UNIT-I
PART-B

Evolution of Software Economics: Software Economics, Pragmatic Software Cost Estimation.

Software Economics

- **Economics** means System of inter relationship of money, industry and employment.
- The cost of the software can be estimated by considering the following things as parameters to a function.

  1. Size
  2. Process
  3. Personnel
  4. Environment
  5. Required Quality

Software Economics

- The **Size** of the end product Which is measured in terms of the number of Source Lines Of Code or the number of function points required to develop the required functionality.
- The **Process** Used to produce the end product, in particular the ability of the process is to avoid non-value-adding activities (rework, bureaucratic delays, communications overhead).
- The capabilities of software engineering **Personnel**, and particularly their experience with the computer science issues and the application domain issues of the project.

Software Economics

- The **Environment** which is made up of the tools and techniques available to support efficient software development and to automate the process.
- The required **Quality** of the product includes its features, performance, reliability, and flexibility.
- The relationship among these parameters and the estimated cost can be calculated by using,

$$\text{Effort} = (\text{Personnel})\ (\text{Environment})\ (\text{Quality})\ (\text{Size}^{\text{Process}})$$

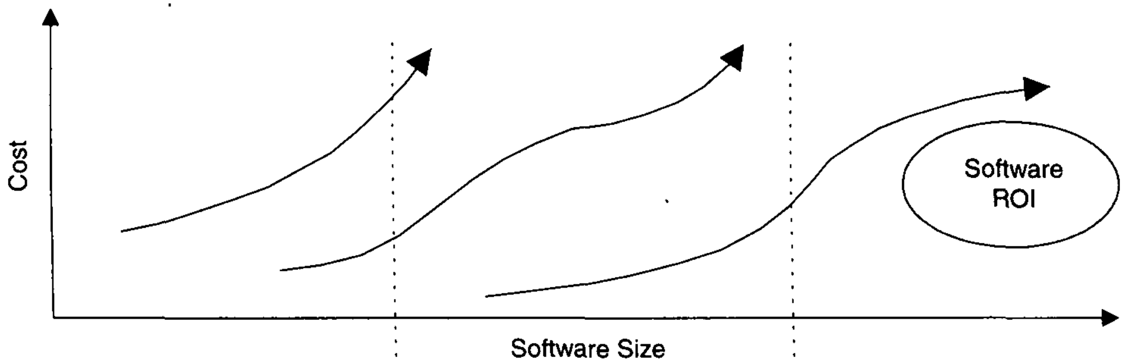
Software Economics

- Several parametric models have been developed to estimate software costs; all of them can be generally abstracted into this form.
- 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.

Software Economics

- The below figure 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.

Target objective: improved ROI



- 1960s–1970s - Waterfall model - Functional design - Diseconomy of scale	- 1980s–1990s - Process impovement - Encapsulation-based - Diseconomy of scale	- 2000 and on - Iterative development - Component-based - Return on investment

Corresponding environment, size, and process technologies

Conventional	Transition	Modern Practices
Environments/tools: Custom	**Environment/tools:** Off-the-shelf, separate	**Environment/tools:** Off-the-shelf, integrated
Size: 100% custom	**Size:** 30% component-based, 70% custom	**Size:** 70% component-based, 30% custom
Process: Ad hoc	**Process:** Repeatable	**Process:** Managed/measured

Typical project performance

Predictably bad	***Unpredictable***	**Predictable**
Always: Over budget Over schedule	Infrequently: On budget On schedule	Usually: On budget On schedule

Software Economics

- The abscissa represents the life cycle of the software business engaged in by the 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.

Software Economics

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.

Software Economics

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.
   - With advances in software technology and integrated production environments, these component-based systems can be produced very rapidly.

Software Economics

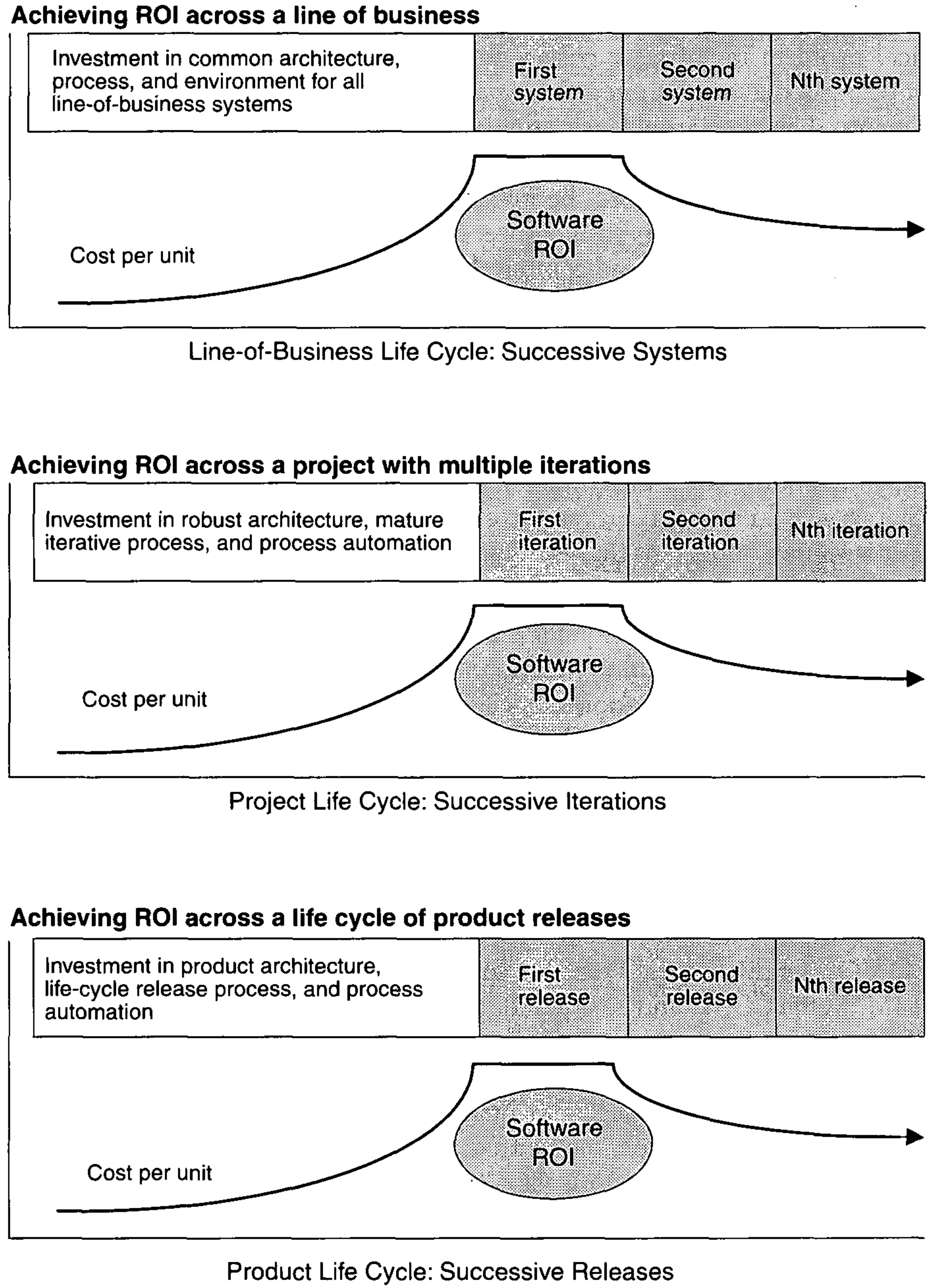
- Technologies for environment automation, size reduction, and process improvement are not independent of one another.
- In each new era, the key is complementary growth in all technologies.
- For example, the process advances could not be used successfully without new component technologies and increased tool automation.

Software Economics

- The transition to modern practices and the promise of improved software economics are by no means guaranteed.
- We must be realistic in comparing the promises of a well-executed, next-generation process using modern technologies against the ugly realities of history.
- It is a sure bet that many organizations attempting to carry out modern projects with modern techniques and technologies will end up with the same old snafu.

Software Economics

- 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.
- Below figure provides an overview of how a return on investment (ROI) profile can be achieved in subsequent efforts across life cycles of various domains.


Achieving ROI across a line of business
Investment in common architecture, process, and environment for all line-of-business systems
First system
Second system
Nth system
Software ROI
Cost per unit
Line-of-Business Life Cycle: Successive Systems
Achieving ROI across a project with multiple iterations
Investment in robust architecture, mature iterative process, and process automation
First iteration
Second iteration
Nth iteration
Software ROI
Cost per unit
Project Life Cycle: Successive Iterations
Achieving ROI across a life cycle of product releases
Investment in product architecture, life-cycle release process, and process automation
First release
Second release
Nth release
Software ROI
Cost per unit
Product Life Cycle: Successive Releases

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.
- Although cost model vendors claim that their tools are suitable for estimating iterative development projects, few are based on empirical project databases with modern iterative development success stories.
- 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.

Pragmatic Software Cost Estimation

- The exact definition of a function point or a SLOC is not very important, just as the exact length of a foot or a meter is equally arbitrary.
- 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?

Pragmatic Software Cost Estimation

1. **Which cost estimation model to use?**
   - About 50 vendors of software cost estimation tools, data, and services compete within the software industry.
   - There are several popular cost estimation models (such as COCOMO, CHECKPOINT, ESTIMACS, Knowledge Plan, Price-S, ProQMS, SEER, SLIM, SOFTCOST, and SPQR/20), as well as numerous organization-specific models.
   - Among those Ada COCOMO and COCOMO II are the basis of many software economics arguments and perspectives.
   - COCOMO is also one of the most open and well-documented cost estimation models.

Pragmatic Software Cost Estimation

2. **Whether to measure software size in source lines of code or function points.**
   - The measurement of software size has been the subject of much rhetoric.
   - There are basically two objective points of view: **source lines of code** and **function points**.

Pragmatic Software Cost Estimation

- **SLOC**
  - Most software experts argued that the SLOC is a poor measure of size. But it has some value in the software Industry.
  - SLOC worked well in applications that were custom built why because of easy to automate and instrument.
  - Now a days there are so many automatic source code generators are available and there are so many advanced higher-level languages are available. So SLOC is a uncertain measure.

Pragmatic Software Cost Estimation

- **Function points**
  - The primary advantage of using function points is that this method is independent of technology and is therefore a much better primitive unit for comparisons among projects and organizations.
  - The main disadvantage is that the primitive definitions are abstract and measurements are not easily derived directly from the evolving artifacts.

Pragmatic Software Cost Estimation

- Although both measures of size have their drawbacks, an organization can make either one work.
- The use of some measure is better than none at all.
- Anyone doing cross-project or cross-organization comparisons should be using function points as the measure of size.
- Function points are also probably a more accurate estimator in the early phases of a project life cycle.
- In later phases SLOC becomes a more useful and precise measurement basis of various metrics perspectives.

Pragmatic Software Cost Estimation

- 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).
- Below figure illustrates the pre-dominant practice: The software project manager defines the target cost of the software, then manipulates the parameters and sizing until the target cost can be justified.
- The rationale for the target cost may be to win a proposal, to solicit customer funding, to attain internal corporate funding, or to achieve some other goal.

Pragmatic Software Cost Estimation

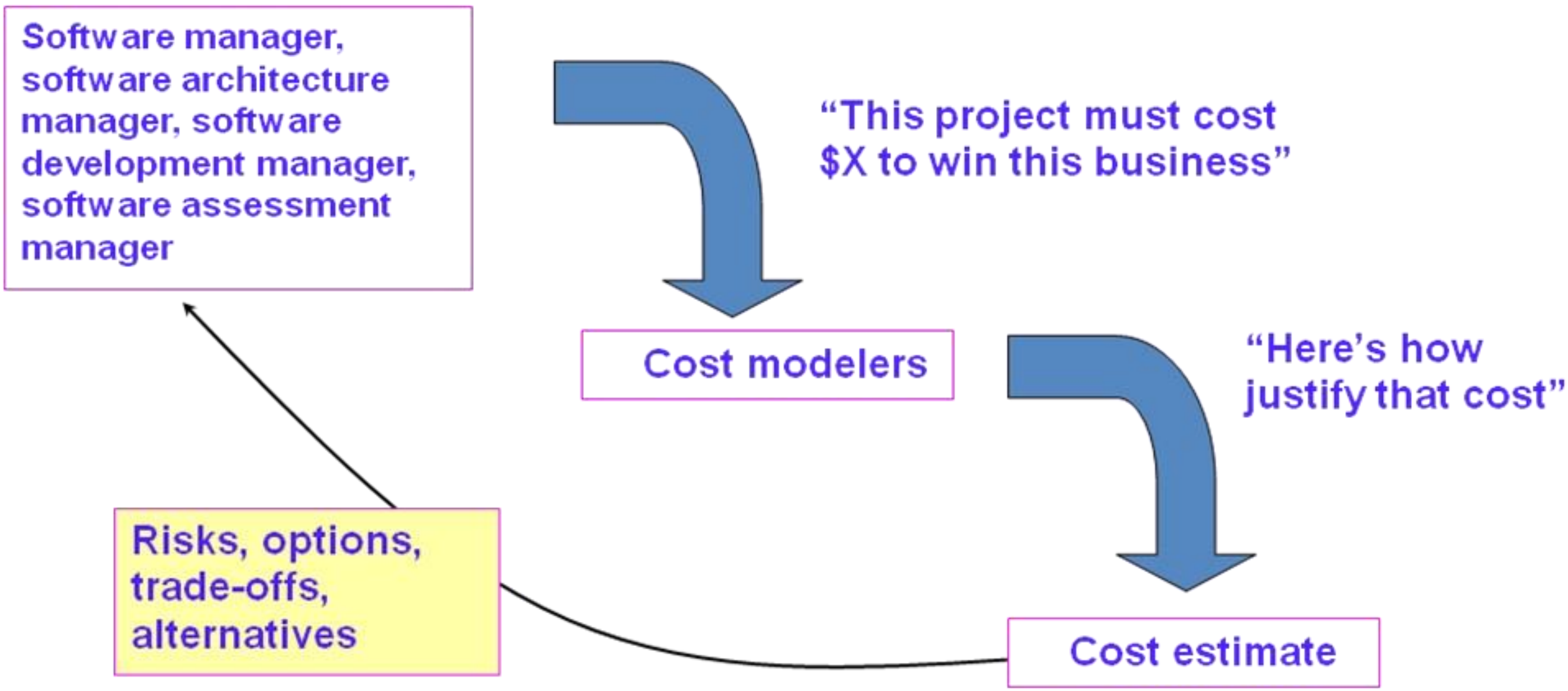


Fig: The predominant cost estimation process

Pragmatic Software Cost Estimation

- The process described in above figure is not all bad.
- 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.
- The result is usually various perturbations in the plans, designs, process, or scope being proposed.
- This process provides a good vehicle for a basis of estimate and an overall cost analysis.

Pragmatic Software Cost Estimation

3. What constitutes a good software cost estimate?

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