Software Quality Engineering

02. Introduction — Part B

Achiya Elyasaf

אוניברסיטת בן-גוריון בנגב
Ben-Gurion University of the Negev

הנדסת מערכות תוכנה ומידע
Software and Information Systems Engineering

In this lecture

- How to arrive at the relevant tests?
- How to evaluate and execute test suites?
- How to measure the software quality?

How to arrive at the relevant tests?

How to arrive at the relevant tests?

- Goal: generating effective tests at a lower cost
- To achieve this, test designers analyze the following sources:
  - Requirements and functional specifications
  - Source code
  - Input and output domains
  - Operational profile
  - Fault model

How to arrive at the relevant tests?

\ Requirements and functional specifications

\ Requirements specification describes the intended behavior of a system

\ Development lifecycle models define the nature and amount of requirements

- Waterfall: a design methodology, in which most of the requirements are defined initially, and then updated and more are added in multiple revisions
  - most of the requirements are captured at the beginning
- Agile (e.g. XP/Scrum):
  - few requirements are identified at the beginning

How to arrive at the relevant tests?

\ Requirements can be specified in various formats

\ informal

- extracted from various formatted files (e.g. text, figures)

\ semi-formal

- (e.g. Unified Modeling Language (UML)) or

\ formal (e.g., Linear Temporal Logic (LTL), Entity Relations Diagram (ERD)) manner

- Bluetooth spec: A 2822 pages PDF file

\ Test engineer must consider all requirements regardless of the lifecycle model or format to select the test cases

How to arrive at the relevant tests?

\ Source code

- \ Source code describes the actual behavior of the system
- \ Specifications can be implemented in various ways (e.g. sorting array can be implemented in multiple ways)
- \ Test cases should be designed based on the code

\ Input and output domains

- \ Some values in the input domain have special meaning and should be treated separately (e.g. 1/x => x!=0)
- \ Some values in the output domain have special meaning (e.g. returning NULL or an integer that is actually a symbol, -999)

How to arrive at the relevant tests?

\ Operational profile

- \ A quantitative measure of the system usage
- \ Main idea:
  - infer, from test results, the system reliability when it is in actual use
- \ Helps test engineers in selecting test cases (inputs) using samples of real system usage
- \ Example:
  - Collect web-application usage data
  - Assign probabilities to test cases inputs according to their real-life usage

\ Fault model

- \ Design new test cases from previously encountered faults
  - Error guessing: using past experience to guess where faults might exist

Can developers arrive at the relevant tests?

- In some modern companies, development teams are responsible for more than just development, a.k.a “shift left”.
- In such companies,
  - the development teams are more aware of the business
  - the implementation is driven not only from the technical specification, but also from the business values
- Problems:
  - Complex interactions between system components are left out
  - Developers’ tests are too similar to the tested code

White Box versus Black Box Testing

\ **Black-box testing** (functional testing) treats software under test as a black-box without knowing its internals.

\ Tests are using software interfaces and trying to ensure that they work as expected

\ Usually done by testers

- **White-box testing** (structural testing) looks inside the software that is being tested and uses that information as part of the testing process
  - Usually done by programmers

White Box vs. Black Box Testing

White box:

- \ Efficient in finding errors and problems
- \ Source code familiarity is beneficial for thorough testing
- \ Programmers introspection
- \ Helps optimizing the code
- \ Maximal coverage
- \ Might not find unimplemented or missing features
- \ Requires knowledge of source code
- \ Requires code access

Black box:

- \ Efficient for large segments of code
- \ Code access is not required
- \ Separation between user's and developer's perspectives
- \ Limited coverage – only a fraction of test scenarios is performed
- \ Inefficient testing due to tester's lack of knowledge in the software internals

How to evaluate and execute test suites?

Test suite

Definition:

- Test suite = a collection of tests.
- Companies have different suites for various occasions (push to main, nightly, version, etc.)
- We will discuss in a future lesson how to define test suites.
- Today's topic — how to evaluate and execute them.

Discussion

- I wrote a test suite with 100 tests, are they good?
- How should I execute them?
- Do I need to look only at the results (fail/pass)? Or is there anything else?

How to Measure Test-Suite Quality?

\ Fault seeding (Bebugging): inject known faults into a program.
Assumption: if the test suite finds seeded faults then it is likely to find other faults

\ Mutation analysis (will be discussed later in the course):

- Altering the program to create mutants
- Test detects different behavior of a mutant from the original programs ("kills the mutant")
- Test suites are measured by the percentage of mutants they kill
- Mutation analysis helps in exposing weaknesses in test suites

\ Two last techniques involve altering the program

- Mutation testing is based on code alteration, whereas fault seeding on fault injection
- Fault seeding is less used due to high cost (time, knowledge, tools)

Test Planning and Design

\ Get ready and organized for test execution

\ Provides a framework, details of resource needed, effort estimation, schedule and a budget

\ Test objectives are identified from different sources

Monitoring and Measuring Execution (1)

\ Monitor the progress of testing and reveal the quality level of the system

\ Why?

\ Example: triggering a revert if system has too many defects

Monitoring and Measuring Execution (2)

\ Metrics for monitoring defects

- \ Coverage
- \ Number of test cases pass, fail, blocked, on hold
- \ Number of defects found, accepted, rejected, etc.
- \ Number of defects by priority (blocker, critical, normal, etc.)
- \ Number of defects found by teams/programmers

\ Using issue tracking tools helps…

Test Tools and Automation (1)

\ Why ?

\ Consistent execution of test cases

- Difficult to reproduce in manual testing
- Setting up the initial conditions of a system, makes it easier to reproduce test results

\ Reduced durations of the testing phases

- Automated test cases can be executed in an unsupervised manner at any time

\ Specific test cases

- Memory leaks detection
- Stress
- …

Test Tools and Automation (2)

\ Test automation costs.

  \ Environment setup and debug time

  \ Sometimes requires more manpower

\ Does not replace manual testing

How to measure the software quality?

SQ Measures: Conformance to Requirements

\ Defect Rate

\ Number of defects per, e.g.:

- 1,000 lines of code
- function points (the estimated functionality of the system, how many features, etc.)

\ Problem: How to count lines of code?

\ Reliability

\ Number of failures per *n* hours of operation

\ Mean time to failure

\ Probability of failure-free operation in a specified time

SQ Measures: Customer Satisfaction

- Measurement Tool
  - Customer Satisfaction Surveys
- Common Measure
  - Percent of satisfied or unsatisfied customers
- IBM CUPRIMDSO Model
  - **C**apability (Functionality)
  - **U**sability
  - **P**erformance
  - **R**eliability
  - **I**nstallability
  - **M**aintainability
  - **D**ocumentation / Information
  - **S**ervice
  - **O**verall satisfaction

Relationships of Software Attributes (Source: Kan, 2003)

	Capability	Usability	Performance	Reliability	Installability	Maintainability	Documentation	Availability
Capability								
Usability								
Performance	●	●						
Reliability	●	○	●					
Installability		○	○	○				
Maintainability	●	○	●	○				
Documentation	●	○				○		
Availability	●	○	○	○	○	○		

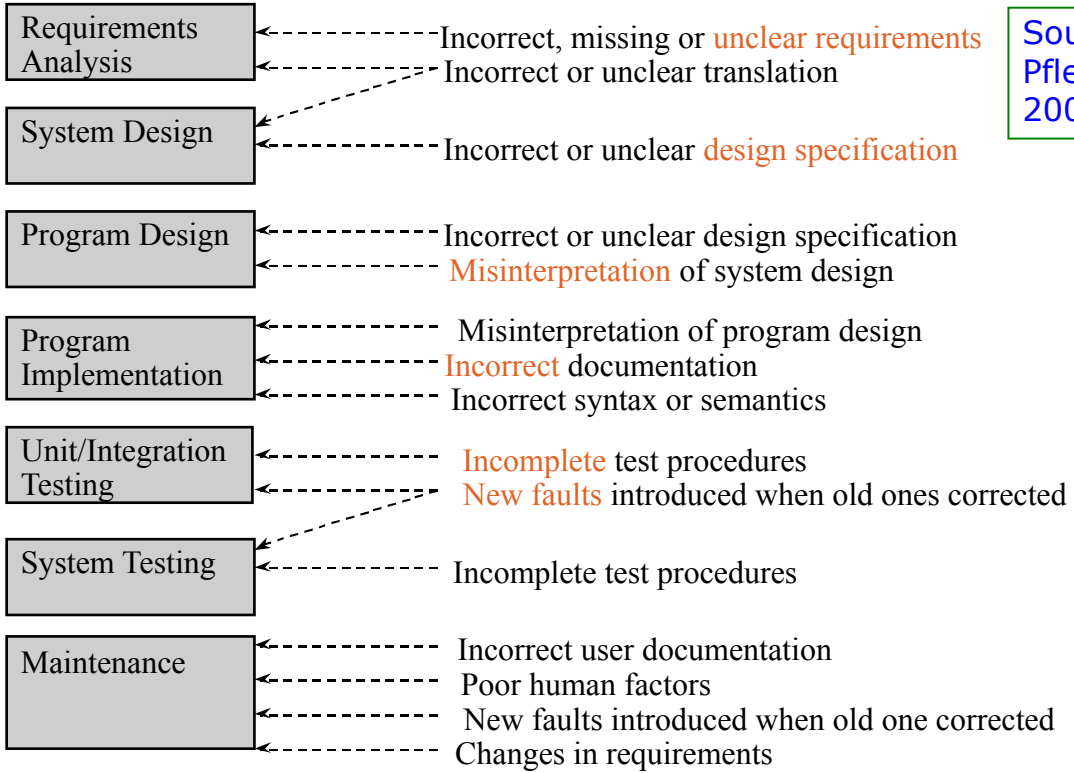
● Conflictive

○ Support One Another

Blank = None

FIGURE 1.1
Interrelationships of Software Attributes—A CUPRIMDA Example

Software Failures: What can go wrong?



Source: Pfleeger, 2001

Bug Types

\ The software does NOT do something that the specification says it should do

\ The software does something that the specification says it should NOT do

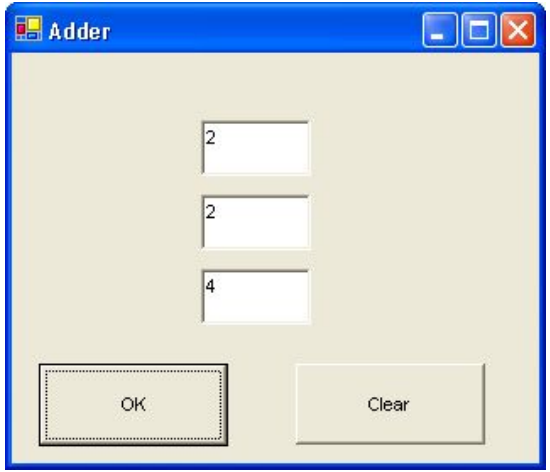
\ The software does something that the specification does NOT mention

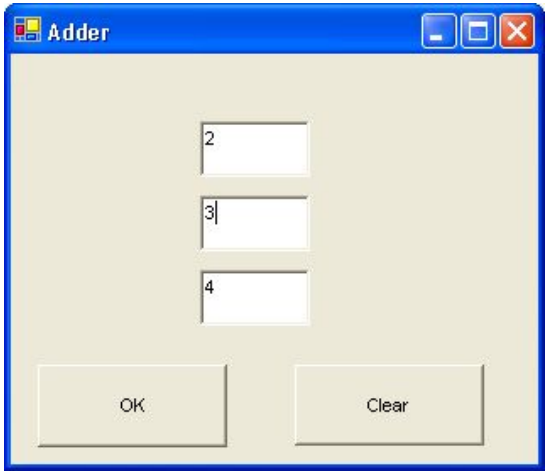
\ The software does NOT do something that the specification does NOT mention but should

\ The software is difficult to understand, hard to use, slow, or not right in the end user's view for any other reason

Case Study – The Adder

A Windows application, which adds two numbers entered by the user and shows the result after the user clicks “OK”





The “Adder” Testing

\ For each test case, please specify
  \ Pre-condition / initial state
  \ Input Event
    - Data entry
    - Action (open, close, tab, click on, etc.)
  \ Expected outputs (cursor position, displayed result, etc.)

\ Execute test cases
  \ Compare expected outputs to actual outputs

\ Perform visual testing
  \ Unique title
  \ Descriptive labels
  \ Text readability
  \ Spelling and grammatical errors

Some Bugs in the Adder application

\ The program shows integer results only

\ The program adds strings with strings, or strings with numbers

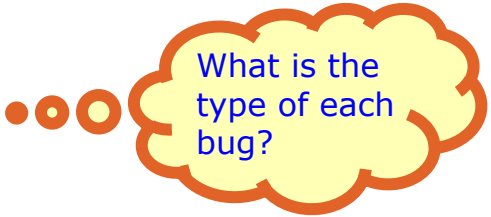
\ Blanks are interpreted as zeros

\ The program shows the previous result until OK is pressed (even if the input data was changed)

\ The addition result can be corrected manually by the user

\ There is no button for exiting the application

\ No help files, no onscreen instructions. Labels for each text box would be very helpful



Software Quality Assurance

“Trying to improve quality by increasing testing is like trying to lose weight by weighing yourself more often.”

S. McConnell, Code Complete

Software Quality Assurance vs. Software Testing

\ SQA Person Responsibility

  \ Create and enforce standards and methods to improve the development process and to prevent bugs from ever occurring

\ Software Tester Responsibility

  \ Find bugs as early as possible and make sure they get fixed

    - "Testers are measured by their skill as **investigators** and **communicators** and by the tools they can create and use to support their investigations—not by their level of control over the product's code or design" (C. Kaner, *IEEE Software*, July-August 2006)

Limitations of Testing

\ No program is error-free

- \ Average number of bugs: 1-3 per 100 statements
- \ 99% of errors are caught and fixed by the programmers
- \ Testing is looking for the remaining 1%

\ You can't test a program completely

- \ You can't test program's response to every possible input
- \ You can't test every possible output
- \ You can't test every path the program can take
- \ The software specification is subjective
  - Software quality is in the eye of the beholder

Limitations of Testing (cont.)

\ Testers are not verifying programs

- \ Testers are looking for problems
- \ A test that reveals a problem is a success
- \ Even “formal proofs” are subject to errors

\ Testing is an ill-defined problem

- \ You do not know where the bugs are or how many of them are left
- \ The testing team wants to maximize the number of discovered bugs under mostly unrealistic time and cost constraints

Limitations of Testing (cont.)

\ The more bugs are found, the more bugs there are

\ One bug may be an indicator of a “sloppy” piece of code

\ Programmers often make the same mistake

\ Several bugs may be caused by the same fundamental problem

\ The Pesticide Paradox (Boris Beizer, Software Testing Techniques, 1990)

\ Similar to applying the same pesticide, insects eventually build up resistance.

\ The same tests can only find the same bugs.

\ Moreover, the developers will go through the tests

\ But what about other bugs?

- Refresh and revise test materials regularly

Software quality in terms of quality factors and criteria

\ Quality factor: behavioral characteristic of a system.

- \ Correctness
- \ Reliability
- \ Testability
- \ Maintainability
- \ Reusability

\ Quality criteria: an attribute of a quality factor that is related to software development

- \ Modularity is an attribute of the system architecture
- \ A highly modular software allows designers to put cohesive components in one module, thereby improving the **maintainability** of the system

Software Metrics Categories

\ Product Metrics
    \ Complexity
    \ Design features
    \ Performance
    \ Quality level
\ Process Metrics
    \ Effectiveness of defect removal
    \ Testing defect arrival
    \ Response time of the fix process
\ Project Metrics
    \ Number of software developers
    \ Cost, schedule, and productivity

Product Related Metrics

\ Reliability Measures

    \ Safety critical systems

\ Defect density

    \ Commercial software

\ Customer problems

\ Customer satisfaction

Reliability Measures

\ How long a product can operate till failure? **(non repairable)**

\ Mean Time To Failure

\ How long a product works?

\ Mean Time Between Failures

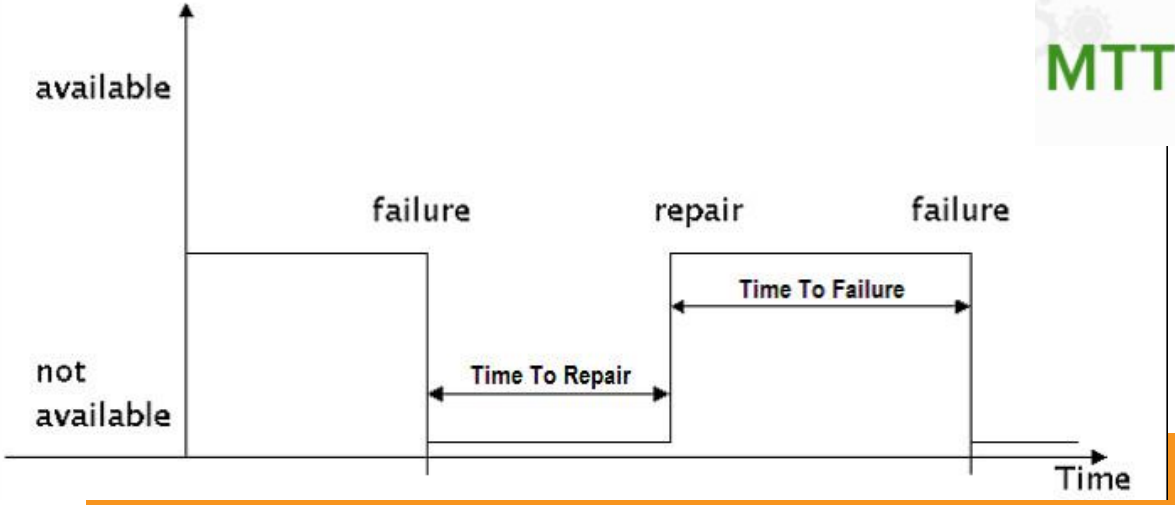
\ How long it takes to repair a product?

\ Mean Time To Repair

$$\text{MTTF} = \frac{\text{total hours of operation}}{\text{total number of units}}$$

$$\text{MTBF} = \frac{\text{total operational time}}{\text{total number of failures}}$$

$$\text{MTTR} = \frac{\text{total maintenance time}}{\text{total number of repairs}}$$



Defect Density

$$Defect\ Rate = \frac{Number\ of\ defects}{Opportunities\ for\ error\ (OFE)} \times K$$

Number of Defects – measured in a given time frame (e.g., one year, a lifetime of a product, etc.)

OFE – source: Lines of Code (LOC / SLOC), Function Points

K – constant (e.g., 1,000)

* limitation – a single error may include many source lines

Product Quality vs. Development Quality

\ The defect rate of the entire product is expected to improve (decrease) from release to release due to aging

\ The defect rate of the new and changed code stays the same, unless there is an improvement in the development process

Let's say we improved the development process. New bugs may be related to old code…

Product Quality vs. Development Quality

Methods for improving defect tracking

\ Change flagging method:

- \ New and changed lines of code are flagged with a specific identification (ID) number (using comments)
- \ The IDs are linked to the requirement numbers

\ Delta-library method:

- \ Original program modules are compared to the new versions in the current release library (no change-flagging is needed)

Example 1: Defect Tracking at IBM

\ Size Metrics

- \ *Shipped Source Instructions (SSI)*
- \ *Changed Source Instructions (CSI)*

\ SSI (current release) =

- \ SSI (previous release)
- \ + CSI (new and changed code instructions for current release)
- \ - deleted code (usually very small)
- \ - changed code (to avoid double count in both SSI and CSI)

Source Instructions ≠ LOC

Defect Rate Metrics (Example 1 Cont.)

1. Code quality of the total product
   - Total defects per KSSI (K=thousands)
2. Defect Rate in production (detected by customers)
   - Production defects per KSSI
3. Development Quality
   - Release-origin defects (production and internal) per KCSI
4. Development Quality per Defects Found by Customers
   - Release-origin production defects (production only) per KCSI

- **Questions:**
  - When metrics (1) and (3) are identical?
  - Which metrics are process measures?
  - Which metrics represent customer's perspective?

Shipped Source Instructions (SSI)

Changed Source Instructions (CSI)

From Customer’s Perspective

- Doesn’t care about defect rate for release-to-release (i.e., that between releases the defect rate improves)
- Does care about total number of defects that might affect their business

=> Good defect-rate target should lead to a release-to-release reduction in the total number of defects.

- If a new release is larger than its predecessors =>
  - the defect rate goal for the new and changed code has to be significantly better than that of the previous release in order to reduce the total number of defects.

Example 2: Customer's Perspective

\ Initial Release
- \ KCSI = KSSI = 50 KLOC
- \ Defects/KCSI = 2.0 // defect rate
- \ Total number of defects = 2.0 X 50 = 100

\ Second Release
- \ KCSI = 20
- \ KSSI = 50 + 20 (new and changed) – 4 (assuming 20% are changed LOC) = 66
- \ Defect/KCSI = 1.8 (10% improvement vs. Release 1)
- \ Total number of additional defects = 1.8 X 20 = 36

(customers experienced a 64% reduction [(100 - 36)/100] because the second release is smaller)

\ Third Release
- \ KCSI = 30
- \ KSSI = 66 + 30 (new and changed) – 6 (20% changed) = 90
- \ Targeted number of additional defects = 36 (not more than previous release)
- \ Defect rate target for the new and changed lines of code: 36/30 = 1.2 defects/KCSI or lower

Aspects of Software Size

\ Size:

\ One of the most useful attributes of a software product that can be measured without execution

\Can be described by ***length***, ***functionality***, and ***complexity***:

| ***Length*** is the physical product size (traditionally code length)

| ***Functionality*** is a measure of the functions supplied by the product to the user

| ***Complexity*** is a multi-faceted attribute which can be interpreted in multiple ways

Length

\ Length is the “physical size” of the product

\ In a software development effort, there are three major development products: **specification**, **design**, and **code**

\ The length of the specification can indicate the expected design length, which may predict the code length

\ Traditionally, code length refers to text-based code length

Length: Lines of Code

Variations of LOC:

- Count of physical lines, including blank lines
- Count of all lines, except blank lines and comments
- Count of all statements except comments
  - statements take more than one line count as only one line
- Count of all lines except blank lines, comments, declarations and headings
- Count of only executable statements, not including exception conditions.

LOC - Example 1

for (i = 0; i < 100; i++) printf("hello"); ***// How many lines of code?***

1. One physical Line of Code (LOC)

   for (i = 0; i < 100; i++) printf("hello");

2. Two Logical Lines of Code (LLOC) (for and printf statements)

```
for (i = 0; i < 100; i++)
        printf("hello");
```

3. One comment line

   // How many lines of code?

LOC – Example 2

```
/* Now how many lines of code is this? */
for (i = 0; i < 100; i++)
{
    printf("hello");
}
```

\ Five physical Lines of Code (LOC) - should placing brackets work to be estimated?

\ Two Logical Lines of Code (LLOC) - what about all the work writing non-statement lines?

\ One comment line - tools must account for all code and comments regardless of comment placement

LOC: Pros and Cons

Advantages of LOC

- \ Simple and automatically measurable
- \ Correlates with programming effort (& cost)

Disadvantage of LOC

- \ Vague definition
- \ Language dependent
- \ Not available for early planning
- \ Developers' skill dependent

Halstead's Theory

A program *P* is a collection of tokens, composed of two basic element types: ***operands*** and ***operators***

- ***Operands*** are variables, constants, addresses, function name (in definition)
  - i.e., a, 10, 20, 1100
- ***Operators*** are defined operations in a programming language (language constructs such as conditional, iterative, and procedural statements, , function name (when called))
  - i.e. include, main, if, (..)

1918-1979

Parameters in Halstead's Theory

\η_1 - number of distinct operators in the program

\η_2 - number of distinct operands in the program

\η - program vocabulary

$$\eta = \eta_1 + \eta_2$$

\ N_1 - total number of occurrences of operators in the program

\ N_2 - total number of occurrences of operands in the program

\ N - Program length - the total number of occurrences of operators and operands:
$N = N_1 + N_2$

Parameters in Halstead's Theory

\ Program volume represents the size to store the program in bits (using uniform binary encoding) is

$$V = N \log_2 \eta$$

\Program estimated length

$$\check{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

\Program Difficulty - difficulty to write or understand the program, e.g., in code review

$$D = (\eta_1/2) * (N_2/\eta_2)$$

(proportional to the number of the unique operator in the program)

Parameters in Halstead's Theory

\ Effort required to generate program *P*:

$$E = D \times V = (\eta_1/2) * (N_2/\eta_2) *(N*\log_2 \eta)$$

- measures the amount of mental activity required to translate the algorithm into implementation in the specified language

\ Time required for developing program *P* is the total effort E divided by the number of decisions per second

$$T = E/\beta \text{ (in seconds)}$$

\Halstead claims that $\beta = 18$

Parameters in Halstead's Theory

\ Remaining bugs: the number of expected delivered errors in the software at the delivery time

Originally $B=E^{2/3}/3000$, nowadays $B = V/3000$

\ Conclusion: bigger program needs more time to be developed and more bugs remained are expected

Example

For the following C program:

```
#include<stdio.h>
main()
{
     int a ;
     scanf (“%d”, &a);  if ( a > 10 )
     if ( a < 20 ) printf ("10 < a< 20 %d\n", a);
     elseif printf(“a>=20  %d\n”, a);
     else printf (“a<=10  %d\n”, a);
}
```

Example

For the following C program:

```
#include<stdio.h>

main()
{
     int a ;
     scanf (“%d”, &a);
     if ( a > 10 )
          if ( a < 20 )
               printf ("10 < a< 20 %d\n" , a);
          else
               if
                    printf(“a>=20  %d\n”, a);
               else
                    printf (“a<=10  %d\n”, a);
 }
```

Example - Operators

For the following C program:

```
main()
{
     int a ;
     scanf (“%d”, &a);
     if ( a >= 10 )
          If ( a < 20 )
               printf ("10 < a< 20 %d\n" , a);
          else
               if
                    printf(“a>=20  %d\n”, a);
               else
                    printf (“a<=10  %d\n”, a);
 }
```

Operators	Number of occurrences	Operators	Number of occurrences
#	1	<=	1
include	1	\n	3
stdio.h	1	printf	3
< … >	1	<	3
main	1	>=	2
(…)	7	if … else	3
{ … }	1	&	1
int	1	,	4
;	5	%d	4
scanf	1	“ … “	4
$\eta_1 = 20$		$N_1 = 47$	

Example - Operands

For the following C program:

```
#include<stdio.h>

main()
{
     int a ;
     scanf (“%d”, &a);
     if ( a >= 10 )
          If ( a < 20 )
               printf ("10 < a< 20 %d\n" , a);
          else
               if
                    printf(“a>=20  %d\n”, a);
               else
                    printf (“a<=10  %d\n”, a);
 }
```

Operands	Number of occurrences
a	10
10	3
20	3
$\eta_2 = 3$	$N_2 = 16$
$\eta_1 = 20$	$N_1 = 47$
Program length: $N = N_1 + N_2 = 63$	
Program Estimated length: $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 = 20 \log_2 20 + 3 \log_2 3 = 91.1934$	

Example 2

Given the following code:

```
1: read x,y,z;
2: type = “scalene”;
3: if (x == y or x == z or y == z) type =“isosceles”;
4: if (x == y and x == z) type =“equilateral”;
5: if (x >= y+z or y >= x+z or z >= x+y) type =“not a triangle”;
6: if (x <= 0 or y <= 0 or z <= 0) type =“bad inputs”;
7: print type;
```

Example 2

(c1) Number of distinct operators in the program

(c2) Number of distinct operands in the program

(c3) Program vocabulary

(c4) Total number of occurrences of operators in the program

(c5) Total number of occurrences of operands in the program

(c6) Program length

(c7) Program Volume

(c8) Program estimated length

(c9) Expected bugs

Example 2

Operators				Operands	
read	1	==	5	strings	5
,	2	or	6	x	9
;	7	and	1	y	8
“ … ”	6	>=	3	z	8
=	5	<=	3	0	3
if	4	+	3	type	6
( )	4	print	1		

Given the following code:

```
1: read x,y,z;
2: type = “scalene”;
3: if (x == y or x == z or y == z) type =“isosceles”;
4: if (x == y and x == z) type =“equilateral”;
5: if (x >= y+z or y >= x+z or z >= x+y) type =“not a triangle”;
6: if (x <= 0 or y <= 0 or z <= 0) type =“bad inputs”;
7: print type;
```

Example 2

(c1) Number of distinct operators in the program: $\eta_1 = 14$

(c2) Number of distinct operands in the program: $\eta_2 = 6$

(c3) Program vocabulary: $\eta = \eta_1 + \eta_2 = 20$

(c4) Total number of occurrences of operators in the program: $N_1 = 50$

(c5) Total number of occurrences of operands in the program: $N_2 = 39$

(c6) Program length: $N = N_1 + N_2 = 89$

(c7) Program Volume: $V = N \log_2 \eta = 89 \log_2 (20) = 384.651$

(c8) Program estimated length: $\check{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$

$= 14\log_2 14 + 6\log_2 6 = 68.81274$

(c9) Expected buts: $V/3000 = 0.128$

Limitations with Halstead's Theory

\ No standards for a line of code

\ Developed in the context of assembly languages and too fine grained for modern programming languages.

  \ Highly biased by software language, developer's skills and experience

\ The treatment of basic and derived measures is somehow confusing.

\ The notions of time to develop and remaining bugs are arguable.

\ Programming language and programmer skills are not incorporated.

Data-Driven Approach for Measuring Quality

- Halstead's used code properties for evaluating the quality.
- We now know that coding style can indicate the developer's level, and the repository text can even indicate the chances for bugs to appear.
- More on this in the last lecture.