Change and Defect Models and Metrics

Changes can be categorized by

purpose	e.g., enhancement, adaptive, corrective, preventive
type	e.g., requirements, specification, design, architecture, planned enhancements, insert/delete debug code, improve clarity, optimize: space or time, feature, enhancement, bug
cause	e.g., market/external and internal needs
size	e.g., number of lines of code, number of components affected,
disposition	e.g., rejected as a change, not relevant, under consideration, being worked on, completed, saved for next enhancement

level of document changed e.g., changes back to requirements document

number of customers affected e.g., 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

Defect Definitions

- Errors

  Defects in the human thought process made while trying to understand given information, to solve problems, or to use methods and tools

- Faults

  Concrete manifestations of errors within the software

  - One error may cause several faults
  - Various errors may cause identical faults

- Failures

  Departures of the operational software system behavior from user expected requirements

  - A particular failure may be caused by several faults
  - Some faults may never cause a failure

  (difference between reliability and correctness)

- IEEE/Standard

Error Data Classes

Error origin - the phase or activity in which the misunderstanding took place. Example subclasses requirements, specification, design, code, unit test, system test, acceptance test, maintenance

Error domain - the object or domain of the misunderstanding. Example subclasses: application, problem-solution, notation <semantics, syntax>, environment, information management, clerical

Document error type - the form of misunderstanding contained in a document. Example subclasses: ambiguity, omission, inconsistency, incorrect fact, wrong section

Change effect - the number of modules changed to fix an error

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

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.

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.)

Function Points – Already discussed

- 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)

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

- 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.
- 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 X 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	$f_{2\,j}$
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:

$$n \sim \hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

$\hat{N}$ = The number of bits necessary to represent all things that exist in the program at least once

PROGRAM VOLUME: (Size of an implementation)

$V = \underset{\wedge}{N} \log_2 n$

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

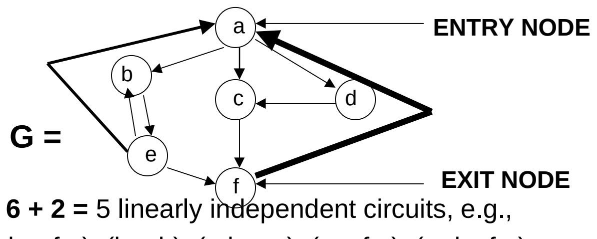
B = the number of errors expected in a program

Cyclomatic Complexity

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

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

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



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)

McCabe

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

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

For a collection of components

$v(C) = \Sigma_v (C_1) \qquad C = \cup C_i$

k

Simplification

Θ	= # Function nodes
Π	= # Predicate nodes
	= # Collecting nodes

THEN

$e = 1 + \Theta + 3\pi$

$n = \Theta + 2\pi + 2$

ASSUMING $p = 1$ AND $v = e - n + 2p$

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

Dynamic Measures

We can divide the dynamic product the characteristics into two basic classes

We can view them as checking on the

- Behavior of the input to the code, e.g., coverage metrics
- Behavior of the code itself, e.g., reliability metrics

Test Coverage Measures

Based upon checking what aspects of the product are affected by a set of inputs

For example,

- procedure coverage - which procedures are covered by the set of inputs
- statement coverage - which statements are covered by the set of inputs
- branch coverage - which parts of a decision node are covered by the set of inputs
- path coverage - which paths are covered by the set of inputs
- requirements section coverage - which parts if the requirements documents have been read

Used to,

- check the quality of a test suite
- support the generation of new test cases

Test Metrics

PASCAL

# TEST CASES:	**32**	**36**
SUBPROGRAM COVERAGE	**.81**	**.92**
BRANCH PATH COVERAGE	**.59**	**.67**
I/O COVERAGE	.23	**.54**
DO LOOP ENTRY	**.92**	**.94**
ASSIGNMENT	**.85**	**.91**
OTHER EXECUTABLE	**.74**	**.78**
CODE COVERAGE	**.70**	**.80**

FORTRAN

# TEST CASES:	**68**
SUBROUTINE COVERAGE	**.91**
FUNCTION COVERAGE	**1.00**
BRANCH PATH	**.63**
I/O	**.35**
DO-LOOP	**.74**
ASSIGNMENT	**.48**
OTHER EXECUTABLE	**.66**

$\rightarrow$ **ANOTHER PROGRAM**

STUCKI - BOEING

Reliability measures

TIME DEPENDENT APPROACHES

- TIME BETWEEN FAILURES (Musa Model)
- FAILURE COUNTS IN SPECIFIED INTERVALS (Goel/Okumoto)

TIME-INDEPENDENT APPROACHES

- ERROR SEEDING
- INPUT DOMAIN ANALYSIS

PROBLEMS WITH USE OF RELIABILITY MODELS

- LACK OF CLEAR UNDERSTANDING OF INHERENT STRENGTHS AND WEAKNESSES
- UNDERLYING ASSUMPTIONS AND OUTPUTS NOT FULLY UNDERSTOOD BY USER
- NOT ALL MODELS APPLICABLE TO ALL TESTING ENVIRONMENTS

Reliability measures - Musa

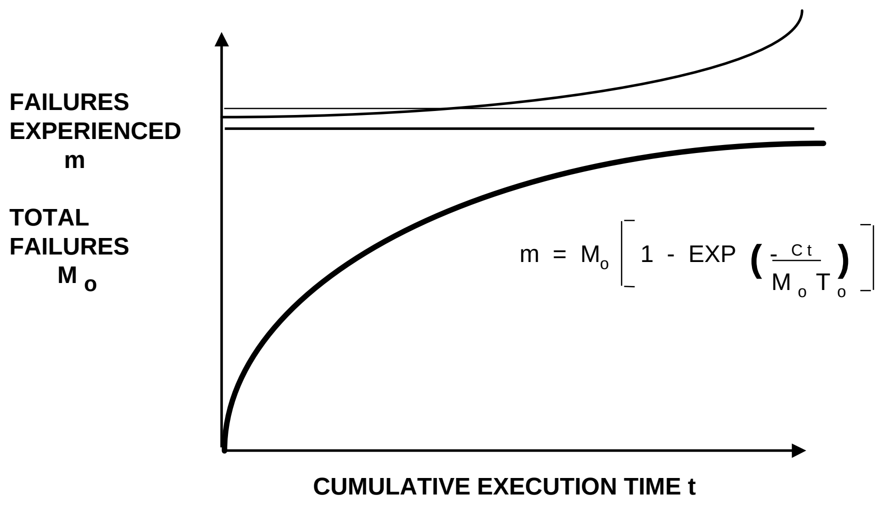
ASSUMPTIONS:

- ERRORS ARE DISTRIBUTED RANDOMLY THROUGH THE PROGRAM
- TESTING IS DONE WITH REPEATED RANDOM SELECTION FROM THE ENTIRE RANGE OF INPUT DATA
- THE ERROR DISCOVERY RATE IS PROPORTIONAL TO THE NUMBER OF ERRORS IN THE PROGRAM
- ALL FAILURES ARE TRACED TO THE ERRORS CAUSING THEM AND CORRECTED BEFORE TESTING RESUMES
- NO NEW ERRORS ARE INTRODUCED DURING DEBUGGING

$$T = \frac{1}{KE} e^{Kt}$$

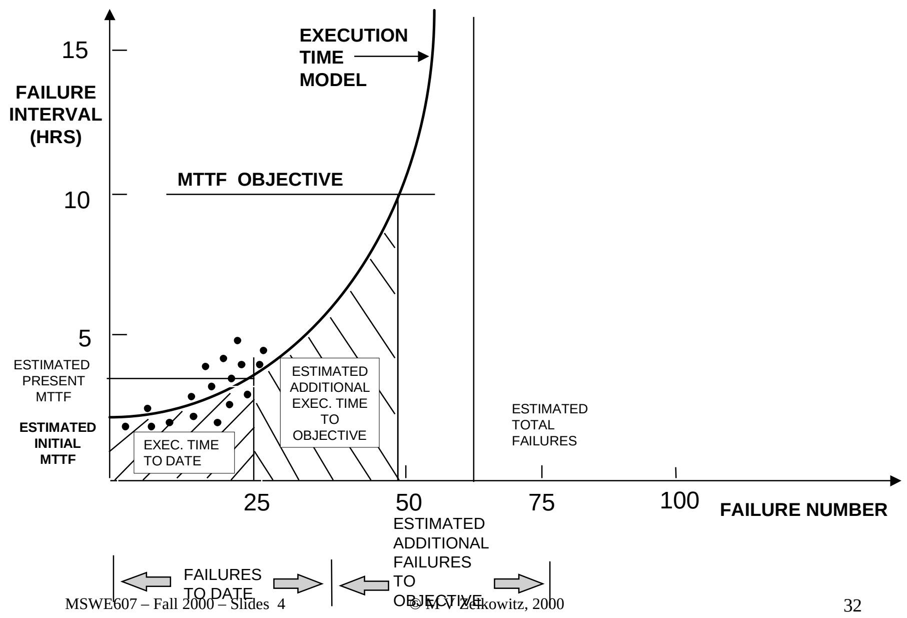
WHERE	**E**	IS TOTAL ERRORS IN THE SYSTEM
	t	IS THE ACCUMULATED RUN TIME (STARTS @ 0)
	T	IS THE MEAN TIME TO FAILURE

Reliability measures – Failure rate



$$m = M_o \left[1 - \mathrm{EXP} \left(\frac{-Ct}{M_o T_o} \right) \right]$$

Reliability measures – Using model



Reliability – Combining approaches

CLEAN ROOM

- DEVELOPER USES READING TECHNIQUES, TOP DOWN DEVELOPMENT
- TESTING DONE BY INDEPENDENT ORGANIZATION AT INCREMENTAL STEPS
- RELIABILITY MODEL USED TO PROVIDE DEVELOPER WITH QUALITY ASSESSMENT

FUNCTIONAL TESTING/COVERAGE METRICS

- USE FUNCTIONAL TESTING APPROACH
- COLLECT ERROR DISTRIBUTIONS, E.G., OMISSION vs COMMISSION
- OBTAIN COVERAGE METRICS
- KNOWING NUMBER OF ERRORS OF OMISSION, EXTRAPOLATE

ERROR ANALYSIS AND RELIABILITY MODELS

- ESTABLISH ERROR HISTORY FROM PREVIOUS PROJECTS
- DISTINGUISH SIMILARITIES AND DIFFERENCES TO CURRENT PROJECT
- DETERMINE PRIOR ERROR DISTRIBUTIONS FOR CURRENT PROJECT
- SELECT CLASS OF STOCHASTIC MODELS FOR CURRENT PROJECT
- UPDATE PRIOR DISTRIBUTIONS AND COMPARE ACTUAL DATA WITH THE PRIORS FOR THE CURRENT PROJECT

Additional Augustine Laws

Related to Metrics

- One tenth of the participants produce over one third of the output. Increasing the number of participants merely reduces the average output. (A variation on Brook’s Law-Adding people to speed up a late project just makes it later.)
- The last 10% of performance generates one third of the cost and two thirds of the problems.
- (Law of Unmitigated Optimism) Any task can be completed in only one-third more time than is currently estimated.
- (Law of Inconstancy of Time) A revised schedule is to business what a new season is to an athlete or a new canvas to an artist.
- Law of Propagation of Misery) If a sufficient number of management layers are superimposed on top of each other, it can be assured that disaster is not left to chance.
- VP of Boeing on 767 project: "is further ahead at the halfway point than any new airliner program in Boeing history."