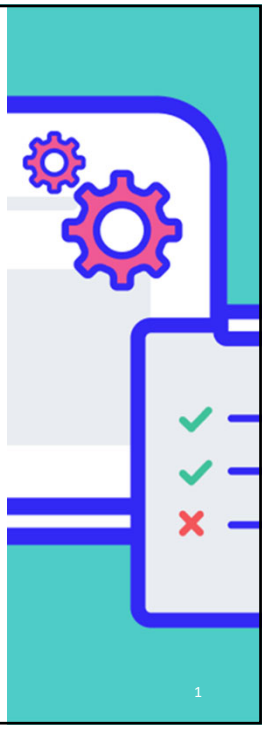
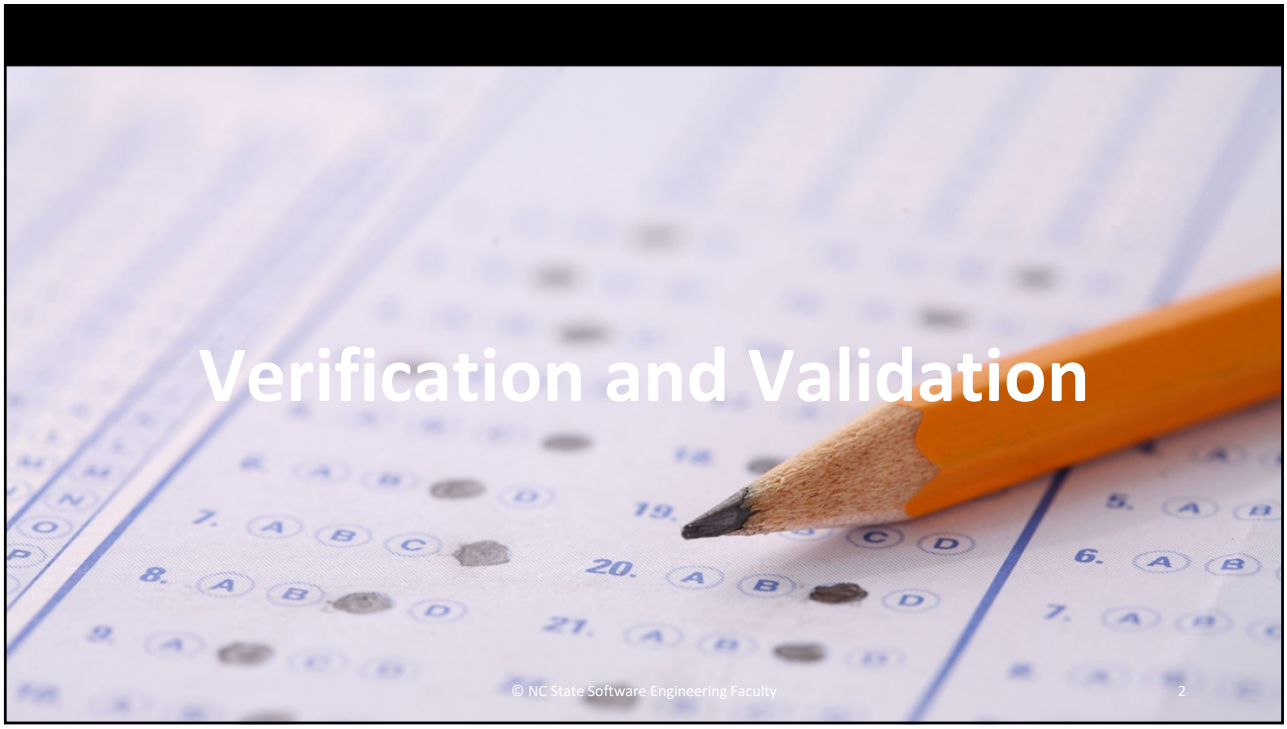
Verification & Validation (Testing)

CSC 326



© NC State Software Engineering Faculty

1



© NC State Software Engineering Faculty

2

Verification & Validation (V&V)

Verification

- "Are we building the product **right**?"
- Deals with testing and code review
- Happens throughout development
- Technical Requirements

tend to be open-box

Validation

- "Are we building the **right** product?"
- Deals with comparison against the customer needs
- Features have been verified, do they satisfy the customer?
- Customer Requirements

tend to be closed-box

3

3

Verification Testing

- Find inputs or input sequences where the behavior of the software is incorrect, undesirable, or does not conform to its specification
- Trying to root out problems by stressing the system (e.g., diabolical inputs, under heavy load)
- ***This is defect testing against technical requirements/specification***
- Focusing on units – methods, classes, modules, interfaces
- And the relationships between units

4

4

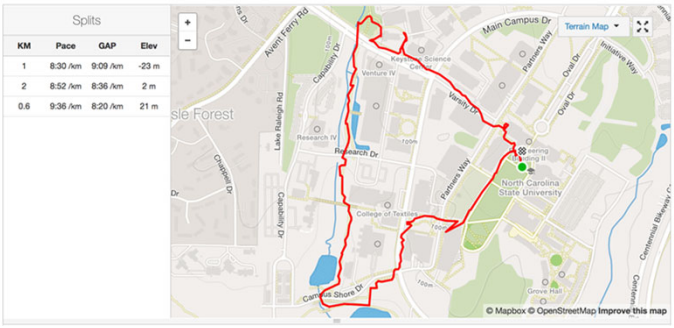
Validation Testing

- Demonstrate to the developer and the customer that software meets its ***customer requirements***
- At least one test per requirement in the requirements document
- Determines if the customer is receiving the product they expect – does it fill the actual needs of the customer
- Defects may be revealed, but that is not the primary goal

© NC State Software Engineering Faculty

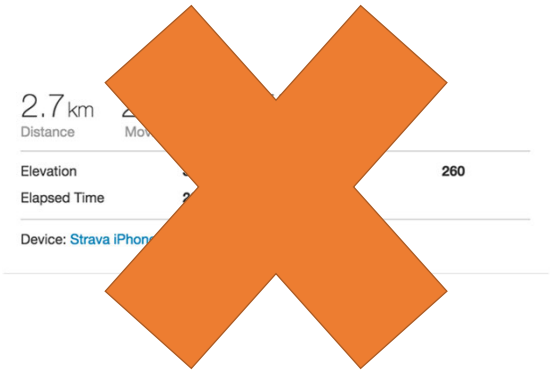
Verification Example - Strava

- **Requirement**: when a user is logged in and views a walking activity, the app shall display the user's walking route on a map.



Validation Example - Strava

- **Requirement:** the app displays summary statistics of fitness activities
- **Customer Feedback**: "*I want to see my daily fitness activity*"



While the original system meets the requirements, it fails to meet the customer needs/wants. This is why requirements solicitation is so hard and important!

7

7



Validation bugs



© NC S

8

8

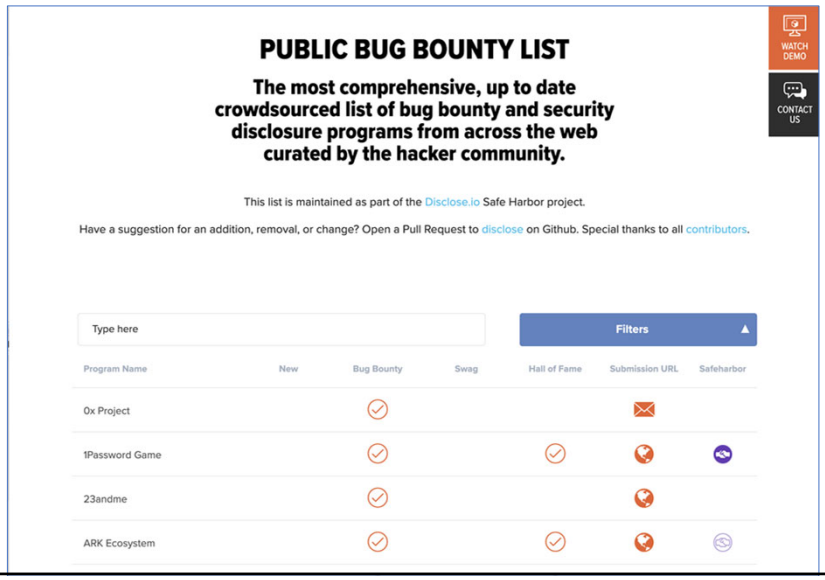
Testing



© NC State Software Engineering Faculty

9

Testing is hard.



PUBLIC BUG BOUNTY LIST

The most comprehensive, up to date crowdsourced list of bug bounty and security disclosure programs from across the web curated by the hacker community.

WATCH DEMO

CONTACT US

This list is maintained as part of the Disclose.io Safe Harbor project.

Have a suggestion for an addition, removal, or change? Open a Pull Request to disclose on Github. Special thanks to all contributors.

Type here

Filters

Program Name	New	Bug Bounty	Swag	Hall of Fame	Submission URL	Safeharbor
0x Project		✓			✉	
1Password Game		✓		✓	✓	✓
23andme		✓			✓	
ARK Ecosystem		✓		✓	✓	✓

10

11

"Testing can only show the presence of errors, not their absence"

Edsger Dijkstra, 1972

© NC State Software Engineering Faculty

11

26/26 tests pass….

What is being tested?

Is it correct?

Does it meet customer needs?



12

12

Types of Testing

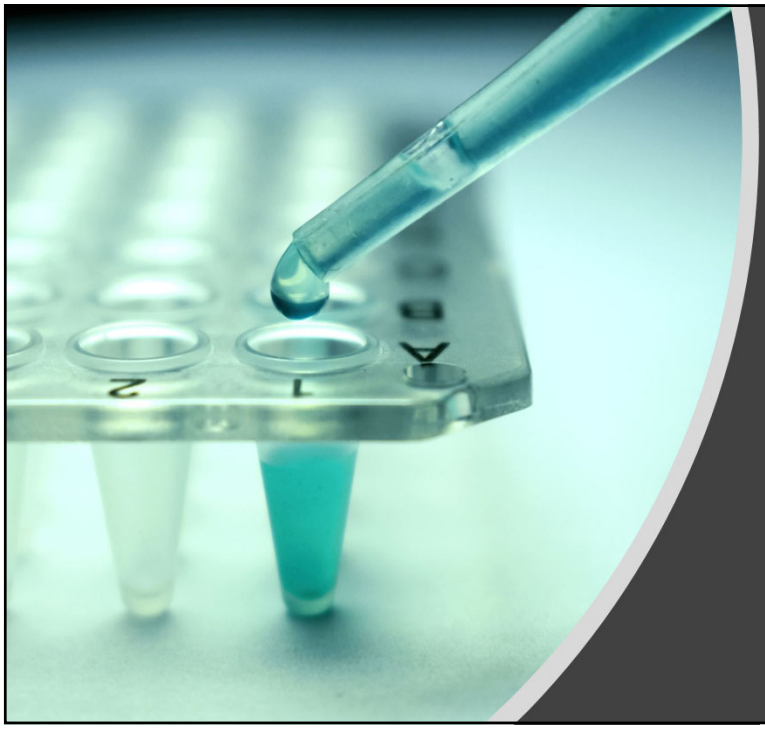
- Unit Testing
- Integration Testing
- Functional/System Testing
- Acceptance Testing
- Regression Testing
- Beta Testing
- …. and more

More related to Verification activities

More related to Validation activities

13

13



14

Let's focus first on testing for defects (verification).

This is probably what you think of when you think of testing.

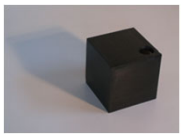
© NC State Software Engineering Faculty

14

Approaches to Testing

Closed / Black Box

- Concerned with requirements
  - Testing each thing the software is supposed to do
  - Independent of implementation







- System/Functional Testing

Open / Glass / White Box

- Concerned with code
  - Focuses on internal structures of the code
  - Testing different paths

- Unit/Integration Testing

© NC State Software Engineering Faculty

Closed-Box

- **Requirement**: for integers between 2 and 50, the method shall return *true* if the number is prime and *false* otherwise.

```
public static boolean isPrime ( int n ) {
}
```

1. Give me a test input!

Things to consider… Closed-box Testing

- Look at requirements/problem statement to generate.
- Test cases need to be traceable to a requirement.

- Test cases must be repeatable so anyone on the team can run the exact test case and get the exact same result/sequence of events.
  - A case for automation (and determinism)!

17

17

Open-Box

- **Requirement**: for integers between 2 and 50, the method shall return *true* if the number is prime and *false* otherwise.

1. Give me a test input!

```
public static boolean isPrime ( int n ) {
    int i;
    if ( n % 5 == 0 ) {
        n--;
    }
    for ( i = 2; i < n; i++ ) {
        if ( n % i == 0 ) {
            return false;
        }
    }
    return true;
}
```

18

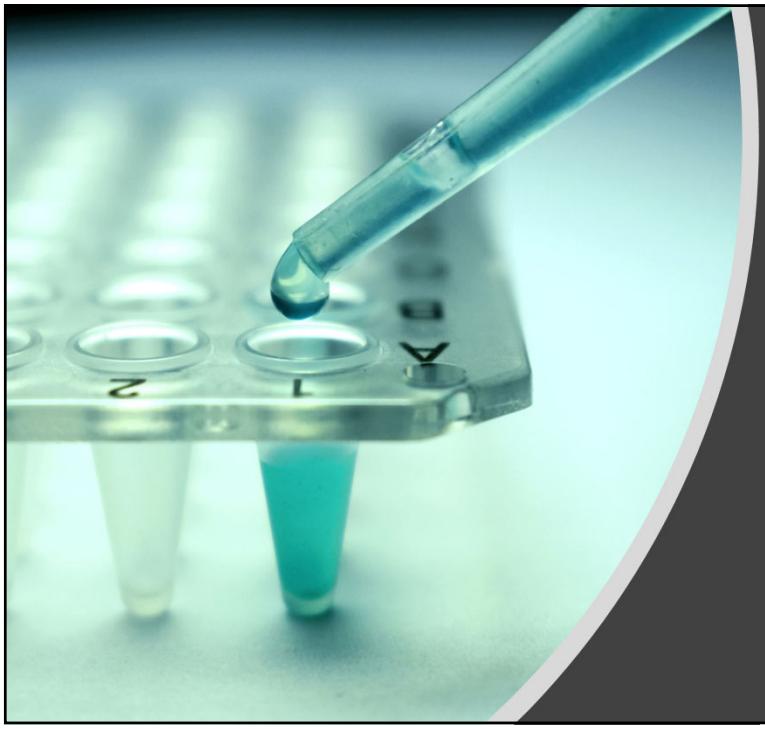
18

Things to consider… Open-box Testing

- Look at structure of the code
- Test cases still need to be traceable to a requirement.
- Test cases still need to be repeatable

19

19



20

Test Suite Quality

© NC State Software Engineering Faculty

20

Types of Coverage

- Class Coverage
- Method Coverage
- Line/Statement Coverage
- Branch Coverage
- Clause Coverage
- Path Coverage
- Requirements Coverage
- …

These require a different view of the code!

This doesn't require code at all!

© NC State Software Engineering Faculty

Benefits of code coverage

- A measure of how complete your ***test cases*** are
  - **High coverage does not guarantee code correctness!**
- Can identify paths through the code that you may have missed

Limits of code coverage

- The assumption that you are *done* testing if you have high coverage is incorrect
  - Coverage tools only tell you if you've covered what's there
  - There may be requirements that you have missed!
  - There may be problematic inputs you've missed!
- Don't write test cases ONLY to make your coverage tool happy (write them because they mean something)
  - Inspect surrounding code for other potential errors, perhaps requirements you've missed

BEWARE: A test suite can achieve 100% line/statement coverage and contain zero assertions!

23

23

Requirements coverage – Use Cases

UC4: Flow of Events for the Edit Recipe Use Case

Preconditions: UC1 is completed.

Main Flow: The user will be shown a list of all recipes in the system. The user may select one recipe that they want to edit [E1]. After selecting a recipe, the form will be pre-filled with all existing values (price, units coffee, units sugar, units milk, and units chocolate) for the recipe selected. The user enters the price [E2][E3][E4], units coffee [E2][E3][E4], units sugar [E2][E3][E4], units milk [E2][E3][E4], and units chocolate [E2][E3][E4] that make up the modified selected recipe [E1]. The user may also return to the main menu [UC1].

Subflows: None

Alternative Flows:

- [E1] If the recipe is deleted by a concurrent user before it's selected for edit, an error message is printed and the user is returned to the edit recipe menu.
- [E2] The price must be a integer. If the price is not an integer then, an error message is printed, and the user is returned to the form to edit. The recipe is not edited.
- [E3] The units of coffee, sugar, milk, and chocolate must be integers. If the unit value on any of the ingredients is not an integer, an error message is printed, and the user is returned to the form to edit. The recipe is not edited.
- [E4] The price and units of ingredients must be positive. If any number is negative, an error message is printed, and the user is returned to the form to edit. The recipe is not edited.

© NC State Software Engineering Faculty

24

24

Branch, Clause and Path Coverage:

Control flow graphs (CFG)

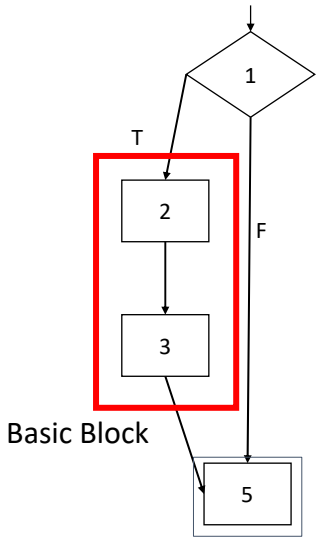
- A **CFG** models all executions of a method by describing control structures
- **Nodes** : Statements or sequences of statements (basic blocks)
  - **Basic Block** : A sequence of statements such that if the first statement is executed, all statements will be (no branches)
- **Edges** : Transfers of control

25

25

CFG: the if statement

```
1. if(x < y){
2.    y = 0;
3.    x = x + 1;
4. }
5. return;
```

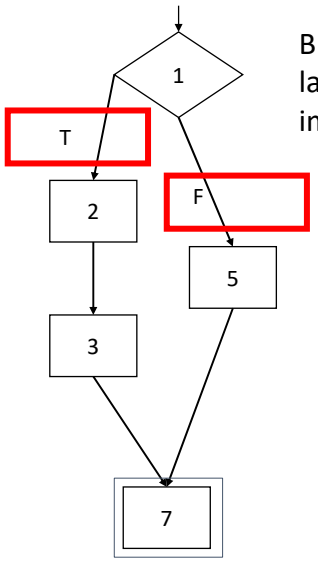


26

26

CFG: the if-else statement

```
1. if(x < y){
2.    y = 0;
3.    x = x + 1;
4. } else {
5.    x = y;
6. }
7. return;
```



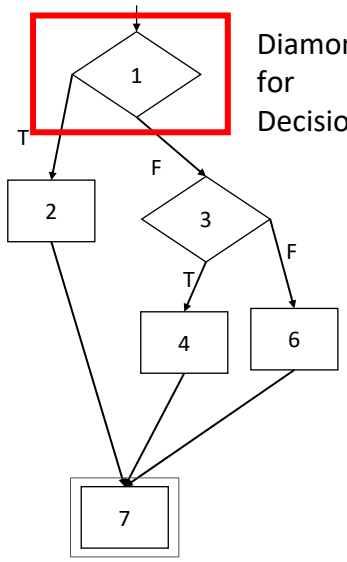
Branch labels are important!

27

27

CFG: the if-elseif-else statement

```
1. if(x < y){
2.    y = 0;
3. } else if (x > y) {
4.    x = y;
5. } else {
6.    x = 0;
7. }
8. return;
```



Diamonds for Decisions

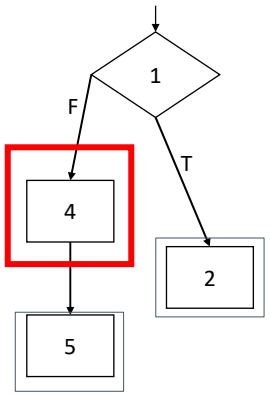
28

28

CFG: the if-return statement

```
1. if(x < y){
2.   return;
3. }
4. print(x);
5. return;
```

Squares for blocks

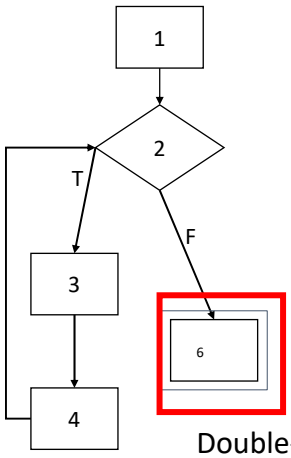


29

29

CFG: while loops

```
1. x = 0;
2. while(x < y) {
3.    y = f(x, y);
4.    x = x + 2;
5. }
6. return;
```



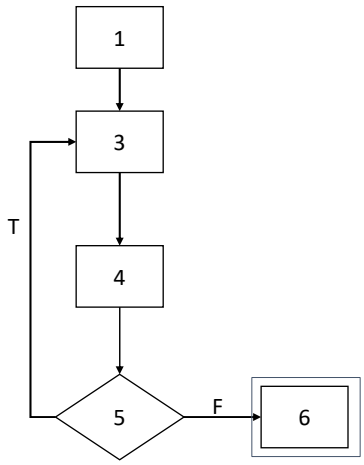
Double-outline for exit

30

30

CFG: the do-while loop

```
1. x = 0;
2. do{
 3.    y = f(x, y);
 4.    x = x + 2;
5. } while(x < y);
6. return;
```



31

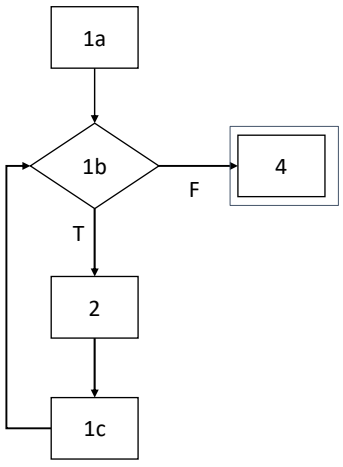
31

CFG: for loop

```
1. for (x = 0; x < y; x++) {
2.     y = f(x, y);
3. }
4. return;
```

1a 1b 1c

No block for a closing parent



32

32

Coverage on Graphs

Exercise 04.01
Work with a small group or by yourself. 2 minutes.

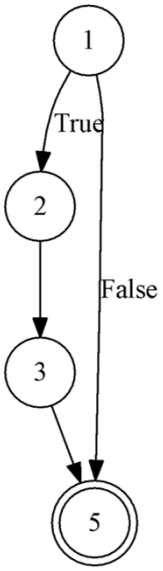
- Node Coverage ≈ Statement Coverage
- Edge Coverage ≈ Branch/Decision Coverage

How many inputs….

To achieve 100% node coverage?

To achieve 100% branch coverage?

```
1. if(x < y){
2.    y = 0;
3.    x = x + 1;
4. }
5. return;
```



© NC State Software Engineering Faculty

33

Branch and Condition Coverage

- Branch (Decision) Coverage: Considers the whole Boolean expression that selects a branch
- Condition (Clause) Coverage: Considers the true/false path of each condition or clause in a compound Boolean expression

Test	a	b	a && b
1	T	T	T
2	T	F	F
3	F	T	F
4	F	F	F

Branch, but not Condition.

Test	a	b	a || b
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

Condition, but not Branch.

34

34

Modified Condition/Decision Coverage (MC/DC)

MC/DC is a stronger criterion for coverage than condition or decision

- Requires each condition to independently affect the outcome
- For each condition x, there is a pair of tests such that all three of the following hold
  - From one test to the next, the outcome of the decision flips from T to F, or vice versa
  - The value of condition x also flips
  - The values of the remaining conditions stay the same across tests
- Goal: Identify the MC/DC pairs for each condition

35

35

Comparison

Test	a	b	a || b
1	T	T	T
4	F	F	F

Branch Coverage: Yes
Condition Coverage: Yes
MC/DC: No

- For a, the value of b isn't consistent
- For b, the value of a isn't consistent

Test	a	b	a || b
2	T	F	T
3	F	T	T
4	F	F	F

Branch Coverage: Yes
Condition Coverage: Yes
MC/DC: No

- For a, Tests 2 & 4
- For b, Tests 3 & 4

36

36

Pairs Tables

Test	a	b	a && b	a	b
1	T	T	T	3	2
2	T	F	F		1
3	F	T	F	1	
4	F	F	F		

Test	a	b	a || b	a	b
1	T	T	T		
2	T	F	T	4	
3	F	T	T		4
4	F	F	F	2	3

37

37

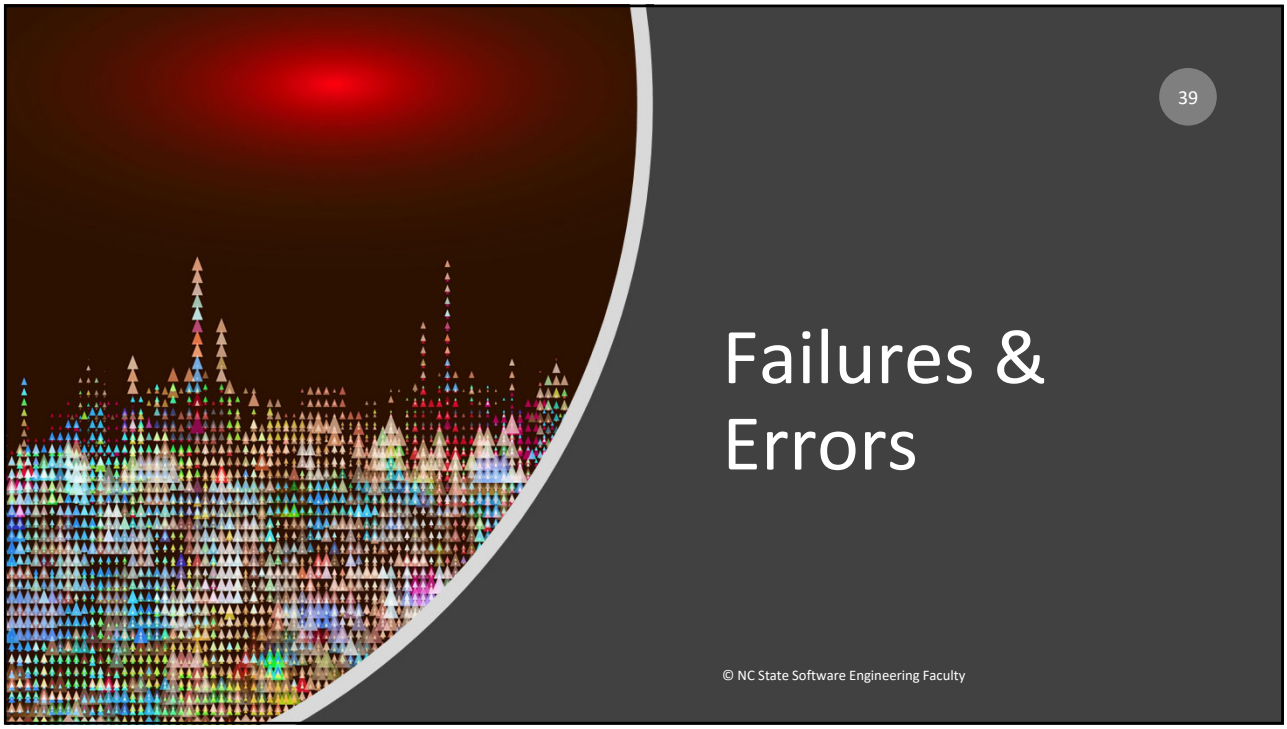
Types of Testing

- Unit Testing
- Integration Testing
- Functional/System Testing
- Acceptance Testing
- Regression Testing
- Beta Testing

Code coverage is only computed for some types of testing!

38

38



39

Mistake, Fault (**bug**), and Failure

bug

Programmer makes a ***Mistake***

↘

Fault (defect, bug) appears in the program

↘

Fault remains undetected during **testing**

↘

The program fails during execution – ***Failure***!

Error: difference between computed, observed, or measured value or condition and true, specified, or theoretically correct value or condition.

© NC State Software Engineering Faculty

©A. Mathur, L. Williams

40

Error Definitions

- ***Error***: difference between computed, observed, or measured value or condition and true, specified, or theoretically correct value or condition.

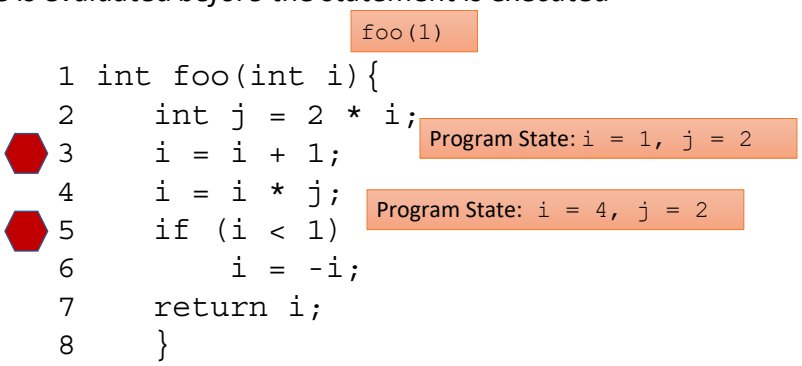
Or

- ***Error***: an incorrect program state resulting from the execution of a fault

© NC State Software Engineering Faculty

Program State – an introduction

- The values of all live variables at a point in a program's execution
  - Think about debugging with breakpoints – this exposes program state!
  - State is evaluated *before* the statement is executed



Example – Pair Share

Exercise 04.02
Work with a small group or by yourself. 2 minutes.

- **Requirement**: the method shall return *true* if the number is prime and *false* otherwise.

1. Identify the fault
2. Identify a test input that results in a failure
3. Identify a test input that results in an error but not a failure

```
public static boolean isPrime ( int n ) {
1       int i;
2       if ( n % 5 == 0 ) {
3           n--;
4       }
5       for ( i = 2; i < n; i++ ) {
6           if ( n % i == 0 ) {
7               return false;
8           }
9       }
10      return true;
    }
```

43

43

Unit Testing

JUnit

- Testing of individual hardware or software units or groups of related units
  - Focus on the method, interleaved methods, and class level
- Done by programmer(s)
- Generally all open box
- Automation desirable for repeatability
- Tools: JUnit

44

44

JUnit

Integration Testing

- Testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them
  - Focus on the interaction between classes due to composition or inheritance
- Done by programmer as they integrate their code into code base
- Generally open box, maybe some closed box (e.g., closed source libraries)
- Automation desirable for repeatability
- Tools: JUnit

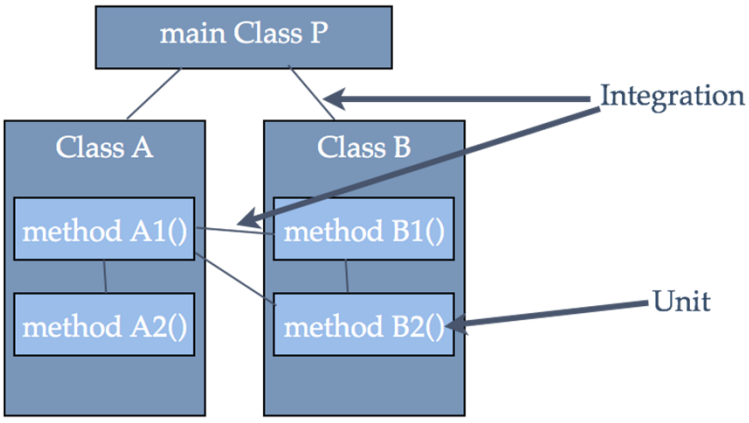
45

45

JUnit

Integration Testing

- Catches mismatched assumptions
  - Who sanitized user input?
  - Will **null** be passed to B2 from A1? Or not?



46

46

Functional / System Testing

- Testing conducted on a complete, integrated system to evaluate the system compliance with its specified requirements
  - Tests are generated from the requirements
  - Includes stress testing, performance testing, and usability testing
- It is recommend that this be done by external test group
- Mostly closed box so that testing is not "corrupted" by too much knowledge of underlying implementation
- Test automation desirable, but non-trivial

47

47

Thought Exercise

Exercise 04.03
Work with a small group or by yourself. 2 minutes.

- You are doing **system testing**.
- You have **tests that cover all your requirements**, including main flows, subflows, and error conditions.
- You know your **coverage tool computes coverage for all the code** in your system (only Java code, for example)
- Your coverage is at **55%**.

What explains the low coverage?

© NC State Software Engineering Faculty

48

48

Acceptance Testing

- Formal testing conducted to determine whether or not a system satisfies its acceptance criteria (the criteria the system must satisfy to be accepted by a customer) and to enable the **customer** to determine whether or not to accept the system
- Generally done by customer/customer representative in their environment through the typical user interface.
- Teaching staff run acceptance tests to evaluate your submissions.
- Closed Box

49

49

Regression Testing

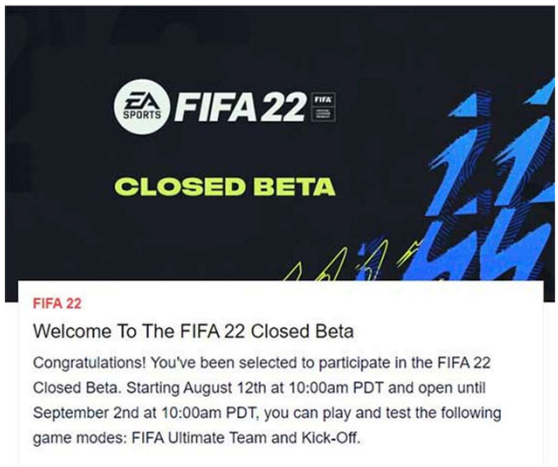
- **Selective retesting** of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.
  - Older tests on newer code
- Combination of open and closed box
- Automated for repeatability

50

50

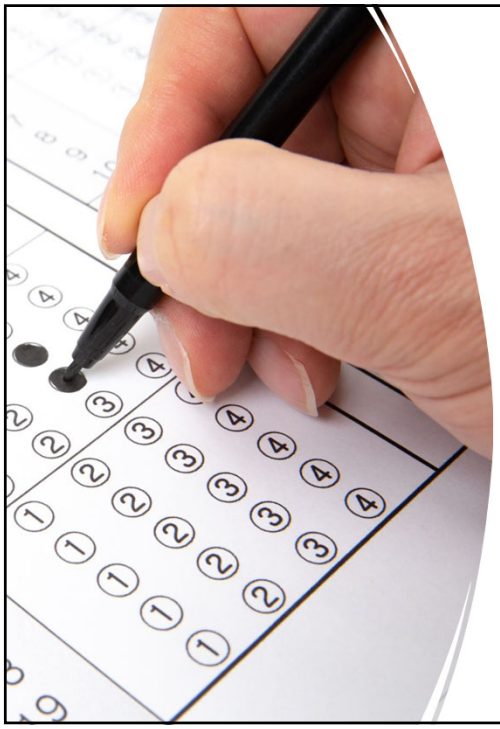
Beta Testing

- 3rd party testing by a subset of customers
- Unstructured/unscripted testing
- Closed box



51

51



Test Selection Strategies

© NC State Software Engineering Faculty

52

52

How to select test inputs?

- Requirements
- Equivalence Classes
- Boundary Values
- Decision Table Testing
- Diabolical Testing

Requirements

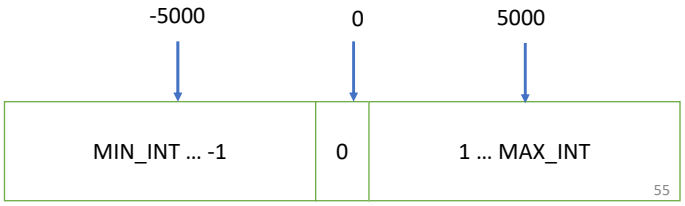
- Ensure each customer requirement is tested at least once
  - Start with most-used *success path*
  - Continue with other *success paths*
  - Also test *failure paths* that contain intentional errors that should fail predictably and gracefully (i.e. expected points of user failure)

Equivalence classes

- Divide your input conditions into groups (classes).
  - Input in the same class should behave similarly in the program.
- Test a mid-range value

- What groups/classes would be useful for:

```
boolean isNegative(int i) {
…
}
```

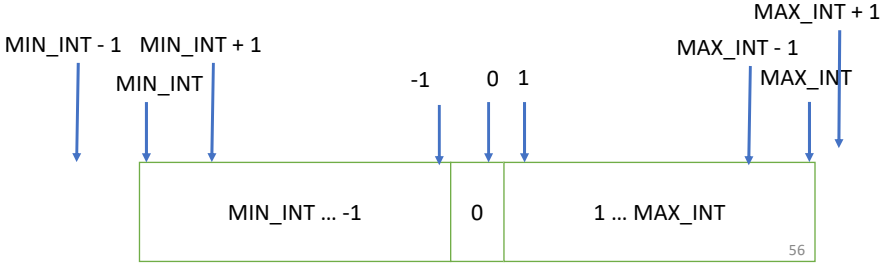


55

Boundary value

- Focus on boundaries . . . because a greater number of faults tend to occur at the boundaries of the input domain

```
boolean isNegative(int i) {
…
}
```



56

Diabolical values

- Can something cause division by zero?
- What if the input type is wrong (You're expecting an integer, they input a float. You're expecting a character, you get an integer.)?
- What if the customer takes an illogical path through your functionality?
- What if mandatory fields are not entered?
- What if the program is aborted abruptly or input or output devices are unplugged?

57

57

OBP – Milestone 1 – Testing Workshop

- You'll have the opportunity to explore testing throughout the full stack of the coffee_maker backend project.
- Complements your debugging skills from the Debugging Workshop



© NC State Software Engineering Faculty

58

58