

CS1699 - Lecture 4 - Test Plans

FizzBuzz

FizzBuzz
Project Details
The fizzbuzz problem is a popular integral part of many introductory computer science courses. It's the first step in teaching the basics of loops, conditionals, and functions. In its most basic form, it's a simple exercise to print numbers from 1 to 100, but with a twist: for numbers divisible by 3, instead of printing the number, you print "Fizz". For numbers divisible by 5, instead of printing the number, you print "Buzz". For numbers divisible by both 3 and 5, instead of printing the number, you print "FizzBuzz".

Example Output
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
18
Fizz
20
Buzz
22
Fizz
24
25
FizzBuzz
28
29
30

Soft's FizzBuzz with a Test Plan
Project Details
Title: FizzBuzz
Description: This project is a simple implementation of the FizzBuzz problem. It takes an integer input and prints the corresponding output based on the rules of the game.
Technologies: Python
Dependencies: None
Status: Complete
Last Update: 2023-01-15
Version: 1.0
File Structure:
- fizzbuzz.py
- README.md
- requirements.txt

Developing a Test Plan

So you have requirements...
how do you test them?

Think About What You're Testing
1. Throw-away script?
2. Development tool?
3. Internal website?
4. Enterprise software?
5. Operating system?
6. Avionics software?

Your test plan should flow from the functional requirements through the non-functional requirements.

Example
System shall fail gracefully.
This will mean something very different for an MRI machine and a website.

The Seven Testing Principles
1. Testing shows presence of defects.
2. Exhaustive testing is impossible.
3. Test early.
4. Defect clustering.
5. The "pesticide paradox"
6. Testing is context-dependent.
7. Absence of errors fallacy.
—S. T. McConnell, Software Quality Toolkit (McGraw-Hill)

Testing Shows Presence of Defects
You can never really "prove" software is free of defects.
Absence of evidence IS NOT evidence of absence.
"You, there are "formally proven" programs, but there is still a whole other rabbit hole to go down, and this concept is going to come up for maybe 1% of testers."
—S. T. McConnell, Software Quality Toolkit (McGraw-Hill)

Testing Can Never Show All Defects or Possible Failure Cases
1. What if an asteroid destroys the entire planet, butting the entire surface with lava?
2. Will system work correctly under 100% load?
3. What happens when memory fails at instruction 1, 2, 3, ... etc?
4. Concurrency and parallelism?
5. What if the value of pi was slightly shifted in our Universe, would program work correctly?

Exhaustive Testing is Impossible
Could you prove a spellchecker could work with every possible input?
Early Testing
It is always, always, always better to discover a problem sooner rather than later.
Report early, report often.
Don't wait to test.

Defect Clustering
Defects tend to cluster in certain areas of code or functionality.
Usually due to technical difficulty of implementing bad requirements, etc.
If you find many errors in one module, don't assume that others are "bad".
—Curtis H. Falvey

Pesticide Paradox
If you keep testing over and over with the same test plan, you'll only ever find bugs in that test plan.
You will need to test other aspects of the system and revise the test plan.

Testing is Context Dependent
How you test.
How much you test.
What tools you use.
What documentation you provide...
All very based on software context.

Absence of Errors Fallacy
Even the greatest quality tests and validation tools can't catch all errors.
Counterexample: If initially owned a horse named Bob, he'd never own a horse named Bob again.

Verification vs Validation
Verification - "Are we building the software right?"
Validation - "Are we building the right software?"

Test Cases / Runs / Plans / Suites



Test Tracking



CS1699 - Lecture 4 - Test Plans

FizzBuzz

FizzBuzz
Project Details
The fizzbuzz problem is a popular integral part of many introductory computer science courses. It's the first step in teaching the concepts of loops, conditionals, and functions. In its most basic form, it asks the student to print numbers from 1 to 100. For each number, if it is divisible by 3, print "Fizz". If it is divisible by 5, print "Buzz". If it is divisible by both 3 and 5, print "FizzBuzz". This is a good exercise for learning how to work with loops and conditionals.

Example Output
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
18
Fizz
20
Buzz
22
23
24
FizzBuzz
26
27
28
Fizz
30
Buzz
32
33
34
FizzBuzz
36
37
38
Fizz
40
Buzz
42
43
44
FizzBuzz
46
47
48
Fizz
50
Buzz
52
53
54
FizzBuzz
56
57
58
Fizz
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
Buzz

Let's Create one with a Test Plan
Test Plan
Title: FizzBuzz
Description: This is a simple test plan for the FizzBuzz problem.
Preconditions:
- The code is written in Python.
- The code is well-documented.
- The code is tested and passes all unit tests.
- The code is reviewed and approved by a peer.
- The code is checked into version control.
- The code is deployed to a production environment.
- The code is monitored for performance and stability.

Developing a Test Plan

So you have requirements...
how do you test them?

Think About What You're Testing
1. Throw-away script?
2. Development tool?
3. Internal website?
4. Enterprise software?
5. Operating system?
6. Avionics software?

Your test plan should flow from the functional requirements through the non-functional requirements.

Example
System shall fail gracefully.
This will mean something very different for an MRI machine and a website.

The Seven Testing Principles
1. Testing shows presence of defects.
2. Complete testing is impossible.
3. Test early.
4. Defect clustering.
5. The "pesticide paradox"
6. Testing is context-dependent.
7. Absence of errors fallacy.
—D. R. T. J. (Inventor of Software Quality Metrics) (Author of Software Quality Metrics)

Testing Shows Presence of Defects
You can never really "prove" software is free of defects.
Absence of evidence IS NOT evidence of absence.
"You, there are "formally proven" programs, but there is still a whole other rabbit hole to go down, and this concept is going to come up for maybe 1% of testers."
—D. R. T. J. (Inventor of Software Quality Metrics) (Author of Software Quality Metrics)

Testing Can Never Show All Defects or Possible Failure Cases
1. What if an asteroid destroys the entire planet, butting the entire surface with lava?
2. Will systems work correctly under 100% load?
3. What happens when memory fails at instruction 1, 2, 3, etc?
4. Concurrency and parallelism?
5. What if the value of pi was slightly shifted in our Universe, would program work correctly?

Exhaustive Testing is Impossible
Could you prove a spellchecker could work with every possible input?
Early Testing
It is always, always, always better to discover a problem sooner rather than later.
Report early, report often.
Don't wait to test.

Defect Clustering
Defects tend to cluster in certain areas of code or functionality.
Usually due to technical difficulty of implementing bad requirements, etc.
If you find many errors in one module, don't assume that others are "bad".
—Guru of Testing

Pesticide Paradox
If you keep testing over and over with the same test plan, you'll only ever find bugs in that test plan.
You will need to test other aspects of the system and revise the test plan.

Testing is Context Dependent
How you test.
How much you test.
What tools you use.
What documentation you provide...
All very based on software context.

Absence of Errors Fallacy
Often the question is a fully loaded and verbal filibuster.
Imagine it takes time to counter.
Counter says for initially owned a huge pile of bugs.
PROFOUNDLY DISPARATE HELP IT'S NOT WHAT THE USER WANTS!

Verification vs Validation
Verification - "Are we building the right software?"
Validation - "Are we building the right software?"

Test Cases / Runs / Plans / Suites



Test Tracking



FizzBuzz

FizzBuzz Requirements

1. The function shall accept one argument, a positive integer. If argument is not a positive integer, do not print anything.
2. The function shall print the numbers from 1 to n, where n is the argument to the function, separated by newlines appropriate to the system, with the exceptions enumerated in requirements 3, 4, and 5.
3. If the current number being printed is evenly divisible by 3 and 5, instead of printing the number, print "FizzBuzz".
4. If the current number being printed is evenly divisible by 3, but not evenly divisible by 5, print "Fizz".
5. If the current number being printed is evenly divisible by 5, but not evenly divisible by 3, print "Buzz".

Argument: 20

1

2

Fizz

4

Buzz

Fizz

7

8

Fizz

Buzz

11

Fizz

13

14

FizzBuzz

16

Example Output

Let's Come up with a Test Plan!

Equivalence Classes?

Boundary Values?

Base case?

Success vs Failure Cases?

Edge cases, Corner/Pathological cases?

Example Output

1
2
Fizz
4
Buzz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16

FizzBuzz

FizzBuzz Requirements

1. The function shall accept one argument, a positive integer.
2. If argument is not a positive integer, do not print anything.
3. The function shall print the numbers from 1 to n, where n is the argument given, separated by newlines.
4. The output shall be appropriate to the system, with the exceptions enumerated in requirements 3, 4, and 5.
5. If the current number being printed is evenly divisible by 3
 a. Print the prefix "Fizz"
4. If the current number being printed is evenly divisible by 5
 b. Print the prefix "Buzz"
5. If the current number being printed is evenly divisible by 3, but not evenly divisible by 5, print "Fizz".
6. If the current number being printed is evenly divisible by 5, but not evenly divisible by 3, print "Buzz".

Developing a Test Plan

So you have requirements...
how do you test them?

Think About What You're Testing

1. Throw-away script?
2. Development tool?
3. Internal website?
4. Enterprise software?
5. Operating system?
6. Avionics software?

Your test plan should flow from the functional requirements through the non-functional requirements.

Example

System shall fail gracefully.

This will mean something very different for an MRI machine and a website.

The Seven Testing Principles

1. Testing shows presence of defects.
2. Exhaustive testing is impossible.
3. Test early.
4. Defect clustering.
5. The "pesticide paradox"
6. Testing is context dependent.
7. Absence of errors fallacy.

-ISTQB (International Software Quality Testing Board)
Foundational Syllabus

Testing Shows Presence of Defects

You can never really "prove" * software is free of defects.

Absence of evidence IS NOT evidence of absence.

* Yes, there are "formally proven" programs out there. This is a whole other rabbit hole to go down, and this concept is going to come up for maybe 1% of testers.

Testing Can Never Show All Defects or Possible Failure Cases

1. What if an asteroid destroys the crust of our planet, bathing the entire surface with lava?
2. Will system work correctly under 100 Tesla of magnetism?
3. What happens when malloc() fails at instruction 1.. 2... 3... etc?
4. Concurrency and parallelism?
5. What if the value of pi was slightly shifted in our Universe, would program work correctly?

Exhaustive Testing is Impossible

Could you prove a spellchecker could work with every possible novel?

Early Testing

It is always, always, always better to discover a problem sooner rather than later.

Report early, report often.

Don't wait to test.

Defect Clustering

Defects tend to cluster in certain areas of code or functionality.

Usually due to technical difficulty of implementing, bad requirements, etc.

If you find many errors in one module, don't assume that others are "due". (Gambler's Fallacy)

Pesticide Paradox

If you keep testing over and over with the same test plan, you'll only ever find bugs in that test plan.

You will need to test other aspects of the system and revise the test plan!

Testing is Context-Dependent

How you test
How much you test
What tools you use
What documentation you provide...

All vary based on software context.

Absence of Errors Fallacy

Let's say we gave our fully tested and vetted FizzBuzz implementation to a customer.

Customer says he actually wanted a hamburger and fries.

FINDING DEFECTS DOESN'T HELP IF IT'S NOT WHAT THE USER WANTS!

Verification vs Validation

Verification - "Are we building the software right?"

Validation - "Are we building the right software?"

So you have requirements...

how do you test them?

Think About What You're Testing

1. Throw-away script?
2. Development tool?
3. Internal website?
4. Enterprise software?
5. Operating system?
6. Avionics software?

Your test plan should flow from the functional requirements through the non-functional requirements.

Example

System shall fail gracefully.

This will mean something very different for an MRI machine and a website.

The Seven Testing Principles

1. Testing shows presence of defects.
2. Exhaustive testing is impossible.
3. Test early.
4. Defect clustering.
5. The "pesticide paradox"
6. Testing is context dependent.
7. Absence of errors fallacy.

*-ISQTB (International Software Quality Testing Board)
Foundational Syllabus*

Testing Shows Presence of Defects

You can never really "prove" * software is free of defects.

Absence of evidence IS NOT evidence of absence.

* Yes, there are "formally proven" programs out there. This is a whole other rabbit hole to go down, and this concept is going to come up for maybe 1% of testers.

Testing Can Never Show All Defects or Possible Failure Cases

1. What if an asteroid destroys the crust of our planet, bathing the entire surface with lava?
2. Will system work correctly under 100 Tesla of magnetism?
3. What happens when malloc() fails at instruction 1.. 2... 3... etc?
4. Concurrency and parallelism?
5. What if the value of pi was slightly shifted in our Universe, would program work correctly?

Exhaustive Testing is Impossible

Could you prove a spellchecker could work with
every possible novel?

Early Testing

It is always, always, always better to discover a problem sooner rather than later.

Report early, report often.

Don't wait to test.

Defect Clustering

Defects tend to cluster in certain areas of code or functionality.

Usually due to technical difficulty of implementing, bad requirements, etc.

If you find many errors in one module, don't assume that others are "due".
(Gambler's Fallacy)

Pesticide Paradox

If you keep testing over and over with the same test plan, you'll only ever find bugs in that test plan.

You will need to test other aspects of the system and revise the test plan!

Testing is Context-Dependent

How you test

How much you test

What tools you use

What documentation you provide...

All vary based on software context.

Absence of Errors Fallacy

Let's say we gave our fully tested and vetted FizzBuzz implementation to a customer.

Customer says he actually wanted a hamburger and fries.

FINDING DEFECTS DOESN'T
HELP IF IT'S NOT WHAT THE
USER WANTS!

Verification vs Validation

Verification - "Are we building the software right?"

Validation - "Are we building the right software?"

Test Cases / Runs / Plans / Suites

Test Cases

A test case is the lowest level of a test plan.
It consists of:
1. Input values
2. Other preconditions
3. Execution steps (or Procedure)
4. Output values
5. Other postconditions

See IEEE 829, "Standard for Software Test Documentation", for more details

Example

Assuming an empty shopping cart, when I click "Buy Widget", the number of widgets in the shopping cart becomes one.

Preconditions: Empty shopping cart.
Execution Steps: Click "Buy Widget".
Postconditions: Shopping cart displays one widget

Example

Assuming that the SORT_ASCENDING flag is set, calling the sort method with [9,3,4,2] will return a new array sorted from high to low, i.e., [2,3,4,9].

Precondition: SORT_ASCENDING flag is set
Input values: [9,3,4,2]
Execution steps: Call sort method
Output values: [2,3,4,9]

Test Plan

A group of test cases makes up a test plan. These are usually associated functionally or in other ways.

Examples:
Database Connectivity Test Plan
Pop-up Warning Test Plan
Calculator Subsystem Test Plan
Pressure Safety Lock Test Plan
Regression Test Plan

Pressure Safety Lock Test Plan

LOW-PRESSURE-TEST
HIGH-PRESSURE-TEST
SAFETY-LIGHT-TEST
SAFETY-LIGHT-OFF-TEST
RESET-SWITCH2-TEST
RESET-SWITCH1-TEST
PAUSE-MOTOR-TEST
RAPID-CHANGE-TEST
GRADUAL-CHANGE-TEST
MEDIAN-PRESSURE-TEST
LIGHT-FAILURE-TEST
SENSOR-FAILURE-TEST
SENSOR-INVALID-TEST

A Group of Test Plans is a Test Suite

Regression Test Suite

Pressure Safety Regression Test Plan
Power Regulation Regression Test Plan
Water Flow Regression Test Plan
Control Flow Test Plan
Security Regression Test Plan
Secondary Safety Process Test Plan

Test Run

The actual execution of a test plan or suite.

During this execution, the status of each test case is recorded. Possible statuses include:

PASSED
FAILED
PAUSED
RUNNING
BLOCKED
ERROR

PASSED - The test case met all postconditions and expected values. No other problems were observed (e.g., test met all postconditions, but screen turned fuchsia afterwards).

FAILED - The test case did not meet one or more postconditions or expected values, or an unexpected problem occurred.

PAUSED - The test started, but is not complete due to internal factors (e.g., the tester went to lunch).

RUNNING - The test is currently executing. This is useful for long-running background tests.

BLOCKED - The test cannot proceed due to external factors (e.g., waiting for a machine to be available).

ERROR - There is a problem with the test itself. For example, it says "Output should be sorted, returning [1,9,7,2,3]." The test could also not meet requirements. In this case, the tester should discuss with the relevant systems engineer, test writer, or requirements analyst.

If a test fails, a DEFECT should be filed.

Note that this need not be official; if you're doing preliminary testing, it may be sufficient to just talk to the developer.

Test Cases

A test case is the lowest level of a test plan.

It consists of:

1. Input values
2. Other preconditions
3. Execution steps (or Procedure)
4. Output values
5. Other postconditions

See IEEE 829, "Standard for Software Test Documentation", for more details

Example

Assuming an empty shopping cart, when I click "Buy Widget", the number of widgets in the shopping cart becomes one.

Preconditions: Empty shopping cart

Execution Steps: Click "Buy Widget"

Postconditions: Shopping cart displays one widget

Example

Assuming that the SORT_ASCENDING flag is set, calling the sort method with [9,3,4,2] will return a new array sorted from high to low, i.e., [2,3,4,9].

Precondition: SORT_ASCENDING flag is set

Input values: [9,3,4,2]

Execution steps: Call .sort method

Output values: [2,3,4,9]

Test Plan

A group of test cases makes up a test plan. These are usually associated functionally or in other ways.

Examples:

Database Connectivity Test Plan

Pop-up Warning Test Plan

Calculator Subsystem Test Plan

Pressure Safety Lock Test Plan

Regression Test Plan

Pressure Safety Lock Test Plan

LOW-PRESSURE-TEST

HIGH-PRESSURE-TEST

SAFETY-LIGHT-TEST

SAFETY-LIGHT-OFF-TEST-

RESET-SWITCH-TEST

RESET-SWITCH2-TEST

FAST-MOVEMENT-TEST

RAPID-CHANGE-TEST

GRADUAL-CHANGE-TEST

MEDIAN-PRESSURE-TEST

LIGHT-FAILURE-TEST

SENSOR-FAILURE-TEST

SENSOR-INVALID-TEST

A Group of Test Plans is a Test Suite

Regression Test Suite

Pressure Safety Regression Test Plan

Power Regulation Regression Test Plan

Water Flow Regression Test Plan

Control Flow Test Plan

Security Regression Test Plan

Secondary Safety Process Test Plan

Test Run

The actual execution of a test plan or suite.

During this execution, the status of each test case is recorded. Possible statuses include:

PASSED

FAILED

PAUSED

RUNNING

BLOCKED

ERROR

PASSED - The test case met all postconditions and expected values. No other problems were observed (e.g., test met all postconditions, but screen turned fuchsia afterwards).

FAILED - The test case did not meet one or more postconditions or expected values, or an unexpected problem occurred.

PAUSED - The test started, but is not complete due to internal factors (e.g., the tester went to lunch).

RUNNING - The test is currently executing. This is useful for long-running background tests.

BLOCKED - The test cannot proceed due to external factors (e.g., waiting for a machine to be available).

ERROR - There is a problem with the test itself. For example, it says "Output should be sorted, returning [1,9,7,2,3]." The test could also not meet requirements. In this case, the tester should discuss with the relevant systems engineer, test writer, or requirements analyst.

If a test fails, a DEFECT should be filed.

Note that this need not be official; if you're doing preliminary testing, it may be sufficient to just talk to the developer.

Test Tracking

Record All Test Runs!

1. Keep track of date, version of software
2. Which test run produced a defect
3. Which test case produced a defect
4. Any other relevant information

Example

Project: Calculator
Test Plan: Multiplication Functionality
Test Run Date: 14 JAN 2014
SW Version: 1.0 Build 141 (RC1)
Defects Found: 108 (Case 6), 110 (Case 7)
Executed on Machine #75

If you're using test tracking software, much of this will be automated.

If I can leave you with one thing...

...more details are better when testing, almost always.

Best to mark down as you go. Don't wait.
File defects ASAP.
Report errors in tests ASAP.
Ask questions ASAP.

When executing a test, the worst thing to do is sit around staring and thinking. You will usually save time by asking someone.

This applies to most software development tasks.

Examples of Test Tracking Software

Mozilla Testopia
HP Quality Manager
IBM Rational Quality Manager
TestLink
Tosca Testsuite

Record All Test Runs!

1. Keep track of date, version of software
2. Which test run produced a defect
3. Which test case produced a defect
4. Any other relevant information

Example

Project: Calculator

Test Plan: Multiplication Functionality

Test Run Date: 14 JAN 2014

SW Version: 1.0 Build 141 (RC1)

Defects Found: 108 (Case 6), 110 (Case 7)

Executed on Machine #75

If you're using test tracking software, much of this will be automated.

If I can leave you with one thing...

**...more details are better when testing,
almost always.**

**Best to mark down as you go. Don't wait.
File defects ASAP.
Report errors in tests ASAP.
Ask questions ASAP.**

**When executing a test, the worst thing to do
is sit around staring and thinking. You will
usually save time by asking someone.**

**This applies to most software development
tasks.**

Examples of Test Tracking Software

Mozilla Testopia

HP Quality Manager

IBM Rational Quality Manager

TestLink

Tosca Testsuite

Tying Defects to Tests

When a defect is found...

1. That test should FAIL.
2. If the defect already exists in the bug tracking system, note that it failed due to defect X.
3. If a defect does not already exist, create a new defect.

We'll Talk More About Describing Defects Next Class...

but for our purposes today, you want to track...

1. Software Version discovered
2. Date discovered
3. Test Run discovered
4. Test Case discovered

Note the pattern :

Everything is traceable and includable.
Traceability and trackability lead to
reproducibility.
Reproducibility leads to resolution.

When a defect is found...

1. That test should FAIL
2. If the defect already exists in the bug tracking system, note that it failed due to defect X
3. If a defect does not already exist, create a new defect

We'll Talk More About Describing Defects Next Class...

but for our purposes today, you want to track...

1. Software Version discovered
2. Date discovered
3. Test Run discovered
4. Test Case discovered

Note the pattern -

Everything is traceable and trackable.

**Traceability and trackability lead to
reproducibility.**

Reproducibility leads to resolution.

Deliverable 1

CS1699 - Lecture 4 - Test Plans

FizzBuzz

FizzBuzz
Project Details
The fizzbuzz problem is a popular integral part of many introductory computer science courses. It's the first step in teaching the basics of loops, conditionals, and arrays. In its most basic form, it asks the student to print numbers from 1 to 100. For each number, if it is divisible by 3, print "Fizz". If it is divisible by 5, print "Buzz". If it is divisible by both 3 and 5, print "FizzBuzz". This is a good exercise for learning how to work with loops and conditionals.

Example Output
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
18
Fizz
20
Buzz
22
23
24
FizzBuzz
26
27
28
Fizz
30
Buzz
32
33
34
FizzBuzz
36
37
38
Fizz
40
Buzz
42
43
44
FizzBuzz
46
47
48
Fizz
50
Buzz
52
53
54
FizzBuzz
56
57
58
Fizz
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
Buzz

Let's Create one with a Test Plan
Test Plan
Title: FizzBuzz
Description: This is a simple test plan for the FizzBuzz problem.
Preconditions:
- The code is written in Python.
- The code is well-documented.
- The code is tested and passes all unit tests.
- The code is reviewed and approved by a peer.
- The code is checked into version control.
- The code is deployed to a production environment.
- The code is monitored for performance and stability.

Developing a Test Plan

So you have requirements...
how do you test them?

Think About What You're Testing
1. Throw-away script?
2. Development tool?
3. Internal website?
4. Enterprise software?
5. Operating system?
6. Avionics software?

Your test plan should flow from the functional requirements through the non-functional requirements.

Example
System shall fail gracefully.
This will mean something very different for an MRI machine and a website.

The Seven Testing Principles
1. Testing shows presence of defects.
2. Complete testing is impossible.
3. Test early.
4. Defect clustering.
5. The "pesticide paradox"
6. Testing is context-dependent.
7. Absence of errors fallacy.
—Sjoerd Visscher, Software Quality Research Group, University of Twente

Testing Shows Presence of Defects
You can never really "prove" software is free of defects.
Absence of evidence IS NOT evidence of absence.
"You, there are "formally proven" programs that have a hole in a vehicle other than the one they were designed for. A rabbit hole to go down, and this concept is going to come up for maybe 1% of testers."

Testing Can Never Show All Defects or Possible Failure Cases
1. What if an asteroid destroys the planet, but the entire surface will be covered?
2. Will systems work correctly under 100% load?
3. What happens when memory fails at instruction 1, 2, 3, etc?
4. Concurrency and parallelism?
5. What if the value of pi was slightly shifted in our Universe, would program work correctly?

Exhaustive Testing is Impossible
Could you prove a spellchecker could work with every possible input?

Early Testing
It is always, always, always better to discover a problem sooner rather than later.
Report early, report often.
Don't wait to test.

Defect Clustering
Defects tend to cluster in certain areas of code or functionality.
Usually due to technical difficulty of implementing bad requirements, etc.
If you find many errors in one module, don't assume that others are "bad".
(Guru's Fallacy)

Pesticide Paradox
If you keep testing over and over with the same test plan, you'll only ever find bugs in that test plan.
You will need to test other aspects of the system and revise the test plan.

Testing is Context-Dependent
How you test.
How much you test.
What tools you use.
What documentation you provide...
All very based on software context.

Absence of Errors Fallacy
Often the question is "Is the software fully tested and verified?"
Implementation is the answer.
Customer says he initially wanted a different product.
PRODUCT IS NOT DEFECT-FREE IF IT'S NOT WHAT THE USER WANTED.

Verification vs Validation
Verification - "Are we building the right software?"
Validation - "Are we building the software right?"

Test Cases / Runs / Plans / Suites



Deliverable 3

Tying Defects to Tests

Test Tracking

