Software Measurement

Objectives

- To introduce the concept of measuring software development
- To introduce the fundamentals of software costing and pricing
- To describe three metrics for software productivity assessment
- To explain why different techniques should be used for software estimation
- To describe the principles of the COCOMO 2 algorithmic cost estimation model

Fundamental estimation questions

- How much effort is required to complete an activity?
- How much calendar time is needed to complete an activity?
- What is the total cost of an activity?
- Project estimation and scheduling are interleaved management activities.

From previous lecture – Need for measurement

Lorenz and Kidd metrics collection in different phases of development

Phase / Metric	Requirements Description	System Design	Program Design	Coding	Testing
Number of scenario scripts	X				
Number of key classes	X	X			
Number of support classes			X		
Average number of support classes per key class			X		
Number of subsystems			X	X	
Class size		X	X	X	
Number of operations overridden by a subclass		X	X	X	X
Number of operations added by a subclass		X	X	X	
Specialization index		X	X	X	X

Chidamber and Kemerer metrics collection in different phases of development

Phase Metric	System Design	Program Design	Coding	Testing
Weighted methods per class	X	X		X
Depth of inheritance	X	X		X
Number of children	X	X		X
Coupling between objects		X		X
Response for a class		X	X	
Lack of cohesion of methods		X	X	X

Where to capture OO metrics

Phase Metric	Use cases	Class diagrams	Inter-action diagrams	Class descriptions	State diagrams	Package diagrams
Number of scenario scripts	X					
Number of key classes		X				
Number of support classes		X				
Average number of support classes per key class		X				
Number of subsystems						X
Class size		X		X		
Number of operations overridden by a subclass		X				
Number of operations added by a subclass		X				
Specialization index		X				
Weighted methods in class		X				
Depth of inheritance		X				
Number of children		X				
Coupling between objects		X				
Response for a class				X		
Lack of cohesion in methods				X		
Average operation size			X			
Average number of parameters per operation			X			
Operation complexity				X		
Percent public and protected				X		
Public access to data members				X		
Number of root classes		X				
Fan-in/fan-out		X				

Software cost components

- Hardware and software costs
- Travel and training costs
- Effort costs (the dominant factor in most projects)
  - The salaries of engineers involved in the project;
  - Social and insurance costs.
- Effort costs must take overheads into account
  - Costs of building, heating, lighting.
  - Costs of networking and communications.
  - Costs of shared facilities (e.g library, staff restaurant, etc.).

Costing and pricing

- Estimates are made to discover the cost, to the developer, of producing a software system.
- There is not a simple relationship between the development cost and the price charged to the customer.
- Broader organizational, economic, political and business considerations influence the price charged.

Software pricing factors

Market opportunity	A development organisation may quote a low price because it wishes to move into a new segment of the software market. Accepting a low profit on one project may give the opportunity of more profit later. The experience gained may allow new products to be developed.
Cost estimate uncertainty	If an organisation is unsure of its cost estimate, it may increase its price by some contingency over and above its normal profit.
Contractual terms	A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged may then be less than if the software source code is handed over to the customer.
Requirements volatility	If the requirements are likely to change, an organisation may lower its price to win a contract. After the contract is awarded, high prices can be charged for changes to the requirements.
Financial health	Developers in financial difficulty may lower their price to gain a contract. It is better to make a smaller than normal profit or break even than to go out of business.

Software productivity

- A measure of the rate at which individual engineers involved in software development produce software and associated documentation.
- Not quality-oriented although quality assurance is a factor in productivity assessment.
- Essentially, we want to measure useful functionality produced per time unit.

Productivity measures

- Size related measures based on some output from the software process. This may be lines of delivered source code, object code instructions, etc.
- Function-related measures based on an estimate of the functionality of the delivered software. Function-points are the best known of this type of measure.

Measurement problems

- Estimating the size of the measure (e.g. how many function points).
- Estimating the total number of programmer months that have elapsed.
- Estimating contractor productivity (e.g. documentation team) and incorporating this estimate in overall estimate.

Product Models and Metrics

There are a large number of product types: requirements documents, specifications, design, code, specific components, test plans, ...

There are many abstractions of these products that depend on different characteristics

- logical, e.g., application domain, function
- static, e.g. size, structure
- dynamic, e.g., MTTF, test coverage
- use and context related, e.g., design method used to develop

Product models and metrics can be used to

- evaluate the process or the product
- estimate the cost of quality of the product
- monitor the stability or quality of the product over time

Static Characteristics

We can divide the static product the characteristics into three basic classes

- Size
- Structure, e.g., Control Structure, Data Structure

Size attempts to model and measure the physical size of the product

Structure models and metrics attempt to capture some aspect of the physical structure of the product, e.g., Control structure metrics measure the control flow of the product. Data structure metrics measure the data interaction of the product

There are mixes of these metrics, e.g., that deal with the interaction between control and data flow.

Lines of code

- What's a line of code?
  - The measure was first proposed when programs were typed on cards with one line per card;
  - How does this correspond to statements as in Java which can span several lines or where there can be several statements on one line.
- What programs should be counted as part of the system?
- This model assumes that there is a linear relationship between system size and volume of documentation.

Size Metrics

There are many size models and metrics, depending on the product,

- source code: lines of code, number of modules
- executables: space requirements, lines of code
- specification: function points
- requirements: number of requirements, pages of documentation
- modules: operators and operands

Size metrics can be used accurately at different points in time

- lines of code is accurate after the fact but can be estimated
- function points can be calculated based upon the specification

Size metrics are often used to

- characterize the product
- evaluate the effect of some treatment variable, such as a process
- predict some other variable, such as cost

Lines of Code Metrics

Lines of code can be measured as:

- all source lines
- all non-blank source lines
- all non-blank, non-commentary source lines
- all semi-colons
- all executable statements
- ...

The definition depends on the use of the metric, e.g.,

- to estimate effort we might use all source lines as they all take effort

- to estimate functionality we might use all executable statements as they come closest to representing the amount of function in the system

Lines of code

- vary with the language being used
- are the most common, durable, cheapest metric to calculate
- are most often used to characterize the product and predict effort

Documentation Metrics

(Law of the Core Dump) The thickness of the proposal required to win a multimillion dollar contract is about one millimeter per million dollars. If all the proposals conforming to this standard were piled on top of each other at the bottom of the Grand Canyon, it would probably be a good idea.

-Norman Augustine (Former CEO, Lockheed Martin Corp.)

Productivity comparisons

- The lower level the language, the more productive the programmer
  - The same functionality takes more code to implement in a lower-level language than in a high-level language.
- The more verbose the programmer, the higher the productivity
  - Measures of productivity based on lines of code suggest that programmers who write verbose code are more productive than programmers who write compact code.

System development times

	Analysis	Design	Coding	Testing	Documentation
Assembly code	3 weeks	5 weeks	8 weeks	10 weeks	2 weeks
High-level language	3 weeks	5 weeks	4 weeks	6 weeks	2 weeks

	Size	Effort	Productivity
Assembly code	5000 lines	28 weeks	714 lines/month
High-level language	1500 lines	20 weeks	300 lines/month

Function Points

- One model of a product is to view it as a set of interfaces, e.g., files, data passed, etc.
- If a system is primarily transaction processing and the "bulk" of the system deals with transformations on files, this is a reasonable view of size.
- Function Points were originally suggested as a measure of size by Al Albrecht at IBM, a a means of estimating functionality, size, effort
- It can be applied in the early phases of a project (requirements, preliminary design)

Function points

- Based on a combination of program characteristics
  - external inputs and outputs;
  - user interactions;
  - external interfaces;
  - files used by the system.
- A weight is associated with each of these and the function point count is computed by multiplying each raw count by the weight and summing all values.

$$\text{UFC} = \sum (\text{number of elements of given type}) \times (\text{weight})$$

Productivity estimates

- Real-time embedded systems, 40-160 LOC/P-month.
- Systems programs , 150-400 LOC/P-month.
- Commercial applications, 200-900 LOC/P-month.
- In object points, productivity has been measured between 4 and 50 object points/month depending on tool support and developer capability.
- But variance about 100%

Factors affecting productivity

Application domain experience	Knowledge of the application domain is essential for effective software development. Engineers who already understand a domain are likely to be the most productive.
Process quality	The development process used can have a significant effect on productivity. This is covered in Chapter 28.
Project size	The larger a project, the more time required for team communications. Less time is available for development so individual productivity is reduced.
Technology support	Good support technology such as CASE tools, configuration management systems, etc. can improve productivity.
Working environment	As I discussed in Chapter 25, a quiet working environment with private work areas contributes to improved productivity.

Quality and productivity

- All metrics based on volume/unit time are flawed because they do not take quality into account.
- Productivity may generally be increased at the cost of quality.
- It is not clear how productivity/quality metrics are related.
- If requirements are constantly changing then an approach based on counting lines of code is not meaningful as the program itself is not static;

Estimation techniques

- There is no simple way to make an accurate estimate of the effort required to develop a software system
  - Initial estimates are based on inadequate information in a user requirements definition;
  - The software may run on unfamiliar computers or use new technology;
  - The people in the project may be unknown.
- Project cost estimates may be self-fulfilling
  - The estimate defines the budget and the product is adjusted to meet the budget.

Changing technologies

- Changing technologies may mean that previous estimating experience does not carry over to new systems
  - Distributed object systems rather than mainframe systems;
  - Use of web services;
  - Use of ERP or database-centred systems;
  - Use of off-the-shelf software;
  - Development for and with reuse;
  - Development using scripting languages;
  - The use of CASE tools and program generators.

Estimation techniques

- Algorithmic cost modelling.
- Expert judgement.
- Estimation by analogy.
- Parkinson's Law.
- Pricing to win.

Estimation techniques

Algorithmic cost modelling	A model based on historical cost information that relates some software metric (usually its size) to the project cost is used. An estimate is made of that metric and the model predicts the effort required.
Expert judgement	Several experts on the proposed software development techniques and the application domain are consulted. They each estimate the project cost. These estimates are compared and discussed. The estimation process iterates until an agreed estimate is reached.
Estimation by analogy	This technique is applicable when other projects in the same application domain have been completed. The cost of a new project is estimated by analogy with these completed projects. Myers (Myers 1989) gives a very clear description of this approach.
Parkinson's Law	Parkinson's Law states that work expands to fill the time available. The cost is determined by available resources rather than by objective assessment. If the software has to be delivered in 12 months and 5 people are available, the effort required is estimated to be 60 person-months.
Pricing to win	The software cost is estimated to be whatever the customer has available to spend on the project. The estimated effort depends on the customer's budget and not on the software functionality.

Pricing to win

- The project costs whatever the customer has to spend on it.
- Advantages:
  - You get the contract.
- Disadvantages:
  - The probability that the customer gets the system he or she wants is small. Costs do not accurately reflect the work required.

Top-down and bottom-up estimation

- Any of these approaches may be used top-down or bottom-up.
- Top-down
  - ❑ Start at the system level and assess the overall system functionality and how this is delivered through sub-systems.
- Bottom-up
  - ❑ Start at the component level and estimate the effort required for each component. Add these efforts to reach a final estimate.

Top-down estimation

- Usable without knowledge of the system architecture and the components that might be part of the system.
- Takes into account costs such as integration, configuration management and documentation.
- Can underestimate the cost of solving difficult low-level technical problems.

Bottom-up estimation

- Usable when the architecture of the system is known and components identified.
- This can be an accurate method if the system has been designed in detail.
- It may underestimate the costs of system level activities such as integration and documentation.

Estimation methods

- Each method has strengths and weaknesses.
- Estimation should be based on several methods.
- If these do not return approximately the same result, then you have insufficient information available to make an estimate.
- Some action should be taken to find out more in order to make more accurate estimates.
- Pricing to win is sometimes the only applicable method.

Pricing to win

- This approach may seem unethical and un-businesslike.
- However, when detailed information is lacking it may be the only appropriate strategy.
- The project cost is agreed on the basis of an outline proposal and the development is constrained by that cost.
- A detailed specification may be negotiated or an evolutionary approach used for system development.

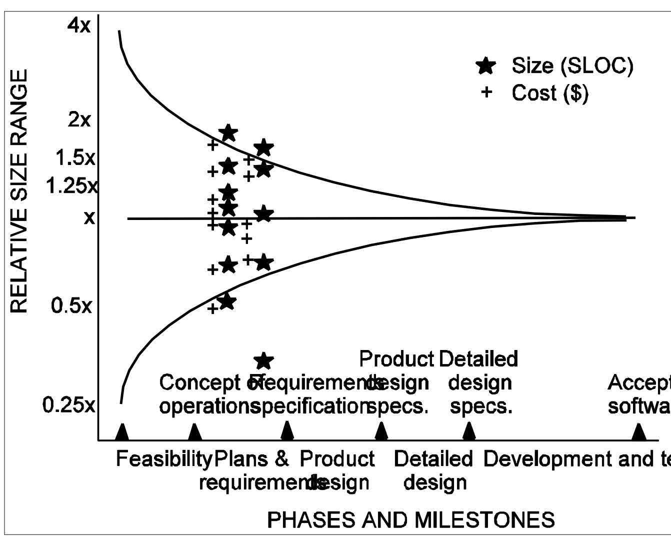
Algorithmic cost modelling

- Cost is estimated as a mathematical function of product, project and process attributes whose values are estimated by project managers:
  - Effort = A $\times$ Size^{B} $\times$ M
  - A is an organisation-dependent constant, B reflects the disproportionate effort for large projects and M is a multiplier reflecting product, process and people attributes.
- The most commonly used product attribute for cost estimation is code size.
- Most models are similar but they use different values for A, B and M.

Estimation accuracy

- The size of a software system can only be known accurately when it is finished.
- Several factors influence the final size
  - ❑ Use of COTS and components;
  - ❑ Programming language;
  - ❑ Distribution of system.
- As the development process progresses then the size estimate becomes more accurate.

Estimate uncertainty



The COCOMO model

- An empirical model based on project experience.
- Well-documented, 'independent' model which is not tied to a specific software vendor.
- Long history from initial version published in 1981 (COCOMO-81) through various instantiations to COCOMO 2.
- COCOMO 2 takes into account different approaches to software development, reuse, etc.

Levels of the COCOMO Model

The COCOMO model levels are:

- **Basic,** which is used for quick, early approximate estimates of software cost and schedule, but its accuracy is limited due to not using detailed data.
- **Intermediate**, which is used for better estimates of cost and schedule, because it considers software project environment factors in terms of their aggregate impact on the project parameters.
- **Detailed,** which is used for even better estimates, because it accounts for the influence of the software project environment factors on individual project phases.

We will concentrate our discussions on the basic and intermediate levels.

Modes of the COCOMO Model

The COCOMO model modes are:

- **Organic mode**, which is appropriate for small, stable projects
- **Embedded mode**, which is appropriate for large projects with tight constraints, that require some degree of innovation and have a complex software interface
- **Semi-detached mode**, which is appropriate for projects that fall in between the above two categories

Basic COCOMO Formulas

- The required effort (**E**) to develop the software system as a function of source size (S) (where E is expressed in Person-Months and S is expressed in KLOC):

BASIC COCOMO MODEL

MODE	EFFORT
Organic	$E = 2.4 * (S^{1.05})$
Semi-detached	$E = 3.0 * (S^{1.12})$
Embedded	$E = 3.6 * (S^{1.20})$

Resources

- The project duration (**TDEV**) as a function of effort (**E**) (where TDEV is expressed in calendar months, and E in Person-Months):

BASIC COCOMO MODEL

MODE	**SCHEDULE**
Organic	TDEV = 2.5* ($E^{0.38}$)
Semi-detached	TDEV = 2.5 * ($E^{0.35}$)
Embedded	TDEV = 2.5 * ($E^{0.32}$)

Intermediate COCOMO Model

- The intermediate COCOMO is an extension of the basic COCOMO model, which used only one predictor variable, the KLOC variable
- The intermediate COCOMO uses 15 more predictor variables called "**cost drivers**." The manager assigns a value to each cost driver from the range:
  - Very low
  - Low
  - Nominal
  - High
  - Very high
  - Extra high
- For each of the above values, a numerical value corresponds, which varies with the cost drivers

How to Use the Cost Drivers

- The manager assigns a value to each cost driver according to the characteristics of the specific software project
- The numerical values that correspond to the manager assigned values for the 15 cost drivers are multiplied
- The resulting value **I** is the multiplier that we use in the intermediate COCOMO formulas for obtaining the effort estimates
- Thus: **I** = RELY * DATA * CPLX * TIME * STOR * VIRT * TURN * ACAP * AEXP * PCAP * VEXP * LEXP * MODP * TOOL * SCED
- Note that although the effort estimation formulas for the intermediate model are different from those used for the basic model, the schedule estimation formulas are the same

Formulas of the Intermediate COCOMO Model

- The required effort to develop the software system (**E**) as a function of the nominal effort (**Enom**), (where E and Enom are expressed in Person-Months, and S in KLOC) is

E = Enom * I,

where:

INTERMEDIATE COCOMO MODEL

MODE	**EFFORT**
Organic	Enom = 3.2 * ($S^{1.05}$)
Semi-detached	Enom = 3.8 * ($S^{1.12}$)
Embedded	Enom = 2.8 * ($S^{1.20}$)

Formulas of the Intermediate COCOMO Model

- The number of months estimated for software development (**TDEV**) (where TDEV is expressed in calendar months, and E in Person-Months):

INTERMEDIATE COCOMO MODEL

MODE	SCHEDULE
Organic	TDEV = 2.5 * ($E^{0.38}$)
Semi-detached	TDEV = 2.5 * ($E^{0.35}$)
Embedded	TDEV = 2.5 * ($E^{0.32}$)

Source Code Size

- The source size (**S**) is expressed in KLOC, i.e. thousands of delivered lines of code, i.e. , the source size of the delivered software (which does ***not*** include the size of test drivers or other temporary code)
- If code is reused, then the following formula should be used for determining the "equivalent" software source size S_e, for use in the COCOMO model:

$$Se = S_n + (a/100) * S_u$$

- where S_n is the source size of the new code, S_u is the source size of the reused code, and **a** is determined by the formula:

$$a = 0.4 * D + 0.3 * C + 0.3 * I$$

- based on the percentage of effort required to adapt the reused design (D) and code (C), as well as the percentage of effort required to integrate the modified code (I)

Software development effort multipliers

Cost Drivers	Very Low	Low	Nominal	High	Very High	Extra High
Product Attributes						
RELY Required software reliability	.75	.88	1.00	1.15	1.40	
DATA Data base size		.94	1.00	1.08	1.16	
CPLX Product complexity	.70	.85	1.00	1.15	1.30	1.65
Computer Attributes						
TIME Execution time constraint			1.00	1.11	1.30	1.66
STOR Main storage constraint			1.00	1.06	1.21	1.56
VIRT Virtual machine volatility [a]		.87	1.00	1.15	1.30	
TURN Computer turnaround time		.87	1.00	1.07	1.15	

Ratings

[a] For a given software product, the underlying virtual machine is the complex of hardware and software (OS, DBMS, etc.) it calls on to accomplish its tasks.

(Con't)

Software development effort multipliers

Cost Drivers	Very Low	Low	Nominal	High	Very High	Extra High
Personnel Attributes						
ACAP Analyst capability	1.46	1.19	1.00	.86	.71	
AEXP Applications exp.	1.29	1.13	1.00	.91	.82	
PCAP Program. capability	1.42	1.17	1.00	.86	.70	
VEXP Virtual machine exp [a]	1.21	1.10	1.00	.90		
LEXP Programming language experience	1.14	1.07	1.00	.95		
Project Attributes						
MODP Use of modern programming practices	1.24	1.10	1.00	.91	.82	
TOOL Use of softw. tools	1.24	1.10	1.00	.91	.83	
SCED Required development schedule	1.23	1.08	1.08	1.04	1.10	

Ratings

[a] For a given software product, the underlying virtual machine is the complex of hardware and software (OS, DBMS, etc.) it calls on to accomplish its tasks.

Other Parameters

- **TDEV** starts when the project enters the product design phase (successful completion of a software requirements review) and ends at the end of software testing (successful completion of a software acceptance review)
- **E** covers management and documentation efforts, but not activities such as training, installation planning, etc.
- COCOMO assumes that the requirements specification is not substantially changed after the end of the requirements phase
- **Person-months** can be transformed to person-days by multiplying by 19, and to person-hours by multiplying by 152

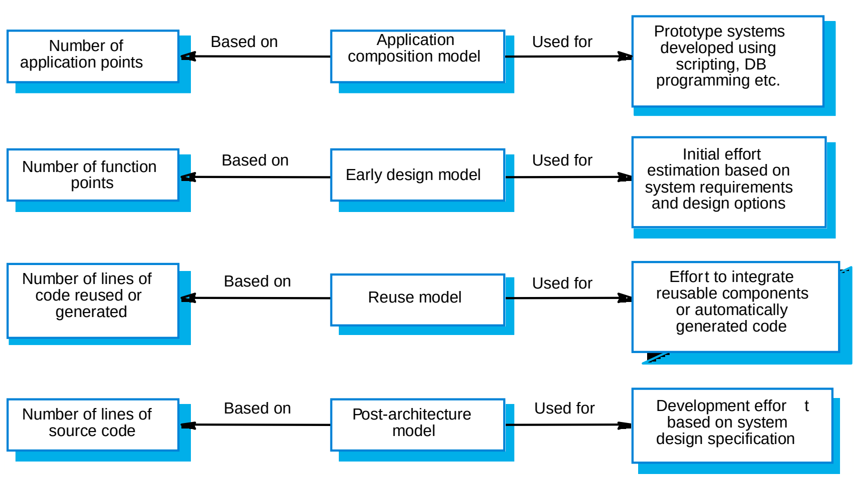
COCOMO 2

- COCOMO 81 was developed with the assumption that a waterfall process would be used and that all software would be developed from scratch.
- Since its formulation, there have been many changes in software engineering practice and COCOMO 2 is designed to accommodate different approaches to software development.

COCOMO 2 models

- COCOMO 2 incorporates a range of sub-models that produce increasingly detailed software estimates.
- The sub-models in COCOMO 2 are:
  - Application composition model. Used when software is composed from existing parts.
  - Early design model. Used when requirements are available but design has not yet started.
  - Reuse model. Used to compute the effort of integrating reusable components.
  - Post-architecture model. Used once the system architecture has been designed and more information about the system is available.

Use of COCOMO 2 models



The exponent term

- This depends on 5 scale factors (see next slide). Their sum/100 is added to 1.01
- A company takes on a project in a new domain. The client has not defined the process to be used and has not allowed time for risk analysis. The company has a CMM level 2 rating.
  - Precedenteness - new project (4)
  - Development flexibility - no client involvement - Very high (1)
  - Architecture/risk resolution - No risk analysis - V. Low .(5)
  - Team cohesion - new team - nominal (3)
  - Process maturity - some control - nominal (3)
- Scale factor is therefore 1.17.

Exponent scale factors

Precedentedness	Reflects the previous experience of the organisation with this type of project. Very low means no previous experience, Extra high means that the organization is completely familiar with this application domain.
Development flexibility	Reflects the degree of flexibility in the development process. Very low means a prescribed process is used; Extra high means that the client only sets general goals.
Architecture/risk resolution	Reflects the extent of risk analysis carried out. Very low means little analysis, Extra high means a complete a thorough risk analysis.
Team cohesion	Reflects how well the development team know each other and work together. Very low means very difficult interactions, Extra high means an integrated and effective team with no communication problems.
Process maturity	Reflects the process maturity of the organization. The computation of this value depends on the CMM Maturity Questionnaire but an estimate can be achieved by subtracting the CMM process maturity level from 5.

Multipliers

- Product attributes
  - Concerned with required characteristics of the software product being developed.
- Computer attributes
  - Constraints imposed on the software by the hardware platform.
- Personnel attributes
  - Multipliers that take the experience and capabilities of the people working on the project into account.
- Project attributes
  - Concerned with the particular characteristics of the software development project.

Effects of cost drivers

Exponent value	1.17
System size (including factors for reuse and requirements volatility)	128, 000 DSI
Initial COCOMO estimate without cost drivers	**730 person-months**
Reliability	Very high, multiplier = 1.39
Complexity	Very high, multiplier = 1.3
Memory constraint	High, multiplier = 1.21
Tool use	Low, multiplier = 1.12
Schedule	Accelerated, multiplier = 1.29
Adjusted COCOMO estimate	**2306 person-months**
Reliability	Very low, multiplier = 0.75
Complexity	Very low, multiplier = 0.75
Memory constraint	None, multiplier = 1
Tool use	Very high, multiplier = 0.72
Schedule	Normal, multiplier = 1
Adjusted COCOMO estimate	**295 person-months**

Project planning

- Algorithmic cost models provide a basis for project planning as they allow alternative strategies to be compared.
- Embedded spacecraft system
  - Must be reliable;
  - Must minimize weight (number of chips);
  - Multipliers on reliability and computer constraints > 1.
- Cost components
  - Target hardware;
  - Development platform;
  - Development effort.

Management option costs

Option	RELY	STOR	TIME	TOOLS	LTEX	Total effort	Software cost	Hardware cost	Total cost
A	1.39	1.06	1.11	0.86	1	63	949393	100000	1049393
B	1.39	1	1	1.12	1.22	88	1313550	120000	1402025
C	1.39	1	1.11	0.86	1	60	895653	105000	1000653
D	*1.39*	*1.06*	*1.11*	*0.86*	*0.84*	*51*	*769008*	*100000*	*897490*
E	1.39	1	1	0.72	1.22	56	844425	220000	1044159
F	1.39	1	1	1.12	0.84	57	851180	120000	1002706

Option choice

- Option D (use more experienced staff) appears to be the best alternative
  - However, it has a high associated risk as experienced staff may be difficult to find.
- Option C (upgrade memory) has a lower cost saving but very low risk.
- Overall, the model reveals the importance of staff experience in software development.

Project duration and staffing

- As well as effort estimation, managers must estimate the calendar time required to complete a project and when staff will be required.
- Calendar time can be estimated using a COCOMO 2 formula
  - $TDEV = 3 \times (PM)^{(0.33+0.2*(B-1.01))}$
  - PM is the effort computation and B is the exponent computed as discussed above (B is 1 for the early prototyping model). This computation predicts the nominal schedule for the project.
- The time required is independent of the number of people working on the project.

Staffing requirements

- Staff required can't be computed by diving the development time by the required schedule.
- The number of people working on a project varies depending on the phase of the project.
- The more people who work on the project, the more total effort is usually required.
- A very rapid build-up of people often correlates with schedule slippage.

Other measures

- Defects
- Complexity
- Reliability
- …

Change and Defect Models and Metrics

Changes can be categorized by

purpose - enhancement, adaptive, corrective, preventive

type - requirements, specification, design, architecture, planned enhancements, insert/delete debug code, improve clarity, optimize: space or time, feature, enhancement, bug

cause - market/external and internal needs

size - number of lines of code, number of components affected,

disposition - rejected as a change, not relevant, under consideration, being worked on, completed, saved for next enhancement

level of document changed - changes back to requirements document

number of customers affected - effects certain customer classes

Sample Change Metrics

- number of enhancements per month
- number of changes per line of code
- number of changes during requirements
- number of changes generated by the user vs. internal
- number of changes rejected/ total number of changes
- Change Report history profile

Fault Data Classes

Fault detection time – the phase or activity in which the fault was detected. Example subclasses: requirements, specification, design, code, unit test, system test, acceptance test, maintenance

Fault Density - number of faults per KLOC

Effort to Isolate/Fix - time taken to isolate of fix a fault usually in time intervals. Example subclasses: 1 hour or less, 1 hour to 1 day, 1 day to 3 days, more than 3 days

Omission/commission - where omission is neglecting to include some entity and commission is the inclusion of some incorrect executable statement or fact

Algorithmic fault - the problem with the algorithm. Example subclasses: control flow, interface, data <definition, <initialization, use>.

Failure Data Classes

Failure detection time - the phase or activity in which the failure was detected. Example subclasses: unit test, system test, acceptance test, operation

System Severity - the level of effect the failure has on the system. Example subclasses: operation stops completely, operation is significantly impacted, prevents full use of features but can be compensated, minor or cosmetic

Customer Impact - the level of effect the failure has on the customer. Example subclasses: usually similar to the subclasses for system severity but filled out from the customer perspective so the same failures may be categorized differently because of subjective implications and customer satisfaction issues

Sample Defect Metrics

- Number of faults per line of code
- Number of faults discovered during system test, acceptance test and one month, six months, one year after system release
- Ratio of faults in system test on this project to faults found after system test
- Number of severity 1 failures that are caused by faults of omission
- Percent of failures found during system test
- Percent of interface faults found by code reading

Code inspection statistics from AT&T

Measurements	*First sample project*	*Second sample project*
Number of inspections in sample	27	55
Total thousands of lines of code inspected	9.3	22.5
Average lines of code inspected (module size)	343	409
Average preparation rate (lines of code per hour)	194	121.9
Average inspection rate (lines of code per hour)	172	154.8
Total faults detected (observable and nonobservable) per thousands of lines of code	106	89.7
Percentage of reinspections	11	0.5

Yield calculation

Activity	Fault found	Faults injected					
		Design inspection	*Code*	*Code inspection*	*Compile*	*Test*	*Post-development*
Planning	0	2	2	2	2	2	2
Detailed design	0	2	4	5	5	6	6
Design inspection	4						
Code	2			2	7	10	12
Code inspection	3						
Compile	5						
Test	4						
Post-development	2						
TOTAL	20						
Design inspection yield		4/4 = 100%	4/6 = 67%	4/7 = 57.1%	4/7 = 57.1%	4/8 = 50%	4/8 = 50%
Code inspection yield				3/5 = 60%	3/10 = 30%	3/14 = 25.5%	3/16 = 18.8%
Total yield		4/4 = 100%	6/6 = 100%	9/9 = 100%	9/14 = 64.3%	9/16 = 56.3%	9/20 = 45%

Software Science

- Suppose we view a module or program as an encoding of an algorithm and seek some minimal coding of its functionality
- The model would be an abstraction of the smallest number of operators and operands (variables) necessary to compute a similar function
- And then the smallest number of bits necessary to encode those primitive operators and operands
- This model was proposed by Maurice Halstead as a means of approximating program size in the early 1970s. It is based upon algorithmic complexity theory concepts previously developed by Chaitin and Kolmogorov in the 1960s, as extensions to Shannon's work on information theory.

Algorithmic complexity-1

- The randomness (or algorithmic complexity) of a string is the minimal length of a program which can compute that string.
- Considered the 192 digits in the string *123456789101112…9899100*.
    - Its complexity is less than 192 since it can be described by the 27 character Pascal statement *for I:=1 to 100 do write(I)*.
    - But a random string of 192 digits would have no shorter encoding. The 192 characters in the string would be its own encoding.
    - Therefore the given 192 digits is less random and more structured than 192 random digits.

Algorithmic complexity-2

- If we increase the string to 5,888,896 digits *1234…9999991000000*, then we only need to marginally increase the program complexity from 27 to 31 *for I:=1 to 1000000 do write(I).*
    - The 5,888,704 additional digits only add 4 characters of complexity to the string.
    - It is by no means that 31 is even minimal. It is assuming a Pascal interpreter. The goal of Chaitin's research was to find the absolute minimal value for any sequence of data items.
- Halstead tried to apply these concepts to estimating program size.

Software Science

MEASURABLE PROPERTIES OF ALGORITHMS

n_1 = # Unique or distinct operators in an implementation

n_2 = # Unique or distinct operands in an implementation

N_1 = # Total usage of all operators

N_2 = # Total usage of all operands

$f_{1,j}$ = # Occurrences of the j^{th} most frequent operator $j = 1, 2,.. n_1$

$f_{2,j}$ = # Occurrences of the j^{th} most frequent operand $j = 1, 2,.. n_2$

THE VOCABULARY **n** IS $n = n_1 + n_2$

THE IMPLEMENTATION LENGTH IS $N = N_1 + N_2$

and

$$N_1 = \sum_{j=1}^{n_1} f_{1,j} \qquad N_2 = \sum_{j=1}^{n_2} f_{2,j} \qquad N = \sum_{i=1}^{2} \sum_{j=1}^{n\,i} f_{ij}$$

Example: Euclid's Algorithm

```
            IF (A = O)
LAST:       BEGIN
                GCD  := B;
                RETURN
                END;
            IF (B = O)
            BEGIN
                GCD  := A;
                RETURN
                END;
HERE:       G  := A/B;  R  := A - B * G;
            IF (R = O) GO TO LAST;
            A  := B; B  := R; GO TO HERE
```

Operator Parameters
Greatest Common Divisor Algorithm

OPERATOR	j	f_{1j}
;	1	9
: =	2	6
( ) or BEGIN...END	3	5
IF	4	3
=	5	3
/	6	1
-	7	1
x	8	1
GO TO HERE	9	1
GO TO LAST	10	1
	$n_1 = 10$	$N_1 = 31$

Operand Parameters
Greatest Common Divisor Algorithm

OPERAND	j	f_{2j}
B	1	6
A	2	5
O	3	3
R	4	3
G	5	2
GCD	6	2
	$n_2 = 6$	$N_2 = 21$

Software Science Metrics

PROGRAM LENGTH:

$n \sim N = n_1 \log_2 n_1 + n_2 \log_2 n_2$

N = The number of bits necessary to represent all things that exist in the program at least once

PROGRAM VOLUME: (Size of an implementation)

$V = N \log_2 n$

B = The number of bits necessary to represent the program

Software Science Metrics

PROGRAMMING EFFORT:

$E = V D = V/L = V^2/V$

E = The effort required to comprehend an implementation rather than produce it

E = A measure of program clarity

TIME:

$T = E/S = V/SL = V^2/SV$

T = the time to develop an algorithm

ESTIMATED BUGS:

$B = (LE)/E = V/E$

WHERE E = The mean effort between potential errors in programming

B = the number of errors expected in a program

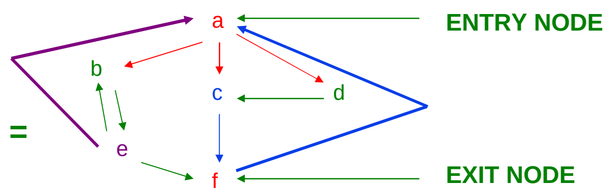
Cyclomatic Complexity -1

The **Cyclomatic Number** V (G) of a graph G with **n** vertices, **e** edges is

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

In a strongly connected graph G, the cyclomatic number is equal to the maximum number of linearly independent circuits

G =

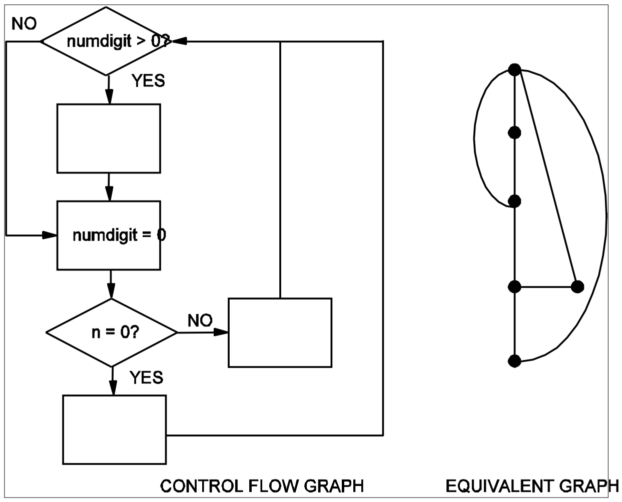


Cyclomatic Complexity -2

V(G) = 9 - 6 + 2 = 5 linearly independent circuits, e.g., (a b e f a), (b e b), (a b e a), (a c f a), (a d c f a)

Suppose we view a program as a directed graph, an abstraction of its flow of control, and then measure the complexity by computing the number of linearly independent paths, v(G)

Cyclomatic complexity



N=6
E=8
V=E-N+2=4

Properties of Cyclomatic Complexity

1) v(G) $\geq$ 1
2) v(G) = # linearly independent paths in G; it is the size of a basis set
3) Inserting or deleting functional statements to G does not affect v(G)
4) G has only one path iff v(G) = 1
5) Inserting a new edge in G increases v(G) by 1
6) v(G) depends only on the decision structure of G

For more than 1 component

v(M u A u B) = e - n + 2p = 13 - 13 + 2(3) = 6

For a collection of components

$v(C) = \Sigma_v (C_1)$ $\quad$ $C_k = u\ C_i$

Simplification

Θ = # Function nodes
Π = # Predicate nodes
Π = # Collecting nodes
THEN
$e = 1 + \Theta + 3\pi$
$n = \Theta + 2\pi + 2$
$v = e - n + 2$
YIELDS $v = (1 + \Theta + 3\pi) - (\Theta + 2\pi + 2) + 2 = \pi + 1$

I.E., v(G) of a structured program equals the number of predicate nodes plus 1

Simplification

THE RESULT GENERALIZES TO NONSTRUCTURED PROGRAMS

V(G) = NUMBER OF DECISIONS + 1

The concept of cyclomatic complexity is tied to complexity of testing program. As a metric, it is easy to compute. It has been well studied

McCABE RECOMMENDS A MAXIMUM V(G) OF **10** FOR ANY MODULE

How to use the models

The models should be an aid to software development management and engineering- not be taken as the sole source

An Approach

- First do a prediction
- Apply one or more models
- Examine the range of prediction offered by the model
- Compare the results

If they agree

- I can be more secure about the estimate

If they don't agree

- Examine why not
- What model assumptions did we not satisfy
- What makes this project different
- Am I comfortable with my explanation of the difference

Barry Boehm, Software Engineering Economics, Prentice Hall
Tom DeMarco, Controlling Software Projects, Yourdon Press

Key points

- There is not a simple relationship between the price charged for a system and its development costs.
- Factors affecting productivity include individual aptitude, domain experience, the development project, the project size, tool support and the working environment.
- Software may be priced to gain a contract and the functionality adjusted to the price.

Key points

- Different techniques of cost estimation should be used when estimating costs.
- The COCOMO model takes project, product, personnel and hardware attributes into account when predicting effort required.
- Algorithmic cost models support quantitative option analysis as they allow the costs of different options to be compared.
- The time to complete a project is not proportional to the number of people working on the project.