**Course Name: Software Engineering Economics**

**Credit Hours:** 3

**Prerequisites: (Intro. to) Software Engineering**

**Objectives:** Determine how new software development technologies affect the economics and risks of software development. Understand and characterize how the paradigm shift affects or replaces our current methods of software cost, schedule and risk estimation. Identify best practices and lessons learned with Web-based developments. Identify acquisition and lifecycle risks

**Outcomes:** The student will be able to learn,

1. Understand and be able to apply the key software engineering economic fundamentals to real-world software economic issues. 2. Illustrate through example the key software life cycle economics, including product and process life cycles; portfolios; proposals; investment decisions; pricing and costing, and earned value management (EVM). 3. Apply the concepts of risk and uncertainty to real world software development projects, including goals; estimates; prioritization and decision making. 4. Perform best-practice economic analysis methods. 5. Relate and interpret the “good-enough” principle; friction-free economy; ecosystems and outsourcing

**Course Outline:**

Programming aspects, economic aspects, human relations aspects, software trends: cost, social impact, the plurality of SE Means, The GOALS Approach to Software Engineering, The Software Work Breakdown Structure (WBS), Software Maintenance, introduction to COCOMO, definitions and assumptions, development effort and schedule, phase distribution, The Raylaigh Distribution, interpolation, basic software maintenance effort estimation. Performance Models, Optimal Performance, Sensitivity Analysis, Cost-Effectiveness Models.

**Reference Materials:**

1. *Software Engineering Economics*, Boehm, Prentice Hall, 1981(or Latest Edition).

2. *Software Cost Estimation with COCOMO II*, Boehm et al., Prentice Hall, 2000 (or Latest Edition).

3. *Making the Software Business Case: Improvement, Reifer, Don*, Addison Wesley, 2001, (or Latest Edition).

**Course Outline:**

Economic aspects of Software Engineering, programming aspects of Software Engineering, human relations aspects of Software Engineering, COCOMO,

General views of software engineering, software development effort,Software failure, software maintenance estimation, software development phase distribution, software trends: cost, social impact, Cost-Effectiveness Model.

***1). Economic aspects of Software Engineering:***

***Economics*** *is the study of how resources (people, time, facilities, money) are used to produce and distribute commodities and how services are provided in society.*[*Engineering economics*](https://en.wikipedia.org/wiki/Engineering_economics)*is a branch of microeconomics dealing with engineering related economic decisions.*

Software engineering economics is about making decisions related to software engineering in a business context. The success of a software product, service, and solution depends on good business management. Yet, in many companies and organizations, software business relationships to software development and engineering remain vague. This knowledge area (KA) provides an overview on software engineering economics. Economics is the study of value, costs, resources, and their relationship in a given context or situation. In the discipline of software engineering, activities have costs, but the resulting software itself has economic attributes as well. Software engineering economics provides a way to study the attributes of software and software processes in a systematic way that relates them to economic measures.

**Software Engineering Economics Fundamentals**

### ***Finance***

### ***Accounting***

### ***Controlling***

### ***Cash Flow***

### ***Decision-Making Process***

### ***Valuation***

***2). software development effort:***

The most important activity in software project management process is the estimation of Software development effort. The literature shows many algorithmic cost estimation models such as Boehm’s COCOMO, Albrecht's Function Point Analysis, Soft computing based techniques etc. but each model have their own advantages and disadvantages in predicting development cost and effort. This is because of the availability of project data in the initial stages of development process is often incomplete, inconsistent and vague.

# 5 Steps to Software Development Effort Estimation

## 1- Scoping

You need first to scope the project even if you do not have the full detailed requirements but you can assume some of them or add margins later. While in most cases you will have a defined scope to start with.

You can always list your assumptions to justify the outcome of the estimation process and its results.

## 2- Decomposition

In this step, you will need to break your software into smaller components and functions and you can categorize them to a different set of elements, this is similar to work breakdown structure but only for the software components not all the working activities for the software.

You may also collect different data from the project team or the customer to ensure that you have listed all functionalities.

## 3- Sizing

In this step, the actual estimation will be done for each component alone, and I will illustrate more about how you will do that using the techniques mentioned above, this will be illustrated in 8 steps in details below.

In this step, and for more validation, you can use different estimation techniques to analyze the different estimation outputs and you may take an average of these estimates as well.

## 4- Expert and Peer Review

After initial estimate, you will need at some point to ask for expert opinion for some new functionalities you may not aware off, or for considering a review from your peers that you have done the correct estimation. Moreover, you may need to do some analogy based techniques for similar components or functions developed before or maybe a similar project to ensure that you are on the correct path.

## 5- Estimation Finalization

This can be considered the final step as you aggregate all the estimations from all components and functions and have a baseline estimate. You can go another round across the process until reaching the correct estimate which will be approved by the Project team and the Management as well.

***3) Software failure***

Most software projects fail completely or partial because they don’t meet all their requirements. These requirements can be the cost, schedule, quality, or requirements objectives. According to many studies, failure rate of software projects ranges between 50% – 80%. There are a variety of causes for software failures but the most common are:

* Lack of user participation
* Changing requirements
* Unrealistic or unarticulated project goals
* Inaccurate estimates of needed resources
* Badly defined system requirements
* Lack of resources
* Poor communication among customers, developers, and users
* Use of immature technology
* Inability to handle the project’s complexity
* Poor Project Management
* Lack of Stakeholder involvement

***4) Software maintenance estimation***

Reports suggest that the cost of maintenance is high. A study on estimating software maintenance found that the cost of maintenance is as high as 67% of the cost of entire software process cycle.

On an average, the cost of software maintenance is more than 50% of all SDLC phases. There are various factors, which trigger maintenance cost go high, such as:



### **Real-world factors affecting Maintenance Cost**

* The standard age of any software is considered up to 10 to 15 years.
* Older softwares, which were meant to work on slow machines with less memory and storage capacity cannot keep themselves challenging against newly coming enhanced softwares on modern hardware.
* As technology advances, it becomes costly to maintain old software.
* Most maintenance engineers are newbie and use trial and error method to rectify problem.
* Often, changes made can easily hurt the original structure of the software, making it hard for any subsequent changes.
* Changes are often left undocumented which may cause more conflicts in future.

### **Software-end factors affecting Maintenance Cost**

* Structure of Software Program
* Programming Language
* Dependence on external environment
* Staff reliability and availability

## Maintenance Activities

IEEE provides a framework for sequential maintenance process activities. It can be used in iterative manner and can be extended so that customized items and processes can be included.



***5) Programming aspect of Software Engineering:-***

### Programming paradigm**.**

A [programming paradigm](https://en.wikipedia.org/wiki/Programming_paradigm) is a fundamental style of [computer programming](https://en.wikipedia.org/wiki/Computer_programming), which is not generally dictated by the project management methodology (such as waterfall or agile). Paradigms differ in the concepts and abstractions used to represent the elements of a program (such as objects, functions, variables, constraints) and the steps that comprise a computation (such as assignations, evaluation, continuations, data flows). Sometimes the concepts asserted by the paradigm are utilized cooperatively in high-level system architecture design; in other cases, the programming paradigm's scope is limited to the internal structure of a particular program or module.

A [programming language](https://en.wikipedia.org/wiki/Programming_language) can support [multiple paradigms](https://en.wikipedia.org/wiki/Multi-paradigm_programming_language). For example, programs written in [C++](https://en.wikipedia.org/wiki/C%2B%2B) or [Object Pascal](https://en.wikipedia.org/wiki/Object_Pascal) can be purely [procedural](https://en.wikipedia.org/wiki/Procedural_programming), or purely [object-oriented](https://en.wikipedia.org/wiki/Object-oriented_programming), or contain elements of both paradigms. Software designers and programmers decide how to use those paradigm elements. In [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming), programmers can think of a program as a collection of interacting objects, while in [functional programming](https://en.wikipedia.org/wiki/Functional_programming) a program can be thought of as a sequence of stateless function evaluations. When programming computers or systems with many processors, [process-oriented programming](https://en.wikipedia.org/wiki/Process-oriented_programming) allows programmers to think about applications as sets of concurrent processes acting upon logically shared [data structures](https://en.wikipedia.org/wiki/Data_structures).

Just as different groups in [software engineering](https://en.wikipedia.org/wiki/Software_engineering) advocate different *methodologies*, different [programming languages](https://en.wikipedia.org/wiki/Programming_language) advocate different *programming paradigms*. Some languages are designed to support one paradigm ([Smalltalk](https://en.wikipedia.org/wiki/Smalltalk) supports object-oriented programming, [Haskell](https://en.wikipedia.org/wiki/Haskell_(programming_language)) supports functional programming), while other programming languages support multiple paradigms (such as [Object Pascal](https://en.wikipedia.org/wiki/Object_Pascal), [C++](https://en.wikipedia.org/wiki/C%2B%2B), [C#](https://en.wikipedia.org/wiki/C_Sharp_(programming_language)), [Visual Basic](https://en.wikipedia.org/wiki/Visual_Basic), [Common Lisp](https://en.wikipedia.org/wiki/Common_Lisp), [Scheme](https://en.wikipedia.org/wiki/Scheme_(programming_language)), [Python](https://en.wikipedia.org/wiki/Python_(programming_language)), [Ruby](https://en.wikipedia.org/wiki/Ruby_(programming_language)), and [Oz](https://en.wikipedia.org/wiki/Oz_(programming_language))).

Many programming paradigms are as well known for what methods they *forbid* as for what they enable. For instance, pure functional programming forbids using [side-effects](https://en.wikipedia.org/wiki/Side-effect_(computer_science)); [structured programming](https://en.wikipedia.org/wiki/Structured_programming) forbids using [goto](https://en.wikipedia.org/wiki/Goto" \o "Goto) statements. Partly for this reason, new paradigms are often regarded as doctrinaire or overly rigid by those accustomed to earlier styles. Avoiding certain methods can make it easier to prove theorems about a program's correctness, or simply to understand its behavior.

Examples of high-level paradigms include:

* [Aspect-oriented software development](https://en.wikipedia.org/wiki/Aspect-oriented_software_development)
* [Domain-specific modeling](https://en.wikipedia.org/wiki/Domain-specific_modeling)
* [Model-driven engineering](https://en.wikipedia.org/wiki/Model-driven_engineering)
* [Object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming) methodologies
  + [Grady Booch](https://en.wikipedia.org/wiki/Grady_Booch)'s [object-oriented design](https://en.wikipedia.org/wiki/Object-oriented_design) (OOD), also known as object-oriented analysis and design (OOAD). The Booch model includes six diagrams: class, object, state transition, interaction, module, and process.[[15]](https://en.wikipedia.org/wiki/Software_development#cite_note-15)
* [Search-based software engineering](https://en.wikipedia.org/wiki/Search-based_software_engineering)
* [Service-oriented modeling](https://en.wikipedia.org/wiki/Service-oriented_modeling)
* [Structured programming](https://en.wikipedia.org/wiki/Structured_programming)
* [Top-down and bottom-up design](https://en.wikipedia.org/wiki/Top-down_and_bottom-up_design)
  + [Top-down programming](https://en.wikipedia.org/wiki/Top-down_programming): evolved in the 1970s by IBM researcher [Harlan Mills](https://en.wikipedia.org/wiki/Harlan_Mills) (and [Niklaus Wirth](https://en.wikipedia.org/wiki/Niklaus_Wirth" \o "Niklaus Wirth)) in developed [structured programming](https://en.wikipedia.org/wiki/Structured_programming).

**6) COCOMO Model:-**

Cocomo (Constructive Cost Model) is a regression model based on LOC, i.e **number of Lines of Code**. It is a procedural cost estimate model for software projects and often used as a process of reliably predicting the various parameters associated with making a project such as size, effort, cost, time and quality. It was proposed by Barry Boehm in 1970 and is based on the study of 63 projects, which make it one of the best-documented models.

The key parameters which define the quality of any software products, which are also an outcome of the Cocomo are primarily Effort & Schedule:

* **Effort:** Amount of labor that will be required to complete a task. It is measured in person-months units.
* **Schedule:** Simply means the amount of time required for the completion of the job, which is, of course, proportional to the effort put. It is measured in the units of time such as weeks, months.

Different models of Cocomo have been proposed to predict the cost estimation at different levels, based on the amount of accuracy and correctness required. All of these models can be applied to a variety of projects.

Boehm’s definition of organic, semidetached, and embedded systems:

1. **Organic –** A software project is said to be an organic type if the team size required is adequately small, the problem is well understood and has been solved in the past and also the team members have a nominal experience regarding the problem.
2. **Semi-detached –** A software project is said to be a Semi-detached type if the vital characteristics such as team-size, experience, knowledge of the various programming environment lie in between that of organic and Embedded. The projects classified as Semi-Detached are comparatively less familiar and difficult to develop compared to the organic ones and require more experience and better guidance and creativity. Eg: Compilers or different Embedded Systems can be considered of Semi-Detached type.
3. **Embedded –** A software project with requiring the highest level of complexity, creativity, and experience requirement fall under this category. Such software requires a larger team size than the other two models and also the developers need to be sufficiently experienced and creative to develop such complex models.

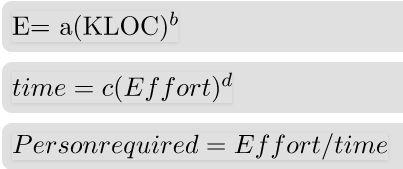
All the above system types utilize different values of the constants used in Effort Calculations.

**Types of Models:** COCOMO consists of a hierarchy of three increasingly detailed and accurate forms. Any of the three forms can be adopted according to our requirements. These are types of COCOMO model:

* 1. Basic COCOMO Model
  2. Intermediate COCOMO Model
  3. Detailed COCOMO Model

**-Cost Estimation Formula**

**1) Basic Model:**

****

**2) Intermediate Model:**

The basic Cocomo model assumes that the effort is only a function of the number of lines of code and some constants evaluated according to the different software system. However, in reality, no system’s effort and schedule can be solely calculated on the basis of Lines of Code. For that, various other factors such as reliability, experience, Capability. These factors are known as Cost Drivers and the Intermediate Model utilizes 15 such drivers for cost estimation.

Classification of Cost Drivers and their attributes:

**(i) Product attributes –**

* + - Required software reliability extent
    - Size of the application database
    - The complexity of the product

**(ii) Hardware attributes –**

* + - Run-time performance constraints
    - Memory constraints
    - The volatility of the virtual machine environment

**(iii) Personnel attributes –**

* + - Analyst capability
    - Software engineering capability
    - Applications experience
    - Virtual machine experience
    - Programming language experience

**(iv) Project attributes –**

* + - Use of software tools
    - Application of software engineering methods
    - Required development schedule

**3) Detailed Model:**  
Detailed COCOMO incorporates all characteristics of the intermediate version with an assessment of the cost driver’s impact on each step of the software engineering process. The detailed model uses different effort multipliers for each cost driver attribute. In detailed cocomo, the whole software is divided into different modules and then we apply COCOMO in different modules to estimate effort and then sum the effort.

The Six phases of detailed COCOMO are:

Planning and requirements

* System design
* Detailed design
* Module code and test
* Integration and test
* Cost Constructive model

**7) Human relations aspects of Software Engineering:-**

**Introduction**

As Chapanis defined, “Human Factors discovers and applies information about human behavior, abilities, limitations, and other characteristics to the design of tools, machines, systems, tasks, jobs, and environments for productive, safe, comfortable, and effective human use” .

**Objective of HCI and Human Factors Design**

The central purpose of HCI and Human Factors design is to create products that meet the operability and learn ability goals. This design should meet the user’s needs by being effective, efficient, and consistent but also of high quality yet keeping an eye on the major concern of the consumer in most cases, that is, affordability.

*The engineering discipline for designers and developers must focus on the following:*

* Users and their psychology.
* Amount of work that the user must do, including task goals, performance requirements, and group communication requirements.
* Quality and performance.
* Information required by users and their jobs.
* Network of relationships between the environment, users, jobs, technologies, and resources.

**Benefits**

* Elevated user satisfaction
* Decreased training time and costs
* Reduced operator stress
* Reduced product liability
* Greater system performance and dependability
* Improved efficiency and effect ability

**Biased Approach to Human Factors**

It is often visible that people take Human Factors not too seriously because it is often regarded as common sense. Many companies heavily channel their resources and time towards factors of software development like planning, management, control, and progress. They often neglect the fact that they must present their product in such a way that it is easy to learn and implement and that it should be aesthetic in nature.

**Usability Testing:**

Usability is a software quality characteristic that surveys on software usability cost and benefits and it can simply be defined as the external attributes of software quality. The process of involving the users in the development life cycle ensures that the product is user-friendly and is widely accepted.

Usability aims at the following:

* Shortening the time to accomplish tasks,
* Reducing the number of mistakes made,
* Reducing learning time,
* Improving people's satisfaction with a system.

**Components of Usability:**

* User interface design
* Website design
* Rapid prototyping techniques
* Groupware
* Usability evaluation
* Digital design
* Digital typography

**Benefits of Usability**

* Elevated sales and consumer satisfaction
* Advertising advantage
* Increased productivity and efficiency
* Decreased training costs and time
* Lesser support and maintenance costs
* Reduced documentation and support cost
* Increased satisfaction, performance, and productivity (For users)
* Reduced development and maintenance costs and improved sales (For companies)

8) General views of software engineering.

**Answer:** - The process of a software development has three Generic views which are:

1. **Definition Phase** - It is the base of Definition phase. The experts get the knowledge about "What".
   * Information needed for processing.
   * Which functions are required.
   * Expectations about the capacity.
   * Interface which is established.
   * Area of the validation.

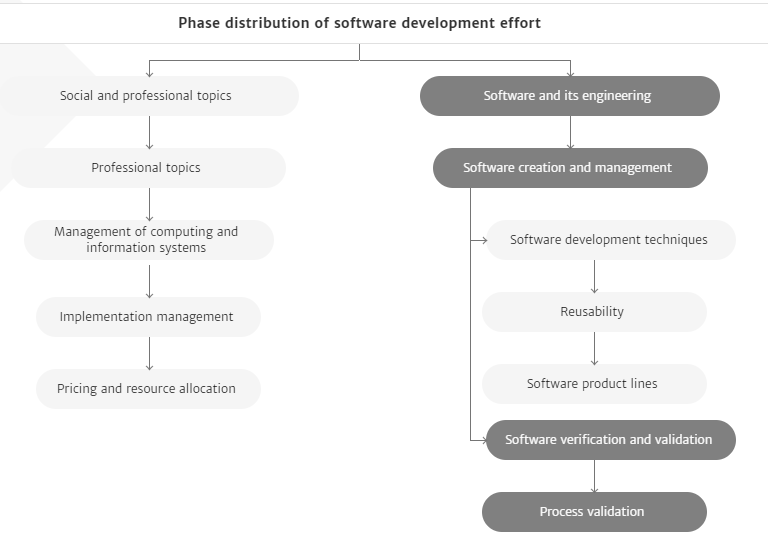
This phase defines all the expectations depending on the standard of the software Engineering. It contains three steps.

* + Analysis of system
  + Planning of project
  + Requirement Analysis

1. **Development phase** - Focus point of development phase is "How". After the explanation of "What" it turn to "How". Various type of question raised in developer mind that how to design the data structure and Architecture of software, Procedural detail how to implemented and how design convert in a programming language and testing of software how to perform. Three special steps always taken in this phase which are
   * Design of software
   * Coding
   * testing of software system
2. **Maintenance phase** - The main focus of maintenance phase is change which cause is correction of errors, adaption of new idea, According to the needs of software after change in customer mood.

***9)* software development phase distribution**

Effort distribution by phase or activity is an important but often overlooked aspect compared to other steps in the cost estimation process. Poor effort allocation is among the major root causes of rework due to insufficiently resourced early activities. This paper provides results of an empirical study on phase effort distribution data of 75 industry projects, from the China Software Benchmarking Standard Group (CSBSG) database. The phase effort distribution patterns and variation sources are presented, and analysis results show some consistency in effects of software size and team size on code and test phase distribution variations, and some considerable deviations in requirements, design, and transition phases, compared with recommendations in the COCOMO model. Finally, this paper discusses the major findings and threats to validity and presents general guidelines in directing effort allocation. Empirical findings from this study are beneficial for stimulating discussions and debates to improve cost estimation and benchmarking practices.

**

**10) Cost-Effectiveness Model:**

Cost effectiveness analysis (CEA) models are a means to predict the costs and effectiveness of an intervention based on the best available evidence. They provide a robust analytical approach for policy makers and regulators to assess if the cost associated with an intervention is worth paying. We principally use Microsoft® Excel®, but are also adept in the use of other industry standard software packages as required. Our team is led by senior modellers, and supported by specialist statistics professionals, who can help you decide on the solution for your particular modelling needs.

## Unique tailored solutions

There is no ‘one size fits all’ approach from our modelling teams, with each project’s unique requirements being investigated by the modelling team and supported by techniques built up over 20 years of modelling; these include:

* Task conceptualization
* Model scoping and storyboarding
* Model engine development
* Data population
* Testing and report generation

We always recommend that clients include clinical Key Opinion Leader (KOL) input to ensure that the resulting models accurately reflect the decision problem. Furthermore, all stages of the modelling process have integrated client review and sign-off, validation of the model by internal and, when appropriate, external modellers.