**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.



***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.[*[citation needed](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed" \o "Wikipedia:Citation needed)*] 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).

***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, whose characteristics determine the value of constant to be used in subsequent calculations. These characteristics pertaining to different system types are mentioned below.

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

The first level, **Basic COCOMO** can be used for quick and slightly rough calculations of Software Costs. Its accuracy is somewhat restricted due to the absence of sufficient factor considerations.

**Intermediate COCOMO**takes these Cost Drivers into account and **Detailed COCOMO** additionally accounts for the influence of individual project phases, i.e in case of Detailed it accounts for both these cost drivers and also calculations are performed phase wise henceforth producing a more accurate result. These two models are further discussed below.

**Estimation of Effort: Calculations –**

* 1. **Basic Model –**

The above formula is used for the cost estimation of for the basic COCOMO model, and also is used in the subsequent models. The constant values a,b,c and d for the Basic Model for the different categories of system:

| **SOFTWARE PROJECTS** | **A** | **B** | **C** | **D** |
| --- | --- | --- | --- | --- |
| Organic | 2.4 | 1.05 | 2.5 | 0.38 |
| Semi Detached | 3.0 | 1.12 | 2.5 | 0.35 |
| Embedded | 3.6 | 1.20 | 2.5 | 0.32 |

The effort is measured in Person-Months and as evident from the formula is dependent on Kilo-Lines of code.  
The development time is measured in Months.

These formulas are used as such in the Basic Model calculations, as not much consideration of different factors such as reliability, expertise is taken into account, henceforth the estimate is rough.

* 1. **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
    - Required turnabout time

**(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

;

| **COST DRIVERS** | **VERY LOW** | **LOW** | **NOMINAL** | **HIGH** | **VERY HIGH** |
| --- | --- | --- | --- | --- | --- |
| **Product Attributes** |  |  |  |  |  |
| Required Software Reliability | 0.75 | 0.88 | 1.00 | 1.15 | 1.40 |
| Size of Application Database |  | 0.94 | 1.00 | 1.08 | 1.16 |
| Complexity of The Product | 0.70 | 0.85 | 1.00 | 1.15 | 1.30 |
| **Hardware Attributes** |  |  |  |  |  |
| Runtime Performance Constraints |  |  | 1.00 | 1.11 | 1.30 |
| Memory Constraints |  |  | 1.00 | 1.06 | 1.21 |
| Volatility of the virtual machine environment |  | 0.87 | 1.00 | 1.15 | 1.30 |
| Required turnabout time |  | 0.94 | 1.00 | 1.07 | 1.15 |
| **Personnel attributes** |  |  |  |  |  |
| Analyst capability | 1.46 | 1.19 | 1.00 | 0.86 | 0.71 |
| Applications experience | 1.29 | 1.13 | 1.00 | 0.91 | 0.82 |
| Software engineer capability | 1.42 | 1.17 | 1.00 | 0.86 | 0.70 |
| Virtual machine experience | 1.21 | 1.10 | 1.00 | 0.90 |  |
| Programming language experience | 1.14 | 1.07 | 1.00 | 0.95 |  |
| **Project Attributes** |  |  |  |  |  |
| Application of software engineering methods | 1.24 | 1.10 | 1.00 | 0.91 | 0.82 |
| Use of software tools | 1.24 | 1.10 | 1.00 | 0.91 | 0.83 |
| Required development schedule | 1.23 | 1.08 | 1.00 | 1.04 | 1.10 |

The project manager is to rate these 15 different parameters for a particular project on a scale of one to three. Then, depending on these ratings, appropriate cost driver values are taken from the above table. These 15 values are then multiplied to calculate the EAF (Effort Adjustment Factor). The Intermediate COCOMO formula now takes the form:

The values of a and b in case of the intermediate model are as follows:

| **SOFTWARE PROJECTS** | **A** | **B** |
| --- | --- | --- |
| Organic | 3.2 | 1.05 |
| Semi Detached | 3.0 | 1.12 |
| Embeddedc | 2.8 | 1.20 |

* 1. **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

The effort is calculated as a function of program size and a set of cost drivers are given according to each phase of the software lifecycle.

Also read: [Classical Waterfall Model](https://www.geeksforgeeks.org/software-engineering-classical-waterfall-model/), [Iterative Waterfall Model](https://www.geeksforgeeks.org/software-engineering-iterative-waterfall-model/), [Prototyping Model](https://www.geeksforgeeks.org/software-engineering-prototyping-model/), [Spiral Model](https://www.geeksforgeeks.org/software-engineering-spiral-model/)

Attention reader! Don’t stop learning now. Get hold of all the important DSA concepts with the [**DSA Self Paced Course**](https://practice.geeksforgeeks.org/courses/dsa-self-paced?utm_source=geeksforgeeks&utm_medium=article&utm_campaign=gfg_article_dsa_content_bottom) at a student-friendly price and become industry ready.

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” (1).

With its origins in Industrial Revolution, Human Factors became widely incorporated discipline during the World War II. Many giant companies came to recognize that the success of a product depends upon a solid Human Factors design and Human Computer Interface (HCI) design, which “is a sub-discipline concerned with the specification, design, evaluation/testing and implementation of interaction computing systems for human use and with the study of major phenomena surrounding them” (2).

Since then, the design and development of HCI has been rapidly developing into a full-fledged engineering field for achieving proper system usability. The particular field aims in providing users with a cost effective and satisfactory way of software development. HCI continues evolve into a discipline, which has its own defined and managed processes. In turn, individuals from various fields who utilize a varied range of methods to develop user-friendly software techniques would use these processes.

Study of Human Factors is essential for every software manager since he/she must be acquainted with how his/her staff members interact with each other. Generally, software products are used by a variety of populace and it is necessary to take into account the abilities and precincts of such a group to make the software more useful and popular.

**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
* Decreased user fatigue
* Decreased incidence of cumulative disturbance
* Decrement of operating costs
* Limited dependence on operation manuals
* Lesser operational errors
* 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**

A systematic approach is required in the design process in Human Factors Design and, thus, Usability is required. As America’s Former Vice-President Al Gore once stated, “American industry and government will become even more productive if they take advantage of usability engineering techniques. (3)” Dedication to usability in the software development cycle can elevate sales and user satisfaction.

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

**Some Factoids about Usability**

* A common notion that the software developers have is that HCI Engineering adds only to the development time and cost but does not really yield any satisfactory results. However, according to the recent research has shown that the Usability engineering has presented reductions in the product-development cycle by over 33-50%. Moreover, 63% of all software projects overran their estimates, with the top four reasons relating to usability.
* A hard and fast rule that software companies are following now a days after realizing the importance of HCI and Usability is that “user should be involved before and not after development” (3).
* During the holiday season, Creative Goods tested ten commerce sites in 1999. It found that 39 percent of the shoppers failed in their shopping attempts because the sites were too difficult to use. When using the e-commerce site’s search engine, 56 percent of the search attempts failed. $3 billion profits would have been made even if only 25 percent of these search attempts were successful. (3)
* A Human Resource Department of a company complained of disused data entry screens for processing a job application. Human resources and usability experts guesstimate that cutting processing time in 25% if one screen was used. It takes about four hours to process one application, which costs around one hundred dollars. If the company receives around a thousand applications each year, it would take one hundred thousand dollars to process these applications. However, the company can save twenty five thousand dollars if one screen approach is to be implemented since it will cost only two thousand four hundred to implement the system. (3)

**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)