

CS1699 - Lecture 4 - Test Plans



Developing a Test Plan

Developing a Test Plan

Test Cases / Runs / Plans / Suites

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.

Test Cases
A test case is a set of steps to be followed by a tester to exercise a specific feature or function of a system. It includes the expected outcome for each step.

Run
A run is a collection of test cases executed against a specific build of a system. It includes the results of each test case, such as pass/fail status and any associated notes.

Plan
A test plan is a document that defines the scope, objectives, and approach for testing a system. It includes details about the test environment, resources required, and timelines.

Suite
A test suite is a collection of test cases grouped together based on a common purpose or context. It can include multiple runs and plans.

Test Case Template
A template for creating test cases, typically used in a spreadsheet format. It includes fields for test ID, description, steps, expected result, and actual result.

Test Run Template
A template for tracking the execution of test cases. It includes columns for test case ID, status, date/time, and any notes or attachments.

Test Plan Template
A template for creating test plans. It includes sections for project information, test strategy, test environment, and test schedule.

Test Suite Template
A template for managing test suites. It includes fields for suite name, description, and a list of included test cases.

The Seven Testing Principles
1. Testing shows presence of defects.
2. Exhaustive testing is impossible.
3. Test early and often.
4. Defects cluster.
5. Avoid the "pesticide paradox".
6. Testing is context dependent.
7. Absence of errors is fallacy.
Source: The Seven Testing Principles, Michael Feathers

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 out there. We'll talk about it later this semester. They are rare.

Testing Can Never Show All Defects or Possible Failure Cases
1. What if an asteroid destroys the crust of our planet, bashing the entire surface will leave? 2. Will system work correctly under 100 Tesla of magnetism? 3. What would happen when millions fails at instruction 1, 2, 3, ... etc? 4. Concurrency and parallelism? 5. What if the value of pi was slightly shifted to 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 are clustered 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 "fine".
(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
Let's say we gave a lottery ticket and without doubt the person who won is the only one who has verified his validation.

Customer says he suddenly started to buy lottery and has "verified" his validation.

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

FINDING DEFECTS DOESN'T HELP IF IT'S NOT RELEVANT (IN THE RIGHT WAY)

CS1699 - Lecture 4 - Test Plans



Developing a Test Plan

Developing a Test Plan

Test Cases / Runs / Plans / Suites

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.

Test Cases
A test case is a set of steps to be followed by a tester to exercise a specific feature or function of a system. It includes the expected outcome for each step.

Run
A run is a collection of test cases executed against a specific build of a system. It includes the results of each test case, such as pass/fail status and any associated notes.

Plan
A test plan is a document that defines the scope, objectives, and approach for testing a system. It includes details about the test environment, resources required, and timelines.

Suite
A test suite is a collection of test cases grouped together based on a common purpose or context. It can include multiple runs and plans.

Test Case Template
A template for creating test cases, typically used in a spreadsheet format. It includes fields for test ID, description, steps, expected result, and actual result.

Test Run Template
A template for tracking the execution of test cases. It includes columns for test case ID, status, date/time, and any notes or attachments.

Test Plan Template
A template for creating test plans. It includes sections for project information, test strategy, test environment, and test schedule.

Test Suite Template
A template for managing test suites. It includes fields for suite name, description, and a list of included test cases.

The Seven Testing Principles
1. Testing shows presence of defects.
2. Exhaustive testing is impossible.
3. Test early and often.
4. Defects cluster.
5. Avoid the "pesticide paradox".
6. Testing is context dependent.
7. Absence of errors is fallacy.
Source: The Seven Testing Principles, Michael Feathers

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 out there. We'll talk about it later this semester. They are rare.

Testing Can Never Show All Defects or Possible Failure Cases
1. What if an asteroid destroys the crust of our planet, bashing the entire surface will leave? 2. Will system work correctly under 100 Tesla of magnetism? 3. What would happen when millions fails at instruction 1, 2, 3, ... etc? 4. Concurrency and parallelism? 5. What if the value of pi was slightly shifted to 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 are clustered 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 "fine".
(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
Let's say we gave a lottery ticket and without doubt the person who won is the only one who has verified his validation.

Customer says he suddenly started to buy lottery and has "verified" his validation.

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

FINDING DEFECTS DOESN'T HELP IF IT'S NOT RELEVANT (IN THE RIGHT WAY)

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 provided, separated by newlines.
4. The output shall be 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 print "Fizz".
4. If the current number being printed is evenly divisible by 5 print "Buzz".
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".

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. Defects cluster.
5. Avoid 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. We'll talk about it later this semester! They are very rare.

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?

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 a fully tested and vetted FizzBuzz implementation to a customer.
Customer says he actually wanted a hamburger and fries (verification vs validation).
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. Defects cluster.
5. Avoid 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. We'll talk about it later this semester! They are very rare.

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 a fully tested and vetted FizzBuzz implementation to a customer.

Customer says he actually wanted a hamburger and fries (verification vs validation).

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. Preconditions
- 3. Execution steps (or Procedure)
- 4. Output values
- 5. 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 Regulation 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.

Developing A Test Plan or Suite

Start top-down: what is a good way to subdivide the system into features (test plans)?

For a given feature (test plan), what aspects do I want to test?

For each aspect, what test cases do I want that will hit different equivalence classes / success or failure cases / edge or corner cases / etc.? How deep should I go down?

Try to have test cases be independent of each other, and reproducible!

Test Cases

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

It consists of:

1. Input values
2. Preconditions
3. Execution steps (or Procedure)
4. Output values
5. 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

RUNNING

FAILED

BLOCKED

PAUSED

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.

Developing A Test Plan or Suite

Start top-down: what is a good way to subdivide the system into features (test plans)?

For a given feature (test plan), what aspects do I want to test?

For each aspect, what test cases do I want that will hit different equivalence classes / success or failure cases / edge or corner cases / etc.? How deep should I go down?

Try to have test cases be independent of each other, and reproducible!

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?

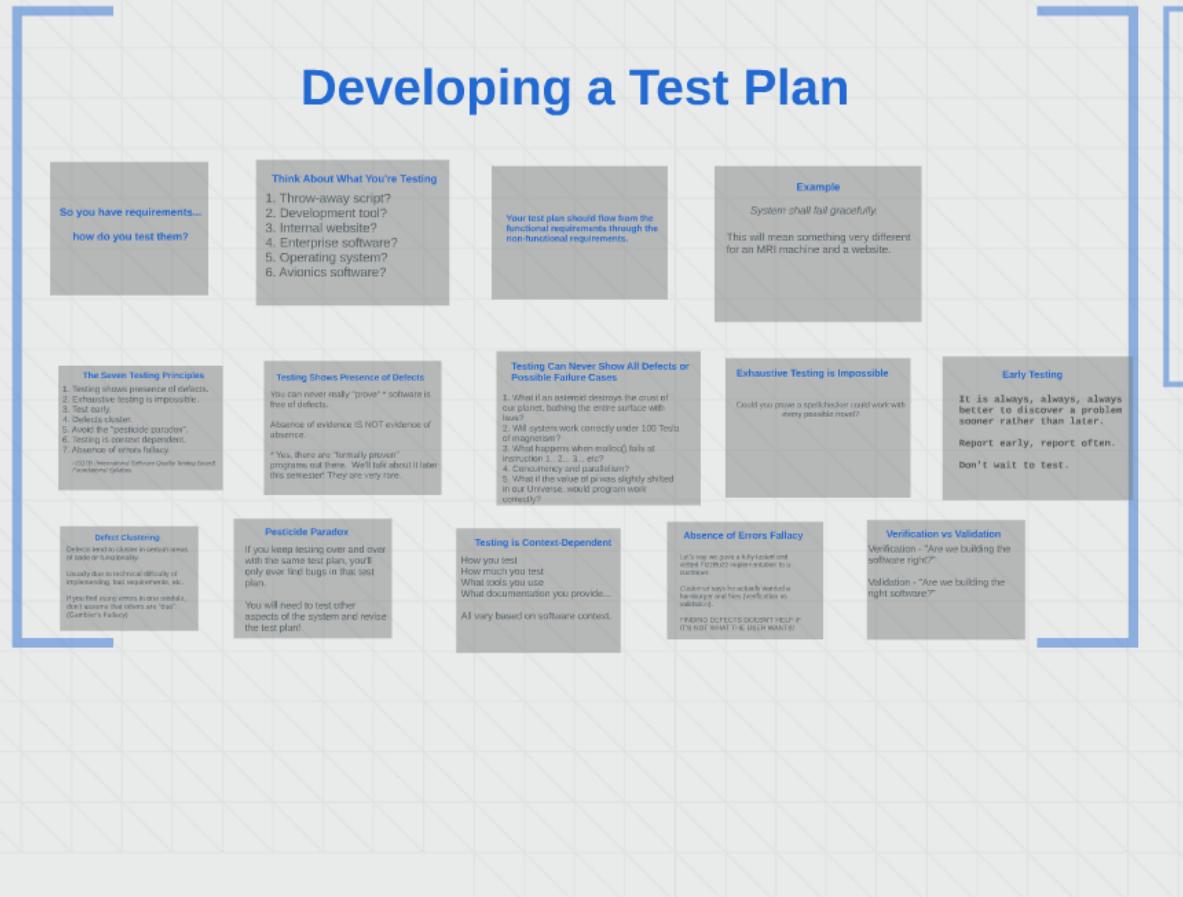
Success vs Failure Cases?

Edge cases, Corner/Pathological cases?

CS1699 - Lecture 4 - Test Plans



Developing a Test Plan



Test Cases / Runs / Plans / Suites

