

# CS1632: Test Plans and TM

Wonsun Ahn

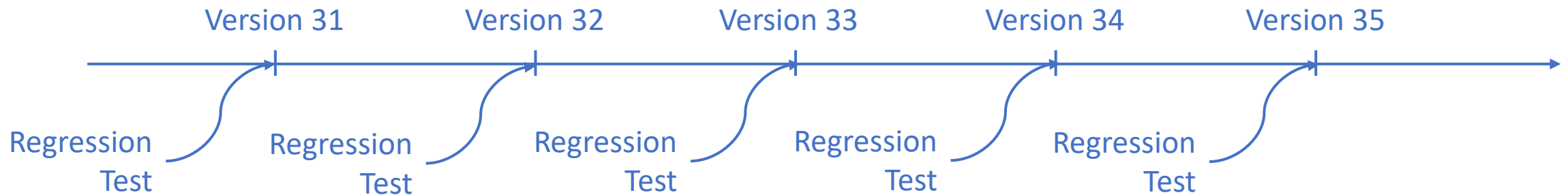
# Test Plans

# What is a Test Plan?

- Test Plan: A document laying out a plan for testing a software system
- Why do we need a **plan**?
  - Goal of testing is to minimize risk of defects given a time/cost budget
  - Careful planning is needed to maximize test coverage
- Why do we need to **document** the plan?
  - Allows project managers to estimate test coverage and manage risk
  - Allows quality engineers to reliably repeat the same tests over and over again
  - **Repeatability** of tests is particularly important for *regression tests*

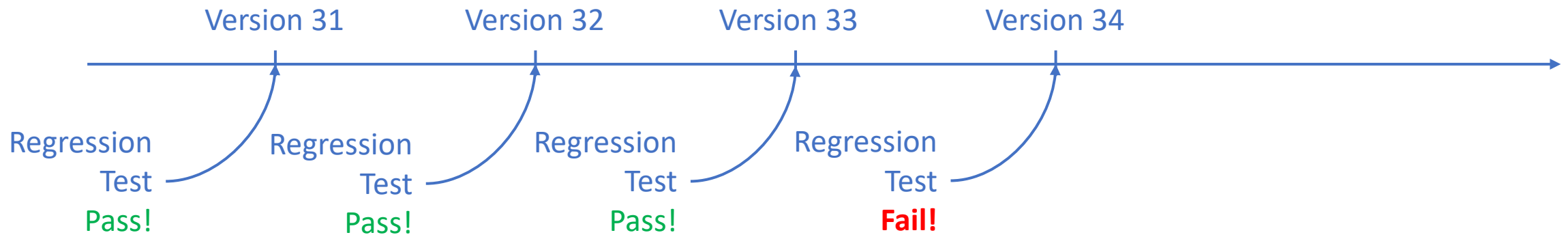
# Regression Tests prevent SW from regressing

- *Regression*: A failure of a previously working feature
  - Can be caused by (seemingly) unrelated enhancements or defect fixes
  - Why? Because code fixes often have non-local effects
  - Regression test must test modified feature but also all other features
- For timely regression detection, regression test is run on each code update



# Repeatable tests can pinpoint defective version

- Suppose a regression test fails on a code update:



- We can pinpoint where the defect crept in, at Version 34.
- Why? Because we are confident that we are repeating the same tests!
- Now, if we ran different tests each time, would we know?

# How formal should the plan be documented?

- As formal or informal as necessary!
- Think about what you are testing
  - How critical is the software that you are testing?
  - How many times is the test plan going to be used?

# What are you testing?

- Throw-away script?
- Development tool?
- Internal website?
- Enterprise software?
- Commercial software?
- Operating system?
- Avionics software?

# 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.



# Test Cases

# Test Plans and Test Cases

- A test plan consists of a list of related test cases that are run together
- *Test case*: a test scenario with precise steps on how to perform it
  - Describes what is to be tested and what steps to perform
  - Describes expected behavior after the steps are performed

# Test Case main body consists of ...

- *Preconditions*: State of system before execution steps. E.g.,
  - Packages X and Y are installed on the system
  - Configuration file X contains entry Y
  - Database has table X set up populated with Y entry
- *Execution Steps*: Steps to perform test
- *Postconditions*: **Expected** state after execution steps
  - Derived from **requirements** based on preconditions + execution steps

# Test Case header identifies and describes it

- *Identifier*: A way to identify the test case
  - Could be numerical, e.g. TC-452
  - Or a descriptive label, e.g. INVALID-PASSWORD-THREE-TIMES-TEST
- *Test Case*: A short description of what is being tested

In full, a test case contains the following items

- Identifier
- Test Case
- Preconditions
- Execution Steps
- Postconditions

See IEEE 829, "Standard for Software Test Documentation", at [resources/IEEE829.pdf](#)

# Example Test Case

- Identifier: SORT-ASCENDING-FOUR-INTEGERS-TEST
- Test Case: When SORT\_ASCENDING flag is set, calling sort([9,3,4,2]) returns a new sorted array [2,3,4,9].
- Preconditions: SORT\_ASCENDING global variable is set to true.
- Execution Steps:
  1. Set test\_array = [9,3,4,2].
  2. Call sort(test\_array).
- Postconditions: Return value of sort(test\_array) is array [2,3,4,9].

# Test Run – Actual execution

- *Test run*: Actual execution of a test case
  - Subsets of test cases may be chosen to run from the entire test suite
  - All depends on the type of code modification and the testing context
- The purpose of a test run is to obtain *observed behavior*
  - Passes or fails after comparing *observed behavior* with *postcondition*

# Status after Test Run

- Possible Statuses
  - PASSED: Completed with expected result
  - FAILED: Completed but unexpected result
  - PAUSED: Test paused in middle of execution
  - RUNNING: Test in the middle of execution
  - BLOCKED: Did not complete because precondition not fulfilled
  - ERROR: Problem with running test itself
- During test run, tester manually (or automatically) executes each test case and sets the status for each
- A FAILED status signals a defect that needs to be reported.



# Creating Good Test Cases

# A good test case is two things

- A good test case verifies **requirements faithfully**
  - No false negative defects: all defective behaviors results in failures
  - No false positive defects: all correct behaviors results in passes
- A good test case is **repeatable**
  - Test results are consistent regardless of who / when / where the tests are run
  - Preconditions + Execution Steps guarantee consistent results

# Pitfall 1:

## Using Observed Behavior for Postcondition

- Never use screenshots (or copy-and-paste) of output even if correct
  - Screenshots contain spurious info that result in **false positive defects**
- Suppose requirement is: “The sum of the 2 arguments is displayed.”
- Execution Step is: “Pass in values of 1 and 2 as arguments.”
- Postcondition is: `Result is: 3` is displayed.
- Observed behavior is: `Value is: 3`
- The test case would fail, but is this a defect? No!
- Postconditions should be derived from **requirements**
  - Correct postcondition: “The result value of 3 is displayed.”

# Pitfall 2: Using Requirement for Postcondition

- Do not paste requirements verbatim as postconditions
  - Forces tester to derive expected behavior based on own interpretation
  - Results in **unrepeatable** tests as well as both false negative and positive defects
- Suppose requirement is:  
“Fibonacci sequence following number given as argument is returned.”
- Execution Step is: “Run program passing 5 as argument.”
- Postcondition is: (a copy of the above requirement)
- Would the tester be able to tell what the expected return value should be?
- Postconditions should describe expected behavior explicitly
  - Correct postcondition: “The value of 8 is returned.”

# Pitfall 3: Imprecise Preconditions / Execution Steps

- Incomplete preconditions (OS / DB / Filesystem / Memory state)
  - E.g. OS environment variable that impacts test case is not specified
  - E.g. A configuration file that impacts test case is not specified
- Imprecise execution steps
  - E.g. “Open new browser” → Multiple ways: Ctrl+N, Menu, Double click
- Results in **unrepeatable** tests

# Pitfall 4 - A Test Case that is not Independent

- Test case shouldn't depend on the execution of a previous test case
  - E.g. Should not depend on database entries inserted by previous test case
- Results in **unrepeatable** tests
  - If the previous test case fails, this test case will be impacted
  - Test cases may be run selectively, and previous case may not be selected to run
  - Test cases may execute out of order, causing previous case to execute later (Often, test cases are executed in parallel to save testing time)

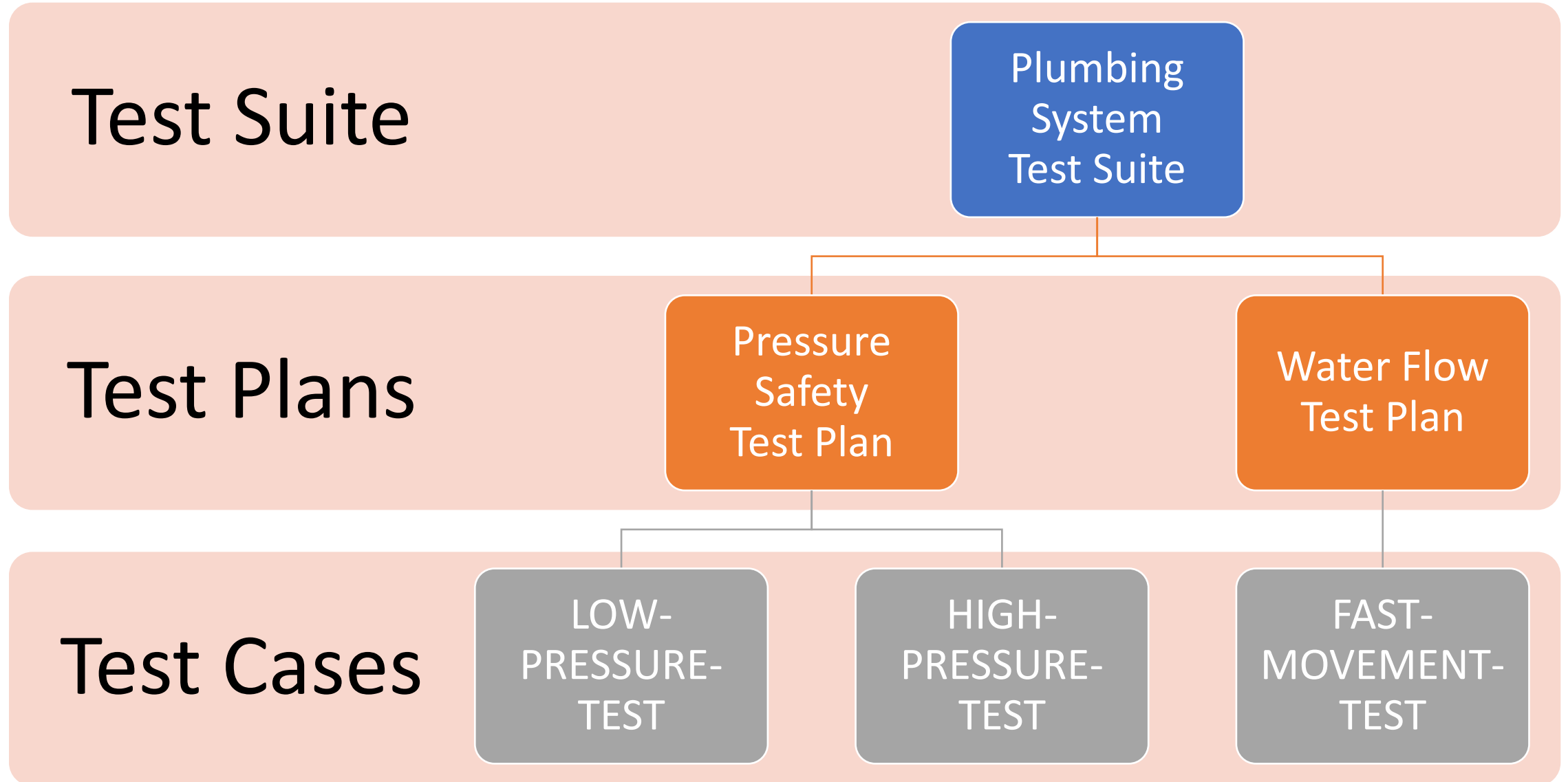
# Pitfall 5 - A Test Case testing Multiple Scenarios

- Test case shouldn't merge multiple scenarios into one
  - On fail, cannot tell easily which scenario resulted in test failure
  - Failure of a previous scenario may prevent accurate testing of a later scenario
- Example of a merged test case:
  - Execution Steps:
    1. Call `sqrt(4)`
    2. Call `sqrt(9)`
    3. Call `sqrt(16)`
  - Postconditions:
    - Results of first, second, and third calls are 2, 3, 4, respectively.

# Testing Hierarchy



A group of test plans make up a *test suite*...



# Creating a test suite from requirements

- Take top-down approach to create hierarchy of test plans and cases.
  1. Subdivide system into features or subsystems
  2. For each feature, create a test plan with varied inputs + preconditions
  3. For each input + precondition, create a test case
- Test base / edge / corner cases for each feature to maximize coverage.

# Traceability Matrix

# Traceability: Ability to trace requirements to test cases (and vice versa)

- Forward Traceability
  - Ability to trace **requirement** → **test cases**
  - Given a requirement, allows listing of all test cases that test it
  - Ensures there are no requirements with insufficient test coverage
- Backward Traceability
  - Ability to trace **test case** → **requirements**
  - Given a test case, allows listing of all requirements that are tested
  - Ensures there are no test cases that are not testing any requirements  
→ “Orphaned” test cases need to be removed, along with the implementation
- Ensures requirements, and only requirements, are implemented

# Traceability Matrix ensures traceability

- **Traceability Matrix:**  
Table describing relationship between requirements and test cases
- Why is it a “matrix”?
  - One test case may test multiple requirements
  - One requirement may be tested by multiple test cases
  - It's a many-to-many relationship, hence the matrix

# Good Forward Traceability Matrix Example

REQ1: TEST\_CASE\_1, TEST\_CASE\_2

REQ2: TEST\_CASE\_1, TEST\_CASE\_3

REQ3: TEST\_CASE\_1

REQ4: TEST\_CASE\_2

REQ5: TEST\_CASE\_4

- Mapping requirements → test cases
- All requirements have at least one test case testing that requirement
- All requirements have *\*some\** test coverage

# Bad Forward Traceability Matrix Example

REQ1: TEST\_CASE\_1, TEST\_CASE\_2

REQ2:

REQ3: TEST\_CASE\_1

REQ4: TEST\_CASE\_2

REQ5: TEST\_CASE\_4

- *No test case is testing requirement 2!*
- *Add test cases for requirement 2!*

# Good Backward Traceability Matrix Example

TEST\_CASE\_1: REQ1, REQ2, REQ3

TEST\_CASE\_2: REQ1, REQ4

TEST\_CASE\_3: REQ2

TEST\_CASE\_4: REQ5

- Mapping test cases → requirements
- All test cases have at least one requirement it is testing



# Bad Backward Traceability Matrix Example

TEST\_CASE\_1: REQ1, REQ2, REQ3

TEST\_CASE\_2: REQ1, REQ4

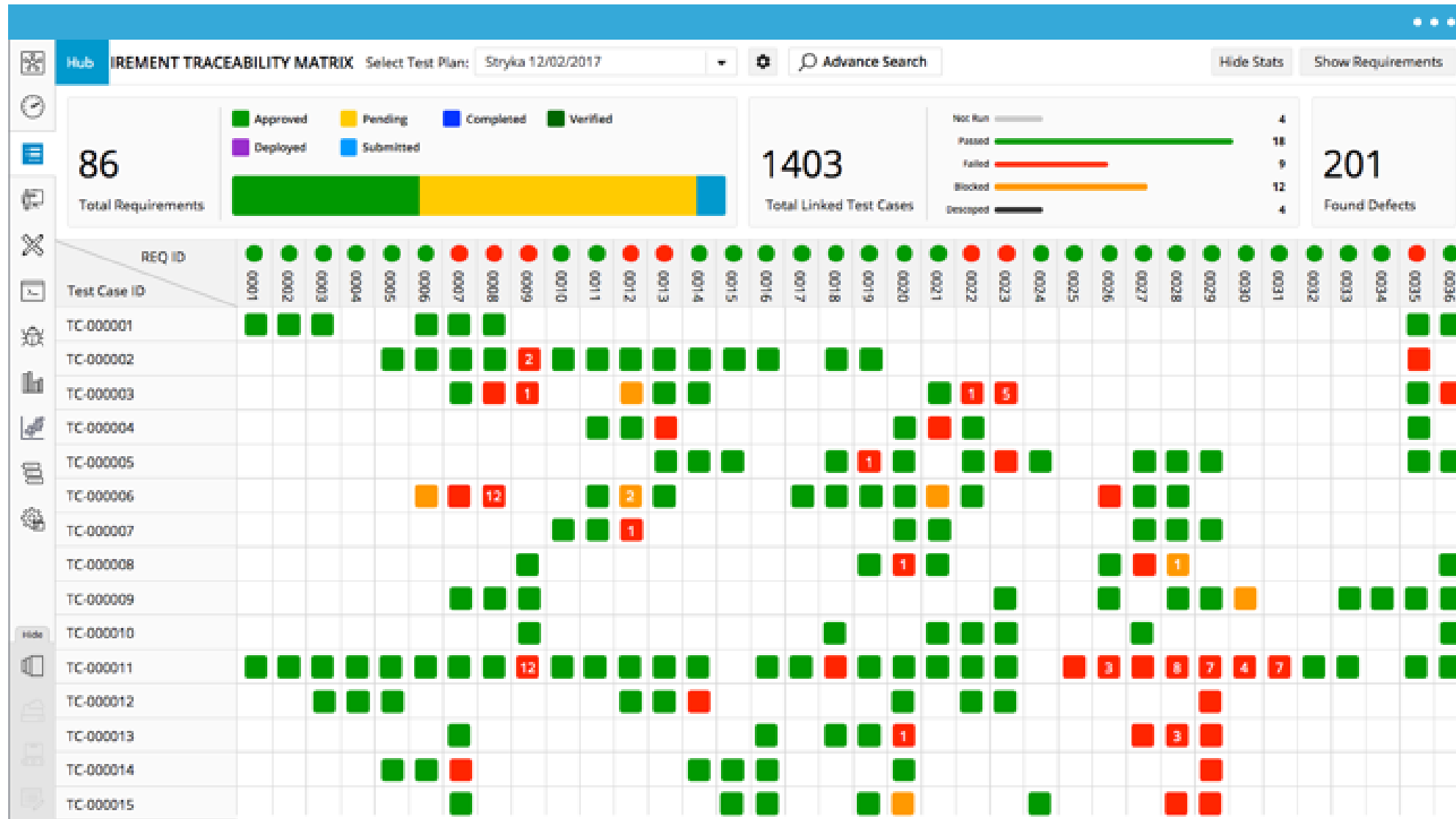
TEST\_CASE\_3: REQ2

TEST\_CASE\_4: REQ5

TEST\_CASE\_5:

- *Test case 5 not checking any requirement*
- *Remove test case 5 along with the implementation code!*

# A Bi-Directional Traceability Matrix



Reference:  
reportportal.io

# Now Please Read Textbook Chapters 6 and 8

- If you are interested in further reading:

## **IEEE Standard for Software Test Documentation (IEEE 829-2008)**

- Can be found in [resources/IEEE829.pdf](#) in course repository