SOFTWARE FAULTS REMOVAL

Ivan Mura, guest lecturer
Adjunct Associate Professor, Duke Kunshan University
Head of Data Science, Infinite Roots

DUKE KUNSHAN
Institute of Applied Physical Sciences and Engineering

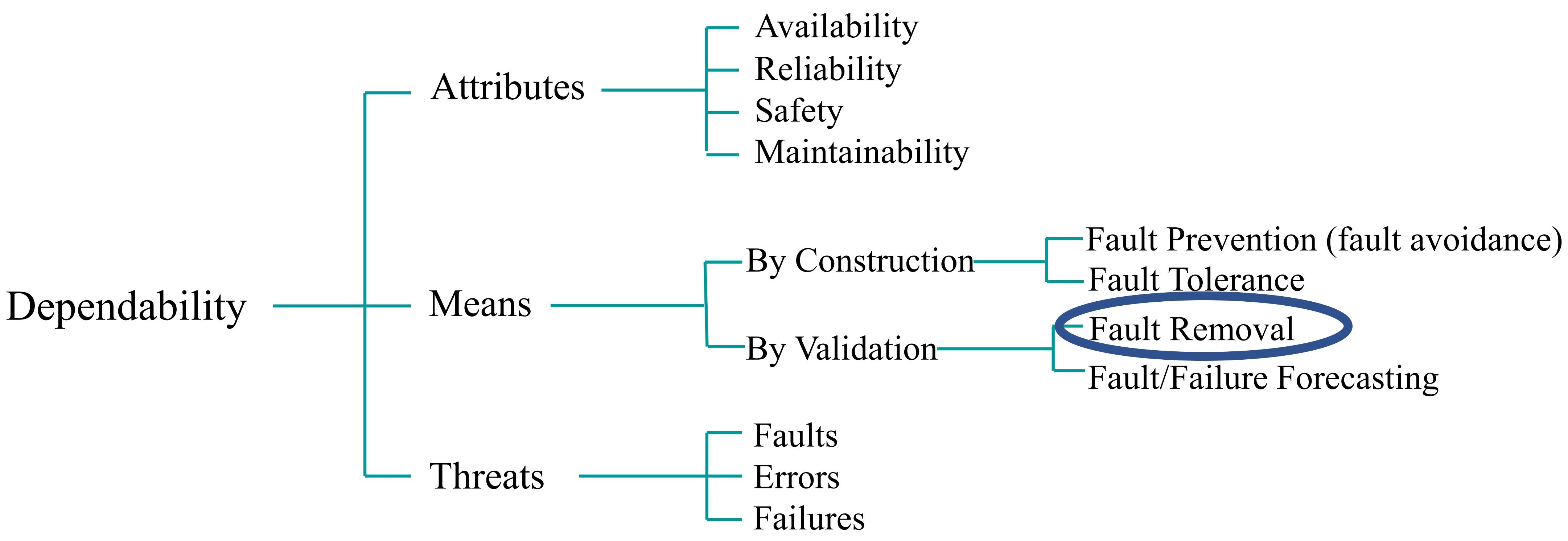
ERRARE HUMANUM EST . . .

Luc us Seneca, roman philosopher

- tut r Nero

PERSEVERARE AUTEM DIABOLICUM

So, you'd better remove those bugs!



The dependability tree: Avizienis, Laprie, Randall, 2001

How to eliminate a bug?

Tempt it: let it manifest!

Find it: see where it hides!

Kill it! Step on it!

How to remove a SW bug?

Tempt it: run a test case! **TEST**

Find it: locate faulty line!

DEBUG

Kill it! Change the line!

Software test definition

Definition of IEEE in 1983

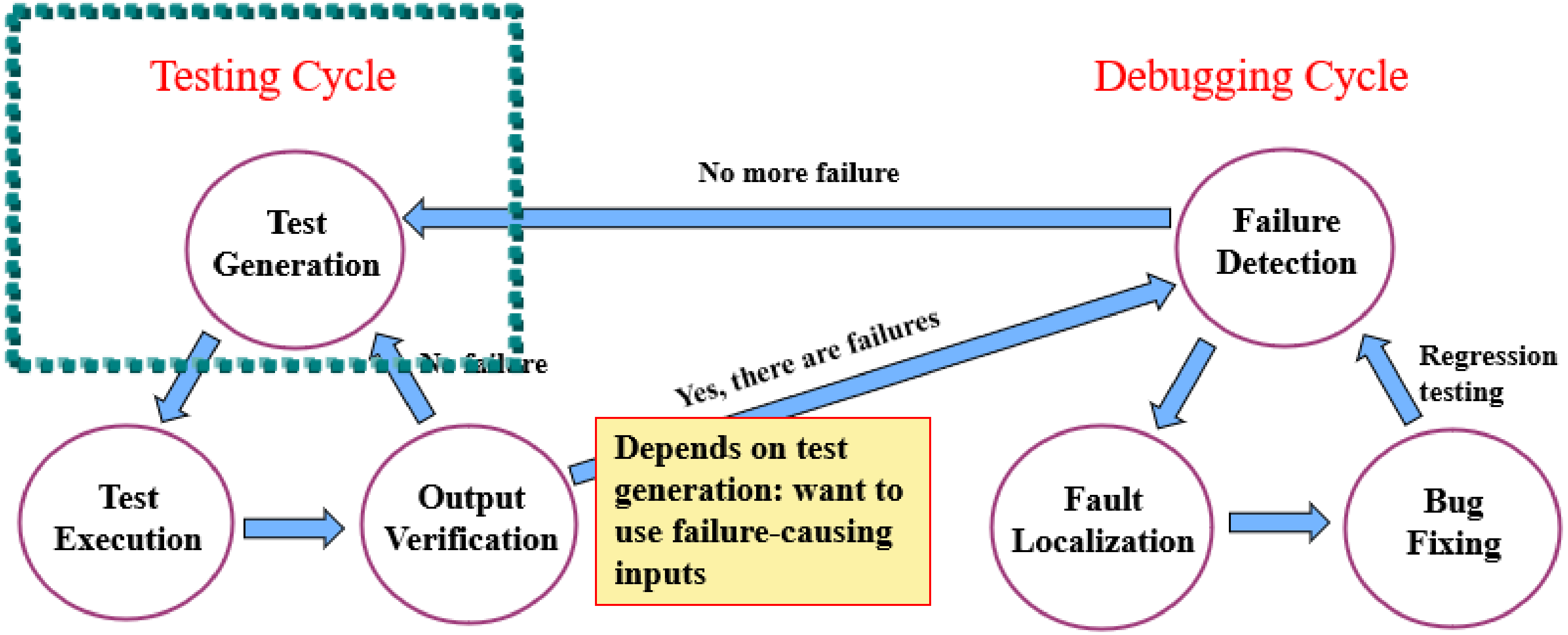
- The process of using a manual or automated means to run or test a system, the purpose of which is to verify that it meets the specified requirements or to clarify the difference between the expected and actual results.

Difference between testing and debugging

- Testing: A **planned**, **repeatable** process whose purpose is to check whether the system satisfies its requirements or to identify problems that are not in accordance with pre-defined specifications and standards.
- Debugging: A process that isolates (locates) and confirms the cause of the problem and then modifies the software to correct the problem (that is, find/locate and fix bugs).

The testing/debugging processes

- Testing and debugging activities constitute one of the most expensive aspects of software development
  - Often more than 50% of the cost



From:
Software Testing, Foundation and Path Forward

Professor W. Eric Wong (UT Dallas)

Example: testing and debugging – naïve

```
// This function returns 1 if its
// argument is a prime integer, and
// 0 if not.
int isPrime(int x) {
  if (x < 2) { // no prime less than 2
    return 0;
  }
  // try all possible divisors
  for (int i = 2; i < sqrt(i); i++) {
    if (x % i == 0) {
      return 0;
    }
  }
  // no divisors found, is prime
  return 1;
}
```

BAIT 1: x = 0

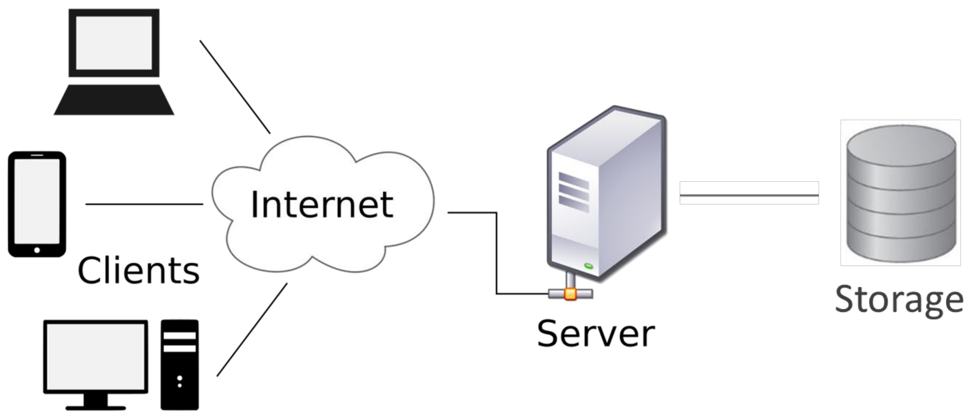
BAIT 2: x = 1

BAIT 3: x = 2

BAIT 4: x = 3

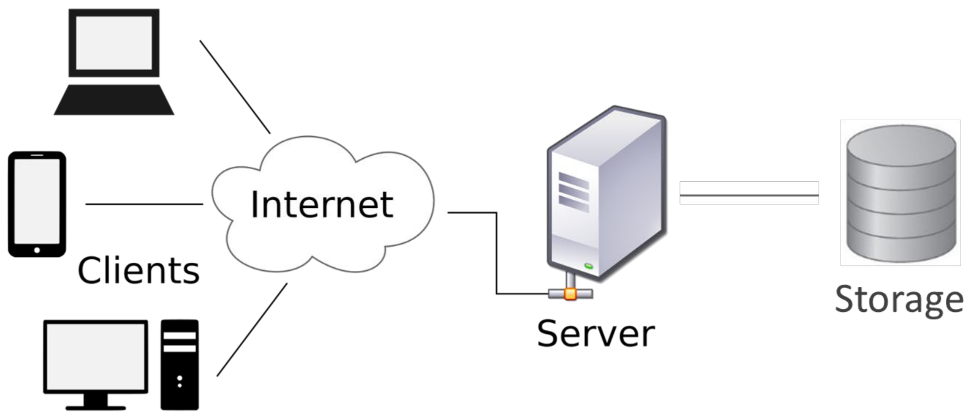
BAIT 5: x = 4

Testing and debugging (real world)



- When you run your tests, in an unpredictable way, the server freezes
  - You suspect an issue with the concurrent implementation (deadlock?)
  - It seems you need some specific order of requests from clients that are running on Android devices (an issue with the communication protocols?)
  - If so, you may change the way threads acquire resources, using trylock() rather than lock()
- How complex a test case for reproducing this sort of failure can be?
- How complex finding the software bug and removing it can be?

Other interesting ones, on the same system



- Your system has the following two requirements:
  - For the nominal workload case of 3,000 clients requests per hour, the 95% percentile of the response time of the server should not exceed 30 seconds
  - The system has to provide a service that is available 99.99% of the time (service down no more than 8.75 hours/year)
- How would you engineer test cases that can verify these requirements?

Conclusions from previous examples

Testing occurs at multiple levels of abstraction, with different purposes

Some requirements (non-functional ones) are in general more difficult to be verified

- Standard software testing may not be sufficient

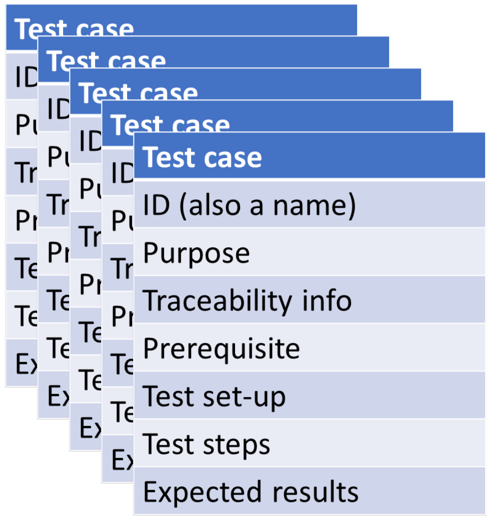
Good quality testing requires a significant amounts of resources

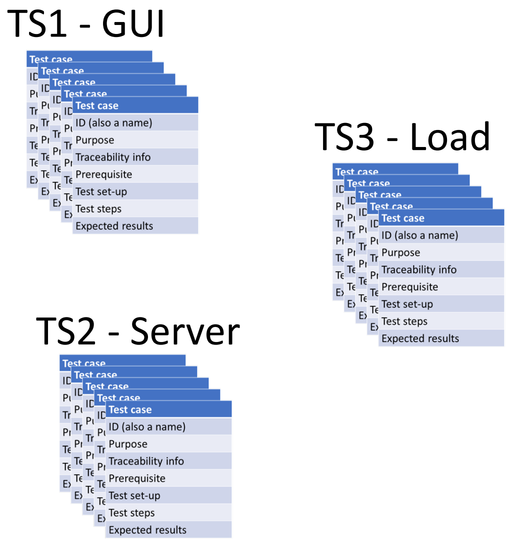
- Plan for testing, craft good test cases, execute and analyze their outcomes, maintain them

Definitions (ISO/IEC/IEEE 29119)

Test case
ID (also a name)
Purpose
Traceability info
Prerequisite
Test set-up
Test steps
Expected results

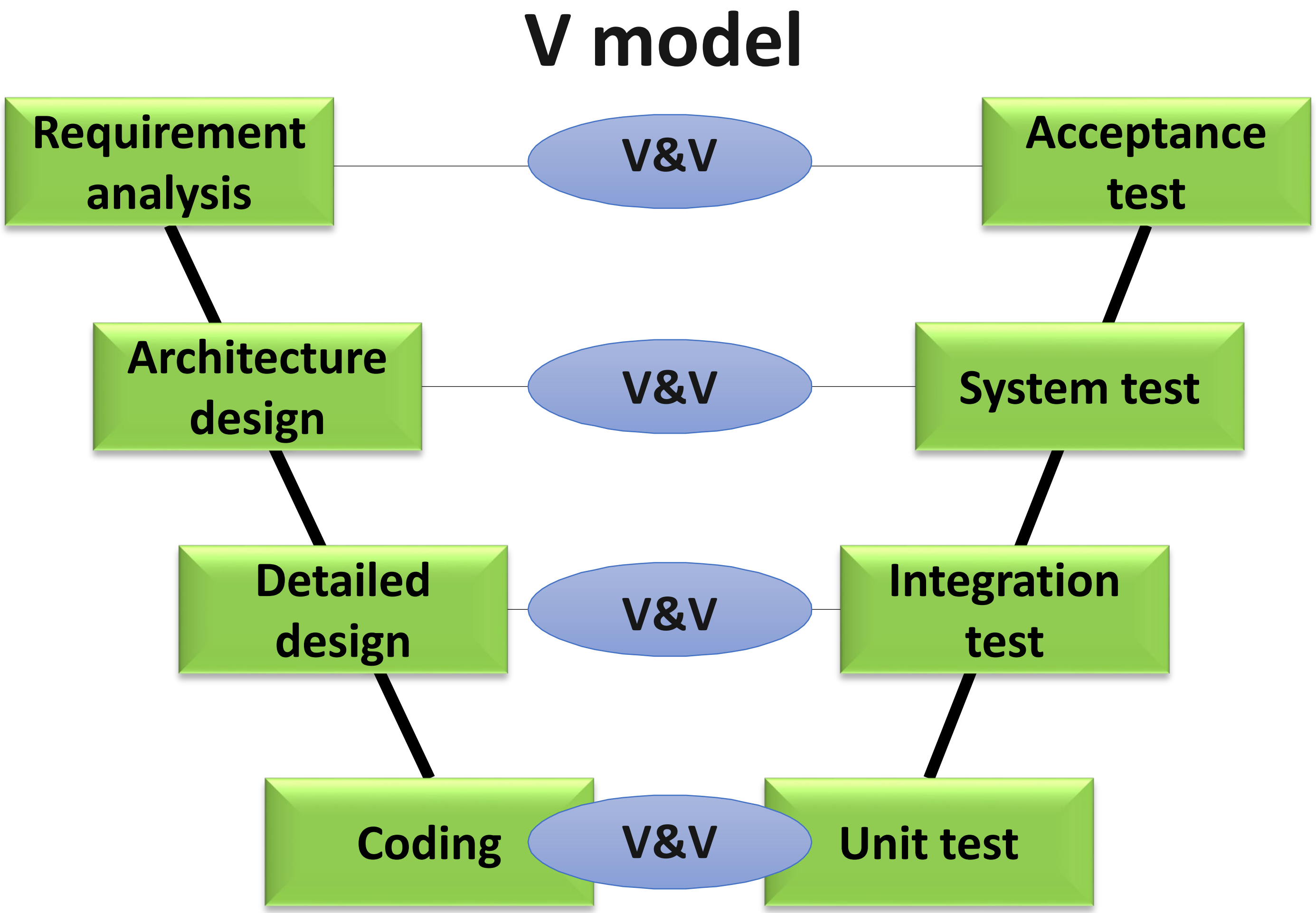
Test suite





Beyond Unit Testing

Testing in real SW projects is structured in multiple phases throughout the development process



Sad jokes about testing

Testing is very easy, and everyone can do it.

If you cannot do programming, you will do testing, and if you cannot do testing, you give talks on software testing!

The more testing, the better!

From:
Software Testing, Foundation and Path Forward

Professor W. Eric Wong (UT Dallas)

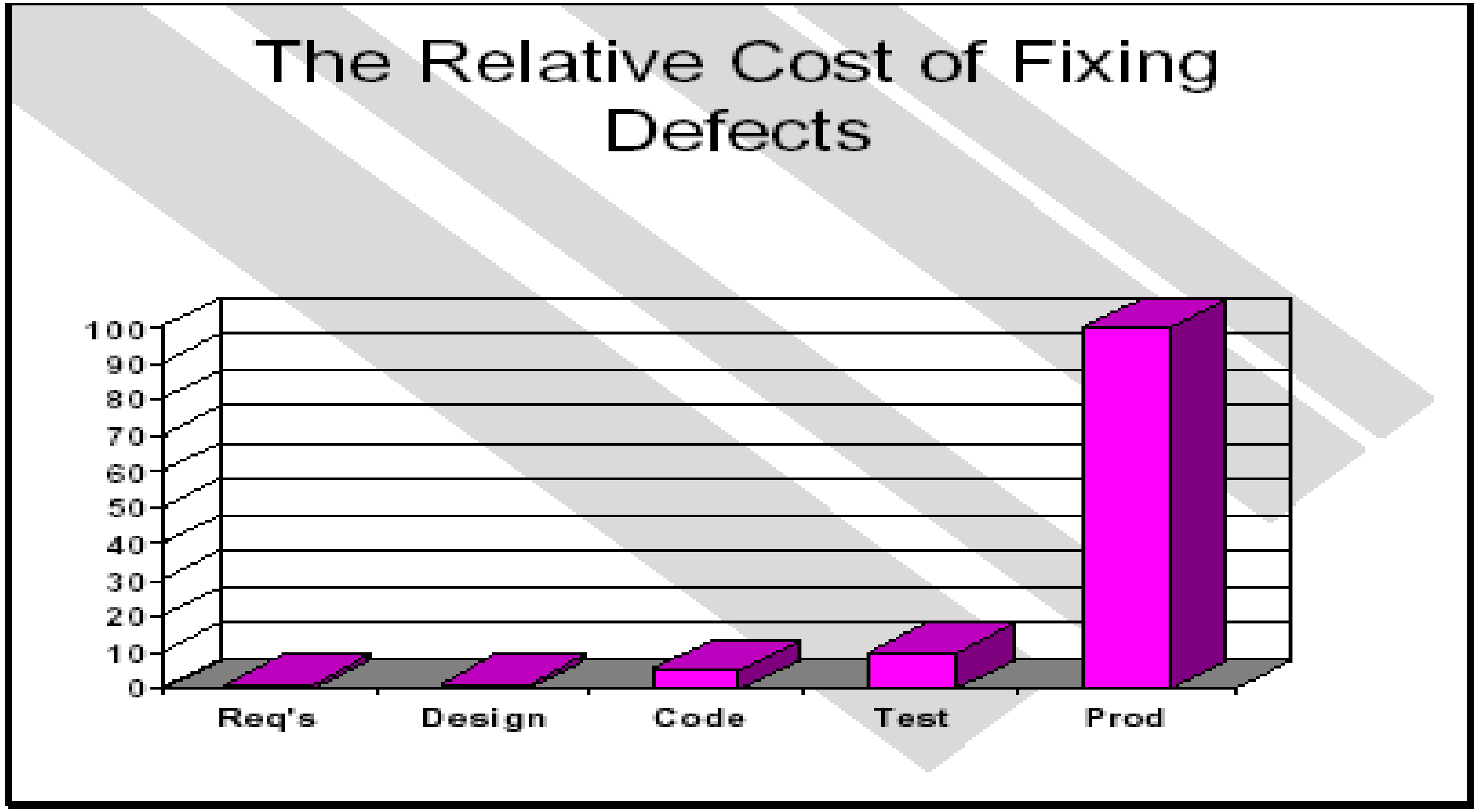
Testing is a part of Software Quality Assurance

SQA: ensuring software development is complying with defined standards

- Testing has several important roles
  - Verify the software meets its specifications
  - Helps fault removal during software development (increase confidence)
  - Provide input data for predictive models that estimate residual faults, and more advanced models of software reliability growth

When to test?

The earlier the better

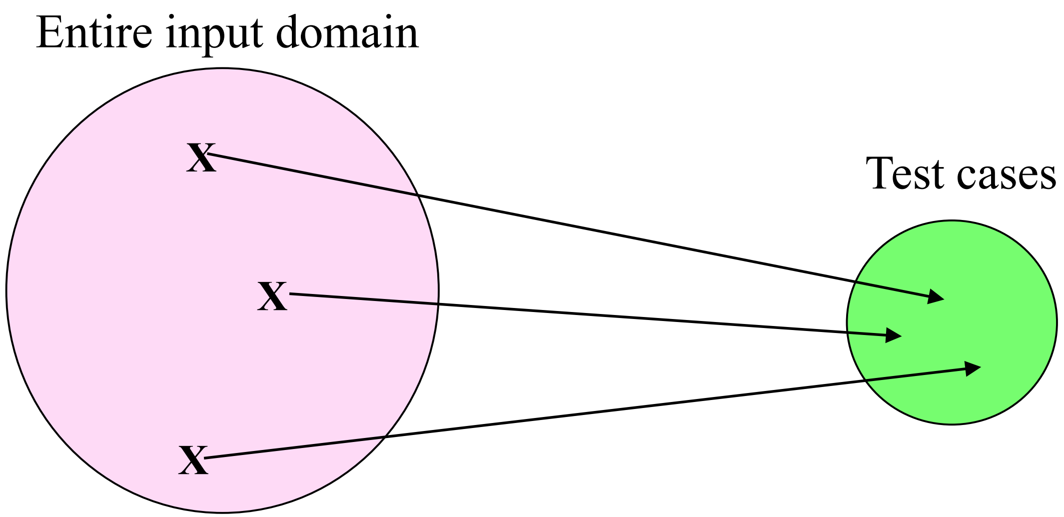


Testing towards more reliable software

Postulate	Testing shows the presence, not the absence, of bugs (E. Dijkstra)
Corollary	It is impossible to verify that software is fault free only by testing

- Fault removal: a continuous process, during specification, design, development and operation
- When to stop?

When to stop testing?



Do we know

- Which inputs are failure causing?
- How many failure causing inputs exist?

Adapted from:
Software Testing, Foundation and Path Forward

Professor W. Eric Wong (UT Dallas)

- Very pragmatically, we stop when quality gate exit criteria are met!

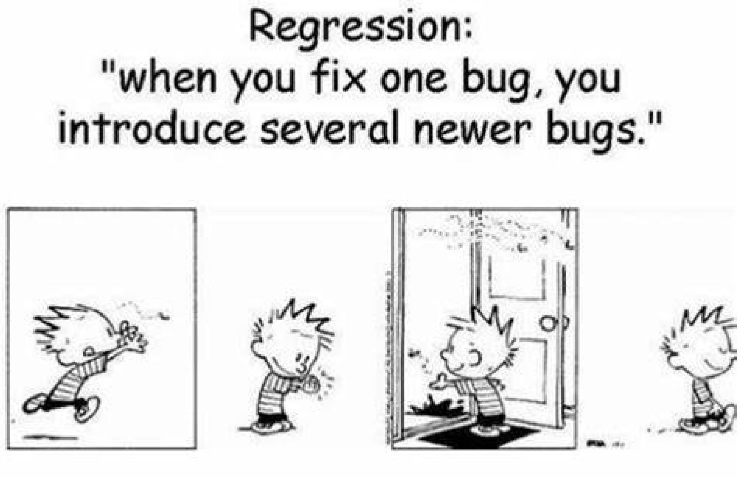
In software development industry

- Software becomes ready for release when it passes ***quality gates***
- A typical quality gate of a testing phase specifies the maximal number of allowable known defects, for instance
  - 0 SEVERE defects
  - 0 MAJOR defects
  - <=5 MINOR defects
  - <=10 COSMETIC defects
- As well as completion of other process related steps (documenting)
  - Detailed information about known defects, i.e. their impact on functionality and how to mitigate it, is released together with the SW (*release notes*)

Are residual SW faults going to be fixed?

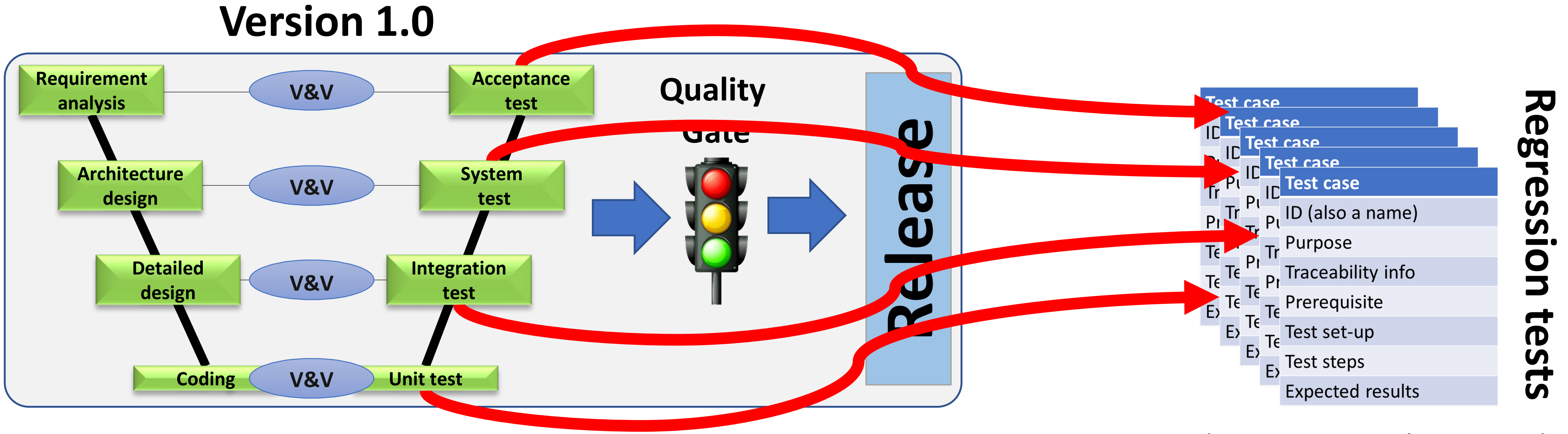
- Possibly, in next releases of the product
- Usually, the development of the next versions considers new functionality as well as the fix of a part of known defects backlog
- In special cases, new version are released that only fix defects
  - dedicated versions can be developed for selected customers (***test objects***)

- Of course, there is an opportunity of introducing a new fault each time software is changed (new feature or debugging)



Regression test suites

Just because a software is not currently failing on a test case today, this does not mean it won't fail on it tomorrow



Good test cases to be reused to check SW changes are not disrupting already implemented functionality

Good test cases

- Clear specification
- Complete specification
- **Specific – tempt 1 bug**
- Be good at finding bugs
- **Leads to observable erroneous state or failure**
- Traceable
- Reusable

Good test suites

- Composed by good test cases
- Structured
  - Product decomposition
  - Importance
- Not redundant
- Coverage
- Evolvable

Types of test cases

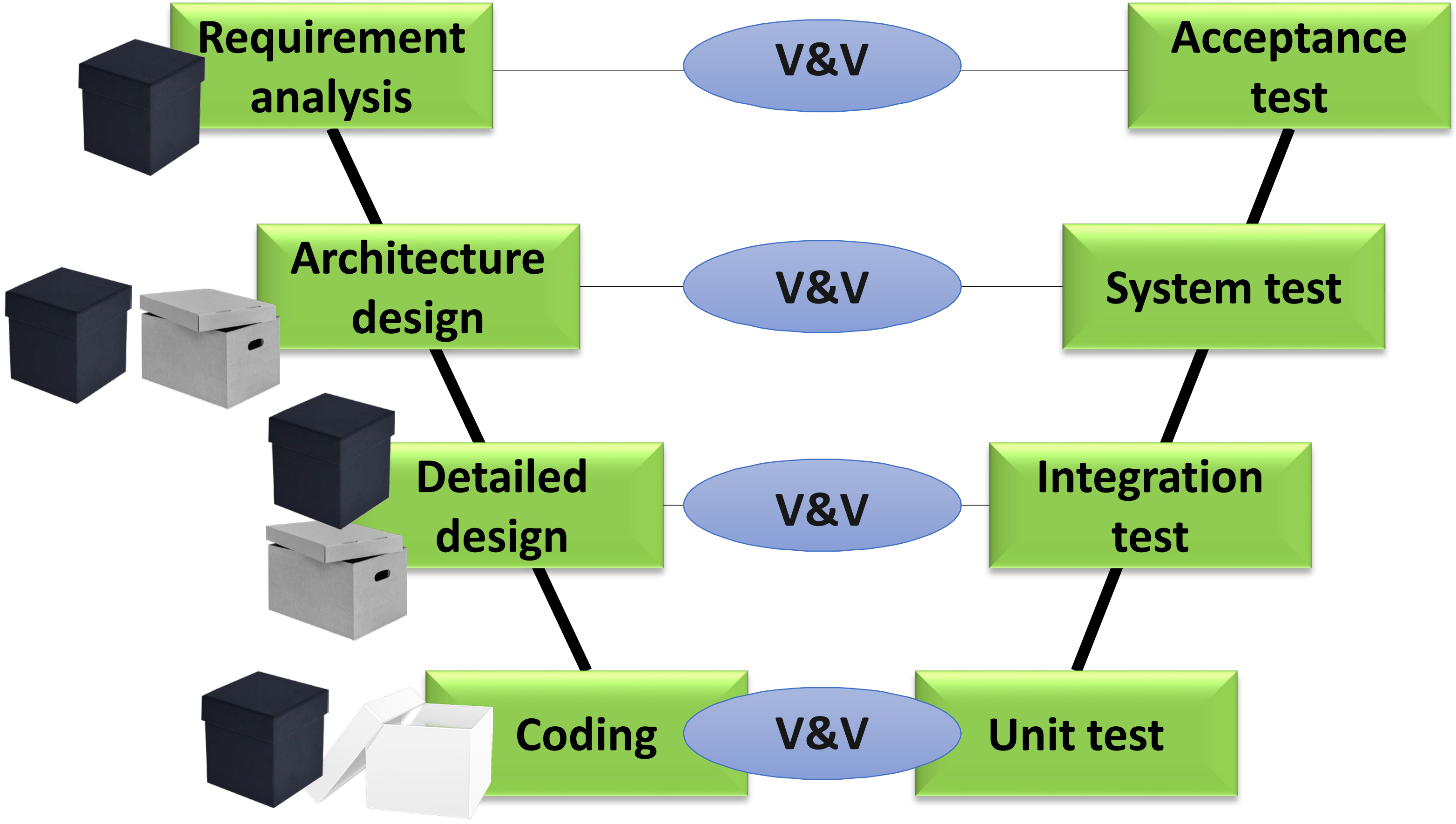
Info necessary for creating the test case

less → *more*

early → *late*

Phase of the project the test case is created

When to create different test types



And of course, think about a regression test suite to use as fault removal goes on

Fixed one bug?

- run regression TS

How to create good BB test cases

- You cannot see the code, but you can identify possible error cases
  - Declared functionality
  - Special corner cases (boundaries, off-by-one)
  - Misinterpretation of requirements

- Cannot cover all input space
  - Partition it into equivalence classes (few representative TC for each class)
  - Check at the boundaries between classes
  - For instance, for a function **`bool isSquare(polygon P){}`**
    it would good to test with a parameter **`P = rectangle(10,10)`**

How to create good WB test cases

- Exercise every code statement

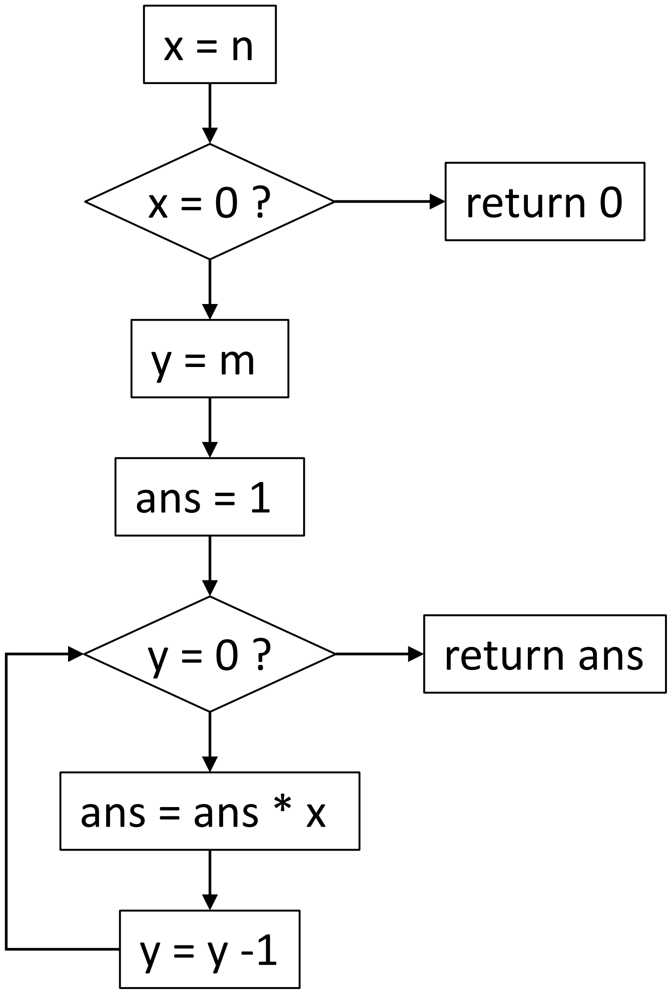
  Statement coverage

- Cover every possible jump

  Branch coverage

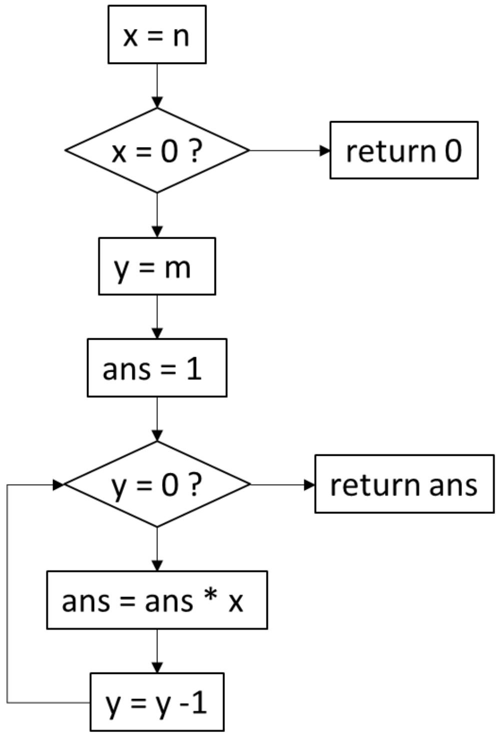
- All possible combination of jumps

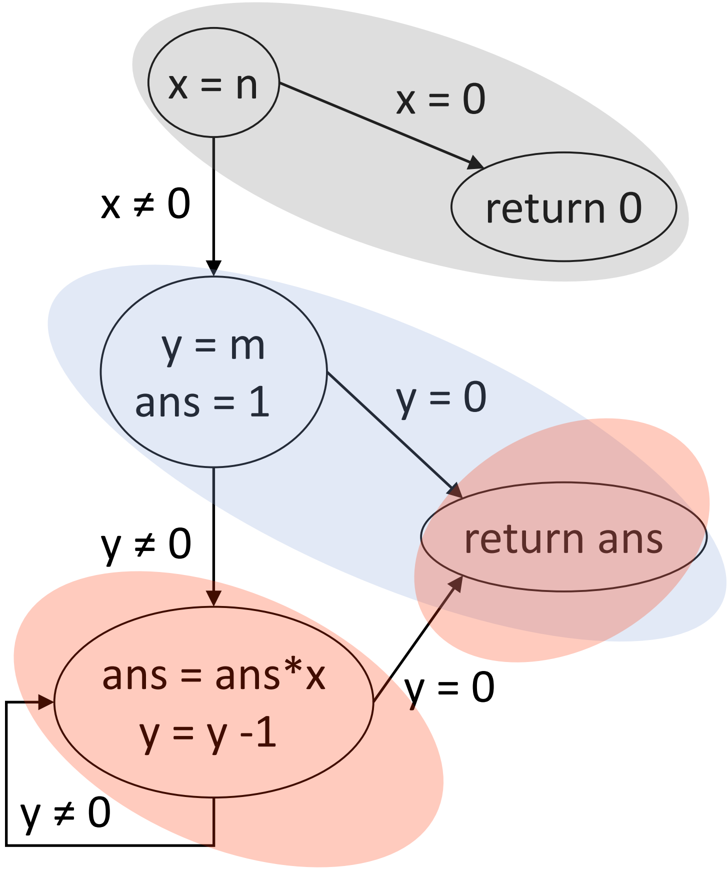
  Path coverage



Checking coverage

- Graph analysis





TC#	n	m
1	0	1
2	2	0
3	3	1

How good are your test cases?

- Is time to check readiness for delivery, and before asking an SQA engineer to do all the checks defined in the quality gate, you consult with your Test Engineer

Great news! Our newly
developed software is perfect!
We ran all the tests and did not
find any single bug!!!

- Then you know your test suite is faulty…

Mutation testing: testing test cases

- A test case is not good if it cannot discriminate faulty SW
- IDEA: pass (on purpose) faulty code to your test cases, to see if they are good enough to catch them
- How to generate faulty versions of the code?
  - random mutation: corrupt operators, switch +/-, remove +1/-1
  - mimicking fault generation

- if the test fails, the test case is sensitive to faults
- If few mutants can kill the majority of the tests, the test suite needs an update

Fault localization

- What a luck, our test suite has found lots of bugs!
- Now the task is to locate the faulty line(s) of code

Fault localization

- We have tools supporting the process (gdb)
  - It can be very time consuming

- An interesting approach to the automation of fault localization
  - Spectrum Based Techniques (SBFL)

What is the spectrum of a program?

- Suppose we test the program below with 3 test cases: "a=0", "a=1", and "a=2". The black dots are the spectrum of the program.

	Code with a bug at s_7	$a=0$	$a=1$	$a=2$
s_1	input(a)	•	•	•
s_2	i = 1;	•	•	•
s_3	$sum = 0;$	•	•	•
s_4	$product = 1;$	•	•	•
s_5	if (i < a){	•	•	•
s_6	$sum = sum + i;$			•
s_7	$product = product \times i;$ $//bug\ product = product \times 2i$			•
s_8	}else{	•	•	
s_9	$sum = sum - i;$	•	•	
s_{10}	$product = product\ /\ i;$	•	•	
s_{11}	}	•	•	
s_{12}	print (sum);	•	•	•
s_{13}	print ($product$);	•	•	•
Execution Results		Successful	Successful	Failed

What info can we mine from the spectrum?

- Each test case's execution result is labelled as "successful" or "failed" according to whether the program returns the expected output

	Code with a bug at s_7	$a=0$	$a=1$	$a=2$
s_1	input(a)	•	•	•
s_2	i = 1;	•	•	•
s_3	$sum = 0;$	•	•	•
s_4	$product = 1;$	•	•	•
s_5	if (i < a){	•	•	•
s_6	$sum = sum + i;$			•
s_7	$product = product \times i;$ //$bug\ product = product \times 2i$			•
s_8	}else{	•	•	
s_9	$sum = sum - i;$	•	•	
s_{10}	$product = product / i;$	•	•	
s_{11}	}	•	•	
s_{12}	print (sum);	•	•	•
s_{13}	print ($product$);	•	•	•
Execution Results		Successful	Successful	Failed

We can obtain four features from the spectrum of a program

The four features from the spectrum

	Code with a bug at s_7	$a=0$	$a=1$	$a=2$
s_1	input(a)	•	•	•
s_2	i = 1;	•	•	•
s_3	*sum* = *0*;	•	•	•
s_4	*product* = *1*;	•	•	•
s_5	if (i < a){	•	•	•
s_6	*sum* = *sum* + *i*;			•
s_7	*product* = *product* × *i*;			•
	//bug product = *product* × *2i*			
s_8	}else{	•	•	
s_9	*sum* = *sum* - *i*;	•	•	
s_{10}	*product* = *product* / *i*;	•	•	
s_{11}	}	•	•	
s_{12}	print (*sum*);	•	•	•
s_{13}	print (*product*);	•	•	•
Execution Results		Successful	Successful	Failed

1 N_{cf}
2 N_{uf}
3 N_{cs}
4 N_{us}

Four Features for each statement:

N_{cf}: Number of ***failed*** test cases that ***EXECUTED*** *the statement*

N_{uf}: Number of ***failed*** test cases that ***DID NOT EXECUTE*** *the statement*

N_{cs}: Number of ***successful*** test cases that ***EXECUTED*** *the statement*

N_{us}: Number of ***successful*** test cases that ***DID NOT EXECUTE*** *the statement*

Example

	Code with a bug at s_7	$a=0$	$a=1$	$a=2$
s_1	input(a)	•	•	•
s_2	i = 1;	•	•	•
s_3	*sum* = 0;	•	•	•
s_4	*product* = 1;	•	•	•
s_5	if (i < a){	•	•	•
s_6	*sum* = *sum* + *i*;			•
s_7	*product* = *product* × *i*; //bug *product* = *product* × 2*i*			•
s_8	}else{	•	•	
s_9	*sum* = *sum* - *i*;	•	•	
s_{10}	*product* = *product* / *i*;	•	•	
s_{11}	}	•	•	
s_{12}	print (*sum*);	•	•	•
s_{13}	print (*product*);	•	•	•
Execution Results		Successful	Successful	Failed

E.g., for statement s1,

1. $N_{cf}^{s1} = 1$
2. $N_{uf}^{s1} = 0$
3. $N_{cs}^{s1} = 2$
4. $N_{us}^{s1} = 0$

E.g., for statement s6,

1. $N_{cf}^{s6} = 1$
2. $N_{uf}^{s6} = 0$
3. $N_{cs}^{s6} = 0$
4. $N_{us}^{s6} = 2$

E.g., for statement s9,

1. $N_{cf}^{s9} = 0$
2. $N_{uf}^{s9} = 1$
3. $N_{cs}^{s9} = 2$
4. $N_{us}^{s9} = 0$

How to use it for fault localization?

The localization is based on the following **two basic intuitions:**

1: Statements covered by more failure-revealing test cases are more likely to be faulty.

2: Statements covered by more passed test cases are less likely to be faulty.

	Code with a bug at s_7	$a=0$	$a=1$	$a=2$
s_1	input(a)	•	•	•
s_2	$i=1;$	•	•	•
s_3	$sum=0;$	•	•	•
s_4	$product=1;$	•	•	•
s_5	if $(i < a)${	•	•	•
s_6	$sum = sum + i;$			•
s_7	$product = product \times i;$			•
	$//bug\ product = product \times 2i$			
s_8	}else{	•	•	
s_9	$sum = sum - i;$	•	•	
s_{10}	$product = product / i;$	•	•	
s_{11}	}	•	•	
s_{12}	print $(sum);$	•	•	•
s_{13}	print $(product);$	•	•	•
Execution Results		Successful	Successful	Failed

We thus can determine a ranking:
{s6, s7} are more suspicious than {s1 – s5, s12, s13}.
Least suspicious are {s8, s9, s10, s11}.

Suspiciousness ranking of statements

In our example: {s6, s7} ← {s1 – s5, s12,s13}← {s8, s9, s10, s11}

Not a total ordering, there are many ties (same suspiciousness rank)
We cannot distinguish among:

- s6, s7
- s8, s9, s10, s11
- s1—s5, s12—s13

Therefore, more precise and tie-less formulae are required to produce a suspiciousness rank for statements of a program.

Alternative rankings

- Tarantula

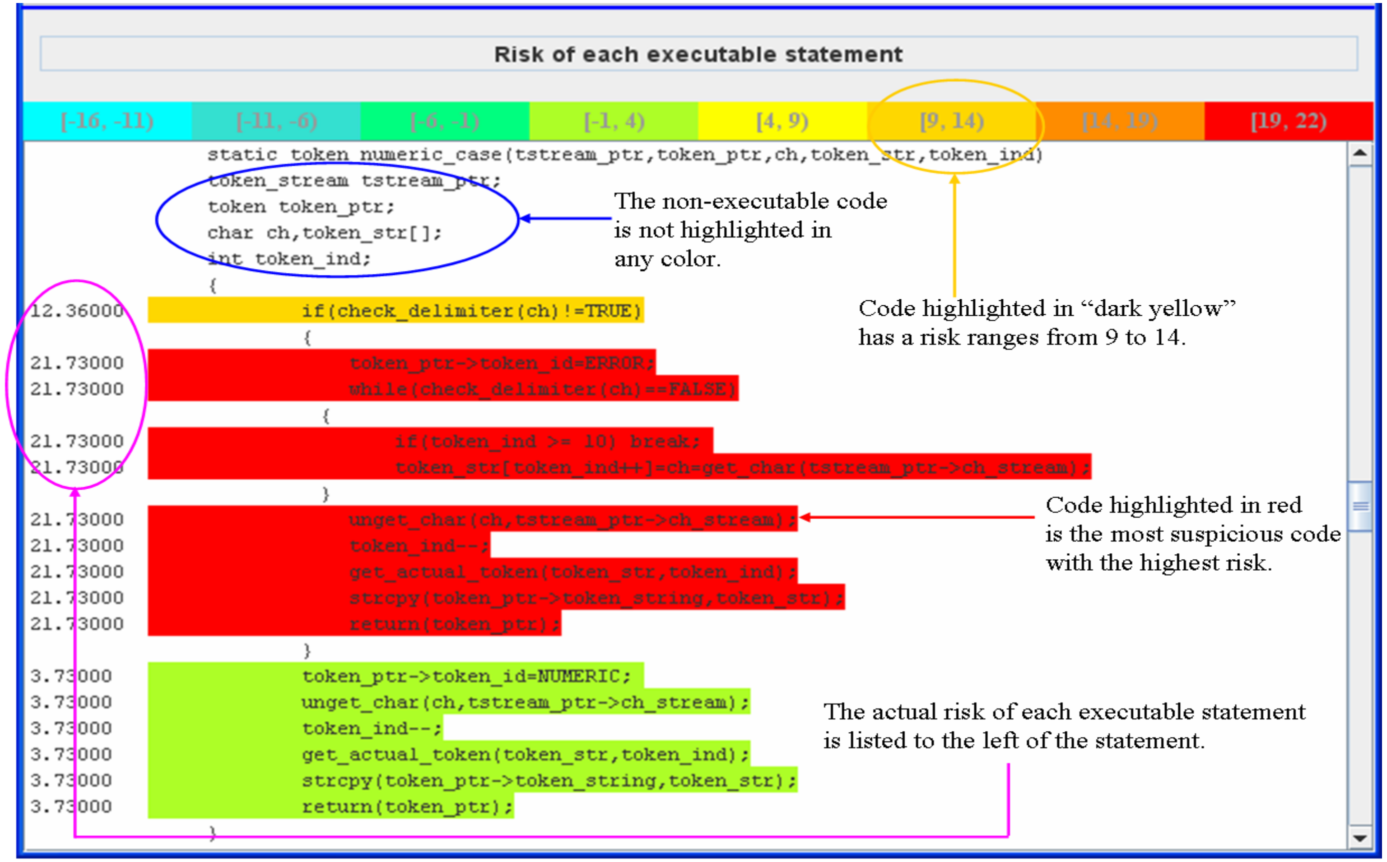
$$Suspiciousness\ index\ = \frac{N_{cf}/N_f}{N_{cf}/N_f + N_{cs}/N_s}$$

N_f, N_s denote the number of failed and successful test cases

In experimental evaluations the above index is found to be more precise than our intuition

  - Several other proposals exist (Pearson, Braun-Banquet, Mountford)

Tool developed at Beihang University (Prof. Zheng)



UnitFL

Available for download

- VS plugin with a GUI
- Allows creating and running NUnit test cases
- Graphically displays coverage
- Analyzes test results to estimate multiple suspiciousness ranks for fault localization
- Combines multiple indexes into an aggregate ranking
- Highlight on the GUI the most suspicious statements

https://visualstudiogallery.msdn.microsoft.com/c5273228-5d3a-487a-acd1-8d2825edfed7

References

- J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization for fault localization," in Proc. Workshop Softw. Vis., 23rd Int. Conf. Softw. Eng., Ontario, BC, Canada, May 2001, pp. 71–75.
- Schuler, David, and Andreas Zeller. "Javalanche: Efficient mutation testing for Java." In Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp. 297-298. 2009.
- Chen, C., & Wang, N. (2016, December). UnitFL: A fault localization tool integrated with unit test. In *2016 5th International Conference on Computer Science and Network Technology (ICCSNT)* (pp. 136-142). IEEE.
- Li, X., Li, W., Zhang, Y., & Zhang, L. (2019, July). Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 169-180).