Unit 10: Project Cost and Time Estimation

AGENDA

- Process Metrics, Project metrics,
- Halstead's Software Science, Function Point(FP), Cyclomatic Complexity Measures;
- Software Project Estimation Models- Empirical, Putnam, COCOMO Estimating Size with Story Points, Velocity,
- Estimating Time: Ideal Days for Estimated Size.
- Techniques for Estimation: Estimates Shared, Estimation Scale, Derive Estimation, Planning Poker.

Introduction

- Effective software project management focuses on the four P’s: **people, product, process, and project.**
- **Product:** What is delivered to the customer is called Product. It may include source code, specification document, manuals, documentation etc. (Set of deliverable only)
- **Process:** is the way in which we produce software - collection of activities that lead to a product.
- An efficient process is required to produce good quality products. If the process is weak, the end product will suffer.
- Nowadays**, CMM** has become almost a standard for process framework.

Introduction

- **Project:** Planned and controlled software is the only known way to manage complexity.
- To avoid project failure, a software project manager and the software engineers must avoid a set of common warning signs, understand the critical success factors and develop a commonsense approach for planning, monitoring, and controlling the project.

Metrics in the Process and Project Domains

- **Process metrics** are collected across all projects and over long periods of time. Their intent is to provide a set of process indicators that lead to long-term software process improvement.
- **Project metrics** enable a software project manager to
- (1) Assess the status of an ongoing project,
- (2) Track potential risks,
- (3) Uncover problem areas before they go “critical,”
- (4) Adjust work flow or tasks, and
- (5) Evaluate the project team’s ability to control quality of software work products.

Process Metrics Vs Project Metrics

- Product metrics are measures of the software product at any stage of its development, from requirements to installed system.
- Product metrics may measure the complexity of the software design, the size of the final program, or the number of pages of documentation produced.
- Process metrics, on the other hand, are measures of the software development process, such as the overall development time, type of methodology used, or the average level of experience of the programming staff.

Software Project Estimation Models

- **Putnam Model**
- The Putnam model is an empirical software effort estimation model. As a group, empirical models work by collecting software project data (For example, effort and size) and fitting a curve to the data.
- Future efforts estimates are made by providing size and calculating the associated effort using the equation which fit the original data (usually with some error).

Halstead's Software Science

- Howard Halstead introduced metrics to measure software complexity.
- Halstead's metrics depends upon the actual implementation of program and its measures, which are computed directly from the operators and operands from source code, in static manner.
- It allows to evaluate testing time, vocabulary, size, difficulty, errors, and efforts for C/C++/Java source code.
- **According to Halstead,**
- **"A computer program is an implementation of an algorithm considered to be a collection of tokens which can be classified as either operators or operands".**
- Halstead metrics think a program as sequence of operators and their associated operands.

Halstead’s Software Science

- **Token Count**
- In these metrics, a computer program is considered to be a collection of tokens, which may be classified as either operators or operands.
- All software science metrics can be defined in terms of these basic symbols. These symbols are called as a token.
- The basic measures are:
  - n_1 = count of unique operators.
  - n_2 = count of unique operands.
  - N_1 = count of total occurrences of operators.
  - N_2 = count of total occurrence of operands.

Halstead’s Software Science

- In terms of the total tokens used, the size of the program can be expressed as $N = N_1 + N_2$.
- Halstead metrics are:
- **Program Volume (V)**
- The unit of measurement of volume is the standard unit for size "bits." It is the actual size of a program if a uniform binary encoding for the vocabulary is used.
    - $V=N*\log_2 n$
- **Program Level (L)**
- This is the ratio of the number of operator occurrences to the number of operand occurrences in the program,

  i.e., $L = n_1/n_2$ OR $1/D$
- The value of L ranges between zero and one, with $L=1$ representing a program written at the highest possible level (i.e., with minimum size).

Halstead's Software Science

- **Program Difficulty**
- The difficulty level or error-proneness (D) of the program is proportional to the number of the unique operator in the program.
    - $D= (n_1/2) * (N_2/n_2)$
- **Programming Effort (E)**
- The unit of measurement of E is elementary mental discriminations.
    - $E=V/L=D*V$
- **Estimated Program Length**
- According to Halstead, The first Hypothesis of software science is that the length of a well-structured program is a function only of the number of unique operators and operands.
    - $N=N_1+N_2$
- And estimated program length is denoted by N^

$$N\text{^} = n_1\log_2 n_1 + n_2\log_2 n_2$$

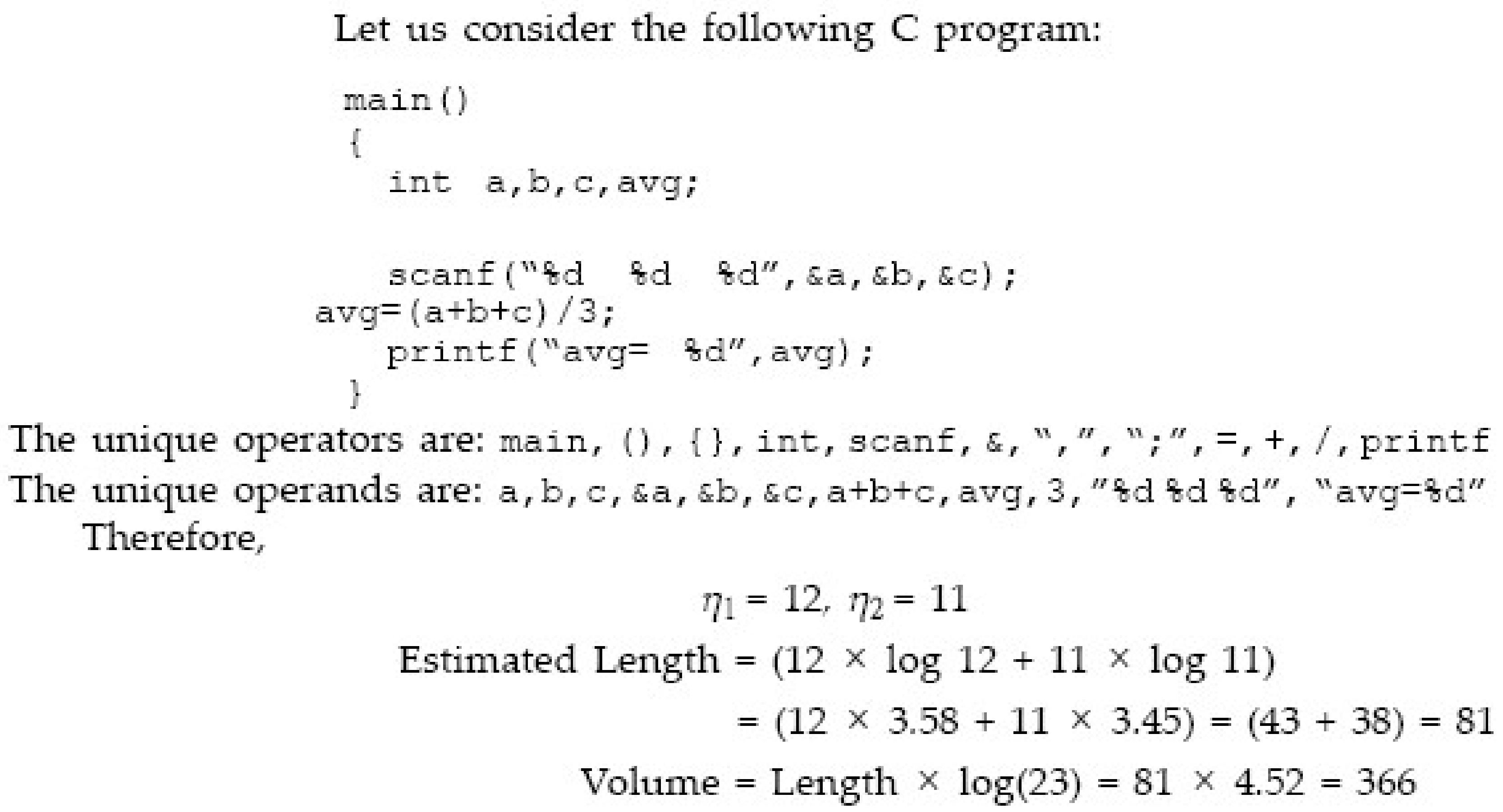
Halstead's Software Science

- **Size of Vocabulary (n)**
- The size of the vocabulary of a program, which consists of the number of unique tokens used to build a program, is defined as:

  $$\mathbf{n=n_1+n_2}$$
- Where,
- n=vocabulary of a program
- n_1=number of unique operators
- n_2=number of unique operands

Halstead’s Software Science

- **Example 1**



Let us consider the following C program:

```
main()
{
  int  a,b,c,avg;

  scanf("%d  %d  %d",&a,&b,&c);
avg=(a+b+c)/3;
  printf("avg=  %d",avg);
}
```

The unique operators are: `main, (), {}, int, scanf, &, ",", ";", =, +, /, printf`

The unique operands are: `a,b,c,&a,&b,&c,a+b+c,avg,3,"%d %d %d", "avg=%d"`

Therefore,

$$\eta_1 = 12,\ \eta_2 = 11$$

$$\text{Estimated Length} = (12 \times \log 12 + 11 \times \log 11)$$

$$= (12 \times 3.58 + 11 \times 3.45) = (43 + 38) = 81$$

$$\text{Volume} = \text{Length} \times \log(23) = 81 \times 4.52 = 366$$

Halstead's Software Science

- **Example 2**

```
void sort ( int *a, int n ) {
int i, j, t;

   if ( n < 2 ) return;
   for ( i=0 ; i < n-1; i++ )  {
          for ( j=i+1 ; j < n ; j++ ) {
                    if ( a[i] > a[j] ) {
                              t = a[i];
                              a[i] = a[j];
                              a[j] = t;
                    }
          }
   }
}
```

$V = 80 \log_2(24) \approx 392$

- Ignore the function definition
- Count operators and operands

3	<	3	{
5	=	3	}
1	>	1	+
1	-	2	++
2	,	2	for
9	;	2	if
4	(	1	int
4	)	1	return
6	[]		

1	0
2	1
1	2
6	a
8	i
7	j
3	n
3	t

	Total	Unique
Operators	N1 = 50	n1 = 17
Operands	N2 = 30	n2 = 7

Halstead’s Software Science

- **Example 3**

```
int sort (int x[ ], int n) {
int i, j, save, im1;
/*This function sorts array x in ascending order */
If (n< 2) return 1;
    for (i=2; i< =n; i++)  {
        im1=i-1;
        for (j=1; j< =im1; j++)
        if (x[i] < x[j]) {
            Save = x[i];
            x[i] = x[j];
            x[j] = save;
        }
    }
return 0;
}
```

Find Program volume, level, difficulty, effort and total length

Function Point

- Function-oriented metrics use a measure of the functionality delivered by the application as a normalization value.
- The most widely used function-oriented metric is the function point (FP).
- Computation of the function point is based on characteristics of the software’s information domain and complexity.
- Proponents claim that FP is programming language independent, making it ideal for applications using conventional and nonprocedural languages.
- It is believed to be based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach.

Function Point

- The size of a software product is directly dependent on the number of different high-level functions or features it supports.
- This assumption is reasonable, since each feature would take additional effort to implement.
- The size of a software product (in units of function points or FPs) is computed using different characteristics of the product identified in its requirements specification.
- It is computed using the following three steps:
  - **Step 1:** Compute the *unadjusted function point (UFP) using a heuristic expression.*
  - **Step 2:** Refine UFP to reflect the actual complexities of the different parameters used in UFP computation.
  - **Step 3:** Compute FP by further refining UFP to account for the specific characteristics of the project that can influence the entire development effort.

Function Point

- **Step 1: UFP computation**
- The unadjusted function points (UFP) is computed as the weighted sum of five characteristics of a product as shown in the following expression.
- The weights associated with the five characteristics were determined empirically by Albrecht through data gathered from many projects.

  UFP = (Number of inputs)*4 + (Number of outputs)*5 + (Number of inquiries)*4 + (Number of files)*10 + (Number of interfaces)*10
- **1. Number of inputs**: Each data item input by the user is counted which is different from user inquiries.
- Inquiries are user commands such as “print-account-balance” that require no data values to be input by the user.
- Inquiries are counted separately.
- Related inputs may be grouped and considered as a single input.

Function Point

- **2. Number of outputs**: include reports printed, screen outputs, error messages produced, etc.
- The individual data items within a report are not considered; but a set of related data items is counted as just a single output.
- **3. Number of inquiries**: An inquiry is a user command (without any data input) and only requires some actions to be performed by the system.
- Thus, the total number of inquiries is essentially the number of distinct interactive queries (without data input) which can be made by the users.
- Examples of such inquiries are print account balance, print all student grades, display rank holders' names, etc.

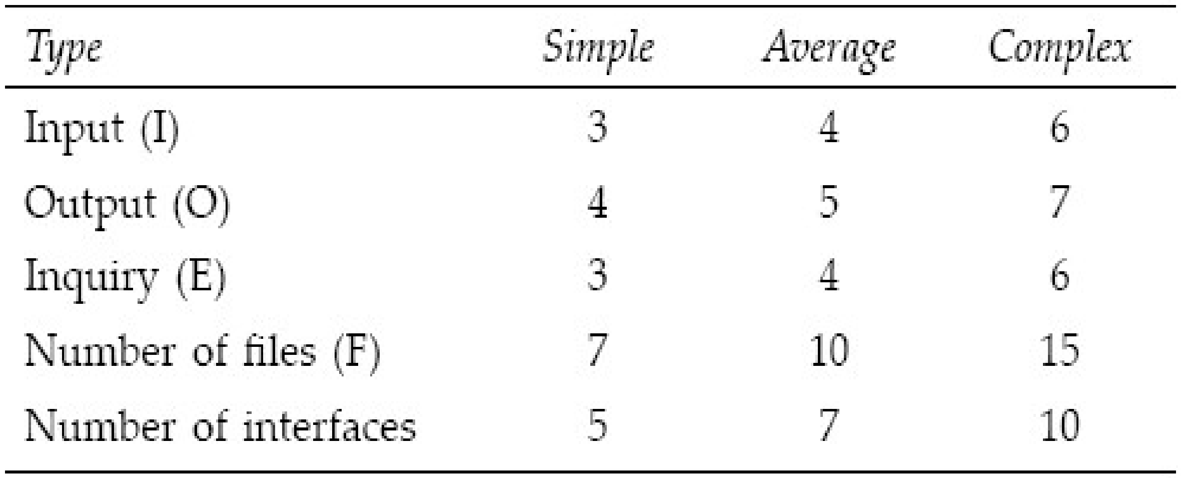
Function Point

- **4. Number of files**: The files referred to here are logical files, which is here a group of logically related data.
- Logical files include data structures as well as physical files.
- **5. Number of interfaces**: the different mechanisms that are used to exchange information with other systems.
- Examples of such interfaces are data files on tapes, disks, communication links with other systems, etc.
- **Step 2: Refine parameters**
- UFP computed at the end of step 1 is a gross indicator of the problem size and needs to be refined by considering various peculiarities of the project.

Function Point

- UFP is refined by taking into account the complexities of the parameters of UFP computation.
- The complexity of each parameter is graded into three broad categories—simple, average, or complex. The weights for are determined based on the numerical values shown in Table below.
- Based on these weights, the parameter values in the UFP are refined.
- For example, rather than each input being computed as four FPs, very simple inputs are computed as three FPs and very complex inputs as six FPs.

Refinement of Function Point Entities



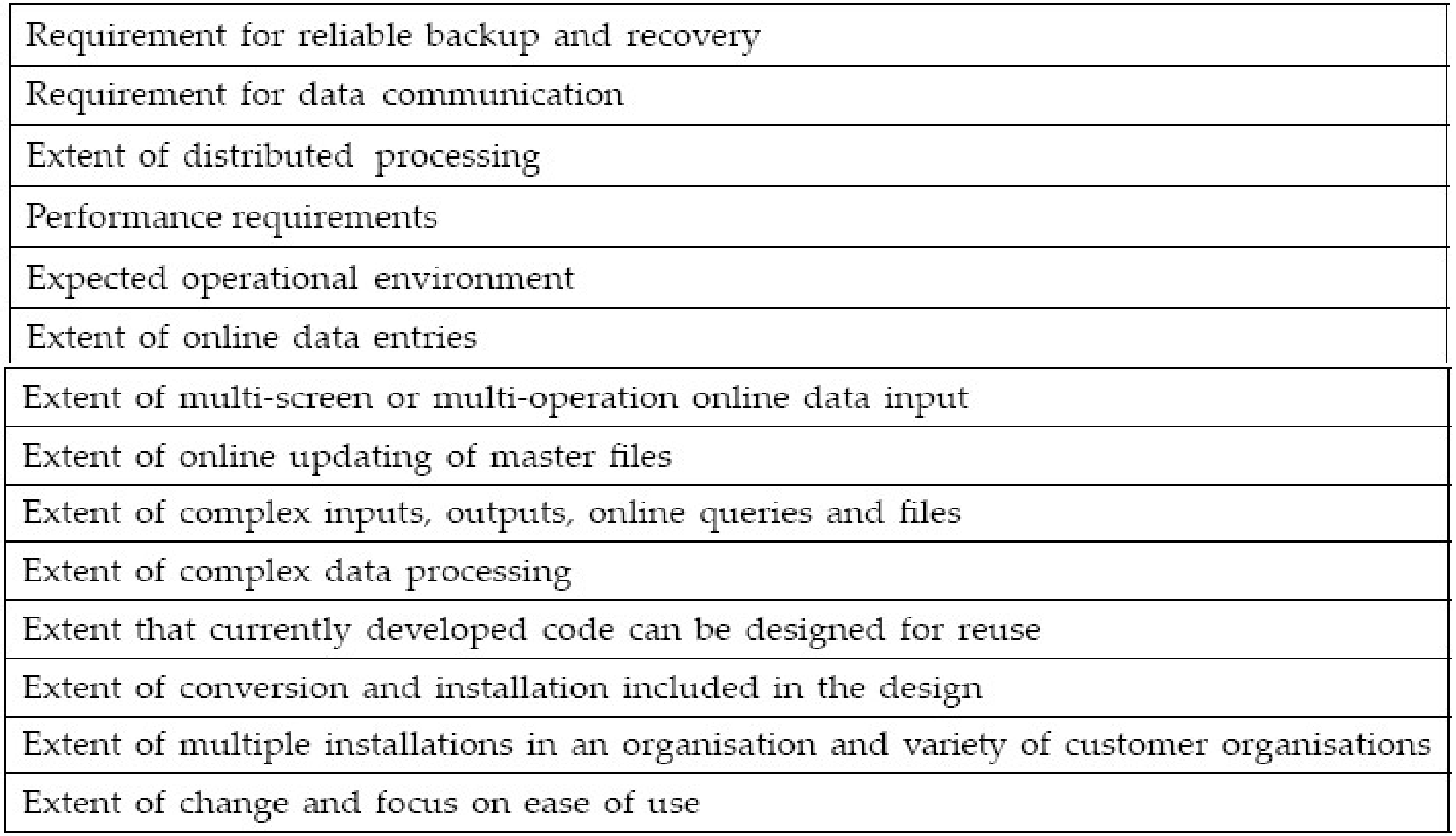
Type	*Simple*	*Average*	*Complex*
Input (I)	3	4	6
Output (O)	4	5	7
Inquiry (E)	3	4	6
Number of files (F)	7	10	15
Number of interfaces	5	7	10

Function Point

- **Step 3: Refine UFP based on complexity of the overall project**
- Several factors that can impact the overall project size are considered to refine the UFP computed in step 2.
- Examples of such project parameters that can influence the project sizes include high transaction rates, response time requirements, scope for reuse, etc.
- Each of these 14 parameters is assigned a value from 0 (not present or no influence) to 6 (strong influence).
- The resulting numbers are summed, yielding the total *degree of influence (DI).*
- *A technical complexity factor (TCF) for the project is computed and the TCF is multiplied with* UFP to yield FP.

Function Point

- The list of these 14 parameters have been shown in Table



Requirement for reliable backup and recovery
Requirement for data communication
Extent of distributed processing
Performance requirements
Expected operational environment
Extent of online data entries
Extent of multi-screen or multi-operation online data input
Extent of online updating of master files
Extent of complex inputs, outputs, online queries and files
Extent of complex data processing
Extent that currently developed code can be designed for reuse
Extent of conversion and installation included in the design
Extent of multiple installations in an organisation and variety of customer organisations
Extent of change and focus on ease of use

Function Point Relative Complexity Adjustment Factors

Function Point

- TCF expresses the overall impact of the corresponding project parameters on the development effort.
- TCF is computed as $(0.65 + 0.01 \times DI)$.
- As DI can vary from 0 to 84, TCF can vary from 0.65 to 1.49.
- Finally, FP is given as the product of UFP and TCF. That is,
    - $FP = UFP \times TCF$.

Example(Function Point) : Determine the function point measure of the size of the following supermarket software

- A supermarket needs to develop the following software to encourage regular customers. For this, the customer needs to supply his/her residence address, telephone number, and the driving license number.
- Each customer who registers for this scheme is assigned a unique customer number (CN) by the computer. Based on the generated CN, a clerk manually prepares a customer identity card after getting the market manager's signature on it.
- A customer can present his customer identity card to the check out staff when he makes any purchase. In this case, the value of his purchase is credited against his CN.
- At the end of each year, the supermarket intends to award surprise gifts to 10 customers who make the highest total purchase over the year.
- Also, it intends to award a 22 caret gold coin to every customer whose purchase exceeded `10,000. The entries against the CN are reset on the last day of every year after the prize winners' lists are generated.
- Assume that various project characteristics determining the complexity of software development to be average.

Cyclomatic Complexity Measures

- Every program encompasses statements to execute in order to perform some task and other decision-making statements that decide what statements need to be executed.
- These decision-making constructs change the flow of the program.
- E.g. If we compare two programs of same size, the one with more decision-making statements will be more complex as the control of program jumps frequently.
- Cyclomatic Complexity Measure quantifies complexity of software based on decision-making constructs of program such as if-else, do-while, repeat-until, switch-case and go to statements.
- It is a Graph driven model.

Cyclomatic Complexity Measures

- Process to make flow control graph:
  - Break program in smaller blocks, delimited by decision-making constructs.
  - Create nodes representing each of these nodes.
  - Connect nodes as follows:
  - If control can branch from block i to block j
  - Draw an arc
  - From exit node to entry node
  - Draw an arc.
- To calculate Cyclomatic complexity of a program module, we use the formula -

  $V(G) = e - n + 2$

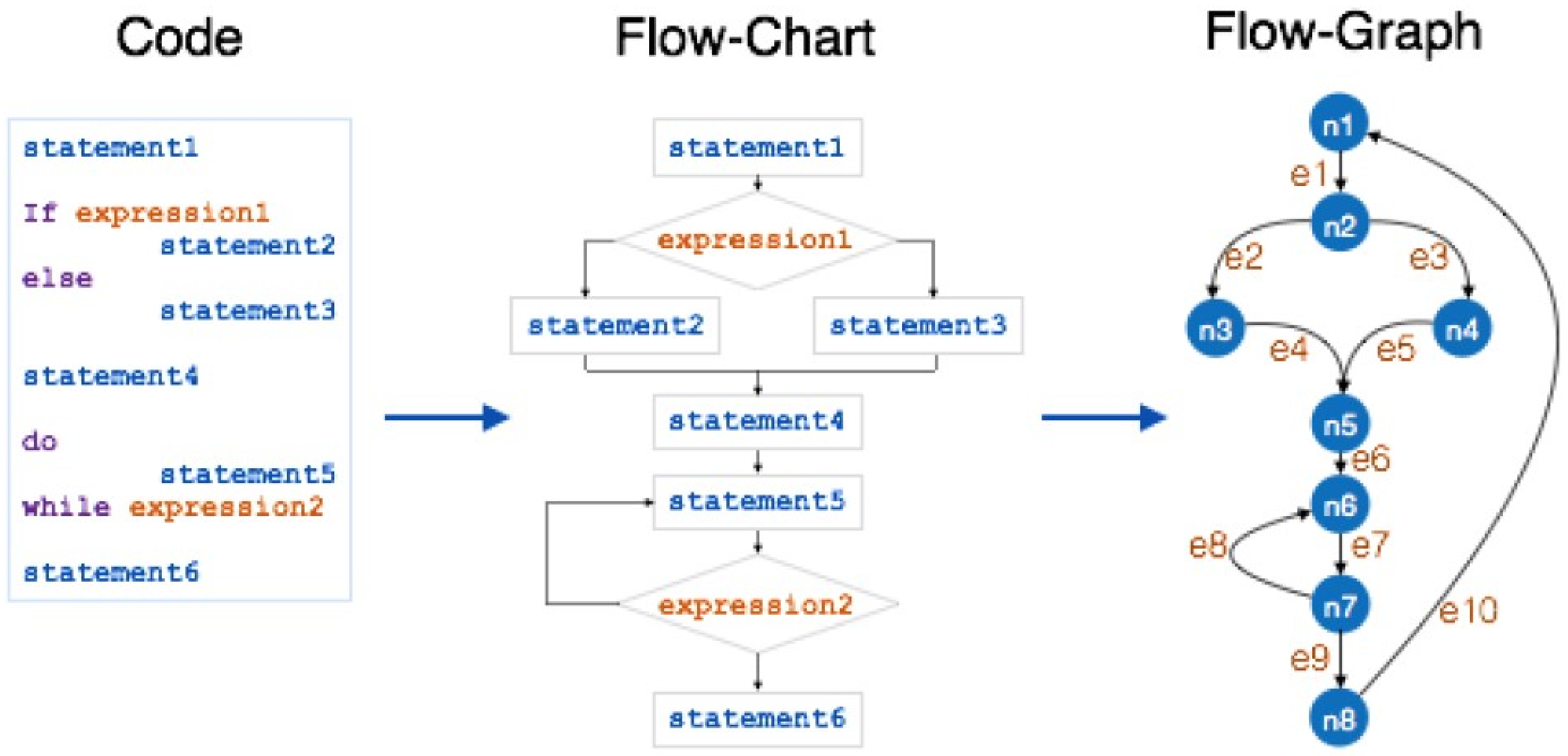
  Where,

  - e is total number of edges

  - n is total number of nodes

Cyclomatic Complexity Measures

- Example:



```
statement1

If expression1
          statement2
else
          statement3

statement4

do
          statement5
while expression2

statement6
```

Cyclomatic Complexity Measures

- The Cyclomatic complexity of the above module is

  $e = 10$

  $n = 8$

- Cyclomatic Complexity $= 10 - 8 + 2$

  $= 4$

- According to P. Jorgensen, Cyclomatic Complexity of a module should not exceed 10.

Software Project Estimation Models

- The different parameters of a project that need to be estimated include—project size, effort required to complete the project, project duration, and cost.
- Accurate estimation not only helps in quoting an appropriate project cost to the customer, but also forms the basis for resource planning and scheduling.
- Estimation techniques can broadly be classified into three main categories:
    - Empirical estimation techniques
    - Heuristic techniques
    - Analytical estimation techniques

Software Project Estimation Models

- **Empirical Estimation Techniques**
- They are based on making an educated guess of the project parameters, works better with having prior experience of development of similar products.
- Although empirical estimation techniques are based on common sense and subjective decisions, over the years, the different activities involved in estimation have been formalised to a large extent.
- Two such formalisations of the basic empirical estimation techniques are known as Expert judgement and the Delphi techniques.

Software Project Estimation Models

- **Expert Judgment**
- An expert makes an educated guess about the problem size after analysing the problem thoroughly.
- Overall estimate is the sum of estimates of the cost of the different components.
- *Shortcomings:*
- *The outcome of the technique is subject to human errors and individual bias and that the expert may overlook some factors inadvertently.*
- *An expert making an estimate may not have relevant experience and knowledge of all aspects of a project.*
- A more refined form of expert judgement is the estimation made by a group of experts. However, the estimate made by a group of experts may still exhibit bias.

Software Project Estimation Models

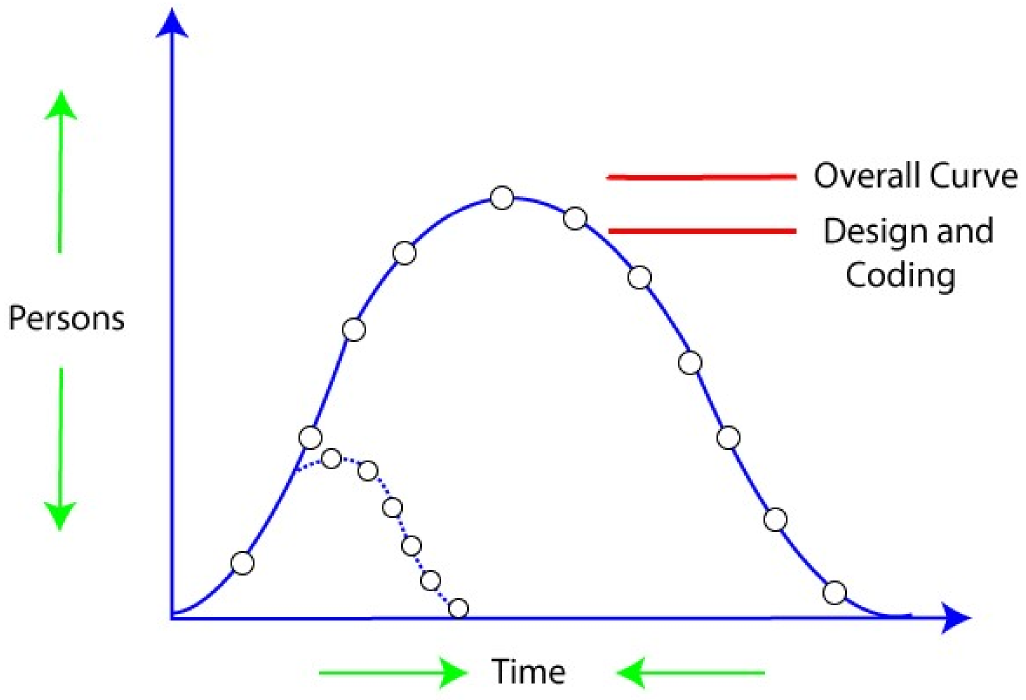
- **Delphi Cost Estimation**
- This technique tries to overcome some of the shortcomings of the expert judgement approach.
- Delphi estimation is carried out by a team comprising a group of experts and a co-ordinator.
- The co-ordinator provides each estimator with a copy of the software requirements specification (SRS) document and a form for recording his cost estimate, to be filled anonymously.
- The co-ordinator prepares the summary of the responses of all the estimators, and also includes any unusual rationale noted by any of the estimators.
- The prepared summary information is distributed to the estimators.

Software Project Estimation Models

- **Delphi Cost Estimation**
- After the completion of several iterations of estimations, the co-ordinator takes the responsibility of compiling the results and preparing the final estimate.
- The Delphi estimation, though consumes more time and effort, overcomes an important shortcoming of the expert judgement technique as the results cannot unjustly be influenced by overly assertive and senior members.

Software Project Estimation Models

- **Putnam Model**
- Putnam noticed that software staffing profiles followed the well known Rayleigh distribution.
- Only a small project staff is required at the beginning of a plan to carry out planning and specification tasks.
- As the project progresses and more detailed work are necessary, the project staff reaches a peak.
- After implementation and unit testing, the number of project staff falls.



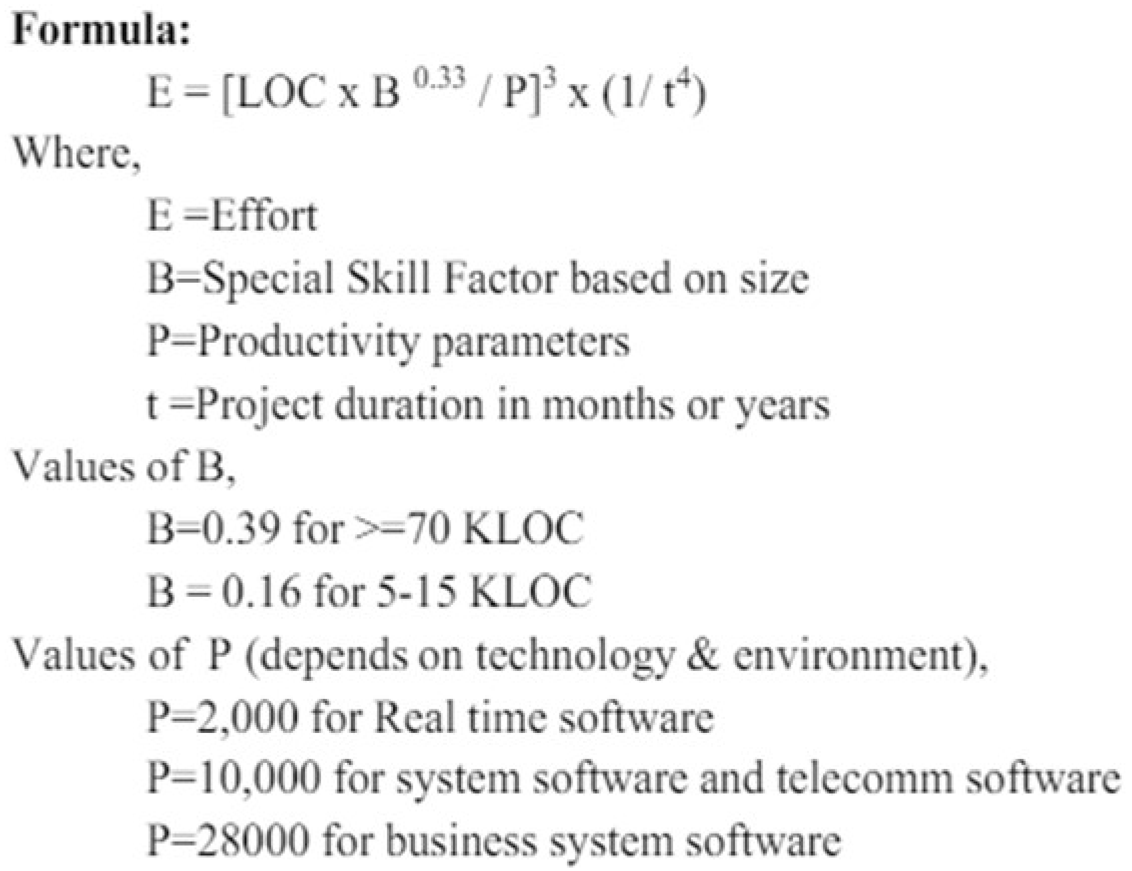
The Rayleigh manpower loading Curve

Software Project Estimation Models

- **Putnam Model**
- Putnam model describes the time and effort required to finish a software project of specified size
- SLIM(Software Lifecycle Management) is the name given by Putnam to the proprietary suite of tools his company QSM, Inc. has developed based on his model.
- It is one of the earliest of these types of models developed, and is among the most widely used.
- Closely related software parametric models are Constructive Cost Model (COCOMO), Parametric Review of Information and Costing and Evaluation – Software (PRICE – S) and Software Evaluation and Estimation of Resources – Software Estimating Model(SEER-SEM).

Software Project Estimation Models

- **Putnam Model**



Formula:

$$E = [LOC \times B^{0.33} / P]^3 \times (1/t^4)$$

Where,

E =Effort

B=Special Skill Factor based on size

P=Productivity parameters

t =Project duration in months or years

Values of B,

B=0.39 for >=70 KLOC

B = 0.16 for 5-15 KLOC

Values of P (depends on technology & environment),

P=2,000 for Real time software

P=10,000 for system software and telecomm software

P=28000 for business system software

- The **Putnam Estimation model** works reasonably well for very large systems, but seriously overestimates the effort on medium and small systems.

Software Project Estimation Models

- **COCOMO—A Heuristic Estimation Technique**
- <u>CO</u>nstructive <u>CO</u>st estimation <u>MO</u>del (COCOMO)
- It was proposed by Boehm, which prescribes a three stage process for project estimation.
- In the first stage, an initial estimate is arrived at. Over the next two stages, the initial estimate is refined to arrive at a more accurate estimate.
- COCOMO uses both single and multivariable estimation models at different stages of estimation.
- The three stages of COCOMO estimation technique are—
    - Basic COCOMO,
    - Intermediate COCOMO, and
    - Complete COCOMO.

COCOMO—A Heuristic Estimation Technique

- **Basic COCOMO**:
- Any software development project can be classified into one of the following three categories based on the development **complexity—*organic, semidetached, and embedded.***
- To identify project type, the characteristics of the product, the development team and development environment have to be considered.
- The three product development classes correspond to **application, utility and system software.**
- Data processing programs are considered to be **application programs.**
- Compilers, linkers, etc., are **utility programs**.
- Operating systems and real-time system programs, etc. are **system programs.**

COCOMO—A Heuristic Estimation Technique

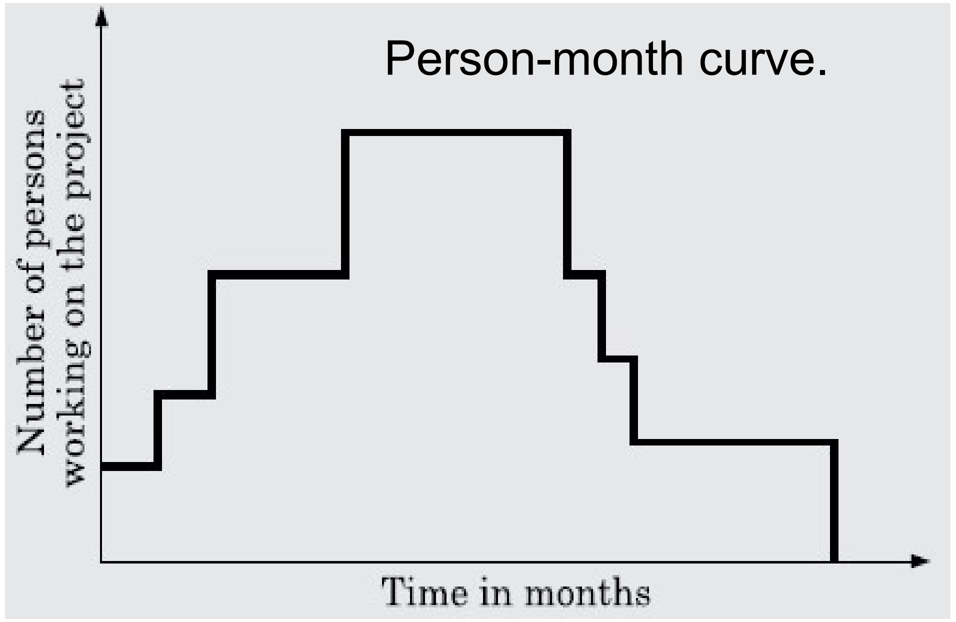
- **Basic COCOMO**:
- System programs interact directly with the hardware and programming complexities also arise out of the requirement for meeting timing constraints and concurrent processing of tasks.
- Utility programs are roughly three times as difficult to write as application programs and system programs are roughly three times as difficult as utility programs.
- Definitions of organic, semidetached, and embedded software are elaborated as follows:

COCOMO—A Heuristic Estimation Technique

- **Organic**: if the project deals with developing a well-understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar types of projects.
- **Semidetached**: if the development team consists of a mixture of experienced and inexperienced staff.
- Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.
- **Embedded:** if the software being developed is strongly coupled to hardware, or if stringent regulations on the operational procedures exist.
- Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.

COCOMO—A Heuristic Estimation Technique

- Along with the characteristics of the product, we need to consider the characteristics of the team members to identify right category of the system.
- The units used: for the **effort (in units of person-months)** and development time from the **size estimation given in kilo lines of source code (KLOC).**
- **What is a person-month?**
- Different number of personnel may work at different points in the project development.
- The number of personnel working on the project usually increases or decreases by an integral number, resulting in the sharp edges.



COCOMO—A Heuristic Estimation Technique

- **General form of the COCOMO expressions**
- The basic COCOMO estimation model, being a single variable model is given by expressions of the following forms:

$$\text{Effort} = a_1 \times (\text{KLOC})^{a_2} \text{ PM}$$

$$\text{Tdev} = b_1 \times (\text{Effort})^{b_2} \text{ months}$$

- Where,
  - KLOC is the estimated size of the software product expressed in Kilo Lines Of Code.
  - a_1, a2, b_1, b_2 are constants for each category of software product.
  - T_{dev} is the estimated time to develop the software, expressed in months.
  - Effort is the total effort required to develop the software product, expressed in person-months (PMs).
- Every line of source text should be calculated as one LOC irrespective of the actual number of instructions on that line.

COCOMO—A Heuristic Estimation Technique

- **General form of the COCOMO expressions**
- Thus, if a single instruction spans several lines (say *n lines), it is considered to be nLOC.*
- **Estimation of development effort:** For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

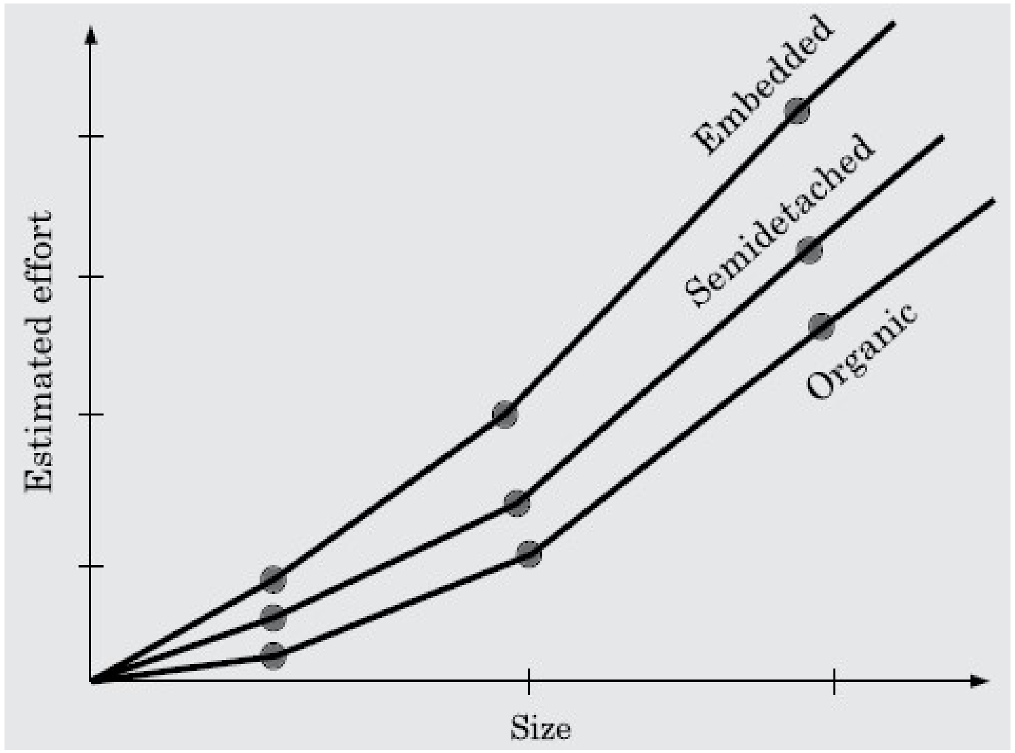
$$\begin{aligned}
&\text{Organic} &&: \text{Effort} = 2.4(\text{KLOC})^{1.05}\ \text{PM}\\
&\text{Semi-detached} &&: \text{Effort} = 3.0(\text{KLOC})^{1.12}\ \text{PM}\\
&\text{Embedded} &&: \text{Effort} = 3.6(\text{KLOC})^{1.20}\ \text{PM}
\end{aligned}$$

- **Estimation of development time**: For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

$$\begin{aligned}
&\text{Organic} &&: \text{Tdev} = 2.5(\text{Effort})^{0.38}\ \text{Months}\\
&\text{Semi-detached} &&: \text{Tdev} = 2.5(\text{Effort})^{0.35}\ \text{Months}\\
&\text{Embedded} &&: \text{Tdev} = 2.5(\text{Effort})^{0.32}\ \text{Months}
\end{aligned}$$

COCOMO—A Heuristic Estimation Technique

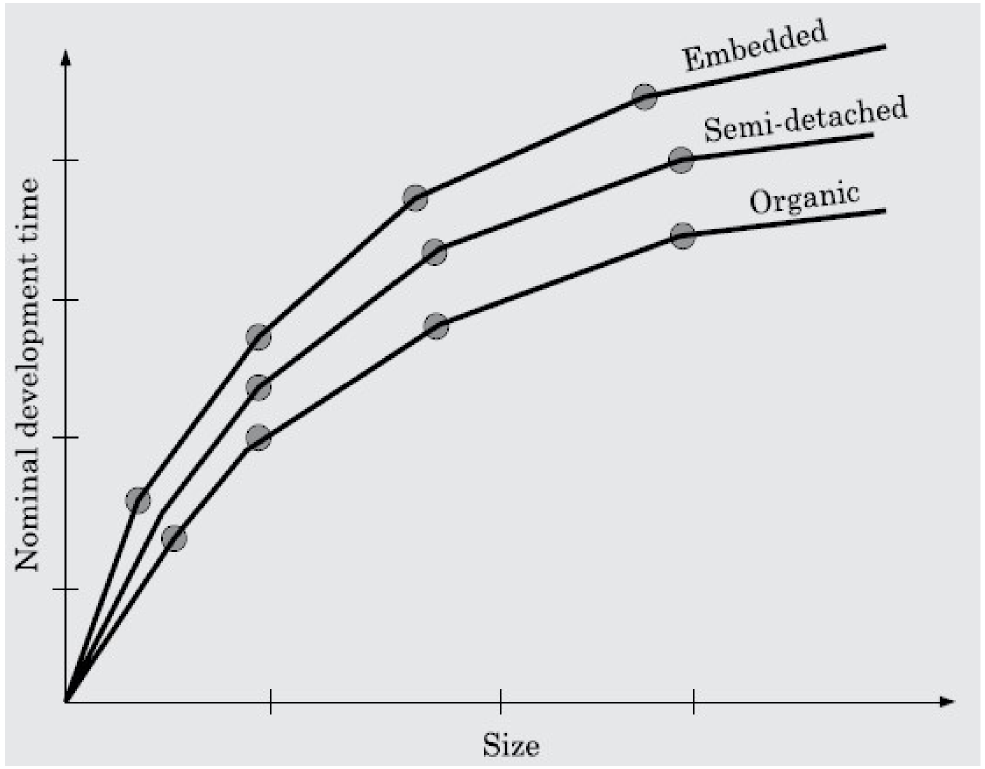
- **Observations from the effort-size plot**
- We can observe that the effort is somewhat super linear (that is, slope of the curve>1) in the size of the software product.
- The effort required to develop a product increases rapidly with project size



Effort versus product size.

COCOMO—A Heuristic Estimation Technique

- **Observations from the time-size plot**
- The development time is a sub linear function of the size of the product.
- That is, when the size of the product increases by two times, the time to develop the product does not double but rises moderately.
- For a project of any given size, the development time is roughly the same for all the three categories of products



Effort time product size.

COCOMO—A Heuristic Estimation Technique

- **Cost estimation**
- From the effort estimation, project cost can be obtained by multiplying the estimated effort (in man-month) by the manpower cost per month. (Cost = efforts * cost/month)
- A project would also incur several other types of costs referred to as the overhead costs.
- The overhead costs would include the costs due to hardware and software required for the project and the company overheads for administration, office space, electricity, etc.
- The project manager has to suitably scale up the cost arrived by using the COCOMO formula to include other costs.

COCOMO—A Heuristic Estimation Technique

- **Staff-size estimation**
- **Example1**: 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 a software developer is `15,000 per month. Determine the effort required to develop the software product, the nominal development time, and the cost to develop the product.
- **Solution:**

From the basic COCOMO estimation formula for organic Software:

Effort = $2.4 \times (32)^{1.05} = 91\ PM$

Nominal development time = $2.5 \times (91)^{0.38} = 14\ months$

Staff cost required to develop the product = 91 × Rs. 15,000 = Rs.1,465,000

COCOMO—A Heuristic Estimation Technique

- **Staff-size estimation**
- **Example2**: Suppose project size is 200 KLOC, then effort estimation is for different types is calculated as shown below.

  Organic $2.4(200)^{1.05}$ = 626 staff-months

  Semi-Detached $3.0(200)^{1.12}$ = 1,133 staff-months

  Embedded $3.6(200)^{1.20}$ = 2,077 staff-months
- For the same project time estimation is as given below.
- **Organic** E = 626 staff months

  TDEV = $2.5(626)^{0.38}$ = 29 months

  Semi-detached E = 1,133

  TDEV = $2.5(1133)^{0.35}$ = 29 months

  Embedded E = 2077

  TDEV = $2.5(2077)^{0.32}$ = 29 months

COCOMO—A Heuristic Estimation Technique

- **Average staff size:**

$$SS = \frac{E}{TDEV} = \frac{[\text{staff - }\cancel{\text{months}}]}{\cancel{[\text{months}]}} = [\text{staff}]$$

- **Productivity:**

$$P = \frac{Size}{E} = \frac{[\text{KLOC}]}{[\text{staff - months}]} = \text{KLOC} \big/ \text{staff - month}$$

COCOMO—A Heuristic Estimation Technique

- Complete Example, Organic
- Suppose an organic project has 7.5 KLOC,

Effort	$2.4(7.5)^{1.05}$ = 20 staff–months
Development time	$2.5(20)^{0.38}$ = 8 months
Average staff	20 / 8 = 2.5 staff
Productivity	7,500 LOC / 20 staff-months = 375 LOC / staff-month

COCOMO—A Heuristic Estimation Technique

- Complete Example, Embedded
- Suppose an embedded project has 50 KLOC,

Effort	$3.6(50)^{1.20}$ = 394 staff–months
Development time	$2.5(394)^{0.32}$ = 17 months
Average staff	394 / 17 = 23 staff
Productivity	50,000 LOC / 394 staff-months = 127 LOC / staff-month

COCOMO—A Heuristic Estimation Technique

- **Intermediate COCOMO**
- The basic COCOMO model assumes that effort and development time are functions of the product size alone.
- However, other project parameters besides the product size also affect the effort as well as the time required to develop the product.
- **“The intermediate COCOMO model refines the initial estimate obtained using the basic COCOMO expressions by scaling the estimate up or down based on the evaluation of a set of attributes of software development.”**

COCOMO—A Heuristic Estimation Technique

- **Intermediate COCOMO**
- The cost drivers identified by Boehm can be classified as being attributes of the following items:
- **Product:** The characteristics of the product to be considered include the inherent complexity of the product, reliability requirements of the product, etc.
- E.g. **RELY**: Required Software Reliability, **DATA**: Data Base Size, **CPLX**: Product Complexity.
- **Computer:** this include the execution speed required, storage space required, etc.
- E.g. **TIME**: Execution Time Constraint, **STOR**: Main Storage Constraint, **VIRT**: Virtual Machine Volatility, **TURN**: Computer Turnaround Time.

COCOMO—A Heuristic Estimation Technique

- **Intermediate COCOMO**
- **Personnel:** this include the experience level of personnel, their programming capability, analysis capability, etc.
- E.g. **ACAP:** Analyst Capability, **AEXP:** Application Experience, **PCAP:** Programming Capability, **VEXP:** Virtual Machine Experience, **LEXP:** Programming Language Experience.
- **Development environment:** these attributes capture the development facilities available to the developers.
- An important parameter is the sophistication of the **automation (CASE) tools** used for software development.

COCOMO—A Heuristic Estimation Technique

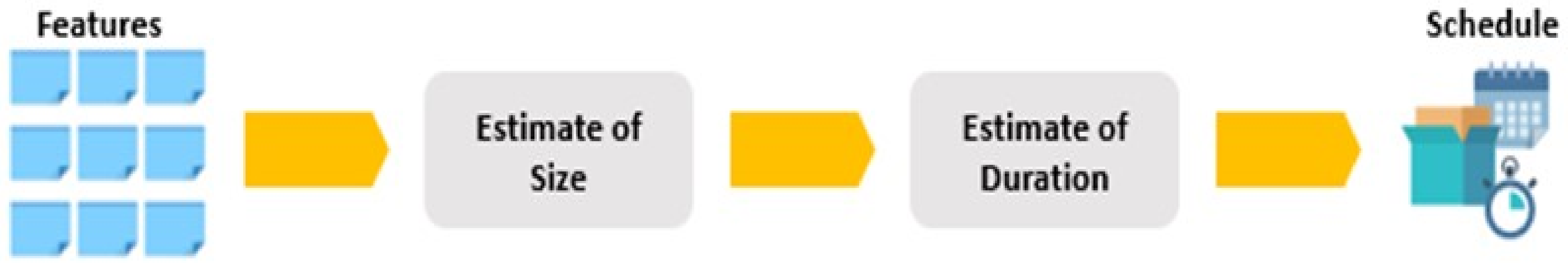
- **Complete COCOMO**
- A major shortcoming of both the **basic** and the **intermediate** COCOMO models is that they consider a software product as a single homogeneous entity.
- However, most large systems are made up of several smaller sub-systems and have widely different characteristics.
- For example, some sub-systems may be considered as organic type, some semidetached, and some even embedded.
- The cost to develop each sub-system is estimated separately, and the complete system cost is determined as the subsystem costs.
- **The complete COCOMO model considers these differences in characteristics of the subsystems and estimates the effort and development time as the sum of the estimates for the individual sub-systems.**

Estimating Size with Story Points, Velocity

- Example:
- “How long it will take to read the last ‘A Song of Ice and Fire’ book?”. I may look at the book and directly estimate that it will take me 5 days to finish reading the book.
- In doing this, I actually skipped any estimate of size and went directly with an estimate of duration.
- On the other hand, suppose now I first look at how many pages the book consists of. Let say it is 800 pages in total.
- This is my estimate of size. However, I also need to know how long it will take me to complete the book.
- Let say, consistently I read about 40 pages an hour in the past, so I just need to divide 800 pages by 40 and know that I could potentially finish reading the book in 20 hours.

Estimating Size with Story Points, Velocity

- To sum it up, the estimate of size derives the estimate of duration as shown below:



- Examples of traditional measures in size include:
    1. Lines of Codes
    2. Function points
- Traditional measures of size are not really a good measurement point to use and are not a good measure of productivity as shown by research.

Estimating Size with Story Points, Velocity

- In Agile, estimates of size and duration are distinct and separated. The two measures of size commonly used in Agile are: **Story points** and **Ideal days**.
- Story points are a unit of measure for expressing the overall size of a user story, feature, or other piece of work.
- The number of story points associated with a story represents the overall size of the story. There is no set formula for defining the size of a story.
- Rather, a story point estimate is an amalgamation of the amount of effort involved in developing the feature, the complexity of developing it, the risk inherent in it, and so on.
- When we estimate with story points we assign a point value to each item. The raw value we assign is unimportant. What matters are the relative values.

Estimating Size with Story Points

- **Estimating Size with Story Points**
- Story points are a relative measure of the size of a user story. Lets try to understand the analogy with an example.
- I go to Starbucks for the first time.
- All I can see is the different cup sizes which name as Short, Tall, Grande, and Venti.
- When I order, I won’t know exactly how many ounces of coffee I will get for each cup size.
- All I do know is that a Grande coffee is larger than a Short or Tall coffee and that it is smaller than a Venti coffee.

Estimating Size with Story Points

- It is important to know that, a particular story or feature is larger or smaller than other stories and features.
- When estimating with story points, a point value is assigned to each user story or feature. This could be a number between 1-10 for example.
- A story point estimate is influenced by the amount of effort, complexity, risk, etc. involved in developing it.
- *A user story estimated as 2 story points should be twice as big, complex, or risky as a story estimated as 1 story point.*
- *Similarly, a 5-point story should be half as big, complex, or risky as a 10-point story.*

Estimating Size with Story Points

- There are two common approaches (assuming a 1-10 scale story points):
- **First**, select a story expected to be one of the smallest stories the team will work on and say the story is estimated with a 1-story point.
- The **second** way is to select a story that is perceived somewhat medium-sized and assign it a story point somewhere in the middle of the range the team expects to use.
- Once a story point value is assigned to the first story, each additional story would be estimated by comparing it to the first story or to any others that have been estimated.

Velocity

- Velocity is a measure of a team’s rate of progress. It is derived by summing up the story points the team completed in an iteration.
- For example, if the team completed four stories each estimated at 5 story points in one iteration, then their velocity is 20.
- Planning-errors are self-correcting due to the application of velocity.
- For example, a team estimated a project to have a total of 400 story points. They thought they could complete 25 story points per iteration, meaning they will finish in 16 iterations.
- However, once the project kick-off, their observed velocity is only 20.
- Without having to re-estimate any work, the team would have correctly recognized that the project will take 20 iterations instead of 16.

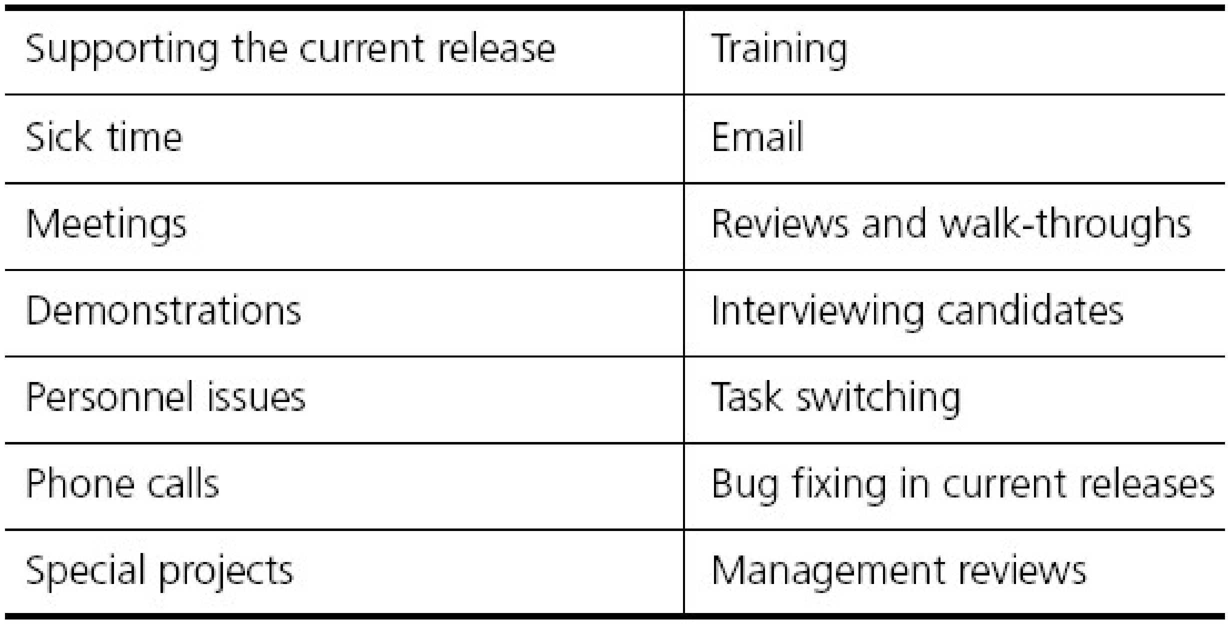
Estimating Time: Ideal Days for Estimated Size

- Ideal Days are generally viewed as the amount of time it takes a Software Developer (for example) to complete the work required in a user story.
- **For example:**
  - Ask any Developer how long they 'actually' get to spend writing code during a workday, and you will generally find most will tell you between 5 and 6 hours on an 8-hour workday (meetings, peer reviews, fixing bugs, etc will consume the other time).
  - This decreases further if you are a 'Senior Developer' because you will usually have extra responsibilities in a team, so let's take another hour from the Ideal Day, and go with 4 - 5 hours per day.

Estimating Time: Ideal Days for Estimated Size

- Ideal days is an estimate of the number of days your team would take to complete a project if they **worked on nothing else and had no interruptions**.
- In this context, “interruptions” means anything that causes the team to stop work on the project, including team meetings, training, other projects, illness, etc.

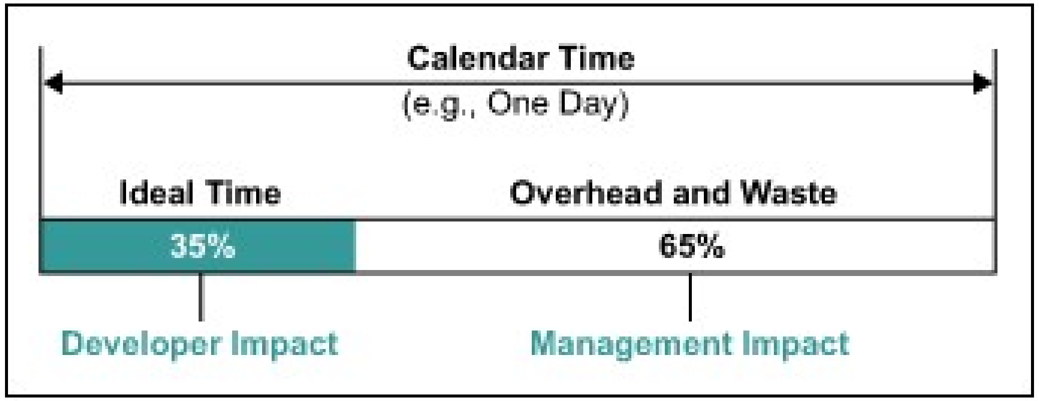
Factors affecting ideal time



Supporting the current release	Training
Sick time	Email
Meetings	Reviews and walk-throughs
Demonstrations	Interviewing candidates
Personnel issues	Task switching
Phone calls	Bug fixing in current releases
Special projects	Management reviews

Estimating Time: Ideal Days for Estimated Size

- **Elapsed Time**



- Let’s take an example. I have to build a new WCF Service and if I can work undisturbed it will take me about 2 days or 16 hours. Actual this is 2 days is Ideal days.
- But of course real days, I will have to interrupt my work for meetings, lunch, phone calls, urgent questions from the end-users etc.
- So to complete this WCF service I need 16 hours hands-on time but it will take me 22 hours **elapsed time** before the task is really finished. Because remaining 6 hours is including meeting, getting help, phone calls etc.

Estimating Time: Ideal Days for Estimated Size

- **Pros of Ideal Days**
- There are two main pros to the ideal days method:
- It’s concrete and intuitive. Clients and outside stakeholders understand it immediately.
- When you tell a client, “this project should take 20 days,” you don’t have to educate them about the process you’re using or a new idea.
- They already understand the concept of time estimates expressed in hours, days, or months.
- In most cases, clients and outside stakeholders will ask for an estimate given in days, weeks, or months .

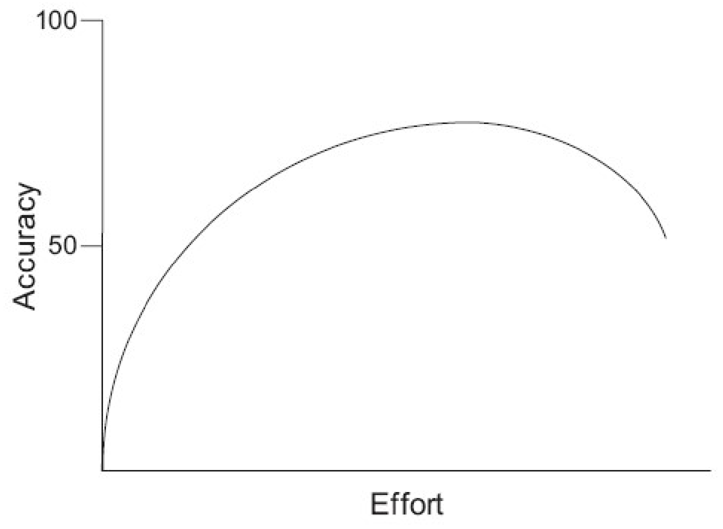
Estimating Time: Ideal Days for Estimated Size

- **Cons of Ideal Days**
- There are three major problems with the ideal days method:
- Human beings are scientifically proven to be outrageously bad at estimating how long things will take;
- Ideal days are often calculated based on the ideal case scenario, not a data-backed review of actual work completed on similar projects;
- In practice, software teams often miss their promise date when using the ideal days method.
- **Conclusion**
- This is not news to developers, of course. It’s no secret that we’re all bad at estimating time.
- Despite these drawbacks, though, ideal days remain a popular method of estimating time. That brings us to the other most popular method: Story points.

Techniques for Estimation

- The more effort we put into something, the better the result but we often need to expend just a fraction of that effort to get adequate results, to improve accuracy.
- No matter how much effort is invested, the estimate is never at the top of the accuracy axis. No matter how much effort you put into an estimate, an estimate is still an estimate.
- No amount of additional effort will make an estimate perfect. Next, notice how little effort is required to dramatically move the accuracy up from the baseline.

- As drawn in Figure, about 10% of the effort gets 50% of the potential accuracy.
- Finally, notice that eventually the accuracy of the estimate declines.



Techniques for Estimation

- When starting to plan a project, it is useful to think about where on the curve of Figure we wish to be.
- Many projects try to be very high up the accuracy axis, forcing teams far out on the effort axis even though the benefits diminish rapidly.
- This only adds up upfront work and documentation to b increased. And even after all of this the estimates still aren’t perfect.
- Agile teams, however, choose to be closer to the left in a figure. They acknowledge that we cannot eliminate uncertainty from estimates but the idea that small efforts are rewarded with big gains.

Techniques for Estimation

- **Estimates Shared**
- Estimates are not created by a single individual on the team. Agile teams do not rely on a single expert to estimate.
- As a proven fact, estimates prepared by those who will do the work are better than estimates prepared by anyone else (Lederer 1992).
- There are two reasons for this.
- First, on an agile project we tend not to know specifically who will perform a given task, anyone may work on anything. Hence, it is important that everyone have input into the estimate.
- Second, even though we may expect a particular person to do some work, others may have something to say about her estimate.

Techniques for Estimation

- **Estimation Scale**
- Studies have shown that we are best at estimating things that fall within one order of magnitude.
- E.g. We can better estimate distance of various places in the city we are living in rather than estimating the distance of moon from the earth or some neighbouring country capital.
- The reason is, we are best within a single order or magnitude we would like to have most of our estimates in such a range.
- Two estimation scales the author has gained good success with are:
  - 1, 2, 3, 5, and 8
  - 1, 2, 4, and 8

Techniques for Estimation

- **Estimation Scale**
- The **first** is the Fibonacci sequence. This is very useful estimation sequence because the gaps in the sequence become appropriately larger as the numbers increase.
- The **second** sequence is spaced such that each number is twice the number that precedes it. These non-linear sequences work well because they reflect the greater uncertainty associated with estimates for larger units of work.
- It's unlikely that a team will encounter many user stories or features that truly take no work, including 0 is often useful because of two reasons:
- **First**, if we want to keep all features within a 10x range, assigning non-zero values to tiny features will limit the size of largest features.
- **Second**, if the work truly is closer to 0 than 1, the team may not want the completion of the feature to contribute to its velocity calculations.

Techniques for Estimation

- **Estimation Scale**
- If the team does elect to include 0 in its estimation scale, everyone involved in the project (especially the product owner) needs to understand that (13 * 0) != 0.
- Some teams prefer to work with larger numbers such as 10, 20, 30, 50, and100. If you go with larger numbers such as 10 –100, its recommend that you pre-identify the numbers you will use within that range.
- Do not, for example, allow one story to be estimated at 66 story points or ideal days and another story to be estimated at 67.
- That is a false level of precision and we cannot discern a 1.5% difference in size.
- It's acceptable to have one-point differences between values such as 1, 2, and 3. As percentages, those differences are much larger than between 66 and 67.

Techniques for Estimation

- **Derive Estimation**
- The three most common techniques for estimating are:
    - Expert opinion
    - Analogy
    - Disaggregation
- These techniques can be either used on their own or may be combined with others to improve the results.

Techniques for Estimation

- **Expert Opinion**
- An expert, who relies on relies on her intuition or gut feel, is asked how long something will take or how big it will be.
- Less useful technique on agile projects than on traditional projects as in earlier case, estimates are assigned to user stories or other user-valued functionality.
- Developing this functionality is likely to require a variety of skills normally performed by more than one person. This makes it difficult to find suitable experts who can assess the effort across all disciplines.
- On a traditional project for which estimates are associated with tasks this is not as significant of a problem because each task is likely performed by one person.
- A nice benefit of estimating by expert opinion is that it usually doesn’t take very long.

Techniques for Estimation

- **Estimating by Analogy**
- There is evidence that we are better at estimating relative size than we are at estimating absolute size.
- The estimator compares the story being estimated to one or more other stories. If the story is twice the size, it is given an estimate twice as large.
- Do not compare all stories against a single baseline or universal reference, rather estimate each new story against an assortment of those that have already been estimated.
- This is referred to as triangulation process, which compares the story being estimated against a couple of other stories.
- To decide if a story should be estimated at five story points, see if it seems a little bigger than a story estimated at three and a little smaller than a story estimated at eight.

Techniques for Estimation

- **Disaggregation**
- It refers to splitting a story or feature into smaller, easier-to-estimate pieces. If most of the user stories are in the range of 2-5 days to develop, it will be very difficult to estimate a single story that may be 100 days.
- The solution to this is to break the large story or feature into multiple smaller items and estimate those.
- However, you need to be careful not to go too far with this approach. Not only does the likelihood of forgetting a task increase if we disaggregate too far, summing estimates of lots of small tasks leads to problems as well.

Techniques for Estimation

- **Planning Poker**
- Planning poker combines expert opinion, analogy, and disaggregation into an enjoyable approach to estimating that results in quick but reliable estimates.
- Participants in planning poker include all of the developers on the team. On an agile project this will typically not exceed ten people. If it does, it is usually best to split into two teams.
- Each team can then estimate independently, which will keep the size down. The product owner participates in planning poker but does not estimate.

Techniques for Estimation

- **Planning Poker**
- At the start of planning poker, each estimator is given a deck of cards pre-prepared. Each card has written on it one of the valid estimates.
- For each user story or theme to be estimated, a moderator(product owner or analyst or anybody) reads the description. The product owner answers any questions that the estimators have.
- After their questions are answered, each estimator privately selects a card representing his or her estimate.
- Cards are not shown until each estimator has made a selection. Then, all cards are simultaneously turned over and shown so that all participants can see each estimate.

Techniques for Estimation

- **Planning Poker**
- If estimates differ, the high and low estimators explain their estimates and have discussions about their own views.
- After the discussion, each estimator re-estimates by again selecting a card.
- Cards are once again kept private until everyone has estimated at which point they are turned over at the same time.
- It rarely takes more than three rounds but continue the process as long as estimates are moving closer together.
- It isn’t necessary that everyone in the room turn over a card with exactly the same estimate written down.

Techniques for Estimation

- **When To Play Planning Poker**
- Teams will need to play planning poker at two different times.
- **First**, to estimate a large number of items before the project officially begins or during its first iterations, with a team and two or three meetings of from one to three hours.
- It depends on how many items there are to estimate, the size of the team, and the product owner's ability to succinctly clarify the requirements.
- **Second**, teams will need to put forth some ongoing effort to estimate any new stories that are identified during an iteration.

Techniques for Estimation

- **Why Planning Poker works?**
- Some of the reasons why it works so well:
- **First**, it brings together multiple expert opinions forming a cross-functional team from all disciplines on a software project they are better suited to the estimation task than anyone else.
- **Second**, a lively dialogue ensues and estimators are called upon by their to justify their estimates.
- This has been found to improve the accuracy of the estimate, especially on items with large amounts of uncertainty.

Techniques for Estimation

- **Why Planning Poker works?**
- **Third**, studies have shown that averaging individual estimates leads to better results as do group discussions of estimates.
- Group discussion is the basis of planning poker and those discussions lead to an averaging of sorts of the individual estimates.
- **Finally, planning poker works because it’s fun.**