Debugging

Dr. Mark C. Paulk

SE 4367 – Software Testing, Verification, Validation, and Quality Assurance

Topics: Software Testing

Part I: Preliminaries

1. Software Testing

- ***Humans, Errors, and Testing***
- ***Software Quality***
- ***Requirements, Behavior, and Correctness***
- ***Correctness vs Reliability***
- **Testing and Debugging**
- **Test Metrics**
- **Software and Hardware Testing**
- **Testing and Verification**
- **Defect Management**
- **Test Generation Strategies**
- **Static Testing**
- **Model-Based Testing and Model Checking**
- **Types of Testing**
- **Saturation Effect**
- **Principles of Testing**

Debugging

The process of determining the cause of failures and removing them.

- **The observation "People make mistakes resulting in defects that cause failures." leads to?**

Dijkstra's Observation

- ***If debugging is the process of removing bugs, then programming must be the process of putting them in.***

programming – The art of debugging a blank sheet of paper.
\- ***Robert Seacord***

Specifying Program Behavior

State transition diagrams can be used to specify program behavior.

The state of a program is the set of current values of all its variables and an indication of which statement in the program is to be executed next.

State can be encoded in a state vector.

Mathur, Example 1.6

Program P1.1

```
1 integer X, Y, Z;
2 input (X,Y);
3 if (X<Y)
4     {Z=Y;}
5 else
6     {Z=X;}
7 endif;
8 output (Z);
9 end;
```

State vectors

(Step X Y Z)

(1 u u u) →

- **what if we initialized variables at step 1 declaration?**

(2 u u u) →

- **variables are initialized and changed AFTER the line of code**

(3 3 15 u) →

(4 3 15 u) →

- **different from Mathur**

(5 3 15 15) →

- **is else a step? ~GOTO?**
- **endif? next statement is target!**

(8 3 15 15) →

(9 3 15 15)

Example, State Transition Diagrams

```
1)  integer X(3), Y, Z;
2)  input (X(1));
3)  X(2) := X(1) * 2;
4)  X(3) := X(2) * 3;
5)  Y := Z := 0;
6)  for i:=1,3 loop
7)      Y := Y + X(i);
8)      Z := Z + X(i) * X(i);
9)  end loop;
10) output (Y, Z);
11) end;
```

Create a set of state vectors to describe the behavior of this program.

Start with Step 1.

The input for X(1) should be 3.

1) integer X(3), Y, Z;	**(Step X_1 X_2 X_3 Y Z i) →**
2) input (X(1));	**(1 u u u u u u) →**
3) X(2) := X(1) * 2;	**(2 u u u u u u) →**
4) X(3) := X(2) * 3;	**(3 3 u u u u u) →**
5) Y := Z := 0;	**(4 3 6 u u u u) →**
6) for i:=1,3 loop	**(5 3 6 18 u u u) →**
7) Y := Y + X(i);	**(6 3 6 18 0 0 u) →**
8) Z := Z + X(i) * X(i);	**(7 3 6 18 0 0 1) →**
9) end loop;	**(8 3 6 18 3 0 1) →**
10) output (Y, Z);	**(9 3 6 18 3 9 1) →**
11) end;	**(6 3 6 18 3 9 2) →**

```
1)  integer X(3), Y, Z;
2)  input (X(1));
3)  X(2) := X(1) * 2;
4)  X(3) := X(2) * 3;
5)  Y := Z := 0;
6)  for i:=1,3 loop
7)      Y := Y + X(i);
8)      Z := Z + X(i) * X(i);
9)  end loop;
10) output (Y, Z);
11) end;
```

(Step X_1 X_2 X_3 Y Z i) $\rightarrow$

(7 3 6 18 3 9 2) $\rightarrow$

(8 3 6 18 <u>9</u> 9 2) $\rightarrow$

(9 3 6 18 9 <u>45</u> 2) $\rightarrow$

(6 3 6 18 9 45 <u>3</u>) $\rightarrow$

(7 3 6 18 9 45 3) $\rightarrow$

(8 3 6 18 <u>27</u> 45 3) $\rightarrow$

(9 3 6 18 27 <u>369</u> 3) $\rightarrow$

(6 3 6 18 27 369 <u>4</u>) $\rightarrow$

(10 3 6 18 27 369 <u>u</u>) $\rightarrow$

(11 3 6 18 27 369 u)

Oracles

The entity that performs the task of checking the correctness of the observed behavior.

May be a human tester.

- **error-prone**
- **slow**
- **may result in trivial checks**

Two Measurement Questions

Are we measuring the right thing?

- **Goal / Question / Metric (GQM)**
- **business objectives ⇔ data**
  - **cost (dollars, effort)**
  - **schedule (duration, effort)**
  - **functionality (size)**
  - **quality (defects)**

Are we measuring it right?

- **operational definitions**

Goals and Measures

One of the dangers in enterprises as complex as software engineering is that there are potentially so many things to measure…

In goal-driven measurement, the primary question is not

"What measures should I use?"

Rather, it is

"What do I want to know or learn?"

Goal-driven measurement is <u>not</u> based on a predefined set of measures.

Goal-Driven Measurement

Goal / Question / Metric (GQM) paradigm

- *V.R. Basili and D.M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, November 1984.*

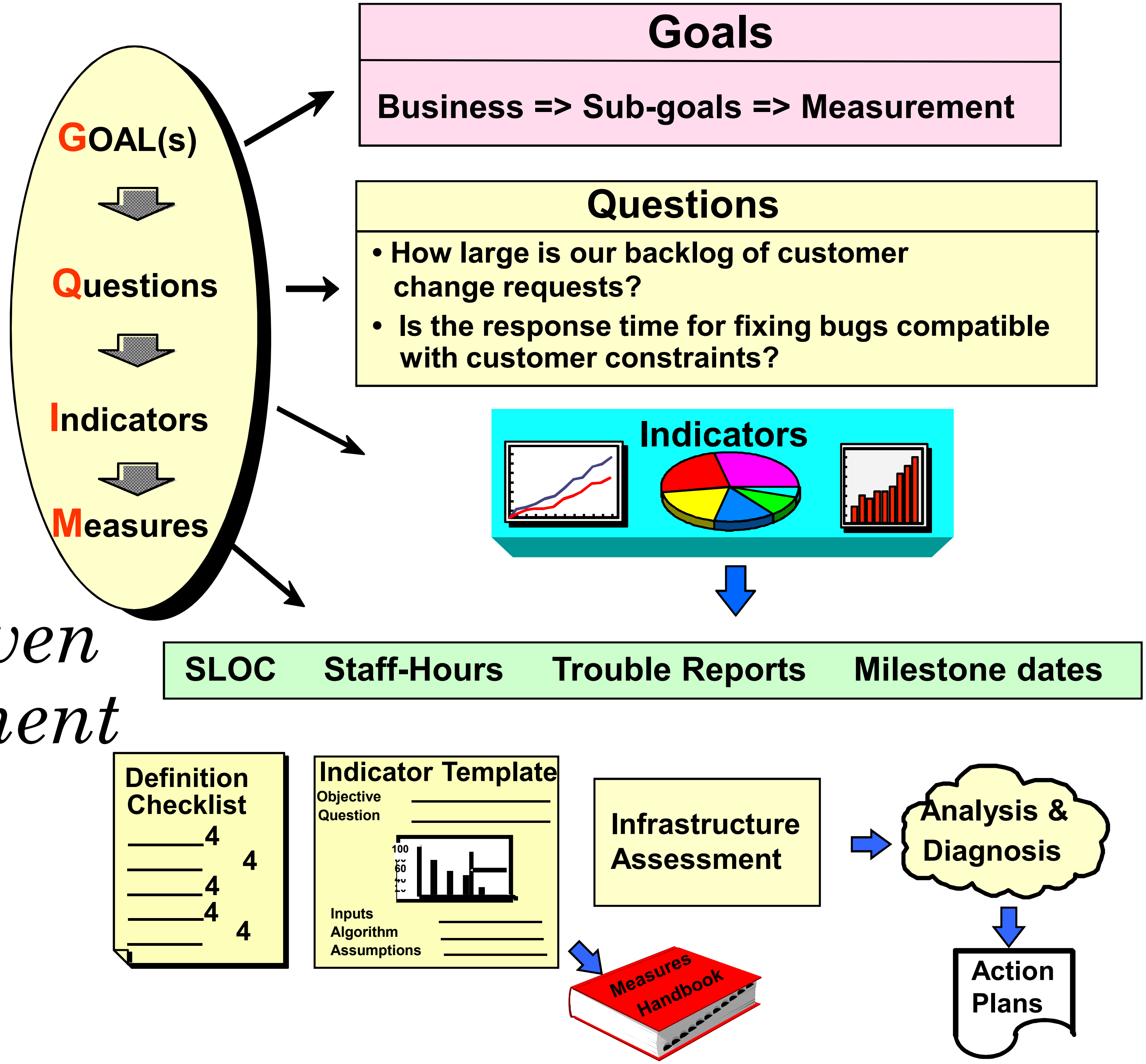
SEI variant: goal-driven measurement

- *Robert E. Park, Wolfhart B. Goethert, and William A. Florac, "Goal-Driven Software Measurement – A Guidebook," CMU/SEI-96-HB-002, August 1996.*

ISO 15939 and PSM variant: measurement information model

- *John McGarry, David Card, et al., <u>Practical Software Measurement: Objective Information for Decision Makers</u>, Addison-Wesley, Boston, MA, 2002.*

Goal-Driven Measurement



Define Operational Measures

An operational definition [is one] which reasonable men can agree on and do business with.

Shewhart believed his work on operational definitions to have been of greater importance than his development of the theory of variation and of the control chart.

There is no true value of anything.

Chapter 7 in Henry R. Neave, <u>The Deming Dimension</u>.

Operational Definitions

The rules and procedures used to capture and record data

What the reported values <u>include</u> and <u>exclude</u>

Operational definitions should meet two criteria

- ***Communication* – will others know what has been measured and what has been included and excluded?**
- ***Repeatability* – would others be able to repeat the measurements and get the same results?**

Examples – Operational Definitions

How is a line of code defined?

- **new, modified, deleted, retained function**

What are the time units?

- **hours vs minutes**

How is a defect defined?

- **severity, criticality, impact**

Is the data collected at the same point in the process each time?

- **peer review before or after compile / unit test**

Human Nature and Measurement

The act of measuring and analyzing will change behavior – potentially in dysfunctional ways.

Use of measurement data to evaluate individuals will negatively affect the correctness and usefulness of the measurement data that are reported.

The squeaky wheel gets the grease…

What gets measured gets attention…

Hawthorne Effect

The act of measuring (paying attention) will change behavior.

- **self-interested behavior on the part of the "measured entity!"**
- **motivational use of measurement (Austin)**

Is the Hawthorne effect bad?

Isn't the intention to change behavior?

Is the change "systematic?" Will it last?

Will management continue to "pay attention?"

Dysfunctional Behavior

Austin's <u>Measuring and Managing Performance in Organizations</u>

- **motivational versus information measurement**

Deming strongly opposed performance measurement, merit ratings, management by objectives, etc.

Dysfunctional behavior resulting from organizational measurement is inevitable unless

- **measures are made "perfect"**
- **motivational use impossible**

Test Metrics

Metric – a standard of measurement

- **syn: measure**

Organizational measures

Project measures

Process measures

Product measures

- **static**
- **dynamic**

Organizational Metrics

Useful in overall project planning and management

May be aggregated across multiple projects

Examples

- **number of defects reported after product release**
- **average defect density in testing**
- **test cost per KLOC**
- **delivery schedule slippage**
- **time to complete system testing**

Project Metrics

Useful in monitoring and controlling a project

Examples

- **ratio of actual to planned system test effort**
- **ratio of successful tests to total tests**

Process Metrics

Used for assessing the goodness of the process

Examples

- **defects found per phase**
  - **unit test, integration test, system test, acceptance test, after release**

Product Metrics

Useful for making decisions about the product

- **Is this product ready for release?**

Examples

- **McCabe's cyclomatic complexity**
- **Halstead complexity measures**
- **Chidamber and Kemerer OO metrics**

Static metrics can be computed without executing the program.

Dynamic metrics require code execution.

McCabe Cyclomatic Complexity

In the control flow graph for a procedure reachable from the main procedure containing

- **N nodes**
- **E edges**
- **p connected procedures**
  - **only procedures that are reachable from the main procedure**

V(G) = E – N + 2p

V(G) should be less than 10

- **Mathur recommends less than 5**
- **some suggest that 10-20 should be classified as “challenging”**

McCabe's p

p is the number of connected components.

McCabe defined a program control graph as having

- **unique entry and exit nodes**
- **all nodes reachable from the entry**
- **the exit reachable from all nodes**

resulting in all control graphs having only one connected component.

Subroutines lead to unconnected components.

- **may arrive from many different places**
- **may return to a variety of places**

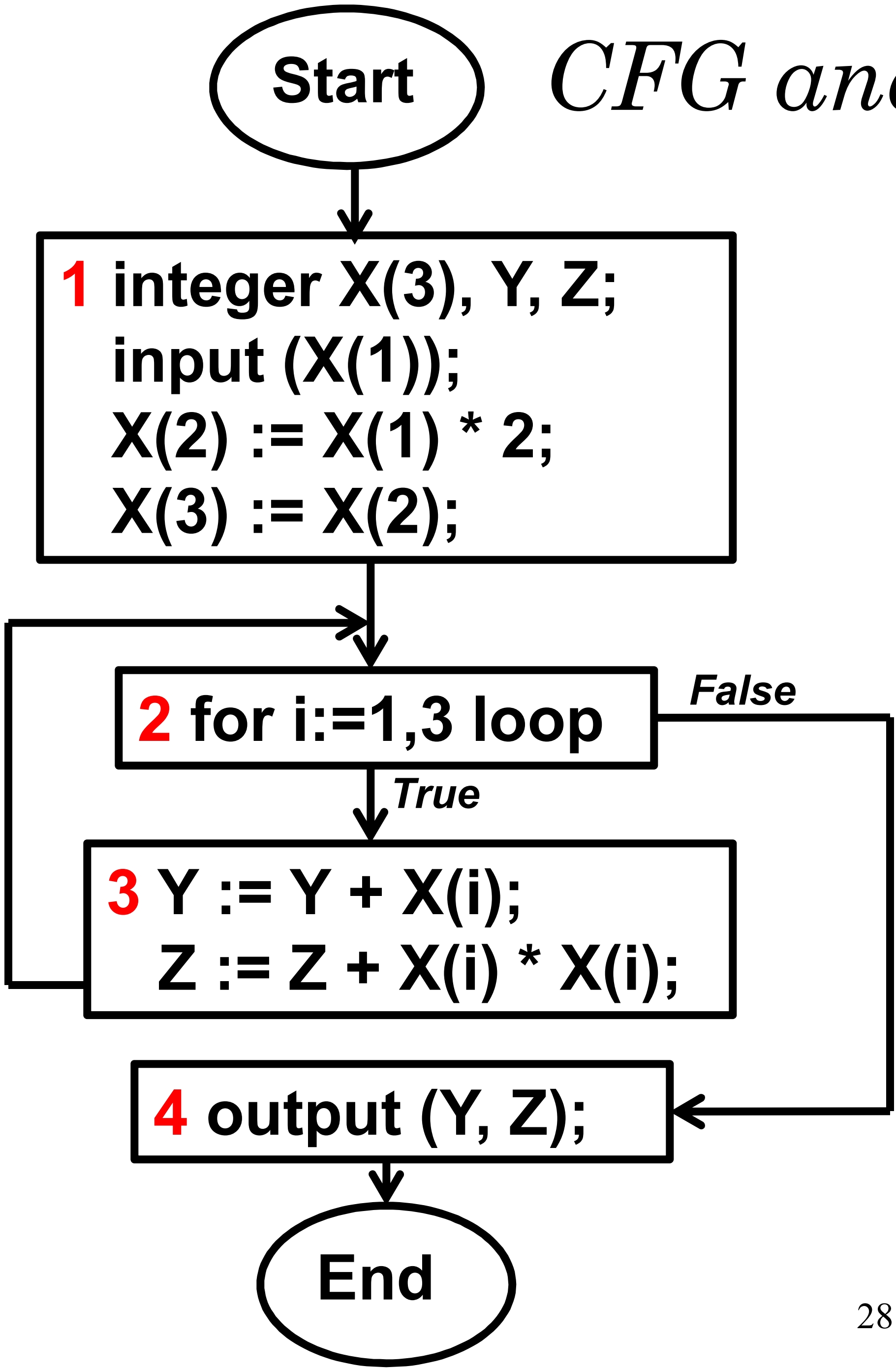
Using p

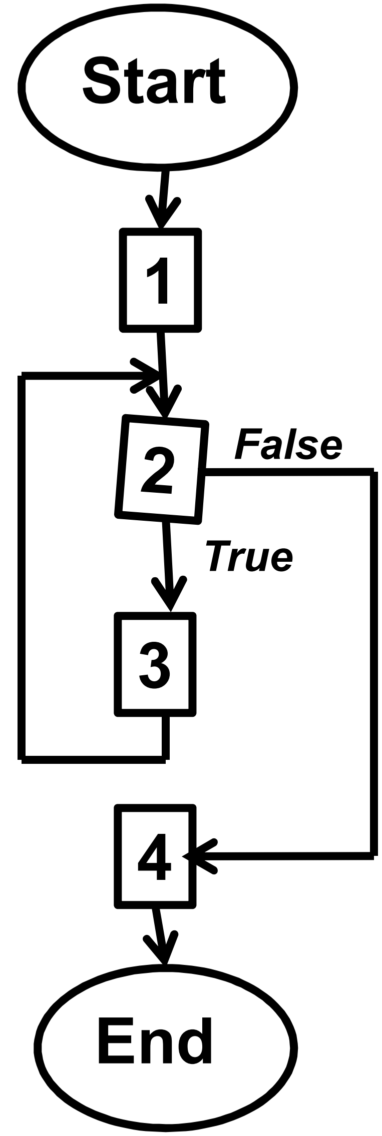
Does p allow analysis of a collection of programs?

- **programs with nested functions**
- **typically only look at a single program rather than a “library” with many disconnected routines**

Herraiz and Hassan (2011) use the maximum or average cyclomatic complexity for all functions in a file.

CFG and McCabe Example



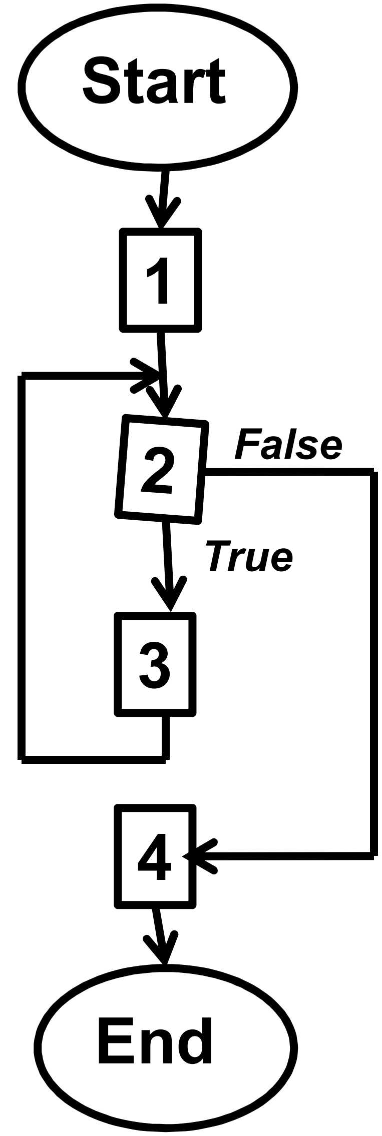


Nodes = {Start, 1, 2, 3, 4, End}
N = 6

Edges = {(Start,1), (1,2), (2,3), (3,2), (2,4), (4,End)}
E = 6

p = 1

V(G) = E – N + 2p
= 6 – 6 + 2(1) = 2



McCabe Cyclomatic Complexity for Structured Programs

A program with no conditions has a CFG with three nodes (including the Start and End nodes) and two edges and V(G) = 2 – 3 + 2(1) = 1.

The addition of each if-then-else statement increases the number of nodes (N) by 3 and edges (E) by 4.

The addition of each if-then or while statement increases the number of nodes (N) by 2 and the number of edges (E) by 3.

Thus the net increase in cyclomatic complexity is 1 for each decision in the program.

Halstead's Software Science

M.H. Halstead, <u>Elements of Software Science</u>, 1977.

N_1 number of operators in a program
N_2 number of operands in a program
η_1 number of unique operators in a program
η_2 number of unique operands in a program
η program vocabulary $= \eta_1 + \eta_2$
N program length $= N_1 + N_2$
V program volume $= N \times \log_2 \eta$
D difficulty $= (\eta_1 / 2) \times (N_2 / \eta_2)$ ***(Mathur text wrong!)***
E effort $= D \times V$
B number of delivered bugs $= V / 3000$
$= (E^{2/3}) / 3000$

Halstead Counts – Alternate Rules

Do not include {}; as operators (Mathur)

Count (), [], {} as one operator

- **begin/end are usually counted as two…**

Count if-then, begin-end, end if, end loop, etc., as a single operator

Count – (minus) as a sign separately from – as an operator

- **count – separately for variables but combine with constants as part of the constant**
- **count – as an operator in all cases**

Halstead's Number of Errors Estimator

Halstead's original formulas for B (<u>Elements of Software Science</u>, page 87) were

$$B = (E^{2/3}) / 3000$$

$$B = V / 3000$$

The formula provided by Mathur

$$B = 7.6 (E^{0.667}) (S^{0.333})$$

comes from Schneider, 1989.

Schneider's Formula for B

What is E? What is S?

You may have assumed that "S" was size, i.e., KSLOC.

- **Mathur does not define S**
- **S is the Stroud number (18) in Halstead's software science**
- **Schneider defines S as KSLOC**

If you read Schneider's paper, on eLearning, you would have also seen at the very beginning that his E is "overall reported months of programmer effort for the project."

Halstead Time

Halstead's E is in terms of discriminations per second

- **Stroud number is 18 discriminations / second**
  - **see the discussion of Halstead Time at http://www.virtualmachinery.com/sidebar2.htm**

One possible correction factor from Halstead's E to person months is 18 * 60 sec/min * 60 min/hr * 8 hr/day * 17 day/mon = 8,812,800

It is common to measure "Halstead time" in terms of minutes.

Halstead Time = E / (18 disc/sec * 60 sec/min)

Halstead Example

```
begin
integer X(3), Y, Z;
input (X(1));
X(2) := X(1) * 2;
X(3) := X(2);
for i:=1,3 loop {
   Y := Y + X(i);
   Z := Z + X(i) * X(i);
   }
output (Y, Z);
end;
```

Operands		
X	***////// ////***	**9**
3	***///***	**3**
Y	***////***	**4**
Z	***////***	**4**
1	***///***	**3**
2	***///***	**3**
i	***////***	**4**

		30

Unique operands = 7

```
begin
integer X(3), Y, Z;
input (X(1));
X(2) := X(1) * 2;
X(3) := X(2);
for i:=1,3 loop {
   Y := Y + X(i);
   Z := Z + X(i) * X(i);
   }
output (Y, Z);
end;
```

Operators

begin	**/**	**1**
integer	**/**	**1**
(	**///// ///// /**	**11**
)	**///// ///// /**	**11**
,	**////**	**4**
;	**///// ///**	**8**
input	**/**	**1**
:=	**/////**	**5**
*****	**//**	**2**
for	**/**	**1**
loop	**/**	**1**
{	**/**	**1**
+	**//**	**2**
}	**/**	**1**
output	**/**	**1**
end	**/**	**1**

		48

Unique operators = 16

N_1 operators = 48
N_2 operands = 30
η_1 unique operators = 16
η_2 unique operands = 7

$\eta = \eta_1 + \eta_2 = 23$
$N = N_1 + N_2 = 78$
$V = N \times \log_2 \eta = 353$
$D = (\eta_1 / 2) \times (N_2 / \eta_2) = 34$
$E = D \times V = 12{,}097$
$B = V / 3000 = 0.12$
$= 7.6\,(E^{2/3})\,(S^{1/3}) = 0.02$

Halstead Time = 11 min

```
begin
integer X(3), Y, Z;
input (X(1));
X(2) := X(1) * 2;
X(3) := X(2);
for i:=1,3 loop {
   Y := Y + X(i);
   Z := Z + X(i) * X(i);
   }
output (Y, Z);
end;
```

Object-Oriented Measures
(Chidamber and Kemerer 1994)

CBO (Coupling Between Objects)

- **number of other classes that a class is coupled to**

LCOM (Lack of Cohesion of Methods)

- **dissimilarities between methods by using attributes used in the methods**

NOC (Number of Children)

- **number classes that directly inherit one class**

DIT (Depth of Inheritance)

- **maximum number of nodes between root and lowest node in the hierarchy**

WMC (Weighted Methods per Class)

- **counting the implemented methods in a class**

RFC (Response for a Class)

- **number of methods a class is accessible to, including methods implemented in own class as well as methods accessible due to inheritance**

Similarities and Differences Hardware vs Software Testing

Software does not degrade over time.

Hardware testers use a variety of fault models at different levels of abstraction.

- **Similarly, software testers use mutation testing, condition testing, finite-state model based testing, combinatorial designs, etc.**

Hardware testers use bit patterns.

Comprehensive test coverage is impractical for hardware or software testing.

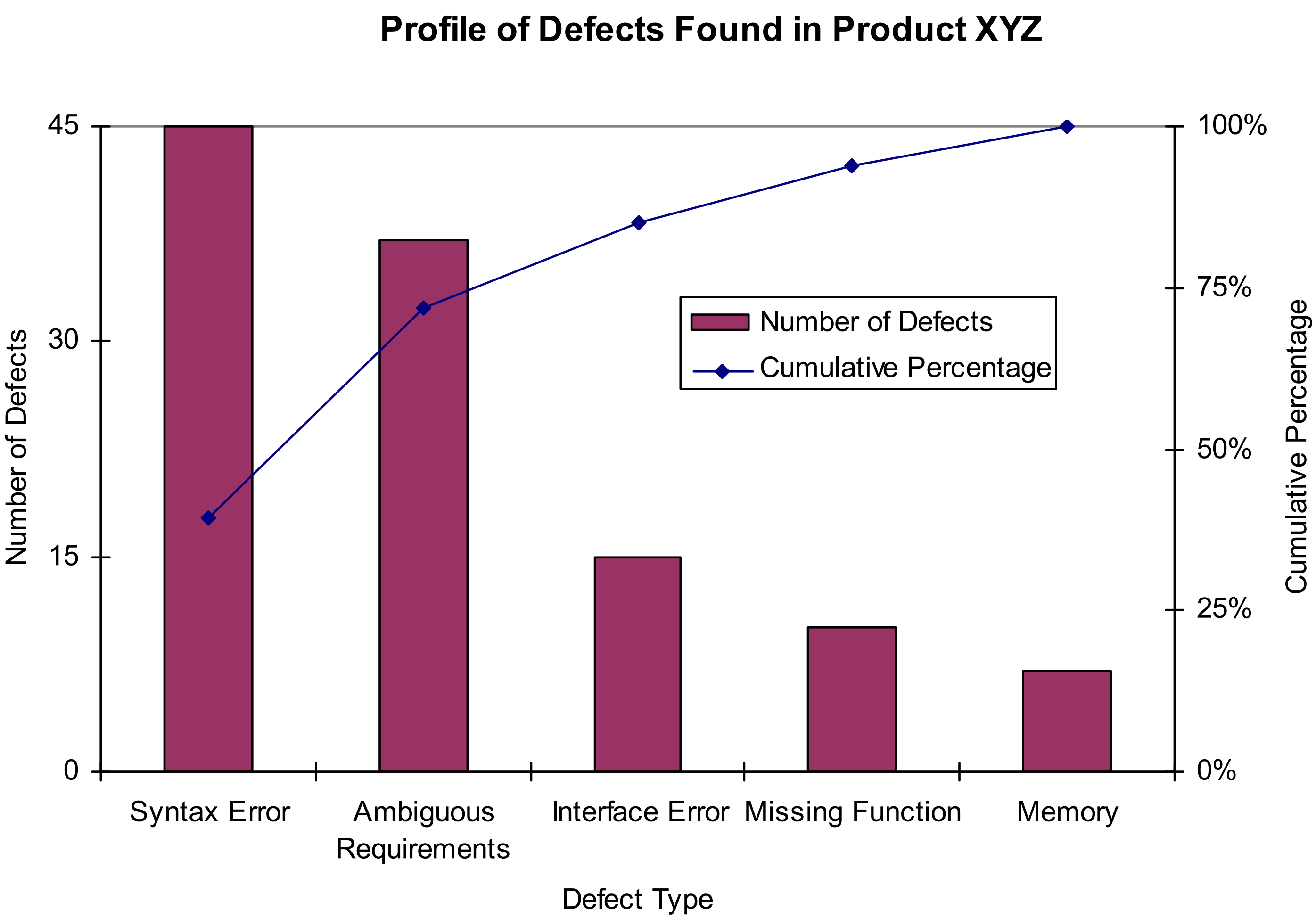
Pareto Charts

Special form of a bar chart.

Interpretation based on the "80/20 rule."

Help focus investigations by ranking problems, causes, or actions in terms of their amounts, frequency of occurrence, or economic consequences.

Pareto Chart Example



About Pareto Charts

What if the 80/20 rule does not apply?

If not, you will see a "flat Pareto."

Possible actions to consider

- **counting a different attribute, while maintaining the same stratification**
- **re-stratifying - use a different classification scheme**
- **use a different attribute of the process under study**

Orthogonal Defect Classification

A taxonomy for defect types

- **documentation**
- **syntax**
- **build, package**
- **assignment**
- **interface**
- **checking**
- **data**
- **function**
- **system**
- **environment**

R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M.Y. Wong, "Orthogonal Defect Classification - A Concept for In-Process Measurements," IEEE Transactions on Software Engineering, November 1992.

Summary – Things to Remember

Goal-driven measurement

Operational definitions

Austin's motivational vs informational measurement

McCabe's cyclomatic complexity

Pareto charts – 80/20 rule

Questions and Answers