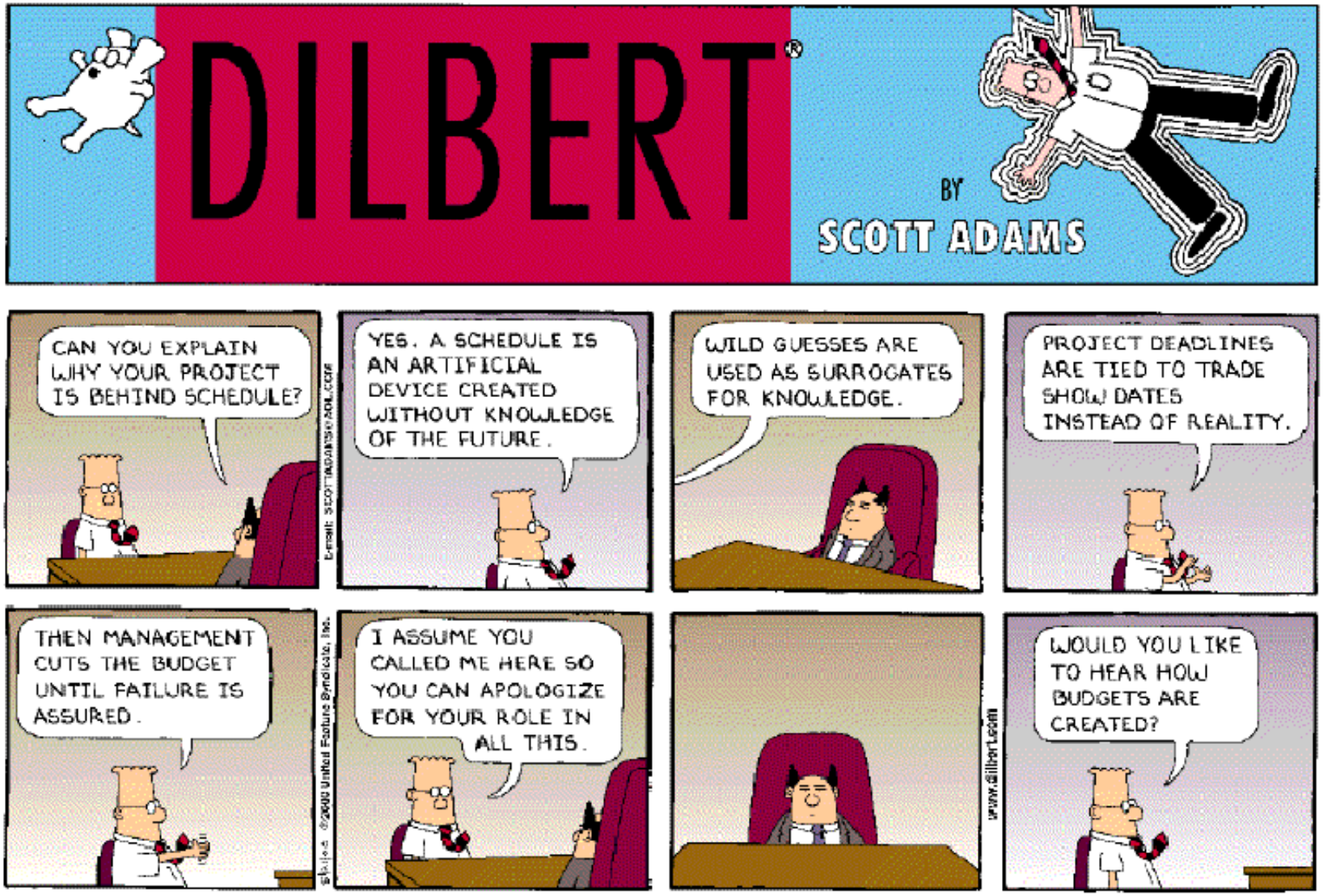
COCOMO Models

Project Management and Mr. Murphy

1. Logic is a systematic method of coming to the wrong conclusion with confide[illegible]
2. Techn[illegible]nage what[illegible]
3. Noth[illegible] on schedule or within budg[illegible]
4. If mathematically you end up with the incorrect answer, try multiplying by the page number.

Mr. Murphy was an optimist



Motivation

The software cost estimation provides:

- The vital link between the general concepts and techniques of **economic analysis** and the particular world of **software engineering.**
- Software cost estimation techniques also provides an essential part of the foundation for **good software management**.

Planning Prerequisites

- The planning process requires the following inputs:
  - Required human effort (man-months)
  - Project duration (months)
  - Project costs ($)
- We would like our estimates to be perfectly precise and accurate
  - But this requirement is impossible until the project is over

Cost of a project

- The cost in a project is due to the requirements for software, hardware and human resources
- The cost of software development is due to the human resources needed
- Most cost estimates are measured in ***person-months (PM)***
- At any point, the accuracy of the estimate will depend on the amount of reliable information we have about the final product.
- The cost of the project depends on the nature and characteristics of the project

Software Cost Estimation

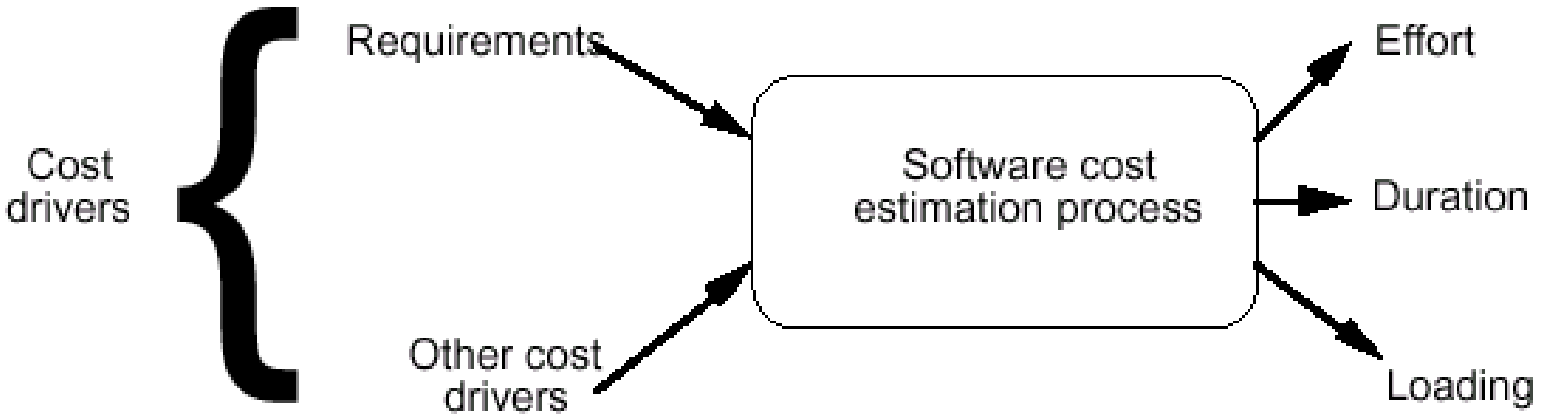


Figure 1. Classical view of software estimation process.

Importance of Estimates

- In the early days of computing,
  - Software costs were a small part of the total system cost
    - Even large errors (order of magnitude) = little impact on the total system cost
  - Today, software costs are the largest component of total system cost
    - Large errors in estimating cost equate to
      - The difference between profit and loss or
      - Survival and demise

Reasons for Inaccuracy in Estimates

- Too many uncertainties in the variables that determine the cost
- In the following categories
  - Human
    - The developers and their skills are not perfectly known
  - Technical
    - May need to use new or unfamiliar technology
  - Environmental
    - May need to run on unfamiliar computer or operating systems
  - Political
    - Internal company or client politics

Estimation Techniques

- Five main categories of techniques
  - Expert judgment
  - Estimation by analogy
  - Pricing to win
  - Parkinson pricing
  - Algorithmic models

Expert Judgment

- Several experts in the application domain independently prepare estimates
- Estimates are compared (together with the rationale for the estimate)
- Differences are resolved by discussion
- It is not
  - Estimation by committee
  - An averaging of the independent estimates

Estimation by Analogy

- Cost of a new project is estimated by analogy to similar systems previously developed
  - Identify differences and estimate cost of these differences
- How to handle no previously developed similar systems?
- What about changes in development environment?
  - How to handle employee turnover?
  - How to handle new language, case tools, …

Pricing to Win

- The cost is what you believe the customer is willing to spend
- What circumstances would lead you to price a project this way?

Parkinson Pricing

- Parkinson's Law
  - Parkinson's Law – The work expands to fill the time available
- Cost is determined by available resources rather than by objective analysis
- Example
  - If the software is needed in 1 year and you have 5 developers available to work on the project, the effort is 60 man-months

Importance of Deviation

- It is important to identify changes from previous projects, especially when employing
  - Expert Judgment or
  - Estimation by Analogy
- Failure to identify change and account for its influence
  - Distorts the estimate
    - Perhaps to the point that the estimate is of little value

Importance of Deviation (cont)

- Examples of change affecting estimates
  - Object-oriented versus structured techniques
  - Client-server systems versus stand-alone applications
  - COTS components versus developed components
  - Reuse versus all new development
  - CASE tools / code generators versus unsupported development

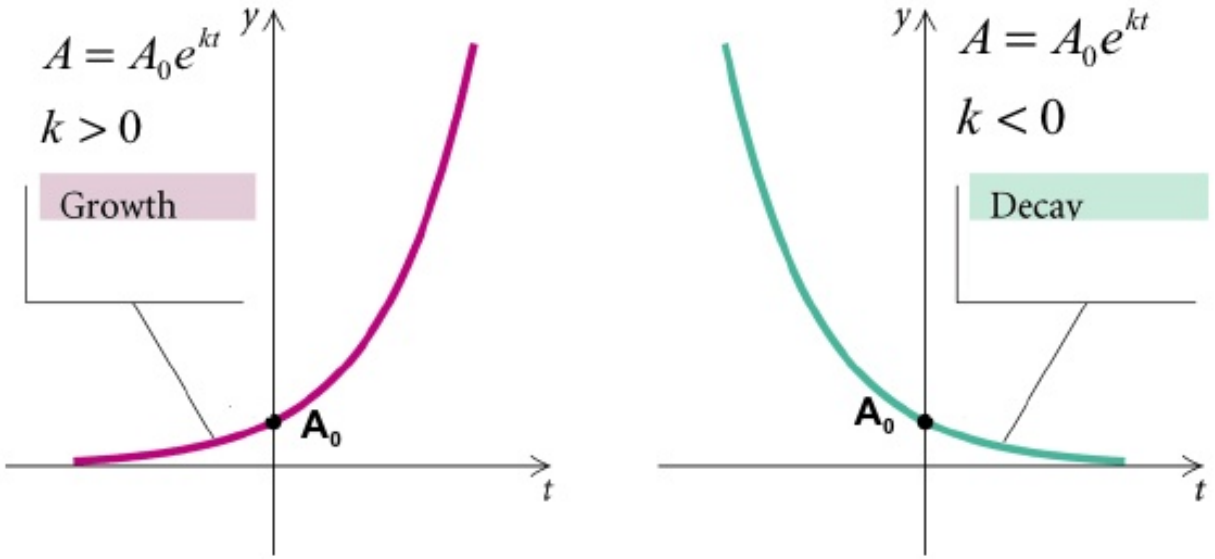
Algorithmic Models

- A formula (or set of formulae) is evaluated to provide an estimate
- Size or functionality metrics are the independent variables
- Constants in the formula are based upon historic cost data

Algorithmic Cost Modeling

- The most systematic approach to cost modeling
  - The most precise method, but
  - Don't confuse with the most accurate
- A formula or set of formulae is used to predict cost based on *project size*, and sometimes other project factors
- Most algorithmic cost models have an *exponential component*
  - Realizing that cost does not scale linearly with size

Graphs



Algorithmic Modeling (cont)

- The simplest model is a static single-variable model

$$Effort\,(PM) = A * Size^{B}$$

- Where
  - A is a constant factor
    - Factor incorporating process, product and development characteristics
  - Size code size or a function oriented value
  - B A constant typically in the range of 1 to 1.5
  - Effort or PM is in person-months
  - 1 person-month =152 working hours

Size Metrics

- Two common categories of size metrics
  - Lines of code
  - Function oriented metrics
- While lines of code (LOC) not the only size metric
  - LOC is the most commonly used measure of size

Productivity

- Productivity equation
  - **(DSI) / (PM)**
    - where **PM** = number of person-month (=152 working hours),
    - **DSI** = "delivered source instructions"

Schedule

- Schedule equation
  - **TDEV = C * (PM)^{n} (**months)
    - where TDEV = number of months estimated for software development.

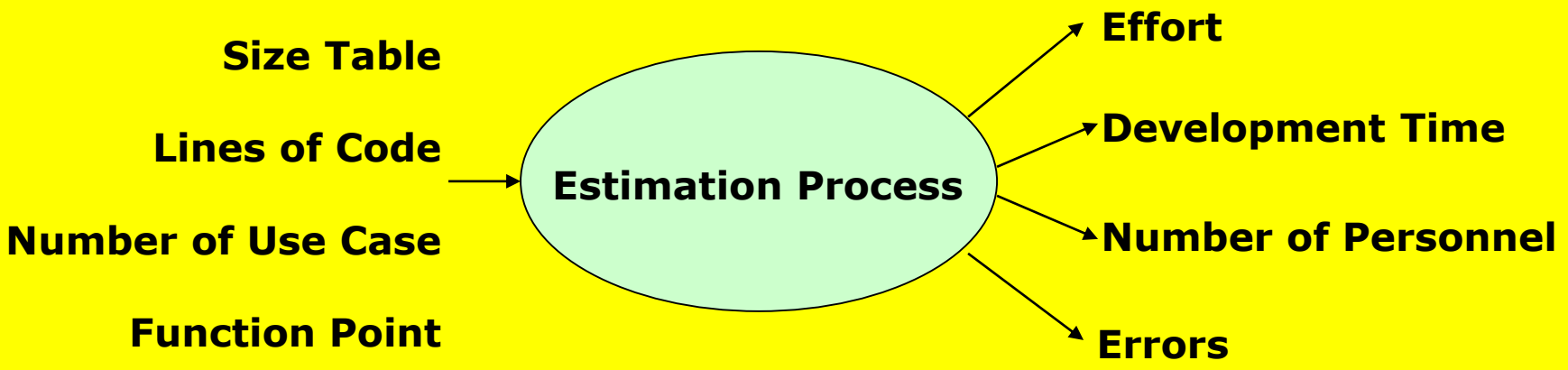
Average Staffing

- Average Staffing Equation
  - **(PM) / (TDEV)** **(**FSP)
    - where FSP means Full-time-equivalent Software Personnel.

Cost Estimation Process

Cost=SizeOfTheProject x Productivity

Cost Estimation Process



Project Size - Metrics

1. **Number of functional requirements**
2. **Cumulative number of functional and non-functional requirements**
3. **Number of Customer Test Cases**
4. **Number of 'typical sized' use cases**
5. **Number of inquiries**
6. **Number of files accessed (external, internal, master)**
7. **Total number of components (subsystems, modules, procedures, routines, classes, methods)**
8. **Total number of interfaces**
9. **Number of System Integration Test Cases**
10. **Number of input and output parameters (summed over each interface)**
11. **Number of Designer Unit Test Cases**
12. **Number of decisions (if, case statements) summed over each routine or method**
13. **Lines of Code, summed over each routine or method**

Project Size – Metrics(.)

Availability of Size Estimation Metrics:

	Development Phase	Available Metrics
a	Requirements Gathering	1, 2, 3
b	Requirements Analysis	4, 5
d	High Level Design	6, 7, 8, 9
e	Detailed Design	10, 11, 12
f	Implementation	12, 13

LOC Metric

- There are two different ways of implementing LOC
  - Lines of Code (LOC or KLOC)
    - Count all lines
  - Thousand of delivered source instructions (KDSI)
    - Count of the physical source statements, includes:
      - Format statements
      - Data declarations
    - Excludes
      - Comments
      - Unmodified utilities

Problems associated with lines of code as a metric

1. Lack of Accountability:
2. Lack of Cohesion with Functionality:
3. Adverse Impact on Estimation:
4. Developer's Experience:
5. Difference in Languages:
6. Advent of GUI Tools:
7. Problems with Multiple Languages:
8. Lack of Counting Standards:
9. Psychology:

- Lack of Accountability:
  - Not useful to measure the productivity of a project using only results from the coding phase, which usually accounts for only 30% to 35% of the overall effort
- Lack of Cohesion with Functionality:
  - Effort may be highly correlated with LOC, but functionality is not so much!
  - skilled developers may be able to develop the same functionality with far less code,
  - developer who develops only a few lines may still be more productive than a developer creating more lines of code
- Adverse Impact on Estimation:
  - Because of point 1 estimates based on lines of code can adversely go wrong
- Developer's Experience:
  - Implementation of a specific logic differs based on the level of experience of the developer. Hence, number of lines of code differs from person to person.
  - An experienced developer may implement certain functionality in fewer lines of code than another developer of relatively less experience does, though they use the same language.
- Difference in Languages:
  - Consider two applications that provide the same functionality (screens, reports, databases). One of the applications is written in C++ and the other application written in a language like COBOL. The number of function points would be exactly the same, but aspects of the application would be different. The lines of code needed to develop the application would certainly not be the same. As a consequence, the amount of effort required to develop the application would be different

- Advent of GUI Tools:
  - GUI-based programming languages and tools such as Visual Basic, allow programmers to write relatively little code and achieve high levels of functionality.
  - a user with a GUI tool can drag-and-drop and other mouse operations to place components on a workspace.
- Problems with Multiple Languages:
  - software is often developed in more than one language depending on the complexity and requirements.
  - Tracking and reporting of productivity and defect rates poses a serious problem in this case since defects cannot be attributed to a particular language subsequent to integration of the system.
- Lack of Counting Standards:
  - There is no standard definition of what a line of code is. Do comments count? Are data declarations included? What happens if a statement extends over several lines?
  - Organizations like SEI and IEEE have published some guidelines in an attempt to standardize counting, it is difficult to put these into practice since new languages being introduced every year.
- Psychology:
  - A programmer whose productivity is being measured in lines of code will have an incentive to write unnecessarily verbose code.
  - This is undesirable since increased complexity can lead to increased cost of maintenance and increased effort required for bug fixing.

FFP

- Proposed by van der Poel and Schach
  - Medium Size Projects (1- 10 man years)
  - Identify and score 3 basic structural elements
    - Files, Flows, and Processes

Structural elements

- Files
  - Permanent files only
    - Do not count temporary or transaction files
- Flows
  - Interfaces between the product and the environment
    - Input / Output Screens
    - Reports
- Processes
  - Functionally coherent manipulations of data
    - Sorting
    - Validating
    - Transforming

FFP (cont)

- Size
  - The size is the sum of the Files, Flows and Processes

$$Size = Files + Flows + Processes$$

- Cost
  - The product of Size and a constant *d*
    - Constant varies from organization to organization
    - Based on historic cost and size data

$$Cost = d * Size$$

FFP (cont)

- Note:
  - This metric is based upon the functionality of the application
    - High level property of the system
    - Can be more accurate earlier in the life-cycle than LOC metrics

Class Exercise

- An application maintains 8 files: a sorted master data file, 3 index files, 1 transaction file and 3 temporary files. It has 3 data input screens, 3 display screens, generates 4 printed reports, and 6 error message boxes. The processing includes sorting the master file, updating transactions, calculating report data from master file data. Assume a value of 800 for *d* .

  Determine the Size and Cost using FFP.

FFP Summary

- Advantages
  - A simple algorithmic model
    - Based on easy-to-count characteristics of a high level design
- Disadvantages
  - All items are equally weighted
  - Requires historic data based upon a particular organization
  - Has not been extended to correctly count databases
  - Something unsettling about adding unlike quantities

Function Points

- A similar approach taken by Albrecht
- Based on 5 functionality characteristics
  - Input items, output items, inquiries, master files, and interfaces
- First calculate the number of unadjusted function points

$$UFP = C_1 * Inp + C_2 * Out + C_3 * Inq + C_4 * Maf + C_5 * Inf$$

Function Points

Measure size in terms of the amount of functionality in a system. Function points are computed by first calculating an ***unadjusted function point count* (UFC).** Counts are made for the following categories

- *External inputs* – those items provided by the user that describe distinct application-oriented data (such as file names and menu selections)
- *External outputs* – those items provided to the user that generate distinct application-oriented data (such as reports and messages, rather than the individual components of these)
- *External inquiries* – interactive inputs requiring a response
- *External files* – machine-readable interfaces to other systems
- *Internal files* – logical master files in the system

Function Points (cont)

- The constants $C_{1 \ldots 5}$ are determined from the following table

Characteristic	Weight (Simple, Average, Complex)		
Input items	3	4	6
Output items	4	5	7
User Queries	3	4	6
Files	7	10	15
External interface	5	7	10

Function Points (cont)

- The next step is to calculate a technical complexity factor
- Each of 14 technical factors is assigned a value from 0 to 5
  - 0 - Not present or no influence
  - 5 - Strong influence throughout
- The degree of influence *DI* obtained by summing the above values

Function Po

- The 14 technical factors are:

1	**Data communication**
2	Distributed data processing
3	Performance criteria
4	Heavily utilized hardware
5	Online data entry
6	End-user efficiency
7	Transaction Rate
8	Online updating
9	Complex computations
10	Reusability
11	Ease of installation
12	Ease of operation
13	Maintainability
14	Multiple Sites

Function Points (cont)

- Calculate the technical complexity factor TCF
$$TCF = 0.65 + 0.01 * DI$$
  - TCF values are in the range of 0.65 to 1.35
- Finally the number of function points FP is $FP = UFP * TCF$

Function Point Calculations

- You may find the following template useful

	Item Complexity			
Characteristic	Low	Medium	High	Total
Input Items	____ * 3 = ____	____ * 4 = ____	____ * 6 = ____	
Output Items	____ * 4 = ____	____ * 5 = ____	____ * 7 = ____	
User Queries	____ * 3 = ____	____ * 4 = ____	____ * 6 = ____	
Master Files	____ * 7 = ____	____ * 10 = ____	____ * 15 = ____	
Ext. Interfaces	____ * 5 = ____	____ * 7 = ____	____ * 10 = ____	
			Unadjusted Function Points	
			Technical Complexity Factor 0.65 + 0.01*____	
			Adjusted Function Points	

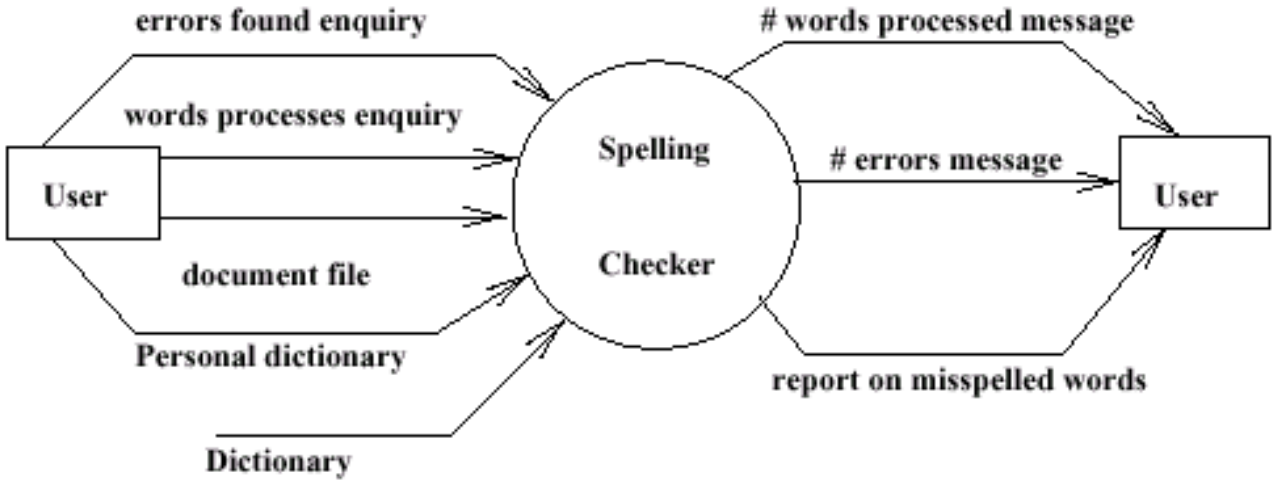
Class Exercise

- An application has 5 simple inputs, 4 complex inputs, 30 average outputs, 5 simple queries, 10 average master files and 8 complex interfaces. The degree of influence is 50. Calculate the number of unadjusted function points and the number of function points.

Class Exercise

Example

The Spell-Checker accepts as input a document file and an optional personal dictionary file. The checker lists all words not contained in either of these files. The user can query the number of words processed and the number of spelling errors found at any stage during processing.



Solution

- 2 users inputs: document file name, personal dictionary name (average)
- 3 users outputs: fault report, word count, misspelled error count (average)
- 2 users requests: #treated words?, #found errors? (average)
- 1 internal file: dictionary (average)
- 2 external files: document file, personal dictionary (av).

$$UFP = 4 \times 2 + 5 \times 3 + 4 \times 2 + 10 \times 1 + 7 \times 2 = 55$$

Solution

- **Technical Complexity Factors:**
  - **1. Data Communication 3**
  - **2. Distributed Data Processing 0**
  - **3. Performance Criteria 4**
  - **4. Heavily Utilized Hardware 0**
  - **5. High Transaction Rates 3**
  - **6. Online Data Entry 3**
  - **7. Online Updating 3**
  - **8. End-user Efficiency 3**
  - **9. Complex Computations 0**
  - **10. Reusability 3**
  - **11. Ease of Installation 3**
  - **12. Ease of Operation 5**
  - **13. Portability 3**
  - **14. Maintainability 3**
    - » **DI =30 (Degree of Influence)**

Solution

- Function Points
  - FP=UFP*(0.65+0.01*DI)= 55*(0.65+0.01*30)=52.25
  - That means **FP=52.25**

Relation between LOC and FP

- Relationship:
  - ***LOC = Language Factor * F***
  - where
    - **LOC** (Lines of Code)
    - **FP** (Function Points)

Relation between LOC and FPs

Language	LOC/FP
assembly	320
C	128
Cobol	105
Fortan	105
Pascal	90
Ada	70
OO languages	30
4GL languages	20

Relation between LOC and FP(.)

Assuming LOC's per FP for:

Java = 53,

C++ = 64

KLOC = FP * LOC_per_FP / 1000

It means for the SpellChekcer Example: (Java)

LOC=52.25*53=2769.25 LOC or 2.76 KLOC

Simple Object-Oriented Estimation

- Simple 4 step model developed by Lorenz and Kidd
- The number of estimated classes served as the size parameter

The Four Steps

1. Determine the number of problem *domain classes* in the application
2. Determine the *interface and the associated weight*
3. Calculate the number of total classes by multiplying the number of problem domain classes by the interface weight and add it to the number of problem domain classes
4. Calculate the number of man-days by multiplying the total number of classes by a productivity constant in the range of 15 - 20

Interface Weights

Interface Type	Weight
No user interface	2.0
Simple text-based interface	2.25
Graphical user interface	2.5
Complex graphical user interface	3.0

Class Exercise

- An object-oriented application has an estimated 50 problem domain cases and a graphical user interface. Assuming a productivity constant of 18, calculate the number of man-days that will needed to develop the application.

COCOMO

- COCOMO is a static single variable model
- COCOMO is an acronym for **Co**nstructive **Co**st **Mo**del that was developed by Barry Boehm
- The COCOMO models are defined for three classes of software projects.

  (1) *organic mode*

  (2) *semi-detached mode*

  (3) *embedded mode*

Introduction to COCOMO models

- The **COstructive COst Model** (COCOMO) is the most widely used software estimation model.
- The COCOMO model predicts the **effort** and **duration** of a project based on inputs relating to the size of the resulting systems and a number of "**cost drives**" that affect productivity.

COCOMO

- COCOMO is actually a hierarchy of models of the following form
  - Basic COCOMO – estimates software development effort and cost as a function of program size in lines of code
  - Intermediate COCOMO - estimates software development and cost as a function of program size in lines of code and a set of "cost drivers"
  - Advanced COCOMO – incorporates the characteristics of intermediate COCOMO with an assessment of the cost driver impact on each phase of the software development cycle
- *The more complex models account for more factors that influence software projects, and make more accurate estimates.*
- *The most important factors contributing to a project's duration and cost is the Development Mode*

Project Types/Levels of COCOMO

The level of difficulty was broken into three modes

- Organic Mode
  - Constraints on development are mild
  - Many similar projects previously developed by the organization
  - *Relatively small, simple software projects in which small teams with good application experience work to a set of less than rigid requirements (e.g., a thermal analysis program developed for a heat transfer group)*
- Semi-detached Mode
  - More constraints on development, but some flexibility remains
  - Few similar projects previously developed by the organization
  - *An intermediate (in size and complexity) software project in which teams with mixed experience levels must meet a mix of rigid and less than rigid requirements (e.g., a transaction processing system with fixed requirements for terminal hardware and data base software)*
- Embedded Mode
  - Very tight constraints
  - No similar projects previously developed
  - *A software project that must be developed within a set of tight hardware, software and operational constraints (e.g., flight control software for aircraft).*

Modes

Feature	Organic	Semidetached	Embedded
Organizational understanding of product and objectives	Thorough	Considerable	General
Experience in working with related software systems	Extensive	Considerable	Moderate
Need for software conformance with pre-established requirements	Basic	Considerable	Full
Need for software conformance with external interface specifications	Basic	Considerable	Full

Modes (.)

Feature	Organic	Semidetached	Embedded
Concurrent development of associated new hardware and operational procedures	Some	Moderate	Extensive
Need for innovative data processing architectures, algorithms	Minimal	Some	Considerable
Premium on early completion	Low	Medium	High
Product size range	<50 KDSI	<300KDSI	All

Effort Computation

- The **Basic COCOMO model** computes effort as a function of program size. The Basic COCOMO equation is:
  - ***Effort = aKLOC^b***
- Effort for three modes of Basic COCOMO.

Mode	**a**	**b**
Organic	2.4	1.05
Semi-detached	3.0	1.12
Embedded	3.6	1.20

Example

Mode	Effort Formula
Organic	$E = 2.4 * (S^{1.05})$
Semidetached	$E = 3.0 * (S^{1.12})$
Embedded	$E = 3.6 * (S^{1.20})$

Size = 200 KLOC

Effort = $a * Size^{b}$

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

Semidetached — $E = 3.0 * (200^{1.12}) = 1133$ staff-months

Embedded — $E = 3.6 * (200^{1.20}) = 2077$ staff-months

Example

Mode	Effort Formula
Organic	$E = 2.4 * (S^{1.05})$
Semidetached	$E = 3.0 * (S^{1.12})$
Embedded	$E = 3.6 * (S^{1.20})$

Size = 200 KLOC

Effort = $a * \text{Size}^b$

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

Semidetached — $E = 3.0 * (200^{1.12}) = 1133$ staff-months

Embedded — $E = 3.6 * (200^{1.20}) = 2077$ staff-months

COCOMO (cont)

- The intermediate model calibrated on
  - 40 software development projects
- Further work revealed certain difficulty factors that dramatically influenced the effort estimates and schedule

Intermediate COCOMO Calculations

- Intermediate COCOMO calculations proceed by
  - First, determine the mode (organic, semi-detached, embedded)
  - This determines the constants A - D

Parameter	Organic	Semi-detached	Embedded
A	3.2	3.0	2.8
B	1.05	1.12	1.20
C	2.5	2.5	2.5
D	0.38	0.35	0.32

Intermediate COCOMO (cont)

– Second, using the appropriate constants from the previous table, calculate the nominal effort and schedule from

$$E_{nominal} = A*(KDSI)^{B} \qquad \text{(Nominal Effort)}$$

– Third, calculate a difficulty multiplier that depends upon the cost drivers (include subjective assessments of attributes in the general areas of)

- Product
- Hardware
- Personnel
- Project

Intermediate COCOMO (cont)

- Fourth, the adjusted effort E is the nominal effort $E_{nominal}$ multiplied by the difficulty multiplier
- Fifth, the project duration is calculated from

$$Schedule = C*(E)^{D} \qquad \text{(Duration in Months)}$$

Effort Computation

- The **intermediate COCOMO model** computes effort as a function of program size and a set of cost drivers. The Intermediate COCOMO equation is:
  - ***E = aKLOC^b*EAF***
- Effort for three modes of intermediate COCOMO.

Mode	**a**	**b**
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

Effort computation(.)

Effort Adjustment Factor

Cost Driver	Very Low	Low	Nominal	High	Very High	Extra High
Required Reliability	.75	.88	1.00	1.15	1.40	1.40
Database Size	.94	.94	1.00	1.08	1.16	1.16
Product Complexity	.70	.85	1.00	1.15	1.30	1.65
Execution Time Constraint	1.00	1.00	1.00	1.11	1.30	1.66
Main Storage Constraint	1.00	1.00	1.00	1.06	1.21	1.56
Virtual Machine Volatility	.87	.87	1.00	1.15	1.30	1.30
Comp Turn Around Time	.87	.87	1.00	1.07	1.15	1.15
Analyst Capability	1.46	1.19	1.00	.86	.71	.71
Application Experience	1.29	1.13	1.00	.91	.82	.82
Programmers Capability	1.42	1.17	1.00	.86	.70	.70
Virtual machine Experience	1.21	1.10	1.00	.90	.90	.90
Language Experience	1.14	1.07	1.00	.95	.95	.95
Modern Prog Practices	1.24	1.10	1.00	.91	.82	.82
SW Tools	1.24	1.10	1.00	.91	.83	.83
Required Dev Schedule	1.23	1.08	1.00	1.04	1.10	1,10

Each of the 15 attributes receives a rating on a six-point scale that ranges from "very low" to "extra high" (in importance or value).

The product of all effort multipliers results in an *effort adjustment factor (EAF)*.

Effort Computation (..)

Total EAF = Product of the selected factors

Calculating Adjusted value of Effort or Adjusted Person Months:

$$E = (\text{Total EAF}) * E_{Nominal}$$

Example

	Organic	Semidetached	Embedded
a	3.2	3.0	2.8
b	1.05	1.12	1.20

Mode	Effort Formula
Organic	$E = 3.2 * (S^{1.05}) * C$
Semidetached	$E = 3.0 * (S^{1.12}) * C$
Embedded	$E = 2.8 * (S^{1.20}) * C$

---

e.g. Size = 200 KLOC

Effort = a * Size^{b} * C

Cost drivers:

- Low reliability .88
- High product complexity 1.15
- Low application experience 1.13
- High programming language experience .95

C = .88 * 1.15 * 1.13 * .95 = 1.086

Organic — $E = 3.2 * (200^{1.05}) * 1.086 = 906$ staff-months

Semidetached — $E = 3.0 * (200^{1.12}) * 1.086 = 1231$ staff-months

Embedded — $E = 2.8 * (200^{1.20}) * 1.086 = 1755$ staff-months

Software Development Time

- **Development Time Equation Parameter Table:**

Parameter	Organic	Semi-detached	Embedded
C	2.5	2.5	2.5
D	0.38	0.35	0.32

Development Time, **$TDEV = C * E^{D}$**

Number of Personnel, **$NP = E / TDEV$**

where E is the Adjusted Effort

COCOMO II

- COCOMO II was developed to address several issues that did not exist when the original COCOMO was developed
  - The waterfall model was **the** software lifecycle development model
  - Most software ran on mainframes
  - Client-server and object oriented technologies were unknown, or at least were not widely used
  - Simple reuse in effect – no inheritance, etc

COCOMO II (cont)

- Major differences
  - COCOMO was based upon lines of codes estimate
    - COCOMO II allows the use of other metrics, i.e. function points
  - COCOMO had a constant exponent, depending upon which of three modes is selected
    - COCOMO II allows the exponent to continuously vary between 1.01 and 1.26
  - COCOMO assumes that savings due to reuse are directly proportional to the amount of reuse
    - COCOMO II uses a non linear model – even a small amount of reuse may incur a huge effort in understanding the code

COCOMO II (cont)

- COCOMO II modified the difficulty factors
- COCOMO II was calibrated with 83 projects

Distribution of Effort

- A development process typically consists of the following stages:
  - **Requirements Analysis**
  - **Design (High Level + Detailed)**
  - **Implementation & Coding**
  - **Testing (Unit + Integration)**

Distribution of Effort (.)

The following table gives the recommended **percentage distribution of Effort (APM)** and **TDEV** for these stages:

Percentage Distribution of Effort and Time Table:

	Req Analysis	Design, HLD + DD	Implementation	Testing	
Effort	**23%**	**29%**	**22%**	**21%**	**100%**
TDEV	**39%**	**25%**	**15%**	**21%**	**100%**

Error Estimation

- Calculate the estimated number of errors in your design, i.e.total errors found in requirements, specifications, code, user manuals, and bad fixes:
  - Adjust the **Function Point** calculated in step1

$$AFP = FP ** 1.25$$

  - Use the following table for calculating error estimates

Error Type	Error / AFP
Requirements	1
Design	1.25
Implementation	1.75
Documentation	0.6
Due to Bug Fixes	0.4

All Together