Software Verification and Validation

L.EIC-ES-2021-22

J. Pascoal Faria & Ademar Aguiar

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

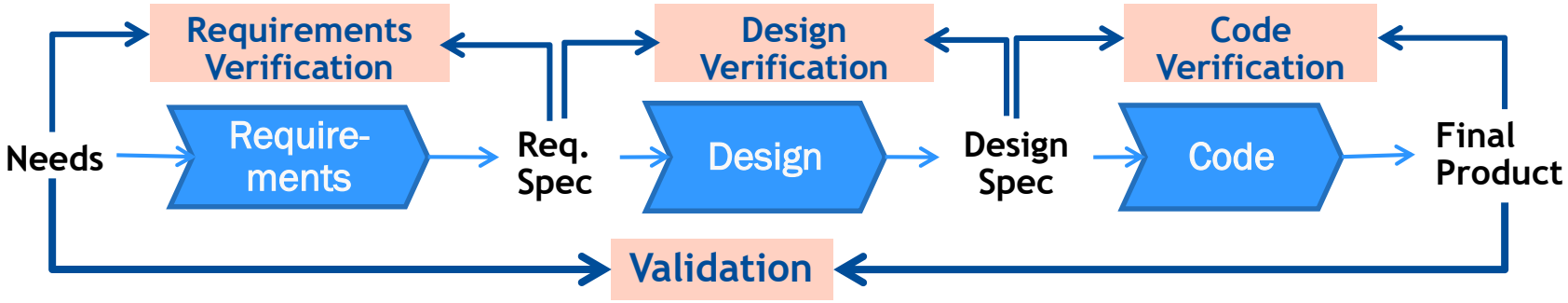
Agenda

- Verification & validation
- Software reviews & inspections
- Software testing
- Demo: Testing Flutter apps

Verification & Validation

Verification versus Validation

- **Verification – are we building the product right?**
  - Ensure (mainly through reviews) that intermediate work products and the final product are "well built", i.e., conform to their specifications
- **Validation – are we building the right product?**
  - Ensure (manly through tests) that the final product will fulfill its intended use in its intended environment
  - Can also be applied to intermediate work products, as predictors of how well the final product will satisfy user needs



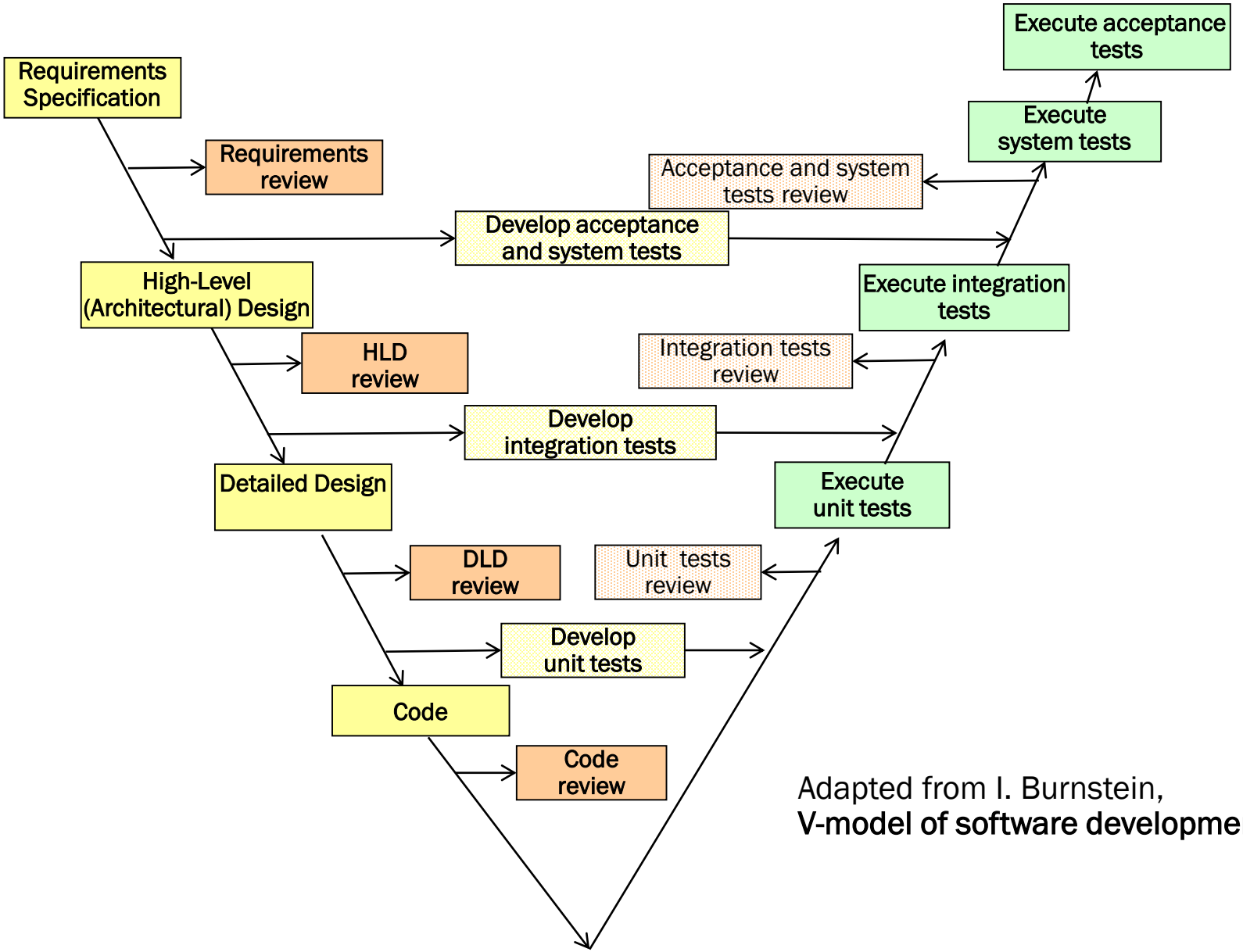
Static & dynamic V&V techniques

- **Static techniques** – involve analyzing the static system representations to find problems and evaluate quality
  - **Reviews** and inspections
  - Automated static analysis (e.g., with **lint**)
  - Formal verification (e.g., with Dafny)

- **Dynamic techniques** – involve executing the system (or an executable representation) and observing its behavior
  - Software **testing**
  - Simulation (e.g., animation of state machine model)

- They are complementary and not opposing techniques.

V&V activities along the life cycle (V-model)



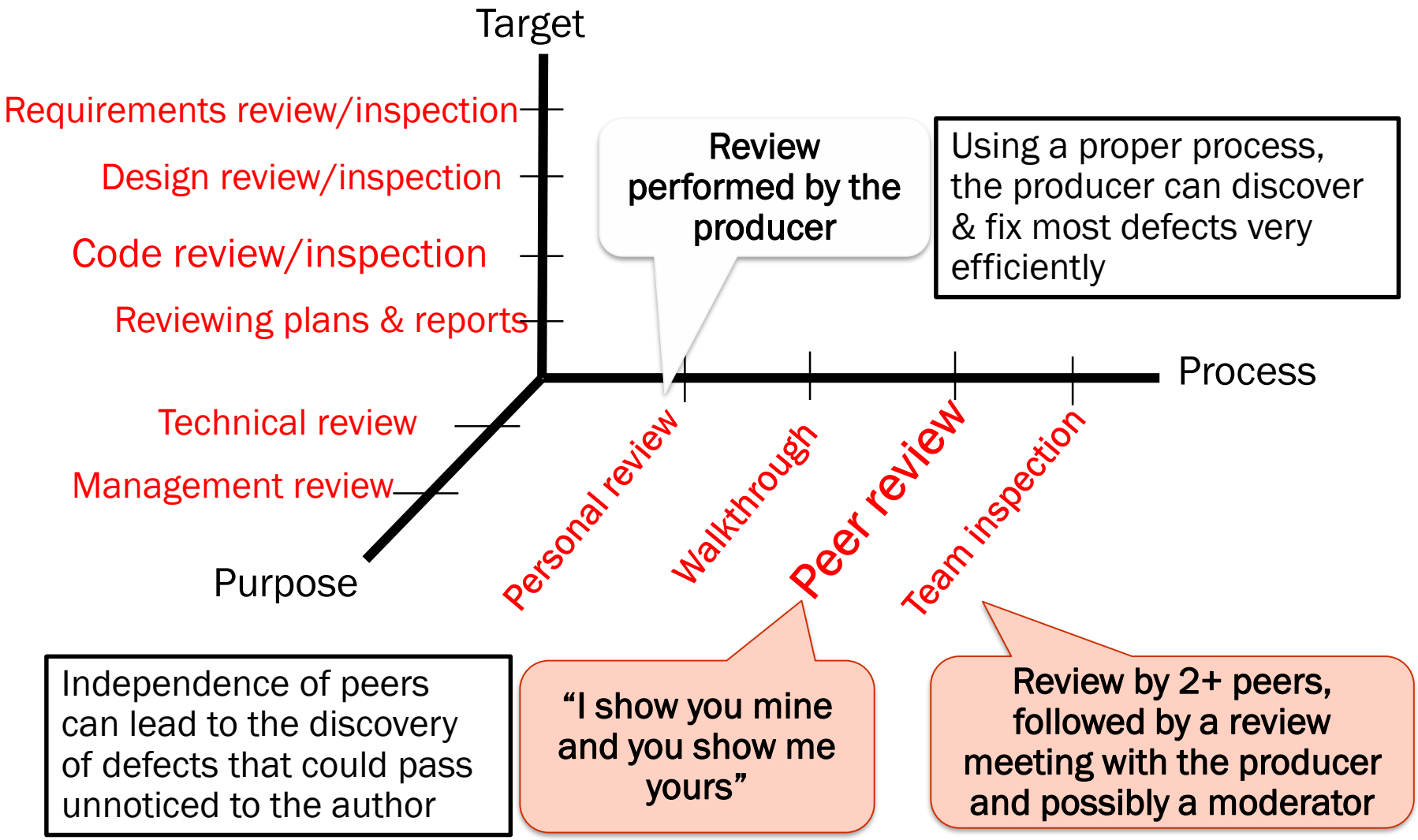
Adapted from I. Burnstein,
V-model of software development

Software reviews & inspections

Software reviews & inspections

- Analysis of static system representations to find problems
  - Manual analysis of requirements specs, design specs, code, etc.
  - May be supplemented by tool-based static analysis
- Advantages (as compared to testing):
  - Can be applied to any artefact, and not only code
  - Can be applied earlier (thus reducing impact and cost of errors)
  - Fault localization (debugging) is immediate
  - Allows evaluating internal quality attributes (e.g., maintainability)
  - Usually more efficient & effective than testing in finding security vulnerabilities and checking exception handling
  - Very effective in finding multiple defects
  - Peer reviews promote knowledge sharing

Types of reviews



Review best practices

- Use a checklist derived from historical defect data
  - Makes the review more effective & efficient, by focusing the attention on the most frequent & important problems.

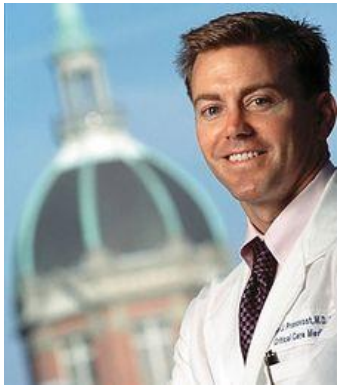
- Take enough review time (e.g., 200 LOC/hour is a recommended *review rate* by some authors)

- Take a break between developing and reviewing (in personal reviews)
- Combine personal reviews with peer reviews or team inspections

- Measure the review process & use data to improve
  - size, time spent, defects found, defects escaped

The importance of checklists

http://www.youtube.com/watch?v=xBPt4j1sOuI



Peter Pronovost
(Dr. Checklist)*

"Most influential people of 2008"
(Time magazin)

Dr. Pronvost's checklist estimated to save:
- $200.000.000
- 8.000 lifes per year in USA

Example of Flutter code review checklist

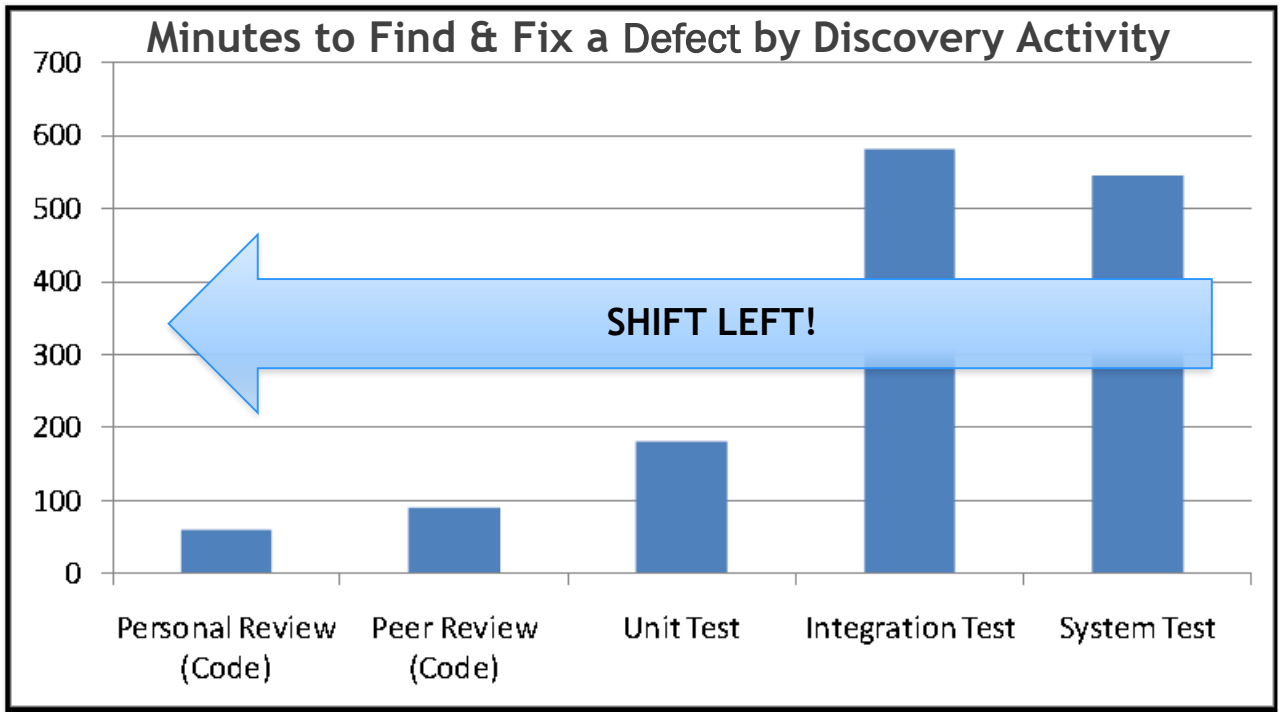
Category	What to verify
Naming conventions	• Classes, enums, typedefs, and extensions name should be in UpperCamelCase. • Libraries, packages, directories, and source files name should be in snake_case • Variables, constants, parameters, and named params should be in lowerCamelCase. • Private variables names preceded with underscores • …
Folder and code organization	• Segregate code into a proper folder structure namely providers, models, screens, utils. • Code is properly formatted with trailing commas used appropriately. • Adequate comments added for documentation. • …
Widget structure and usage	• Use ListView builder when working with infinite lists or very large lists • Use const in widgets which will not change when setState is called • The build method should be pure, without any side effects. • …
Linting rules	• Avoid relative imports for files in lib/. Use package imports. • Follow linting rules (See https://dart-lang.github.io/linter/lints/). • …
State management and separation of logic from UI	• Use provider as the recommended package for state management. • You can also choose to use any other approach for state management like Bloc. • Business logic should be separated from the UI • …
…	

Adapted from: https://walkingtree.tech/flutter-best-practices-and-code-review-checklist/

Adapt to your needs and apply in your project!

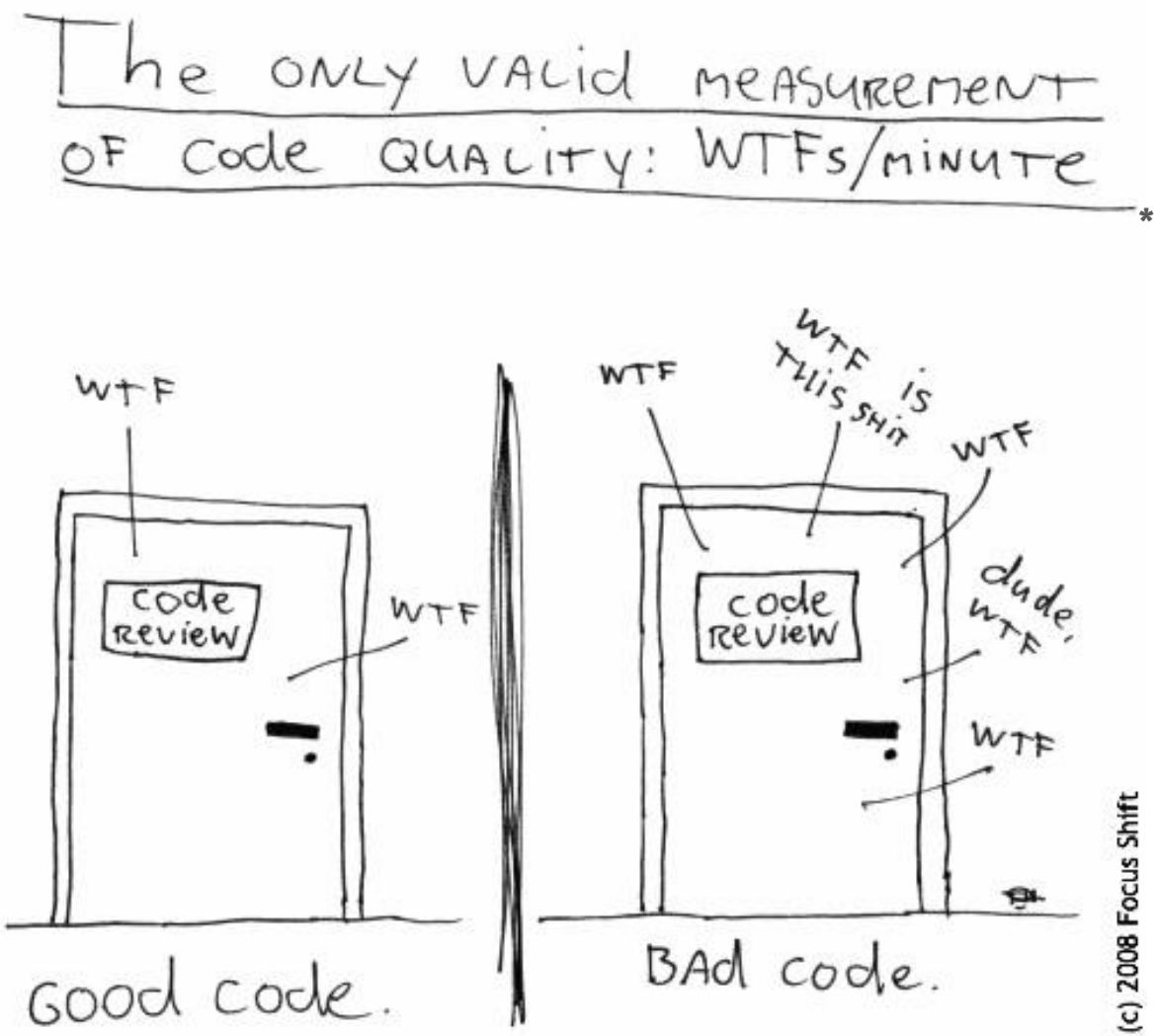
Efficiency of defect removal methods

- Even experienced programmers inject around 100 defects/KLOC (thousands of lines of code).
- Such large number of defects should be removed using the most efficient defect removal methods.



Source: Inspiring, enabling and driving the Evolution of Quality at Adobe leveraging the TSP, Jim Sartain, Senior Director, Quality, TSP Symposium 2009

About measurement ...



*WTF = What a Terrible Fault

Software testing

Testing goals and limitations

- Software testing consists in exercising the software with defined **test cases** and observing its behaviour to discover **defects**
  - **Defect testing** – using test cases with high defect finding capability
  - **Defect, fault** or **bug** – something that has to be changed in a program, and may lead to a **failure** if not fixed

- A secondarily goal is to increase the confidence on the software correctness and to evaluate product quality
  - **Statistical testing** – using representative test data and test cases to estimate a software quality metric (e.g., performance)

- **Testing can show the presence of bugs, not their absence**

  (Edgar Dijkstra)

Test cases

Test case	Inputs		Expected outputs
	a	b	gcd(a, b)
1	2	3	1
2	2	4	2

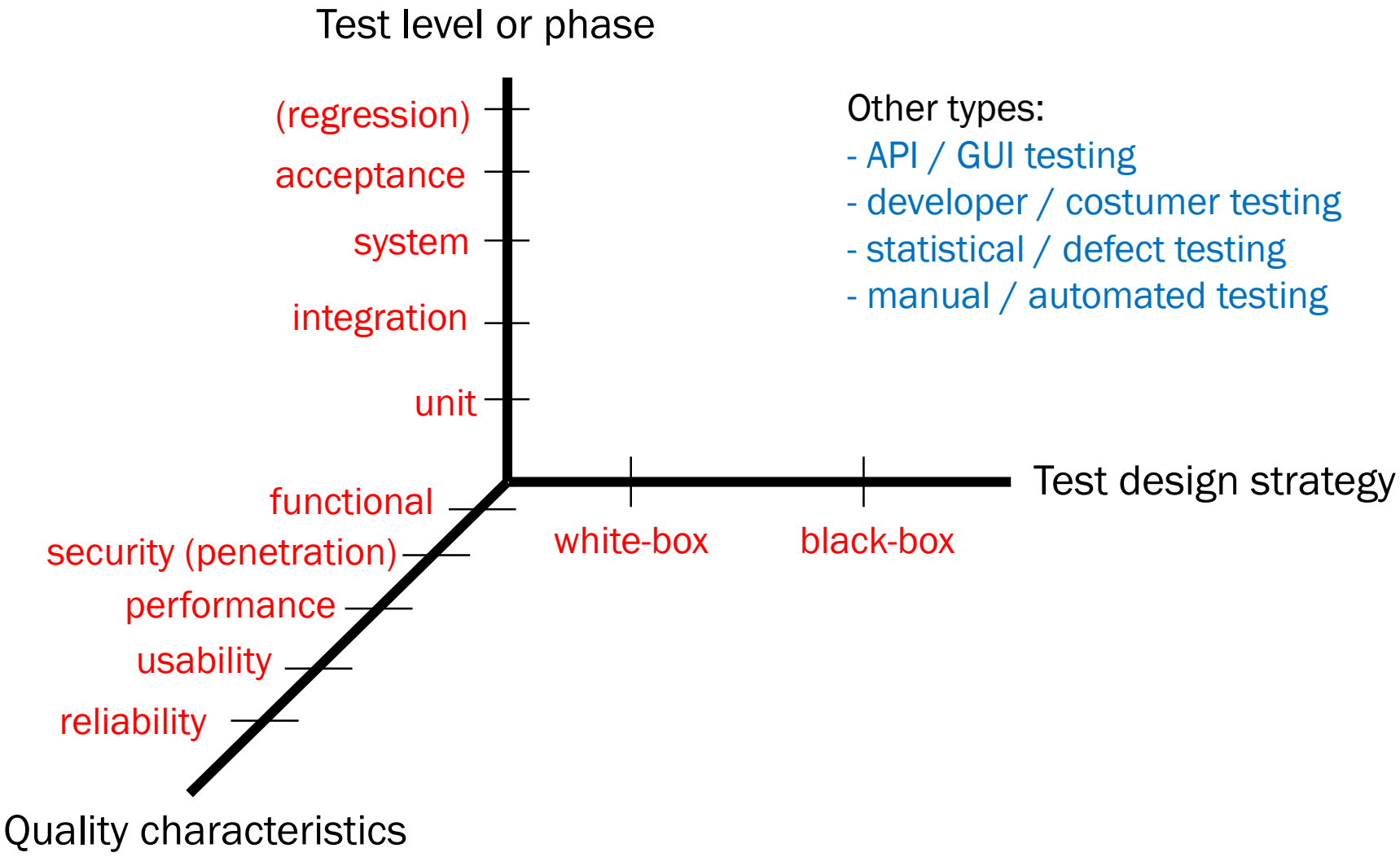
- **Test case**.
  (1) A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.
  (2) Documentation specifying inputs, predicted results, and a set of execution conditions for a test item
  - IEEE Standard Glossary of Soft. Eng. Terminology 610.12-1990
  - Inputs/outputs may include an initial/final state of the system
  - Can also have: id, description, item under test, test procedure, dependencies
- **Test script** – concrete definition of test steps / procedure
  - In natural language, scripting language or programming language
  - Test scripts can be parameterized for reuse with multiple **test data**

Test activities

A test process consists of the following main (groups of) activities:

- **Test planning** – define the objectives of testing and the approach for meeting test objectives within constraints imposed by the context
- **Test monitoring and control** - compare actual progress against the plan, and take actions necessary to meet the objectives of the test plan
- **Test analysis** – identify testable features and test conditions
- **Test design** – derive test cases
- **Test implementation** – create automated test scripts
- **Test execution** – run test suites
- **Test completion** - collect data from completed test activities

Test types



Other types:
- API / GUI testing
- developer / costumer testing
- statistical / defect testing
- manual / automated testing

Test levels: unit testing

- Testing of individual hardware or software units or groups of related units
  - IEEE Standard Glossary of Software Engineering Terminology 610.12-1990
- Also known as component testing (ISTQB) or module testing
- A principal goal is to detect functional (e.g., wrong calculations) and non-functional (e.g., memory leaks) defects in the unit
- Usually API testing
- Usually the responsibility of the developer
- Usually based on experience, specs & code

Test levels: integration testing

- Testing in which software and/or hardware components are combined and tested to evaluate the interaction between them
  - IEEE Standard Glossary of Software Engineering Terminology 610.12-1990
- ISTQB distinguishes two levels of integration testing:
  - Component integration testing –interactions between components
  - System integration testing - interactions between systems
- Sometimes the responsibility of an independent test team
- Usually based on a system spec (technical/design spec)
- Main goal is to detect defects that occur on the units' interfaces
- For easier fault localisation, integrate incrementally/continuously

Test levels: system testing

- Testing conducted on a complete, integrated system to evaluate the system's compliance with specified requirements
  - IEEE Standard Glossary of Software Engineering Terminology 610.12-1990
- Both functional behavior and quality requirements (performance, usability, reliability, security, etc.) are evaluated
- Usually GUI testing
- Usually the responsibility of an independent test team
- Usually based on a requirements document

Test levels: acceptance testing

- Formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable a customer, a user, or other authorized entity to determine whether or not to accept the system
  - IEEE Standard Glossary of Software Engineering Terminology 610.12-1990
- Usually the responsibility of the customer
- Usually based on a requirements document or contract
- A principal goal is to check if customer requirements and expectations are met

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
  - IEEE Standard Glossary of Software Engineering Terminology 610.12-1990

- Changes to software, to enhance it or fix bugs, are a very common source of defects

- Note: Not really a new test level, but just the repetition of testing at any level

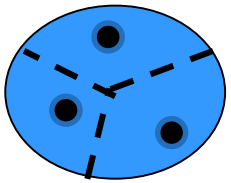
Test case design strategies

- Design strategies:
  - **Black-box** testing - Derivation of test cases based on some external specification
  - **White-box** testing - Derivation of test cases according to program structure

- Design goals:
  - Minimize the number and size of test cases
  - Maximize the fault detection capability

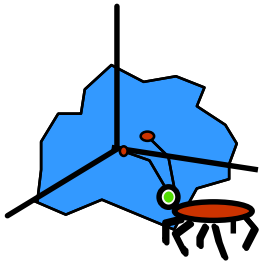
Black-box testing

- **Equivalence class partitioning (ECP):** partition the input domain into classes of equivalent behavior, separating classes of valid and invalid inputs, and select at least one test case from each class



Equivalence class partitioning

- **Boundary value analysis (BVA):** select test values at the boundaries of each partition (e.g., immediately below and above), besides typical values
  - Many bugs are "off-by-one"



"Bugs lurk in corners and congregate at boundaries." (B.Beizer)

- **State transition testing** : derive test cases from a state-machine model of the system (cover all states and transitions)
- **Decision table testing:** test all possible combinations of a set of conditions and actions
- **Use case testing:** derive test cases from a use case model of the system (model detailed with scenarios, pre/post-conditions, etc.)

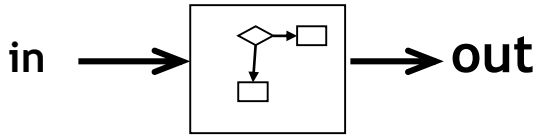
Example of ECP and BVA

- Create test cases for a function that determines the type of a triangle (equilateral, isosceles or scalene), given the sizes of its edges (of type unsigned int)

Equivalence class (test condition)	Inputs			Expected output
	I1	I2	L3	
Valid inputs, all sides equal	1	1	1	Equilateral
Valid inputs, one side different	1	2	2	Isosceles
Valid inputs, all sides different	2	3	4	Scalene
Invalid inputs, not a valid triangle	1	1	2	InvalidArgumentException
Invalid inputs, not a valid size	0	1	1	InvalidArgumentException

Boundary values were selected for the inputs
Permutations of the above inputs could also be considered

White-box testing



- Use a coverage analysis tool (e.g., Eclemma) to analyze code coverage achieved by black-box tests and design additional tests as needed

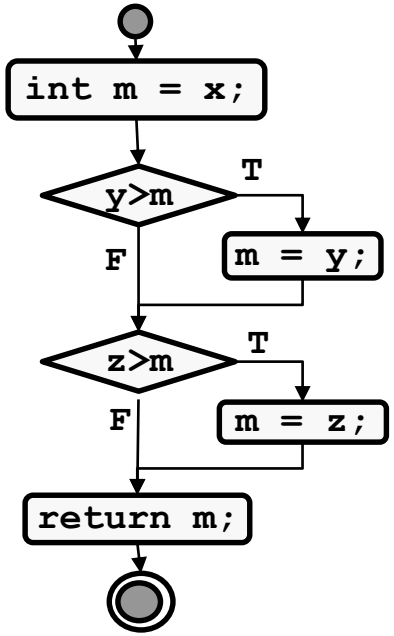
- Code coverage criteria with increasing strength:
  - Statement coverage
    - Ensure that all statements are exercised
  - Decision (or branch) coverage
    - Ensure additionally that all decisions (if, while, for, etc.) take both values true and false
  - ...

Warning: 100% code coverage may not be enough (check this out: http://blog.caplin.com/2012/05/08/why-100-code-coverage-is-not-enough/)

Example of white-box testing

```
int max(int x, int y, int z)
{
    int m = x;
    if (y > m)
        m = y;
    if (z > m)
        m = z;
    return m;
}
```

Program flow graph



- Create minimal sets of test cases that cover:
  - a) all statements (nodes)

#	x	y	z	max	
1	1	2	3	3	TT

  - b) all decisions (branches)

#	x	y	z	max	
1	1	2	3	3	TT
2	1	1	1	1	FF

Testing tools (1/2)

- Unit testing frameworks
  - JUnit, NUnit, etc.
- Mock object frameworks
  - Facilitate simulating external components in unit testing
  - Mockito, EasyMock, jMock, etc.
- Test coverage analysis tools
  - Measure degree of code coverage achieved by the execution of a test suite
  - Useful for white-box testing
  - EclEmma, Clover, etc.
- Mutation testing tools
  - Evaluate the quality of a test suite by determining its ability to 'kill' mutants (with common fault types) of the program under test
  - pitest, muJava, etc.
- Acceptance testing frameworks
  - Allows creating test cases by people without technical knowledge
  - Cucumber (Gherkin), JBehave, Fitnesse, etc.

Testing tools (2/2)

- Capture/replay tools (aka functional testing tools)
  - Capture user interactions in scripts that can be edited and replayed
  - Useful for GUI testing, particularly regression testing
  - Selenium, IBM Rational Functional Tester, etc.
- Performance/load testing tools
  - Execute test suites simulating many users and measure system performance
  - IBM Rational Performance Tester, Compuware QA Load, etc.
- Penetration testing tools
  - Metasploit, ZAP, etc.
- Test case generation tools
  - Automatically generate test cases from code or models
  - EvoSuite (Java), ParTeG (UML), etc.
- Test management tools
  - Manage test information, status, traceability, etc.
  - Xray, TestLink, etc.

Testing best practices

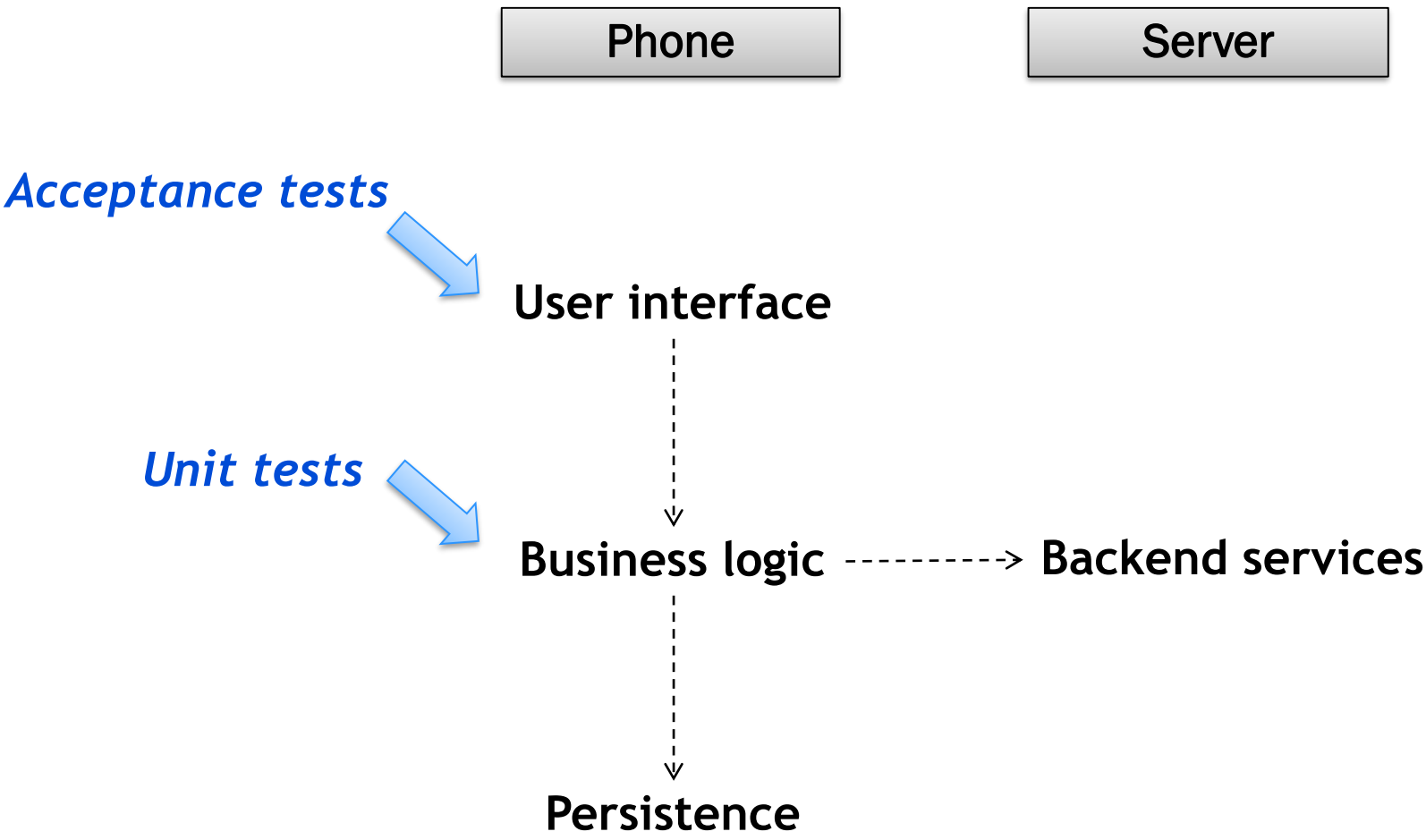
- Test as early as possible
  - Because the cost of finding & fixing bugs increases exponentially with time elapsed
- Automate the tests
  - E.g., using frameworks such as JUnit
  - Because of the need to frequently repeat test execution
  - But it's easier to automate API testing than GUI testing
- Test first (write tests before the code) $\Rightarrow$ (A/U)TDD
  - Helps clarifying requirements and specifications
  - Test cases are partial specifications (examples) of system behavior
- Black-box first
  - Start by designing test cases based on the specification (black-box) (as a "sufficient" set of examples)
  - Then add any tests needed to ensure code coverage (white-box)

Software Verification and Validation

L.EIC-ES-2021-22

Demo: Testing Flutter apps

Unit & acceptance tests



Unit (developer) tests

- Automated in Android Studio (IntelliJ) or VSCode
- Using the “test” package (dart) or “flutter_test” package (Flutter specific)

- You should use in your projects to test the business logic
  - Try to separate business logic code from UI code
  - Try to maximize code coverage

- More information:
  - https://docs.flutter.dev/cookbook/testing/unit/introduction

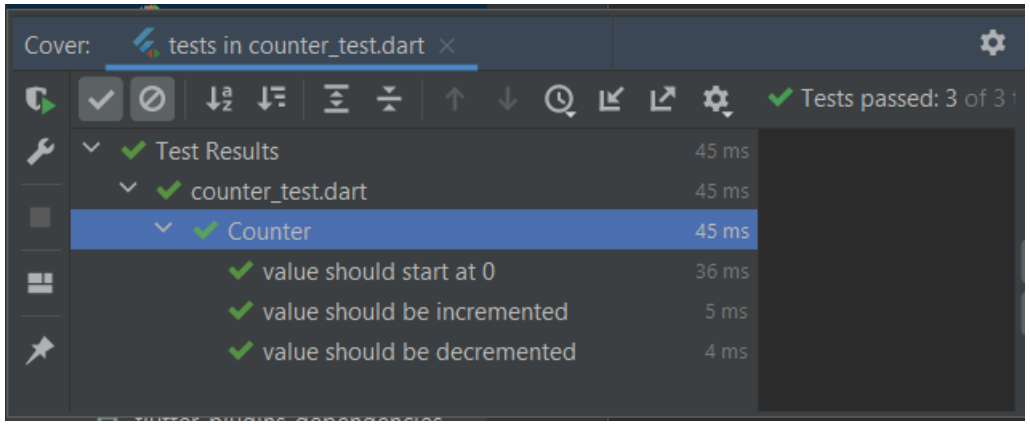
Example

Class under test (lib/counter.dart)

```dart
class Counter {
  int value = 0;
  void increment() => value++;
  void decrement() => value--;
  @override String toString() => value.toString();
}
```

Run 'tests in counter_tes...' with Coverage

```dart
class Counter {
  int value = 0;
  void increment() => value++;
  void decrement() => value--;
  @override String toString() => value.toString();
}
```



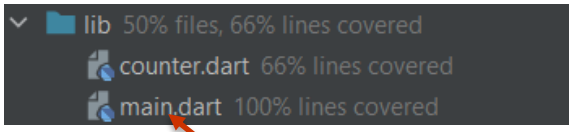
Test code (test/counter_test.dart)

```dart
// Import the test package and Counter class
import 'package:test/test.dart';
import 'package:flutter_demo/counter.dart';

void main() {
  group('Counter', () {
    test('value should start at 0', () {
      expect(Counter().value, 0);
    });

    test('value should be incremented', () {
      final counter = Counter();
      counter.increment();
      expect(counter.value, 1);
    });

    test('value should be decremented', () {
      final counter = Counter();
      counter.decrement();
      expect(counter.value, -1);
    });
  });
}
```

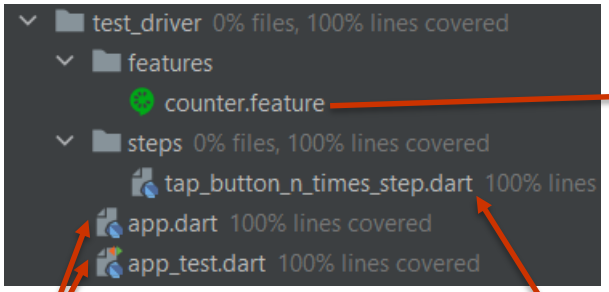
Acceptance (customer) tests

- Acceptance test scenarios written in Gherkin
  - Readable by the customer
- Can be automated using the “flutter_gherkin” package
  - Provides a number of built-in steps for interacting with UI objects
  - Can be easily extended with custom steps
- You should try to automate at least some of your acceptance tests
  - Minimum of two test scenarios automated (in each iteration)
  - All others must be executed manually
- More information:
  - https://pub.dev/packages/flutter_gherkin

Example



Modified demo app, with increase/decrease buttons, and widgets with a unique key assigned for referencing purposes



Boilerplate code

Custom step

```
Feature: Counter
  The counter should be incremented/decremented when increment/decrement
  buttons are pressed, without going below zero.

  Scenario: Counter increases when the button is pressed (1)
    Given I expect the "counter" to be "0"
    When I tap the "increment" button
    And I tap the "increment" button
    Then I expect the "counter" to be "2"

  Scenario: Counter increases when the button is pressed (2)
    Given I expect the "counter" to be "0"
    When I tap the "increment" button 2 times
    Then I expect the "counter" to be "2"

  Scenario Outline: Counter increases when the button is pressed (3)
    Given I expect the "counter" to be "0"
    When I tap the "increment" button <number> times
    Then I expect the "counter" to be "<number>"

    Examples:
      | number |
      | 3      |
      | 10     |
```

```
  Scenario: Counter may be incremented and decremented (2)
    Given I expect the "counter" to be "0"
    When I tap the "increment" button 3 times
    Then I expect the "counter" to be "3"
    When I tap the "decrement" button 2 times
    Then I expect the "counter" to be "1"
    When I tap the "decrement" button 2 times
    Then I expect the "counter" to be "0"
```


```
10 scenarios (10 passed)
38 steps (38 passed)
0:01:31.097000
```

Example of custom step

when2: step with 2 params that can be used in the “When” part

Phrase structure to be used in the test scenario with two params of types string and int

Obtain the widget locator using the given key

Perform the intended action (tap) using an utility function

Provide custom step handlers here

tap_button_n_times_step.dart

```dart
import 'package:flutter_driver/flutter_driver.dart';
import 'package:flutter_gherkin/flutter_gherkin.dart';
import 'package:gherkin/gherkin.dart';

StepDefinitionGeneric TapButtonNTimesStep() {
  return when2<String, int, FlutterWorld>(
    'I tap the {string} button {int} times',
        (key, count, context) async {
      final locator = find.byValueKey(key);
      for (var i = 0; i < count; i += 1) {
        await FlutterDriverUtils.tap(context.world.driver, locator);
      }
    },
  );
}
```

app_test.dart

```dart
Future<void> main() {
  final config = FlutterTestConfiguration()
    ..features = [Glob(r"test_driver/features/**.feature")]
    ..reporters = [
      ProgressReporter(),
      TestRunSummaryReporter(),
      JsonReporter(path: './report.json')
    ]
    ..stepDefinitions = [TapButtonNTimesStep()]
    ..customStepParameterDefinitions = []
    ..restartAppBetweenScenarios = true
    ..targetAppPath = "test_driver/app.dart";
  return GherkinRunner().execute(config);
}
```

Key points

- Verification shows conformance with specification. Validation shows that the program meets the customer's needs.
- Static verification techniques involve examination and analysis of the program for error detection. Dynamic techniques involve executing the program for error detection.
- Program reviews and inspections are very effective in discovering errors.
- Testing can show the presence of faults; but not prove there are no remaining faults.
- Component developers are responsible for unit testing.
- System testing is the responsibility of a separate team.
- Integration testing is testing increments of the system. The main goal is to discover defects in the interfaces of units.
- Test automation reduces testing costs by supporting the test process with a range of tools.

References and further reading

- "Software Engineering", Ian Sommerville, 10th Edition (Chapter 8 – Software Testing)
- "Practical Software Testing", Ilene Burnstein, Springer-Verlag, 2003
- IEEE Std 610.12: 1990 - Standard Glossary of Software Engineering Terminology
- “Peer Reviews in Software: A Practical Guide”, Karl E. Wiegers, Addison-Wesley, 2002
- Certified Tester Foundation Level Syllabus Version 2018 Version, International Software Testing Qualification Board, https://www.istqb.org/downloads/send/51-ctfl2018/208-ctfl-2018-syllabus.html
- https://docs.flutter.dev/testing
- https://pub.dev/packages/flutter_gherkin