balance and coverage are two of the most important features of excellent teams. Whenever a team is out of balance, it is vulnerable.
- It is the responsibility of the project manager to keep track of his teams. Since teamwork is much more important than the sum of the individuals.

Improving Team Effectiveness

- 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.

Improving Team Effectiveness

- In examining how to staff a software project, Boehm offered the following five staffing principles:

1. The principle of top talent: Use better and fewer people.
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.
4. The principle of team balance: Select people who will complement and synchronize with one another.
5. The principle of phase-out: Keeping a misfit on the team doesn’t benefit anyone.

Improving Team Effectiveness

- Software project managers need many leadership qualities in order to enhance team effectiveness. Although these qualities are intangible.
- 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 stake-holders is a prerequisite for success.

Improving Team Effectiveness

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.

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.

Improving Automation Through Software Environments

- Above all, configuration management environments provide the foundation for executing and instrumenting 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.

Improving Automation Through Software Environments

- 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 the automation of the development process.
- This environment should include requirements management, document automation, host/target programming tools, automated regression testing, continuous and integrated change management, and feature/defect tracking.
- A common thread in successful software projects is that they hire good people and provide them with good tools to accomplish their jobs.
- 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.

Improving Automation Through Software Environments

- The following are the some of the configuration management environments which provide the foundation for executing and implementing the process.
- **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 (for example, creating a visual design model from a source code representation).

Improving Automation Through Software Environments

- One word of caution is necessary in describing the 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 niche:

Improving Automation Through Software Environments

- 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.

Achieving Required Quality

- Software best practices are derived from the development process and technologies.
- Below table summarizes some dimensions of quality improvement.

Table: General quality improvement with a modern process

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

Achieving Required Quality

- Key practices that improve overall software quality include the following:
  - Focusing on driving requirements and critical use cases early in the lifecycle
  - Focusing on requirements completeness and traceability late in the life cycle
  - 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

Achieving Required Quality

- 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 close look into performance issues through demonstration-based evaluations

Achieving Required Quality

- Improved insight into run-time performance issues is even more important as projects incorporate mixtures of commercial components and custom-developed components.
- Conventional development processes stressed early sizing and timing estimates of computer program resource utilization.
- The typical chronology of events in performance assessment was as follows:
  - Project inception
  - Initial design review
  - Mid-life-cycle design review
  - Integration and test

Achieving Required Quality

- 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. However, the infrastructure—including the operating system overhead, the database management overhead, and the interprocess and network communications overhead—and all the secondary threads were typically misunderstood.

Achieving Required Quality

- 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 under-stood early design trade-offs.

Peer Inspections: A Pragmatic View

- Peer inspections are frequently overhyped as the key aspect of a quality system.
- 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:

Peer Inspections: A Pragmatic View

- 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

Peer Inspections: A Pragmatic View

- 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.

Peer Inspections: A Pragmatic View

- In all but trivial cases, architectural issues are exposed only through more rigorous engineering activities such as the following:
  - Analysis, prototyping, or experimentation
  - Constructing design models
  - Committing the current state of the design model to an executable implementation
  - Demonstrating the current implementation strengths and weaknesses in the context of critical subsets of the use cases and scenarios
  - Incorporating lessons learned back into the models, use cases, implementations, and plans