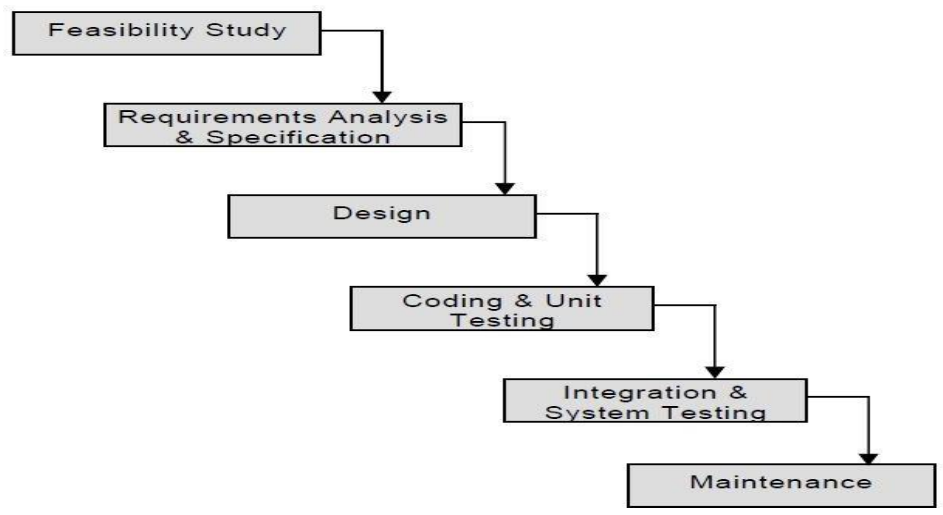
Waterfall Model

- The **Waterfall model** is a simple process model where phases are organized in a linear sequence.
- Although intuitive, the **classical waterfall model** is often impractical for real software development projects.
- Many other lifecycle models are derived from the **waterfall model**, making it essential to understand its phases.
- The **waterfall model** divides the software lifecycle into distinct phases.


Feasibility Study
Requirements Analysis & Specification
Design
Coding & Unit Testing
Integration & System Testing
Maintenance

Feasibility Study

- The main aim of feasibility study is to determine whether it would be financially and technically feasible to develop the product.

Requirements Analysis and Specification

The **Requirements Analysis and Specification** phase aims to fully understand and document what the customer wants. It has two main steps:

1. **Requirements gathering and analysis**: Collect all important information from the customer to understand their needs and remove any missing or unclear details.
2. **Requirements specification**: Organize this information into a formal document called the Software Requirements Specification (SRS).

For example, if developing accounting software, the analyst would interview accountants to understand their needs. Since different users may have incomplete or conflicting views, these issues need to be identified and resolved through further discussions. The SRS document will include:

- **Functional requirements** (what the software should do),
- **Non-functional requirements** (performance, security, etc.),
- **Goals of implementation** (objectives for building the software).

Design

The **design phase** focuses on turning the requirements from the SRS document into a structure that can be implemented in a programming language.

There are two main approaches:

1. **Traditional design approach**:
    - First, a detailed analysis of the problem is done (structured analysis).
    - Then, the results are used to create the software design (structured design).
2. **Object-oriented design approach**:
    - This approach identifies objects in the problem and solution.
    - The relationships between these objects are mapped out, and the design is refined for implementation.

Coding and Unit Testing

- **Coding**: Each part of the design is written as a separate piece of code, or module.
- **Unit Testing**: Each module is tested individually to ensure it works correctly. This helps find and fix errors early by testing each module on its own.

Integration and System Testing

Integration Testing:

- This phase combines different modules of the software after they've been individually coded and unit tested.
- Integration is done incrementally, meaning modules are added and tested step-by-step rather than all at once.

System Testing:

- The goal is to verify that the entire system meets the requirements specified in the Software Requirements Specification (SRS) document.
- **Alpha Testing**: Conducted by the development team.
- **Beta Testing**: Performed by a select group of friendly customers.
- **Acceptance Testing**: Done by the customer to decide if the final product is acceptable.

Maintenance

Maintaining software often takes more effort than developing it. Research shows that maintenance usually requires about 60% of the total effort, compared to 40% for development. Maintenance includes:

1. **Corrective Maintenance**: Fixing errors that weren't found during development.

2. **Perfective Maintenance**: Improving and adding new features based on customer feedback.
3. **Adaptive Maintenance**: Adjusting the software to work in new environments, such as different operating systems or hardware.

Limitations of Waterfall Model

- **Fixed Requirements**: It assumes you can set all requirements before starting design. For small projects, this might work, but for large projects, it's tough because requirements can change as users learn more.
- **Hardware Issues**: It often requires choosing hardware early, which can become outdated by the time the project is finished, especially if the project takes several years.
- **Big Bang Delivery**: The entire software is delivered at the end, which is risky. If something goes wrong or funding runs out, you might end up with nothing.
- **Requirements Creep**: It can lead to adding extra requirements at the beginning that may not be necessary, as everything has to be specified upfront.
- **Heavy Documentation**: It relies heavily on formal documentation at the end of each phase, which can be cumbersome.

When to use the Waterfall Model?

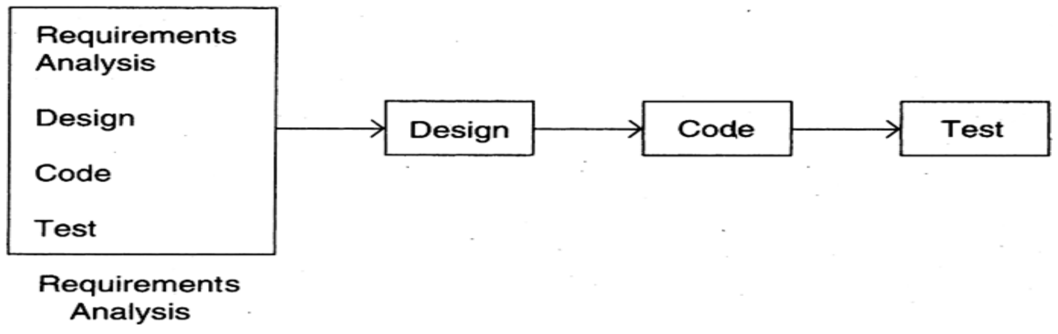
- Requirements are well known and stable
- Technology is understood
- Development team have experience with similar projects
- For small projects

Prototyping

The prototyping approach addresses the limitation of the Waterfall model by not requiring complete requirements before starting design and coding.

How It Works:

- **Prototype Creation**: Build a basic version of the system (a prototype) with the current understanding of requirements. This prototype has limited functionality and is developed informally.
- **Client Feedback**: Present the prototype to clients/users. Their feedback helps refine the requirements.
- **Iteration**: Modify the prototype based on feedback until the client is satisfied.
- **Final Development**: Once the prototype meets the client's needs, proceed with the actual design, coding, and testing of the final system.



Advantages:

- **Early Feedback**: Provides a working model early, increasing user confidence.
- **Better Requirements**: Helps clarify unclear requirements and improves communication.
- **User Involvement**: Ensures the final product better meets user needs.
- **Risk Reduction**: Helps identify issues early, reducing overall project risks.

Disadvantages:

- **Time and Cost**: Developing and refining prototypes can be time-consuming and expensive.
- **Quality Concerns**: Developers might focus too much on the prototype's features and less on the quality of the final system.
- **False Expectations**: Clients might mistakenly believe that the system is nearly complete when it is not.

Where to Use Prototype Model ?

- Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help determine the requirements.
- This might be needed for novel systems.

Iterative Development

The iterative development process model counters the third and fourth limitations of the waterfall model and tries to combine the benefits of both prototyping and the waterfall model.

Concept: Develop software in small, manageable parts (increments). Each part adds new functionality until the full system is completed.

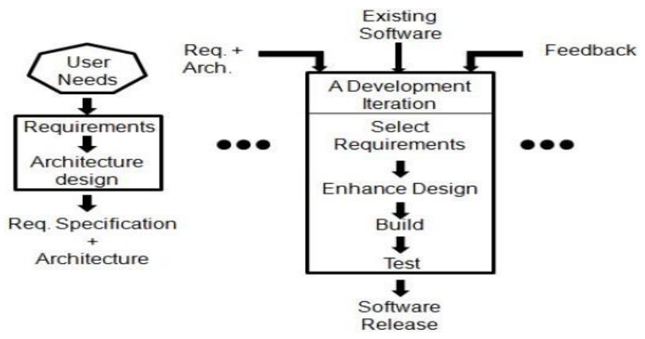
Approaches:

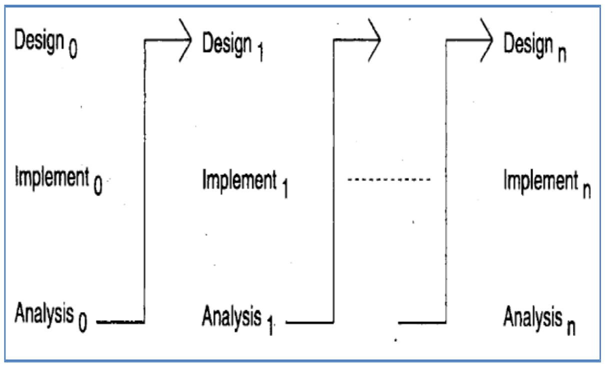
Iterative Enhancement Model:

- Start with a basic version of the system that includes essential features.
- Create a project control list with tasks needed to complete the final system.
- For each task, design, code, test, and analyze the results.
- Repeat until all tasks are completed and the final system is ready.

Iterative Delivery Approach:

- Begin with initial requirements and architecture design.
- Deliver software in phases, adding new features in each iteration.
- Each phase ends with delivering a working version of the software.
- Continue this process until the final system is ready.





Advantages:

- Early Returns: Provides working software after each iteration, allowing clients to see progress and reducing investment risk.
- Reduced Rework: Frequent feedback helps in making necessary adjustments early, minimizing rework.
- Reduced Risk: Avoids the “all-or-nothing” risk by delivering software in stages.
- Flexibility: Easier to incorporate new requirements as development progresses.

Disadvantages:

- Requires Good Planning: Effective planning and design are essential for success.
- Time Constraints: May not be suitable for projects with very tight deadlines.

Rational Unified Process

RUP is an iterative development model designed for object-oriented development, divided into four phases:

- Inception phase
- Elaboration phase
- Construction phase
- Transition phase

Inception Phase:

- **Purpose:** Define the project’s scope and goals.
- **Tasks:**
  - Understand what the software will and won’t do.

- Identify critical use cases and propose a candidate architecture.
- Estimate project cost, schedule, and risks.

Elaboration Phase:

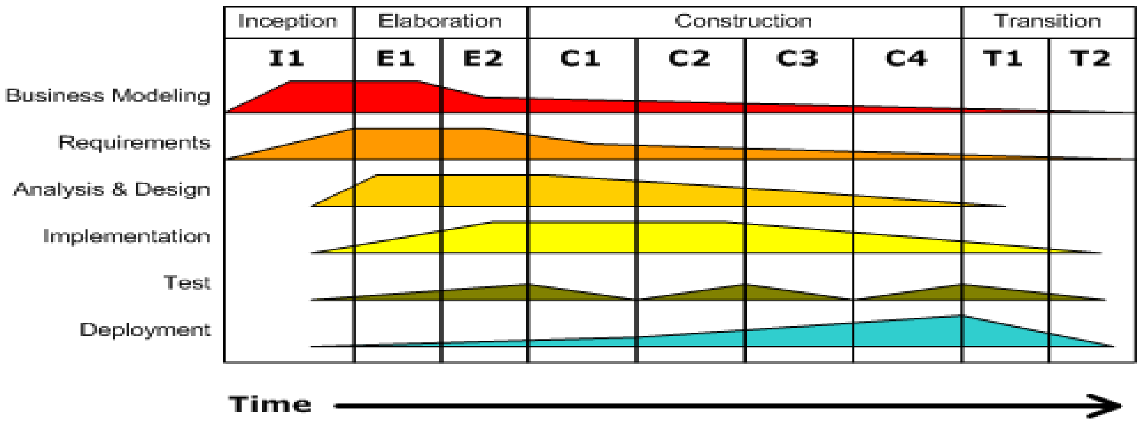
- **Purpose:** Refine the project plans and architecture.
- **Tasks:**
  - Finalize the system architecture.
  - Confirm the project vision and plan for construction.
  - Ensure the architecture supports the vision within budget and time constraints.

Construction Phase:

- **Purpose:** Build the software system.
- **Tasks:**
  - Optimize development to reduce costs and avoid rework.
  - Ensure quality and produce usable versions (e.g., alpha, beta releases).

Transition Phase:

- **Purpose**: Deploy the software to users.
- **Tasks:**
  - Conduct beta testing to validate the system.
  - Install the software at client sites.
  - Convert operational databases and train users.



Extreme Programming and Agile Process

Agile Process:

- **Purpose**: Address the limitations of the Waterfall model by adapting to changes easily.

- **Key Principles**:
    - **Working Software**: The main measure of progress.
    - **Small Increments**: Develop and deliver software in small, rapid updates.
    - **Adaptability**: Accept changes in requirements even late in the process.
    - **Communication**: Face-to-face interactions are preferred over extensive documentation.
    - **Customer Involvement**: Continuous feedback from customers is crucial for quality.
    - **Simple Design**: Start with a simple design and evolve it over time.
    - **Empowered Teams**: Teams decide on delivery dates based on their capabilities.

Popular Agile Methodologies:

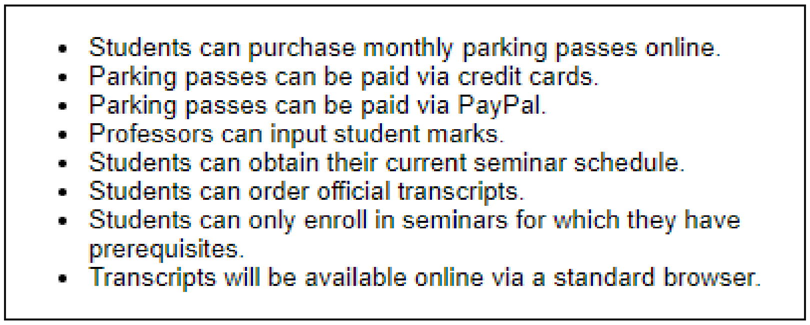
- Extreme Programming (XP)
- Scrum
- Unified Process
- Crystal
- DSDM
- Lean

Extreme programming (XP)

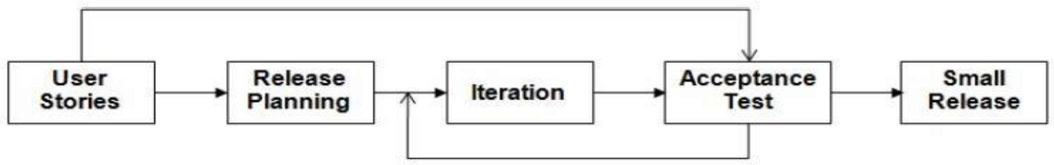
extreme Programming (XP) focuses on frequent, small releases and flexibility in development. Here's a simplified look at how it works:

User Stories:

- **Definition**: Short, simple descriptions of what the software should do, unlike detailed traditional requirements.



- Students can purchase monthly parking passes online.
- Parking passes can be paid via credit cards.
- Parking passes can be paid via PayPal.
- Professors can input student marks.
- Students can obtain their current seminar schedule.
- Students can order official transcripts.
- Students can only enroll in seminars for which they have prerequisites.
- Transcripts will be available online via a standard browser.



Estimation:

- **Process**: The development team estimates how long it will take to complete each user story, usually between one to four weeks.

Release Planning:

- **Planning**: Based on estimates, decide which user stories will be included in each system release and when these releases will happen.
- **Small Releases**: Encourages frequent, small updates.

Iterations:

- **Process**: Development is done in iterations, each focusing on one user story.
- **Planning**: At the start of each iteration, select the user stories to work on.
- **Development**: Develop and test the selected user story, often using acceptance tests to ensure it meets requirements.
- **Release**: After each iteration, a new version of the software is released, even if it doesn't add major new features.

Final Release:

- After several iterations, the complete software is released.

Advantages:

- **Customer Satisfaction:** Rapid and continuous delivery of useful software keeps customers happy.
- **Frequent Interaction:** Regular communication between customers, developers, and testers.
- **Frequent Releases:** Software is delivered in weeks, not months.
- **Face-to-Face Communication:** Encourages direct conversation rather than relying on documentation.
- **Close Cooperation:** Daily collaboration between business and development teams.
- **Technical Excellence:** Focuses on high-quality design and technical practices.
- **Adaptability:** Easily adapts to changing requirements, even late in development.

Disadvantages:

- **Misinterpretation:** Can be misunderstood or applied incorrectly.
- **External Review:** Hard to get outside feedback or review.
- **Maintenance Challenges**: Once the project is finished and the team disbands, maintaining the software can be difficult.

When to Use the Agile Process Model

- **Frequent Changes:** Ideal for projects where requirements change often and rapidly.
- **High Risk**: Suitable for situations with significant risks related to requirements.
- **Customer Involvement:** Works best when the customer is actively involved throughout development, collaborating closely with the team.

Basic COCOMO Model

Development Effort (E) = a x (KLOC)^b PM

Development Time (T) = c x (E)^d Months

Project Category	a	b	c	d
Organic	3.2	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

Problems

- **Assume that a system for simple student registration in a course is planned to be developed and its estimated size is approximately 10,000 LOC. The organization proposed to pay 25,000 per month to s/w engineers. Compute the development effort, development time, and the total cost for product development.**

The project can be considered an organic project.

From basic COCOMO Model

Development effort (E)=2.4*$(10)^{1.05}$=26.93PM

Development effort (T)=2.5*$(26.93)^{0.38}$=8.74months

Total product development cost=

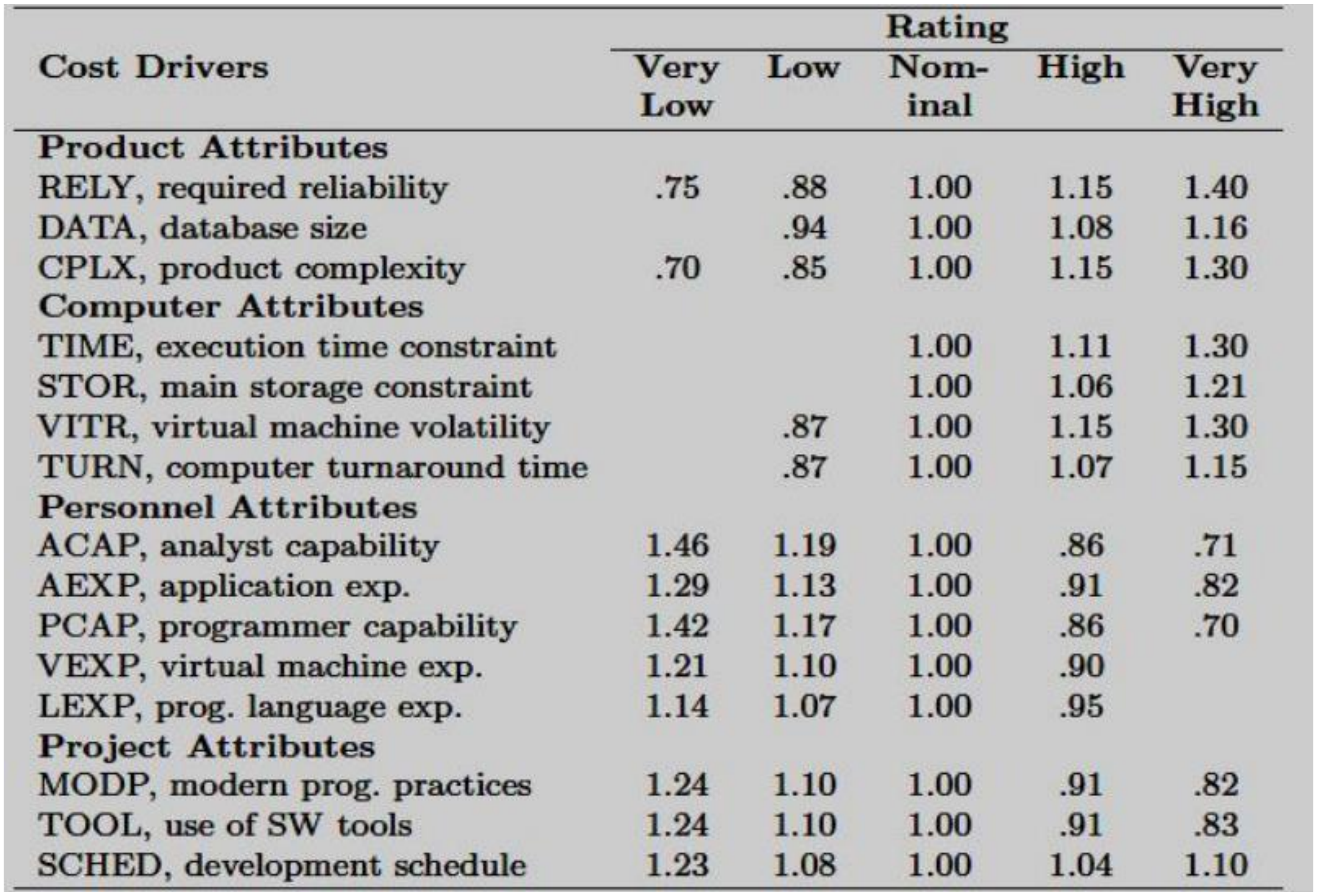
Development time *salaries of engineers

=8.74*25000

=2,18500

- **Assume that the size of an organic type software product has been estimated to be 32,000 lines of source code. Assume that the average salary of software engineers be Rs. 15,000/- per month. Determine the effort required to develop the software product and the nominal development time.**

Intermediate COCOMO



Cost Drivers	Rating				
	Very Low	Low	Nom-inal	High	Very High
Product Attributes					
RELY, required reliability	.75	.88	1.00	1.15	1.40
DATA, database size		.94	1.00	1.08	1.16
CPLX, product complexity	.70	.85	1.00	1.15	1.30
Computer Attributes					
TIME, execution time constraint			1.00	1.11	1.30
STOR, main storage constraint			1.00	1.06	1.21
VITR, virtual machine volatility		.87	1.00	1.15	1.30
TURN, computer turnaround time		.87	1.00	1.07	1.15
Personnel Attributes					
ACAP, analyst capability	1.46	1.19	1.00	.86	.71
AEXP, application exp.	1.29	1.13	1.00	.91	.82
PCAP, programmer capability	1.42	1.17	1.00	.86	.70
VEXP, virtual machine exp.	1.21	1.10	1.00	.90	
LEXP, prog. language exp.	1.14	1.07	1.00	.95	
Project Attributes					
MODP, modern prog. practices	1.24	1.10	1.00	.91	.82
TOOL, use of SW tools	1.24	1.10	1.00	.91	.83
SCHED, development schedule	1.23	1.08	1.00	1.04	1.10

Development effort (E):

Initial effort (Ei) = a × (KLOC) b

Effort Adjustment Factor (EAF) = EAF1× EAF2×... × EAFn

Total development effort (E) = Ei× EAF

Development time (T) = c × (E)d

Suppose a lib management system (LMS) is to be designed for an academic institution. From the project proposal, the following five major components are identified:

online data entry	**-**	**1.0 kloc**
data update	**-**	**2.0 kloc**
file input and output	**-**	**1.5 kloc**
library reports	**-**	**2.0 kloc**
query and search	**-**	**0.5 kloc**

Solution

The LMS project can be considered an organic category project. The total size of the modules is 7 KLOC. The development effort and development time can be calculated as follows:

Development effort

Initial effort (E_i) $= 3.2 \times (7)^{1.05}$

$= 24.6889$ PM

Effort Adjustment Factor (EAF)

$= 1.16 \times 0.82 \times 0.91 \times 1.06 \times 1.10 \times 0.87 = 0.8780$

Total effort (E) $= E_i * EAF$

$= 24.6889 \times 0.8780$

$= 21.6785$ PM

Development time (T) $= 2.5 \times (E)^{0.38}$ Months

$= 2.5\,(21.6785)^{0.38}$ months

$= 8.0469$ months

Detailed COCOMO Model

Compute the phase-wise development effort for the problem discussed in

Example 3.2 above

Solution

There are **five** components in the **organic project** discussed in Example above:

- ✓ Online Data Entry,
- ✓ Data Update,
- ✓ File Input and Output,
- ✓ Library Reports,
- ✓ Query and Search.

The **estimated effort (E)** is 21.6785 PM. The total size is **7** KLOC, which is between **2 KLOC** and **32 KLOC**. Thus, the **actual percentage** of effort can be calculated as follows:

H.KLOC DV + [(L.KLOC DV - H.KLOC DV) / (H.KLOC - L.KLOC)]*Size

Plan and Requirement (%) $= 6 + (6 - 6) / (32 - 2) \times 7 = 6\%$

Effort $= 0.06 \times 21.6785$ PM $= 1.30071$ PM

System Design $= 16 + (16 - 16) / (32 - 2) \times 7 = 16\%$

Effort $= 0.16 \times 21.6785$ PM $= 3.46856$ PM

Detailed Design $= 24 + (26 - 24) / (32 - 2) \times 7 = 25\%$

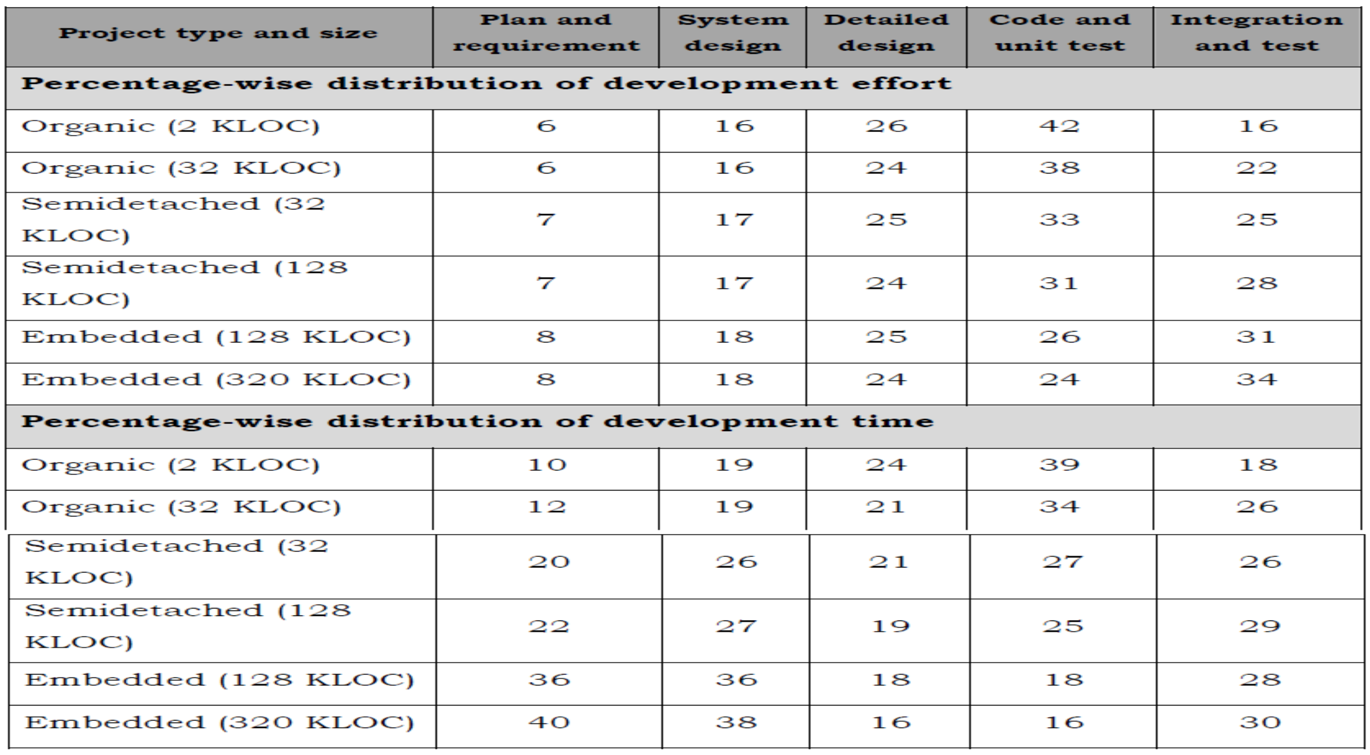
Effort $= 0.25 \times 21.6785$ PM $= 5.419625$ PM

Code and Unit Test $= 38 + (42 - 38) / (32 - 2) \times 7 = 39\%$

Effort $= 0.39 \times 21.6785$ PM $= 8.454615$ PM

Integration Test $= 22 + (16 - 22) / (32 - 2) \times 7 = 24\%$

Effort $= 0.24 \times 21.6785$ PM $= 5.20284$ PM



Project type and size	Plan and requirement	System design	Detailed design	Code and unit test	Integration and test
Percentage-wise distribution of development effort					
Organic (2 KLOC)	6	16	26	42	16
Organic (32 KLOC)	6	16	24	38	22
Semidetached (32 KLOC)	7	17	25	33	25
Semidetached (128 KLOC)	7	17	24	31	28
Embedded (128 KLOC)	8	18	25	26	31
Embedded (320 KLOC)	8	18	24	24	34
Percentage-wise distribution of development time					
Organic (2 KLOC)	10	19	24	39	18
Organic (32 KLOC)	12	19	21	34	26
Semidetached (32 KLOC)	20	26	21	27	26
Semidetached (128 KLOC)	22	27	19	25	29
Embedded (128 KLOC)	36	36	18	18	28
Embedded (320 KLOC)	40	38	16	16	30

Suppose a project was estimated to be 400 KLOC. Calculate the effort , development time and average staff size for each of the three model i.e., organic, semi-detached & embedded.

Solution: The basic COCOMO equation takes the form:

Effort=a_1*(KLOC) ^{a2} PM

Tdev=b_1*(efforts)^{b2} Months

Average staff size=Effort/Tdev

Estimated Size of project= 400 KLOC

(i)Organic Mode

Effort = 2.4 * (400)$^{1.05}$ = 1295.31 PM

Tdev = 2.5 * (1295.31)$^{0.38}$=38.07 Months

Average staff size=Effort/Tdev= 1295.31/38.07=34.03 Persons

(ii)Semidetached Mode

Effort = 3.0 * (400)$^{1.12}$=2462.79 PM

Tdev = 2.5 * (2462.79)$^{0.35}$=38.45 Months

Average staff size=Effort/Tdev= 2462.79/38.45=64.05 Persons

(iii) Embedded Mode

Effort = 3.6 * (400)$^{1.20}$ = 4772.81 PM

Tdev = 2.5 * (4772.8)$^{0.32}$ = 38 Months

Average staff size=Effort/Tdev= 4772.81 /38=125.60 Persons

A project size of 200 KLOC is to be developed. Software development team has average experience on similar type of projects. The project schedule is not very tight. Calculate the Effort, development time, average staff size, and productivity of the project.

Solution: Software development team has average experience on similar type of projects. The project schedule is not very tight. So this is semidetached type of project.

Effort=a_1*(KLOC) ^{a2} PM
Tdev=b_1*(efforts)^{b2} Months

Average staff size=Effort/Tdev

Productivity=KLOC/Effort

- Effort=3.0(200)$^{1.12}$=1133.12PM
  Tdev=2.5(1133.12)$^{0.35}$=29.3 Months
- Productivity=KLOC/Effort

  =200/1133.12

  =0.176 KLOC/PM

  =176 LOC/PM

ANOTHER PPT

Suppose that a project was estimated to be 400 KLOC. Calculate the effort and development time for each of the three modes i.e., organic, semidetached and embedded.

<u>Solution</u>

The basic COCOMO equation take the form:

$E = a_b (KLOC)^\wedge b_b$

$D = c(KLOC)^{db}$

Estimated size of the project = 400 KLOC

(i) Organic mode

E = 2.4(400)$^{1.05}$ = 1295.31 PM

D = 2.5(1295.31)$^{0.38}$ = 38.07 PM

(ii) Semidetached mode

E = 3.0(400)$^{1.12}$ = 2462.79 PM

D = 2.5(2462.79)$^{0.35}$ = 38.45 PM

(iii) Embedded mode

E = 3.6(400)$^{1.20}$ = 4772.81 PM

$D = 2.5(4772.8)^{0.32} = 38$ PM

A project size of 200 KLOC is to be developed. Software development team has average experience on similar type of projects. The project schedule is not very tight. Calculate the effort, development time, average staff size and productivity of the project.

The semi-detached mode is the most appropriate mode; keeping in view the size, schedule and experience of the development team.

$E = 3.0(200)^{1.12} = 1133.12$ PM

$D = 2.5(1133.12)^{0.35} = 29.3$ PM

$$\text{Average staff size } (SS) = \frac{E}{D}\, Persons$$

$$= \frac{1133.12}{29.3} = 38.67\, Persons$$

$$\text{Productivity } = \frac{KLOC}{E} = \frac{200}{1133.12} = 0.1765\, KLOC / PM$$

$$P = 176\ LOC / PM$$

Example: 4.7

A new project with estimated 400 KLOC embedded system has to be developed. Project manager has a choice of hiring from two pools of developers: Very highly capable with very little experience in the programming language being used

Or

Developers of low quality but a lot of experience with the programming language. What is the impact of hiring all developers from one or the other pool ?

Solution

This is the case of embedded mode and model is intermediate COCOMO.

Hence $$E = a_i (KLOC)^{d_i}$$

$$= 2.8\ (400)^{1.20} = 3712 \text{ PM}$$

Case I: Developers are very highly capable with very little experience in the programming being used.

EAF $= 0.82 \times 1.14 = 0.9348$

E $= 3712 \times .9348 = 3470$ PM

D $= 2.5\ (3470)^{0.32} = 33.9$ M

Software Requirements Analysis and Specification

Value of a Good SRS

SRS as an Agreement:

It acts as a contract between the client (who describes what they need) and the developer (who builds the software).

The client uses it to outline expectations, while the developer understands what features to include.

SRS for Validation:

The SRS is a reference point to check if the final product meets the client's needs.

Without an SRS, the client can't verify if the software matches their requirements, and the developer can't prove that all requirements are met.

Quality Impact:

A good SRS is crucial for high-quality software. It minimizes errors in the requirements phase, reducing the chances of changes and mistakes later.

Fixing errors becomes more expensive as the project progresses, so catching them early is important.

Cost Reduction:

A high-quality SRS lowers development costs by improving the clarity and quality of requirements upfront, reducing the need for costly fixes later.

Requirement Process

The Requirement Process is a series of steps followed during the requirements phase to create a high-quality Software Requirements Specification (SRS). It involves three main tasks:

Problem or Requirements Analysis: Understanding what the client needs and what problems the software will solve.

Requirements Specification: Writing down the detailed requirements for the software in a clear and organized way.

Requirements Validation: Checking to ensure that the requirements are correct and that they meet the client’s needs.

1. Problem or Requirements Analysis:

- Starts with a basic problem statement.
- The goal is to fully understand what the software needs to do.
- Analysts meet with clients and users to learn about their work, environment, and needs.

- Early meetings involve listening, collecting documents, and understanding how tasks are done.
- Later, the analyst asks for clarifications and checks with the client to confirm their understanding.

2. Requirements Specification:

- After analyzing the requirements, the next step is to clearly organize and write them down.
- Earlier, stories, scenarios, or Data Flow Diagrams (DFDs) were used; now, the use case approach is common.
- This phase also addresses the tools and languages used to document the requirements.

3. Requirements Validation:

- Ensures that the SRS includes all the necessary requirements and is of high quality.
- The requirements process ends with a well-prepared SRS.

Desirable Characteristics of an SRS:

- **Correct:** The SRS should list only the requirements that the software will meet. There's no guaranteed way to check for complete correctness.
- **Complete:** The SRS should describe everything the software will do, including its responses to all input types.
- **Unambiguous**: Each requirement should have only one clear meaning to avoid confusion.
- **Verifiable:** The requirements should be measurable so it's easy to check if the final product meets them.
- **Consisten**t: No requirements should contradict each other.
- **Ranked for Importance/Stability:** Requirements should be ranked by how important or stable they are. Not all requirements have the same priority.

Components of an SRS:

- **Functionality: Describes what the system will do, such as services it provides. For example, in a library system:**
    - Maintain books and member info
    - Search for and issue books
    - Return books
- **Performance: These are non-functional requirements, like how well the system performs. For example:**
    - Responds to queries in under 5 seconds
    - Zero downtime
    - Supports 100 remote users

- **Design Constraints: Limits on how the system will be built. For example:**
    - Uses standard web browsers
    - Built with Java
    - Uses open-source databases like PostgreSQL
- **External Interfaces: How the system interacts with users, hardware, and other software. Examples include:**
    - Graphical User Interface (GUI) screens
    - File formats for data input/output

Structure of an SRS Document:

Introduction:

Purpose: Explains the main objective of the system.

Scope: Describes what the system will and will not do.

Overview: A brief summary of the system.

Overall Description:

Product Perspective: How this system fits into other systems or environments.

Product Functions: The key functions the system will perform.

User Characteristics: Describes the users and their needs.

Assumptions: Any assumptions made during development.

Constraints: Any limitations or restrictions on the system.

Specific Requirements:

External Interfaces: How the system interacts with users, hardware, or other software.

Functional Requirements: What the system will do (e.g., services and features).

Performance Requirements: How well the system should perform (e.g., speed, reliability).

Design Constraints: Limits on how the system should be built (e.g., tools or languages to be used).

Acceptable Criteria: Defines what is considered acceptable for the system's performance and functionality.

This structure is standardized by IEEE to ensure consistency and clarity in SRS documents.