

# CS1632: Test Plans and TM

Wonsun Ahn

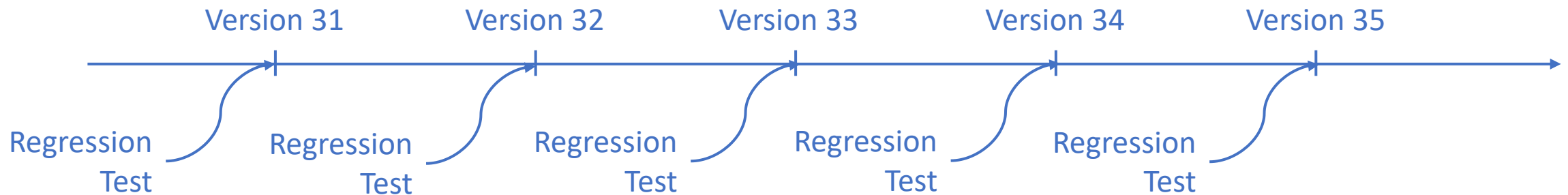
# Test Plans

# 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 can maximize test coverage with a limited number of tests
- 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*

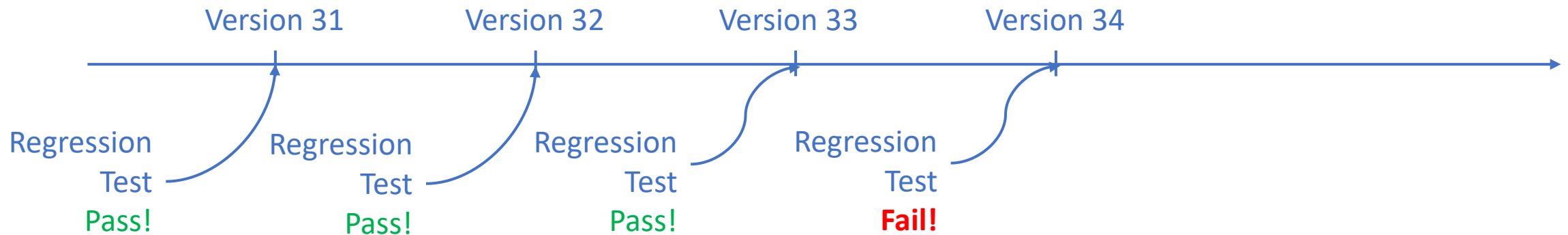
# Regression Tests prevent SW from regressing

- *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 updated 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*: Smallest unit of a test plan that tests an individual behavior
  - You can think of one input value as one test case
  - 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 the system before testing
  - Environment / global variable values, ...
  - State of the screen, state of the database, ...
- *Execution Steps*: Steps to perform test
- *Postconditions*: **Expected** state of the system after testing
  - Environment / global variables are set, ...
  - Output printed to screen, network packet sent, ...

# 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: ADD-ONE-WIDGET-TO-CART-TEST
- Test Case: When shopping cart is empty, when I add one widget to the cart, the number of widgets in the cart should become one.
- Preconditions: Empty shopping cart
- Execution Steps:
  1. Select first widget from the list of widgets by clicking on the checkbox
  2. Click “Add to Cart” button
- Postconditions: Shopping cart displays one widget

# Another Example Test Case

- Identifier: SORT-ASCENDING-FOUR-INTEGERS-TEST
- Test Case: When SORT\_ASCENDING flag is set, calling sort([9,3,4,2]) should return 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]

# Creating Good Test Cases

- Besides good test coverage, what else makes a good test case?
- A good test case as the following additional characteristics:
  - Reproducibility
  - Independence



# A Good Test Case is Reproducible

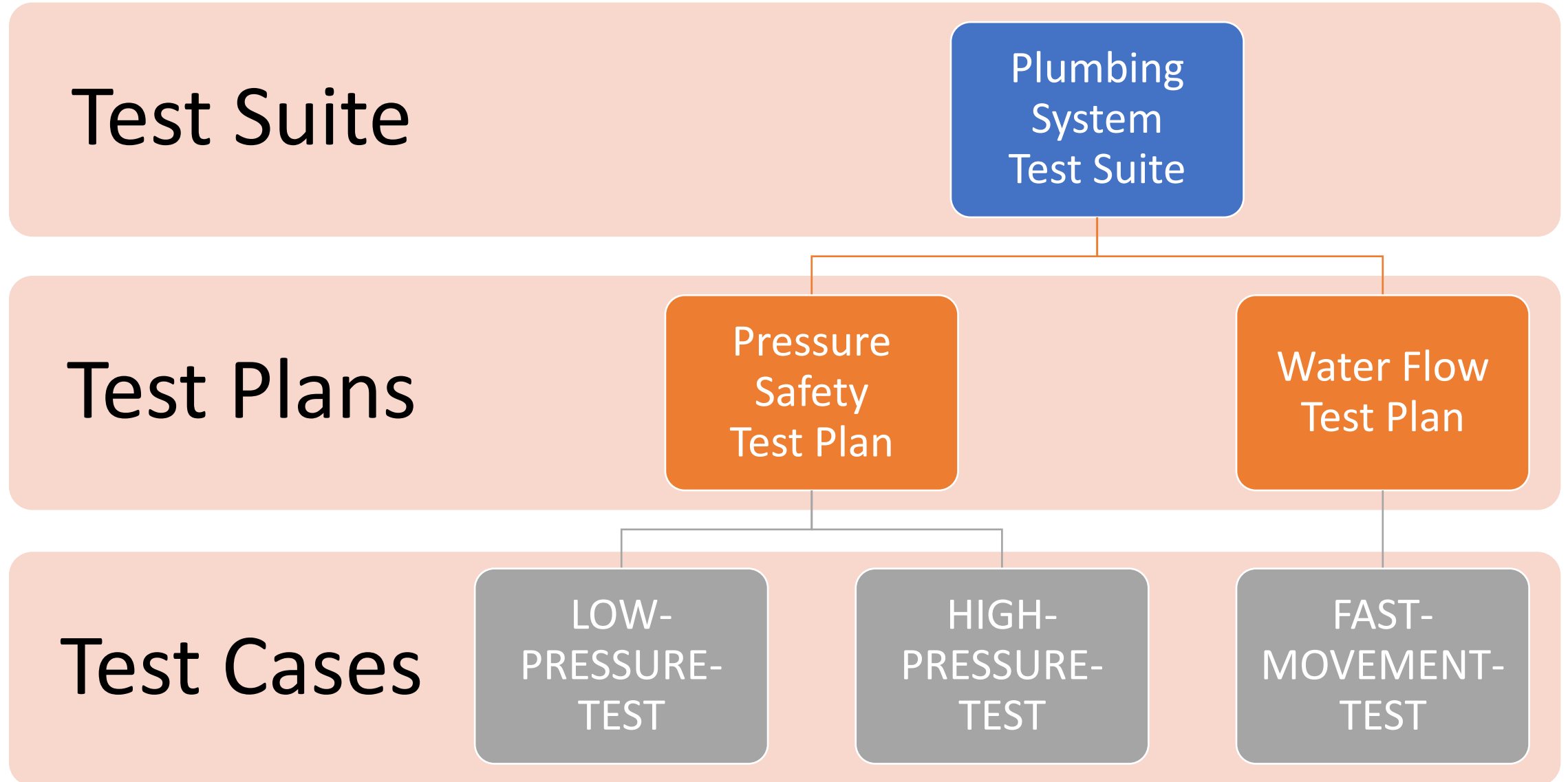
- Preconditions + Execution Steps always result in same behavior
- What happens when a test case is unreproducible?
  - Defect found by test may not manifest when developer tries to debug it
  - Test does not find defect but defect manifests when software is deployed
- What causes a test case to be unreproducible?
  - Incomplete preconditions (OS state, DB state, filesystem state, 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
    - E.g. Java compiler version that impacts test case is not specified
  - Imprecise execution steps
    - E.g. “Open new browser window” → Multiple ways: Ctrl+N, Menu, Icon double click

# A Good Test Case is 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
  - Test cases may be executed selectively, meaning previous may not execute
  - Test cases may execute out of order, causing previous case to execute later  
(Often, test cases are run in parallel to save testing time)
- What causes a test case to be dependent?
  - Relying on previous test case to fulfill part of the preconditions
  - All preconditions should be explicitly specified and set up before each test case

# Test Suites

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



# Creating a test suite from requirements

- Start top-down
- Subdivide system into features or subsystems
  - Create a test plan for each of those features
- For each feature, decide on what aspects to test
- For each aspect, decide on which inputs or user interactions to test
  - Create a test case for each input, under the subsystem test plan
  - Hit different base, edge, and corner cases for good test 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 cases (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

- *All requirements have at least one test case associated with them*
- *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 one or more test cases.*

# Good Backward Traceability Matrix Example

TEST\_CASE\_1: REQ1, REQ2, REQ3

TEST\_CASE\_2: REQ1, REQ4

TEST\_CASE\_3: REQ2

TEST\_CASE\_4: REQ5

- *All test cases have at least one requirement associated with them*

# 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!*

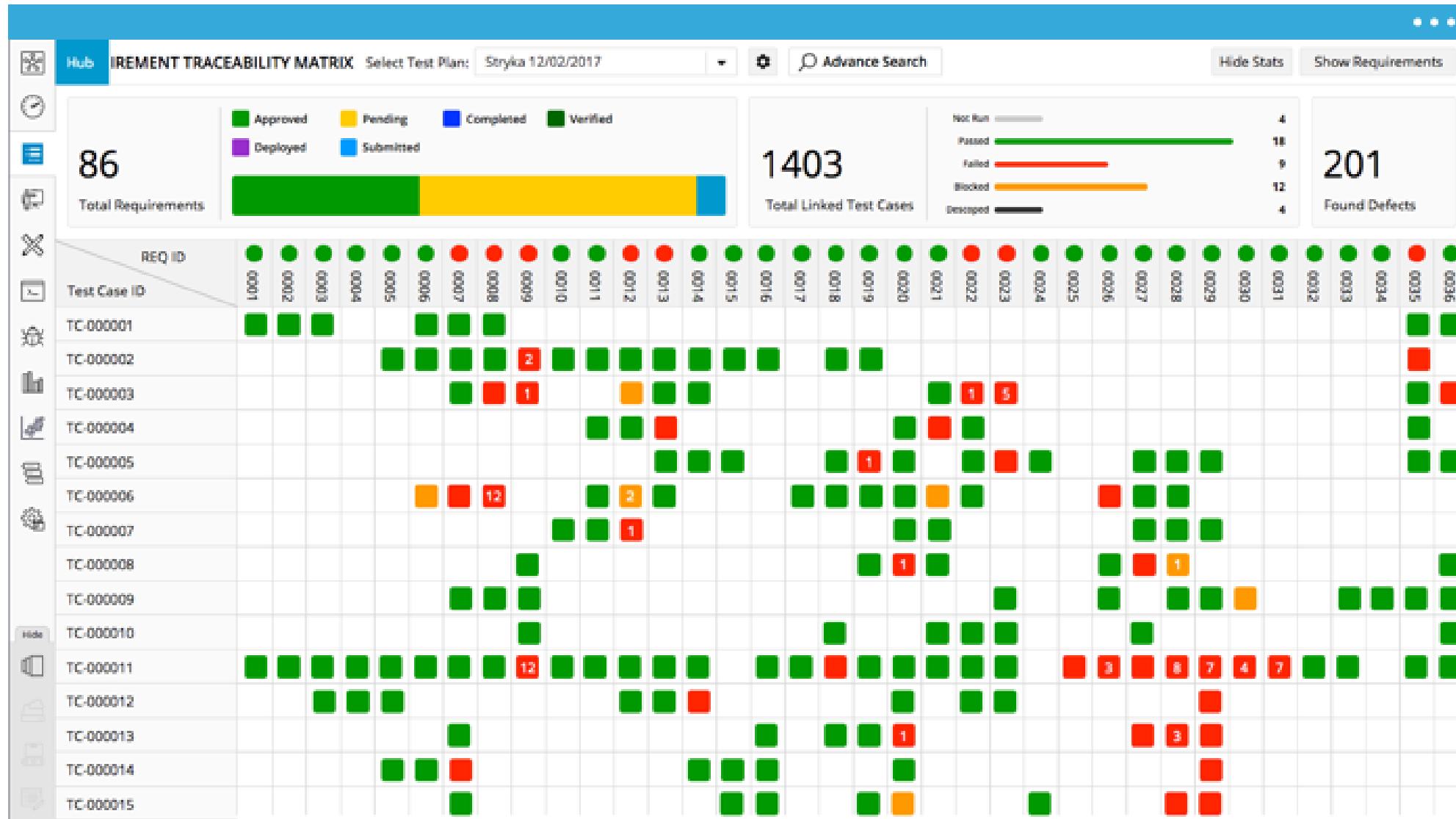
# Test Run – Actual execution

- *Test run*: Actual execution of a test case / test plan / test suite
  - Subsets of test cases may be chosen to run or 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!

# A Bi-Directional Traceability Matrix



Reference:  
reportportal.io

# Defects

- If the test case fails, a defect should be filed
- We will talk about filing defects on the next lecture



# Now Please Read Textbook Chapters 6 and 8

- In particular, read Chapter 8 carefully since that's mostly what you will be doing for our first in-class exercise next week.
- 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