Basic Testing Techniques 1/2

An introduction to testing, unit testing, and test-driven development with examples in Java (and C#)

Roberto Casadei, PhD
roby.casadei@unibo.it

Alma Mater Studiorum - Università di Bologna
Dipartimento di Informatica - Scienza e Ingegneria

https://github.com/metaphori/testing-basic-techniques

Presentation

- **Who I am:** Roberto Casadei
- **What I do:** adjunct professor & post-doc at DISI, UNIBO
  - research on “collective adaptive systems”
  - teaching (BEng): *Object-Oriented Programming (Module 2)*, *Foundations of Informatics A (Module 2)*
- **Me & software testing**
  - Interests in software engineering, agile sw dev, etc.
  - CS4004 Software Testing and Inspection (Spring 2012, Prof. Norah Power, University of Limerick)
  - Application of testing to research tools like ScaFi
- https://robertocasadei.github.io

Course: Goals and Contents

Goal: introduce the practice of testing (mostly by a developer perspective)

- Introduce the basics of ***automated software testing*** (a testing culture)
- Provide elements for testing ***in practice*** and guiding design through tests
- Provide pointers for ***effective*** testing

Contents: Part 1/2 (tentative)

- Key concepts about testing; types and perspectives on testing; testing automation
- Unit testing in xUnit
- Elements of test case design; coverage
- Test-driven development

Contents: Part 2/2 (tentative)

- Test doubles (stubs, mocks etc.)
- Effective testing
- Acceptance test-driven (ATDD) / Behaviour-driven development (BDD)
- Pointers to advanced testing-related techniques

Outline

Outline

Let's start with a simple piece of software

Goal: finding the smallest and largest element in a list of integers

An attempt

```
public class NumFinder {
    private int smallest = Integer.MAX_VALUE;
    private int largest = Integer.MIN_VALUE;

    public void find(int[] numbers){
        for(int n : numbers){
            if(n < smallest) smallest = n;
            else if(n > largest) largest = n;
        }
    }

    public int getSmallest(){ return smallest; }

    public int getLargest(){ return largest; }
}
```

Question: is this correct (bug-free)?

Let's check it by running the program

```java
    public static void main(String[] args){
        // Get inputs by reading integers from stdin
        Scanner in = new Scanner(System.in);
        System.out.print("How many numbers? ");
        int n = in.nextInt();
        List<Integer> list = new ArrayList<>(n);
        for(int i=0; i < n; i++) {
            System.out.print(i + "th number: ");
            list.add(in.nextInt());
        }
        in.close();

        // Build the SUT (System Under Test)
        NumFinder nf = new NumFinder();

        // Launch the system
        nf.find(list.stream().mapToInt(Integer::intValue).toArray());

        // Print out the results
        System.out.println("Smallest: " + nf.getSmallest());
        System.out.println("Largest: " + nf.getLargest());
    }
```

- Each run of the program would be a manual test
- We could test it against different test cases
- **[live] run it**

Let's try to **automate** the tests

- Let's use a script file with inputs for the program
  - still assuming that what we run ***is*** the SUT

```
public static void main(String[] args) throws FileNotFoundException {
    Scanner in = new Scanner(args.length == 1
        ? new DataInputStream(NumFinder.class.getClassLoader()
            .getResourceAsStream(args[0]))
        : System.in);
```

- **[live] run it**
- still working on the SUT (production code)
- still manually checking results
- still needing multiple program runs to check multiple test cases
  - we may script it, though

Let's try to **automate** the tests, better

```
public class AutomaticTest2 {
    public static void main(String[] args){
        // Test case 1: some numbers
        NumFinder numFinder = new NumFinder();
        int[] input1 = new int[]{ 4, 25, 7, 9 };
        numFinder.find(input1);
        System.out.println("Apply to { 4, 25, 7, 9 } => " +
                " - smallest: " + numFinder.getSmallest() +
                " - largest: " + numFinder.getLargest());
        // Test case 2: monotonically increasing sequence
        int[] input2 = new int[]{ 10, 20, 30 };
        numFinder.find(input2);
        System.out.println("Apply to { 10, 20, 30 } => " +
                " - smallest: " + numFinder.getSmallest() +
                " - largest: " + numFinder.getLargest());
        // Test case 3: monotonically decreasing sequence
        numFinder = new NumFinder();
        int[] input3 = new int[]{ 4, 3, 2, 1 };
        numFinder.find(input3);
        System.out.println("Apply to { 4, 3, 2, 1 } => " +
                " - smallest: " + numFinder.getSmallest() +
                " - largest: " + numFinder.getLargest());
}    }
```

- using a ***different piece of SW (program)*** to test our SW (program/SUT)
  - this is the key idea of test automation
- isolate the unit of behaviour to be tested
- found 2 "relatively subtle" bugs (1 in the SUT [TC#3], 1 in the test [TC#2])
- still manually checking results

Let's try to **automate** the tests, **better**

```
public class AutomaticTest3 {
    public static void main(String[] args){
        NumFinder numFinder = new NumFinder();
        int[] input1 = new int[]{ 4, 25, 7, 9 };
        numFinder.find(input1);
        if(!(numFinder.getSmallest() == 4 && numFinder.getLargest() == 25)){
            System.out.println("Test Case #1 failed");
        }

        numFinder = new NumFinder();
        int[] input2 = new int[]{ 10, 20, 30 };
        numFinder.find(input2);
        if(!(numFinder.getSmallest() == 10 && numFinder.getLargest() == 30)){
            System.out.println("Test Case #2 failed");
        }

        numFinder = new NumFinder();
        int[] input3 = new int[]{ 4, 3, 2, 1 };
        numFinder.find(input3);
        if(!(numFinder.getSmallest() == 1 && numFinder.getLargest() == 4)){
            System.out.println("Test Case #3 failed");
}   }   }
```

- automatically checking expected vs. actual results
- ***Are we done?*** Not quite...
  - **test** issues (not comprehensive...); **what testing process?**
  - **quality** issues (repetition...)
  - **structural** issues (what if multiple units are tested? multiple `main`s?)
  - **reliability** issues (what if the SUT raises an exception?)
  - **maintainability** issues (what if we wanna change the reporting?)
  - **harnessing complexity** (how to test complex SUTs? how to express complex expectations? ...)

Good automated tests

- As we work on automated tests, we may
  - study **testing approaches** to define better test cases
  - define **processes** for good team- or enterprise-wide testing
  - define **libraries** with testing utilities for reuse
  - define **structures** for better productivity (writing new tests) and maintainability
  - devise **patterns/idioms** for addressing recurrent testing-related problems
  - discover that tests are also an excellent **design tool**

- Fundamental testing techniques [1]
- Test automation frameworks (test harnesses) [2]
- Testing patterns [3]
- XP [4], Test-Driven Development [5], Agile testing [6]

[1] A. P. Mathur, *Foundations of Software Testing*, 1st. Addison-Wesley Professional, 2008
[2] C. Tudose, *JUnit in Action, Third Edition*. Manning Publications, 2020
[3] G. Meszaros, *XUnit Test Patterns: Refactoring Test Code*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006
[4] K. Beck, *Extreme programming explained: embrace change*. addison-wesley professional, 2000
[5] K. Beck, *Test Driven Development. By Example (Addison-Wesley Signature)*. Addison-Wesley Longman, Amsterdam, 2002
[6] L. Crispin and J. Gregory, *Agile testing: A practical guide for testers and agile teams*. Pearson Education, 2009

Revised program + effective automated tests (1/3)

After a few iterations of a “game” played between implementations and tests...

```
public class MinMaxFinder {
    public static Optional<ImmutablePair<Integer,Integer>> find(int[] numbers){
        if(numbers == null) return Optional.empty();

        Optional<Integer> smallest = Optional.empty();
        Optional<Integer> largest = Optional.empty();

        for(int n : numbers){
            smallest = smallest.map(x -> x < n ? x : n).or(() -> Optional.of(n));
            largest = largest.map(x -> x > n ? x : n).or(() -> Optional.of(n));
        }

        final Optional<Integer> min = smallest;
        final Optional<Integer> max = largest;
        return min.flatMap(minv -> max.map(maxv -> ImmutablePair.of(minv,maxv)));
    }
}
```

- from state-based to functional

Revised program + effective automated tests (2/3)

```java
public class MinMaxFinderTest {

    @Test
    public void test_empty_array(){
        assertEquals(Optional.empty(), MinMaxFinder.find(
                new int[]{ }
        ));
    }

    @Test
    public void test_increasing_values(){
        assertEquals(nonEmptyPair(1, 4), MinMaxFinder.find(
                new int[]{ 1, 2, 3, 4 }
        ));
    }

    // ...

    private static Optional<ImmutablePair<Integer,Integer>> nonEmptyPair(Integer left,
     Integer right){
        return Optional.of(ImmutablePair.of(left, right));
    }
}
```

- Tests are simple and read just fine... but can we remove some overhead/repetition?
    - after all, we are mainly interested in (1) inputs and (2) expected outputs

Revised program + effective automated tests (3/3)

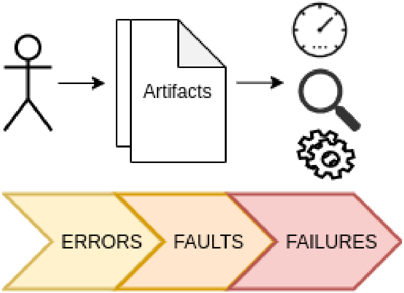
```java
public class MinMaxFinderParameterizedTest {

    @ParameterizedTest(name = "{index} {2}")
    @MethodSource("minMaxFinderTestCases")
    public void test_empty_array(int[] inputs,
                                 Optional<ImmutablePair<Integer,Integer>> expected,
                                 String name){
        assertEquals(expected, MinMaxFinder.find(inputs));

    }

    private static Stream<Arguments> minMaxFinderTestCases(){
        return Stream.of(
                Arguments.of(null, Optional.empty(),
                             "null array"),
                Arguments.of(new int[]{ }, Optional.empty(),
                             "empty array"),
                Arguments.of(new int[]{ -5 }, nonEmptyPair(-5, -5),
                             "singleton array"),
                Arguments.of(new int[]{ 1, 2, 3, 4 }, nonEmptyPair(1, 4),
                             "monotonically increasing sequence"),
                Arguments.of(new int[]{ 4, 3, 2, 1 }, nonEmptyPair(1, 4),
                             "monotonically decreasing sequence"),
                Arguments.of(new int[]{ Integer.MAX_VALUE, Integer.MIN_VALUE },
                             nonEmptyPair(Integer.MIN_VALUE, Integer.MAX_VALUE),
                             "boundary values")
                // ...
        );
    }
```

- **[live] run it** (by IntelliJ, by Gradle, by GH-Actions)

Outline

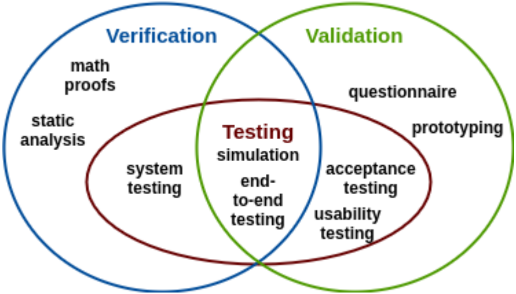
Error vs. failure vs. fault

- **Failure (problem, anomaly)**: difference from expected result (e.g., wrt to specs)
- **Fault (defect, bug)**: cause of a failure; manifestation of an error
  - A failure may be caused by many faults
  - A fault can cause many failures.
- **Error (mistake)**: mistake which caused the faults to occur



Testing and V&V

- **Testing**: activity ***looking for failures, revealing faults***
  - Much more, actually
- **Verification**: checks system conformance with ***specifications***
  - ***"Build the thing right"***
  - "A design without specifications cannot be right or wrong, it can only be surprising!" [7]
  - However, specifications may be wrong or incomplete
- **Validation**: does the system meet ***stakeholder expectations***?
  - ***"Build the right thing"*** (i.e., what is valuable)



Definitions of “software testing”

1) *“An **investigation** conducted to provide **stakeholders** with information about the **quality** of the software product or service under test”* (Cem Kaner/Wikipedia)
2) *“Activity in which a system or component is **executed** under **specified conditions**, the **results** are **observed** or recorded, and an **evaluation** is made of some aspect of the system or component”* (ISO/IEC/IEEE 24765:2010 Systems and software engineering – Vocabulary)
3) *“The **process** consisting of all lifecycle activities, **both static and dynamic**, concerned with **planning, preparation and evaluation of software products** and related work products to determine that they satisfy specified **requirements**, to demonstrate that they are **fit for purpose** and to **detect defects**.”* (ISTQB–International Software Testing Qualifications Board)
4) *“The **overall process** of planning, preparing, and carrying out a **suite** of **different types of tests** designed to **validate** a system under development, in order to achieve an **acceptable level of quality** and to **avoid unacceptable risks**”*

$\rightarrow$ activity/process; variety; v&v; quality; risk; various stakeholders

General testing principles (cf. [8], [1])

1. **Exhaustive testing typically unfeasible** in any but small systems
   - ➜ consider ***risks*** and ***resources***
2. Testing reveals the ***presence*** of faults, not their absence
   - i.e., you cannot prove a SW product is defect-free
   - cf. coverage
3. **Absence of errors fallacy**: just because a SW product is defect-free, it does not mean it is ready to be shipped
   - cf. verification vs. validation
4. **Defect clustering**: bugs tend to come in groups
5. **Pesticide paradox (saturation effect)**: tests tend to loose effectiveness
   - code tends to be “resistant” to existing tests
   - **solution:** add new test cases
6. **Reliability vs. confidence**
   - testing may increase your confidence on the correctness of your SW, but your confidence may not match actual reliability (statistical measure of probability of success in a given environment).

[8] R. Patton, *Software Testing*. Sams, 2006
[1] A. P. Mathur, *Foundations of Software Testing*, 1st. Addison-Wesley Professional, 2008

Testing: why

Testing is the widest industrial approach to V&V

Why *doing* testing

- Testing as a key means for
  1) quality
  2) project risk mitigation
  3) (long-term) productivity
  4) specification/documentation

Why *exploring/studying* testing

- Powerful tool in any software engineer's toolbox
- Career paths
- Intensively researched topic

Testing for **quality** (see e.g. [9]$_{2.1.3}$)

Testing as a process that ***fosters quality in software***

- What is software quality?
  - The degree to which a system meets requirements/expectations (cf. V&V)
  - Defined in terms of ***quality attributes***
  - [superseded] **ISO 9126-1** (product quality): functionality, reliability, usability, efficiency, maintainability, portability
  - **ISO/IEC 25010** "System and software quality models"
    - ***quality in use***: effectiveness, satisfaction, fredom from risk, context coverage
    - ***product quality***: functional sustainability, efficiency, compatibility, usability, reliability, security, maintainability, portability
- Two kinds of processes for quality
  - ***measuring*** quality (ex post)
    - **ISO/IEC 25023** "Measurement of system and software product quality"
  - ***building*** quality (ex ante)
- Testing
  - ***estimates*** quality
  - ***locates*** quality issues
  - ***promotes*** processes for quality (cf. TDD, refactoring)
  - ***enforces*** quality (cf. testability)

⚠ Automated tests are software themselves → quality also applies to tests!

- are tests efficient? reliable? maintainable? ...

Testing for **risk mitigation** (see e.g. [9]$_{2.1.4}$)

Testing as a process for ***mitigating risk***

- **Risk**: likelihood $\times$ impact
  - Defects can cause failures that may cause high costs
  - ? ***What is your experience with defects? any anecdotes?***
  - Risks should be ***estimated*** ($\rightarrow$ estimation of likelihood, estimation of impact)
- Testing, by tracking failures/defects, ***lowers the likelihood*** of bad events, hence the risk
  - "Testing should continue as long as costs of finding and correcting a defect are lower than the costs of failure" [10]
- Moreover
  - regression tests lower the risk of changing software (e.g., to improve quality)
  - TDD lowers the risk of bad design
- **Principle**: define ***test intensity*** and ***test extent*** depending on risk [9]

Testing for specification/documentation

Tests as a ***formalisation of requirements*** and ***as (complementary) documentation***

- Tests ***specify (formally)*** how your software is expected to behave
  - ***Source-of-truth***
  - Promotes ***learning*** about the SUT and its domain (good for users, new developers etc.)
    - good tests are ***readable***
- A kind of unambiguous, executable, ***living documentation***
  - “living” in the sense that it is “synchronised” with the SUT (i.e., always up-to-date)
- Promotes ***collaboration*** between developers and among different stakeholders
  - cf. ATDD/BDD

⚠ Tests can rarely be the only form of documentation

- tests may be incomplete, obsolete, hard-to-read, at multiple levels of abstractions...

Testing for (long-term) **productivity**

Testing as a process that can increase (long-term) ***productivity***

- Quality + early detection of faults lower ***debugging/maintenance effort***
  - cf. sustainable development
- TDD helps getting ***design right*** right away
  - tests as a tool for gaining ***knowledge***
- Regression tests enables us to refactor / move quickly and with confidence (***agility***)
  - cf. technical debt
- Tests may promote ***collaboration*** (cf. previous slide)

⚠ Uneffective testing may not improve productivity

Outline

Testing: what it is all about[1]

- **Why** you test
  - I.e., what is the goal?
- **What** to test
  - **Subject/System** Under Test (SUT); functional & non-functional requirements
- **Who** prepares tests, run tests, evaluates results, etc.
  - and relationships with other stakeholders (e.g., developers, customers)
- **When** tests are prepared and executed
  - w.r.t. when the SUT is designed, implemented, validated, and released?
- **Where** tests are defined and executed
  - Cf. Maven-style project tree structures
  - Cf. tests execution in developer machine vs. CI server
- **How** testing is done
  - I.e., what techniques, tools, processes etc.?

[1] Cf., Six Honest Serving Men (Rudyard Kipling)

Perspectives on testing

"Schools" of testing

- **Analytic School**: testing as a rigorous task, via formal methods
  - Motto: Programs are logical artifacts, subject to math laws
- **Factory School**: testing to measure project progress
  - Motto: Testing must be planned/scheduled/managed
- **Quality School**: testing as a quality process
  - Motto: A quality process for a quality product
- **Context-Driven School**: testing as a stakeholder-oriented, adaptable process
  - Key question: which testing would be more valuable right now?
- **Agile School**: testing to facilitate change
  - Techniques: test automation, test-driven approaches

Takeaway: perspectives on testing impact the *why/what/how..* of testing

Many different kinds of testing (what, how, when..)

Multiple classifications

- According to
  - **Goal**: security testing vs. performance testing vs. load testing ... ; regression testing; ...
  - **Who**: programmer tests vs. quality assurance vs. customer tests
- **Technology-facing** vs. **Business-facing testing**
  - Verification vs. validation; component vs. usability testing
- **Granularity/scope of testing**
  - Unit – integration – system-level/end-to-end/acceptance
- **Static** vs. **dynamic**
  - Code-level vs. runtime
- **Manual (human-based)** vs. **automated (computer-based)**
  - Examples of human testing: inspection (code reviews), walkthrough
  - Examples of computer-based testing: static analysis, “standard” testing
- **Structural (aka white-box)** vs. **functional (aka black-box)**
  - Examples of structural testing: control flow testing, path testing
  - Examples of functional testing: equivalence partitioning, BVA
- **Frequency, whether planned or not, temporal scope..**
  - Waterfall vs. shift-left; build/manually/change-triggered (continuous)

V&V techniques by static/dynamic $\times$ manual/automated

	Human	Computer
Static	Inspection, technical reviews	Static analysis
Dynamic	Walkthrough, usability testing	Testing

- Note: terminology is sometimes inconsistent or ambiguous.
  - What precisely is "testing" and what "V&V" and what "analysis" depend on definitions, hence to where lines are drawn.
  - For sure: testing includes both V&V techniques
  - For sure: V&V include testing techniques
  - Often, testing is considered only as a dynamic activity

Levels of testing

Levels of testing based on granularity/scope

1) **Unit testing**: testing at the level of individual units (of functionality/behaviour)
2) **Integration testing**: testing of the functionality provided by multiple integrated/interacting units
3) **System testing**: testing the whole system for correctness
   - In a black-box way
   - Non-functional requirements (e.g., efficiency, reliability, security) are typically tested at this level

Acceptance tests

Acceptance testing: testing a system against the needs and expectations of users and stakeholders

Acceptance testing

- Is the SW system **"acceptable"** (for delivery)?
- It's about testing whether the SW system satisfies the **acceptance criteria** (e.g., as specified in the requirements)
- Acceptability is matter of customers, product owners, or users
- It is very much about **what** has to be built
  - $\Rightarrow$ **Acceptance test-driven development (ATDD)**
- Tools: FitNesse (wiki-based), JBehave, Cucumber, SpecFlow (.NET)

Acceptance testing vs. system testing

- System tests $\Rightarrow$ build the **thing right** (things should behave correctly)
- Acceptance tests $\Rightarrow$ build the **right thing** (validate what your stakeholders really want)

A note on **standards** for software testing

- **IEEE Std 1044-1** ***Classification for Software Anomalies***
- IEEE Std 829-2008 *Software Test Documentation* — **superseded by 29119-3**
- **ISO/IEC/IEEE 29119**: ***Software and systems engineering – Software testing***
  - **29119-1:2013** ***Concepts and Definitions***
  - **29119-2:2013** ***Test processes***
  - **29119-3:2013** ***Test documentation***
  - **29119-4:2015** ***Test techniques***
    - Specification-based test design techniques
    - Structure-based test design techniques
    - Experience-based test design techniques
  - **29119-5:2016** ***Keyword-driven testing***

⚠ **29119 is controverse!** due to (1) lack of true consensus; (2) heavy focus on docs; (3) theory vs. practice inconsistencies; (4) exclusion of context-driven testing; (5) too prescriptive; (6) political and monetary connotations

Outline

Outline

Build automation

Motivation

- Test automation requires **tools**
  - For ***expressing*** tests
  - For ***running*** tests
- Results of tests may affect other project-related activities
  - merge of pull requests; delivery; e-mail triggers; ...

Build automation

- The process of **automating** the creation of software builds and ***associated processes***
- It eases the **configuration of a build workflow** comprising activities like:
  - **dependency management**
  - compilation of source code
  - execution of automated tests
  - application packaging
  - delivery
- Build automation is supported by **build (automation) tools**
  - Ant, Maven, Gradle, MSBuild, ...

Gradle (1/2)

Create a project

```
$ gradle init
# Select type of project to generate: 1) basic 2) application 3) library => 2
# Select implementation language: 1) C++ 2) Groovy 3) Java 4) Kotlin     => 3
# Select build script DSL: 1) Groovy 2) Kotlin => 2
# Select test framework: 1) JUnit 4 2) TestNG 3) Spock 4) JUnit Jupiter  => 4
# ...
```

Default directory structure (Maven-style)

```
.gradle/
build/
gradle/wrapper/*
src/
    main/
        java/*
        resources/*
    test/
        java/*
        resources/*
build.gradle.kts
gradlew
gradlew.bat
settings.gradle.kts
```

- NB: separation of **`main/`** and **`test/`** sources is “standard”

Gradle (2/2)

build.gradle.kts

```
plugins { // declare what "build" modules you use
    java
    application
}
repositories { // repository: where you resolve dependencies
    mavenCentral()
}
dependencies {
    // Main application dependencies
    implementation("org.apache.commons:commons-lang3:3.12.0")
    // Testing dependencies
    val jupiterVersion = "5.4.2"
    testImplementation("org.junit.jupiter:junit-jupiter-api:$jupiterVersion")
    testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine:$jupiterVersion")
}
tasks.withType<Test> { // configure the "test" task (provided by Java plugin)
    useJUnitPlatform()
    testLogging.events("failed","passed","skipped")
}
```

- “Convention over configuration”
- NB: separation of “main” and “test” dependencies
  - you don’t want testing dependencies to be included in your application packages
- NB: separation of “compilation and “runtime dependencies”

Gradle (3/2)

Use gradle or gradlew command to run project-related activities

```
$ ./gradlew tasks --all # list tasks
$ ./gradlew test        # run tests
$ ./gradlew compileJava # compiles main Java sources
```

Key abstractions of Gradle (and many build tools in general)

- Builds
- Projects
- Settings
- Resources
- Tasks and task graphs
- Configurations (i.e., groups of dependencies)
- Scopes (e.g., a project, a configuration, a task)
- Plugins

dotnet SDK (.NET 5)

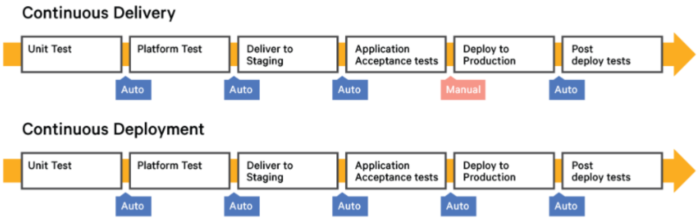
- **dotnet** command

```
# Create solution
dotnet new sln --name testing-basics

# Create and build main project
dotnet new classlib -n u02-unit-testing
dotnet build u02-unit-testing

# Create test project and wire the dependency
dotnet new xunit -n test-u02-unit-testing
dotnet add test-u02-unit-testing reference u02-unit-testing

# Run tests
dotnet test test-u02-unit-testing

# Add projects to solution
dotnet sln add u02-unit-testing
dotnet sln add test-u02-unit-testing

# Run tests on solution
dotnet test
```

Outline

Continuous Integration (CI) / Continuous Delivery (CD)

- Continuous Integration (CI): the practice of ***frequently*** merging all developer working copies to a shared mainline
  - this includes checking that the ***merging*** is fine
- Continuous Delivery (CD): the practice of ***frequently*** producing software in short cycles
  - **Goal:** building, testing, and releasing software ***frequently***
- Continuous Deployment: the practice of ***frequently*** deploying “sound” changes to prod



- these practices build heavily on
  - ***version control systems (VCS)*** (cf. **git**)
    - VCSs help establishing and tracking well-defined ***changes*** to project artifacts
    - VCSs support development ***workflows***
  - build automation
  - virtualisation / containerisation
  - CI/CD platforms (to orchestrate everything—see **GitHub Actions** next)

Continuous-*: why

- Continuous Testing (CT): the practice of running automated tests ***for every change***
  - CI involves CT
  - ***tools running tests in background*** or ***filesystem watchers*** can support CT at a fine granularity
  - **infinitest** (JVM)
    - has plugins for Eclipse and IntelliJ
  - **`dotnet watch test`** (.NET)
- ***Why continuous-*?***
  - ***shortening time-to-feedback reduces risks***
    - the more changes you apply / time passes on before catching regressions, the harder to locate the faults

Github Actions Workflows (1/1)

- **Key concepts**
  - **Workflow**: an **automated, event-triggered procedure made of 1+ jobs for your repo**
  - **Event**: an activity that **triggers a workflow**
  - **Job**: a set of steps that execute on the same runner (so can share data), in a fresh virtual env
  - **Step**: an individual task of a job; either an action or a shell command
  - **Action**: standalone commands that can work as a step in a job
  - **Runner**: a **server** (hosted on GH or on-premise) with the **GA runner application installed**
    - Listens for available jobs, runs one job at a time, reports progress, logs, and results back to GH.

Github Actions Workflows (2/1)

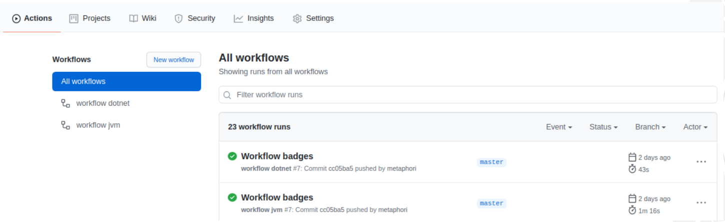
- **How:** You configure GAs via YAML files in `.github/workflows/` ➔
- **Viewing job's activity**: Actions tab in your repo home

.github/workflows/workflow-jvm.yaml

```yaml
name: workflow jvm
on: ['push']
jobs:
  run-tests:
      runs-on: ubuntu-latest
      steps:
        - uses: actions/checkout@v2
        - run: cd jvm./gradlew test
```

.github/workflows/workflow-dotnet.yaml

```yaml
name: workflow dotnet
on: ['push']
jobs:
  run-tests:
      runs-on: ubuntu-latest
      strategy:
        matrix:
          dotnet-v: ['5.0.x']
      steps:
        - uses: actions/checkout@v2
        - name: Setup .NET Core SDK ${{ matrix.dotnet-v
       }}
          uses: actions/setup-dotnet@v1.7.2
          with:
            dotnet-version: ${{ matrix.dotnet-v }}
        - name: Test
          run: dotnet test
```



Outline

Unit Testing in JUnit 5: a Quick Overview

System/Unit-under-test

```
// Abstraction
public interface Device {
  void on(); void off();
  boolean isOn();
}

// Implementation
public class DeviceImpl
  implements Device {
  // Encapsulation
  private boolean isOn;

  // implements Device...
}
```

Run with gradle

```
./gradlew test --tests *
```

Unit testing code

```
public class DeviceTest {           // Test suite
  private Device device;              // SUT / Fixture

  @BeforeEach
  public void init(){                 // (1. Arrange)
    this.device = new DeviceImpl();   // Fixture setup
  }

  @Test
  public void test_turning_on_when_off() {
    assumeTrue(!this.device.isOn());  // Assumption
    this.device.on();                 // (2. Act)
    assertTrue(this.device.isOn());   // (3. Assert)
  }
  //..........
}
```

Concepts

- Test suite (1+ test classes), test cases (test methods)
- Assertions; Assumptions (conditions in which a test is meaningful)
- Fixture; System Under Test (SUT)
- Arrange, Act, Assert
- Test automation, Test discovery, Test execution, Suite/Test lifecycle/hooks, Test reporting

Unit Testing in xUnit.NET: a Quick Overview

System/Unit-under-test

```
// Abstraction
public interface Device
{
    void on(); void off();
    bool isOn { get; set; }
}

// Implementation
public class DeviceImpl : Device
{
    public bool isOn { get; set; }

    public DeviceImpl(){ this.isOn = false; }

    public void on() { isOn = true; }
    public void off() { isOn = false; }
}
```

Unit testing code

```
public class DeviceTest
{
    private readonly Device sut;

    public DeviceTest(){ sut = new DeviceImpl(); }

    [Fact] public void
     Test_Device_TurnOn_From_Starting_State() {
        Assert.True(!sut.isOn);
        sut.on();
        Assert.True(sut.isOn);
}   }
```

Run with dotnet

```
dotnet test
```

Concepts

- Test suite (1+ test classes), test cases (test methods)
- Assertions; Assumptions (conditions in which a test is meaningful)
- Fixture; System Under Test (SUT)
- Arrange, Act, Assert
- Test automation, Test discovery, Test execution, Suite/Test lifecycle/hooks, Test reporting

Outline

Testing Automation

Concerns (typically dealt with an *ecosystem* of tools—aka test harness)

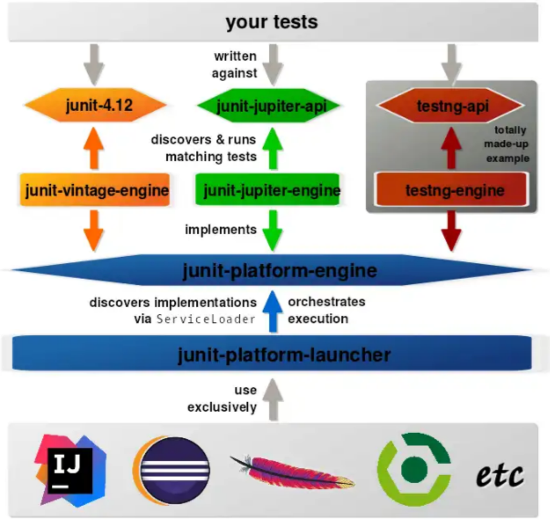
- Test **definition**
  - Specifying test cases
  - Specifying expectations

```
public class DeviceTest {
    private readonly Device sut;

    public DeviceTest(){ sut = new DeviceImpl(); }

    [Fact] public void Test_Device_TurnOn_From_Starting_State() {
        Assert.True(!sut.isOn);
        sut.on();
        Assert.True(sut.isOn);
}   }
```

- Test **execution**
  - Test **discovery**
    - Usually via reflection & annotations/attributes
  - Test **lifecycle management**
    - Tests may require setup / teardown activities
    - Tests may run sequentially / in parallel
    - Fail-fast vs. fail-safe
- Test **reporting**
  - Passed / Failed tests
  - Coverage

```
dotnet test # builds on "integrated" test automation tools
```

JUnit 5 architecture & artifacts (1/3)



(Picture by Nicolai Parlog CC BY-NC 4.0)

Key takeaway

- JUnit 5 provides clear APIs for tools (Platform) and testers (Jupiter)
  - Engines promote discovery and execution of tests
  - JUnit Platform supported by popular ***IDEs*** (e.g. IntelliJ IDEA, Eclipse, NetBeans, Visual Studio Code..) and ***build tools*** (Gradle, Maven, sbt..).

JUnit 5 architecture & artifacts (2/3)

- JUnit 4: (1) requires Java$\geq$5; (2) all in one JAR
- **JUnit 5**: (1) requires Java$\geq$8; (2) modular $\rightarrow$ separation of concerns
  1. **JUnit Platform**: platform for test execution + Engine API (for tools)
  2. **JUnit Jupiter**: (1) API for writing tests + (2) corresponding engine
  3. **JUnit Vintage**: engine for running JUnit 3/4 tests in JUnit Platform

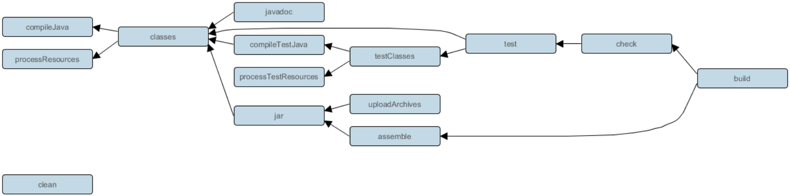
build.gradle.kts

```
plugins {
  java   // configures a 'test' task
  jacoco // coverage
}

dependencies {
  testImplementation("org.junit.jupiter:junit-jupiter-api:5.7.1")
  testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine:5.7.1")
  testRuntimeOnly("org.junit.vintage:junit-vintage-engine:5.7.1")
}

tasks.named<Test>("test") { // configuration for the Test task
  // useJUnit() // (default) scans for JUnit 3/4 tests
  useJUnitPlatform { // use JUnit 5 Platform + Vintage
    includeEngines("junit-jupiter","junit-vintage")
  }
}
```

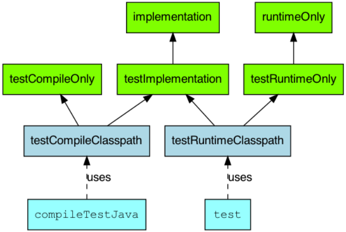
- Note: runners for test classes may be explicitly specified via `@RunWith`

Gradle: testing in Java/JVM projects

- Support for running tests is provided by the **`Java`** plugin
  - Two **source sets**: `main` and `test`
  - Default locations: `src/main/*`, `src/test/*`



  - Task **`test`** depends on `testClasses` + all tasks producing to `testRuntimeClasspath` dependency config.



xUnit.NET project files and relevant packages

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.7.1" />
    <PackageReference Include="xunit" Version="2.4.1" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.4.3">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="coverlet.collector" Version="1.3.0">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
  </ItemGroup>

    <ItemGroup>
    <ProjectReference Include="<path-to-project-under-test>.csproj" />
  </ItemGroup>
</Project>
```

- `xunit` package brings in three child packages
  - `xunit.core` (the testing framework itself)
  - `xunit.assert` (the library which contains the Assert class)
  - `xunit.analyzers` (which enables Roslyn analyzers to detect common issues with unit tests)
- `xunit.runner.visualstudio`: for running tests in Visual Studio
- `Microsoft.NET.Test.Sdk`: for running your test project with **`dotnet test`**
- `coverlet.collector` package: allows collecting **code coverage**.

Outline

1. Testing: a Concise Introduction
2. Preliminaries
3. Unit Testing
   - xUnit Automation Architecture
   - Unit testing in practice
4. Working on tests
5. Test-Driven Development
6. Wrap-up

Test objects, test items

A very simple SUT

```
public interface Device
{
    void on(); void off();
    bool isOn { get; set; }
}

public class DeviceImpl : Device
{
    public bool isOn { get; set; }

    public DeviceImpl(){ this.isOn =
     false; }

    public void on() { isOn = true; }
    public void off() { isOn = false; }
}
```

Another very simple SUT

```
public class Calculator {
    public int add(int v1, int v2){
        return val1 + val2;
    }

    public int subtract(int v1, int v2){
        return v1 - v2;
    }
}
```

Definitions

- **Test object**: The work product to be tested. [11]
  - e.g. `DeviceImpl`, `Calculator`
  - **System-Under-Test (SUT)**: A type of test object that is a system. [11]
- **Test item**: A part of a test object used in the test process. [11]
  - e.g. `{DeviceImpl.off(),DeviceImpl.isOn}`, `Calculator.add()`, `Calculator.subtract()`
  - **Test process**: The set of interrelated activities comprising of test planning, test monitoring and control, test analysis, test design, test implementation, test execution, and test completion. [11]

Test cases (1/2)

- Test cases are usually represented via test methods
  - **Test methods** are, typically, annotated public void instance methods
  - NB: a test method may represent one ***or more*** test cases

JUnit 5

```
@Test void my_test(){ /* test case impl */ }
@ParameterizedTest .. void my_test2(){ /* test case impl */ }
```

xUnit.NET

```
[Fact] public void My_Test(){ /* test case impl */ }
[Theory] .. public void My_Test2(...){ /* test case impl */ }
```

Test cases (2/2)

Definitions

Test case: A set of preconditions, inputs, actions, expected results and postconditions, developed based on test conditions. [11]

- **Test condition** (test requirement, test situation): A testable aspect of a component or system identified as a basis for testing. [11]
- **Expected result**: The observable predicted behavior of a test item under specified conditions based on its test basis. [11]
- **Test basis**: The body of knowledge used as the basis for test analysis and design. [11]
- **Precondition**: The required state of a test item and its environment prior to test case execution. [11]
- **Postcondition**: The expected state of a test item and its environment at the end of test case execution. [11]

Test cases (3/2)

```
public class SomeTests {
  private Device device;

  @BeforeEach
  public void init(){ device = new DeviceImpl(); }

  @Test
  public void test_turning_on_when_off() {
    assumeTrue(!device.isOn()); // checking precondition
    device.on(); // action
    assertTrue(device.isOn()); // checking postcondition
  }
}
```

- **Test object:** `Device`
- **Test item:** the parts of `Device` related to monitoring & control of its state
- **Test case:** "turning on the device, when off"
  - **Test condition:** the ability to be turned on when off
  - **Inputs:** none
  - **Action:** requesting to be turned on
  - **Expected result:** no exception
  - **Precondition:** the device is off
  - **Postcondition:** the device is on

Test cases (4/2)

```
public class CalculatorTest
{
    private readonly Calculator calc;

    public CalculatorTest(){
        calc = new Calculator();
    }

    [Theory]
    [InlineData(1, 2, 3)]
    [InlineData(-1, -2, -3)]
    [InlineData(0, 5, 5)]
    [InlineData(-5, 0, -5)]
    [InlineData(int.MinValue, -1, int.MaxValue)]
    public void Test_Calculator_Add(int v1, int v2, int expected) {
        Assert.Equal(expected, calc.Add(v1,v2));
    }
}
```

- **Test object:** `Calculator`
- **Test item:** `Calculator.Add()`
- **Test cases:**
  - **Test conditions:** the different ways of summing two integers
  - **Inputs:** given
  - **Action:** requesting the sum
  - **Expected result:** given
  - **Pre-/Post-condition:** none

Test suites (1/2)

- **Test suites (sets of test cases)** are usually represented via test classes (and test packages/projects) or by tagging tests
- This is related to test filtering (choosing which tests to run)

JUnit 5

```
public class MySuite {
  @Test @Tags({ @Tag("basics"), @Tag("foo") })
  void my_test1(){ /* test case impl */ }
  @Test void my_test2(){ /* test case impl */ }
}

@DisplayName("Appliances suite") @RunWith(JUnitPlatform.class)
@SelectClasses({DeviceImplTest.class, CalculatorTest.class})
public class AppliancesSuite { }

@DisplayName("Basics suite") @RunWith(JUnitPlatform.class)
@SelectPackages("it.unibo") @IncludeTags({ "basics" })
public class BasicsSuite { }
```

```
tasks.register("metatest", Test::class) {
    useJUnitPlatform {
        includeTags("basics")
        includeTestsMatching("*Test")
        // also: ./gradlew test --tests *Test
    }
}
```

Test suites (2/1)

xUnit.NET

```
public class MySuite {
  [Fact] [Trait("Category","Basics")]
  public void My_Test1(){ /* test case impl */ }
  [Fact] public void My_Test2(){ /* test case impl */ }
}
// dotnet test --filter "Category=Basics"
```

Definitions

- **Test suite** (**test set**): A set of test scripts or test procedures to be executed in a specific test run. [11]
  - **Test script**: A sequence of instructions for the execution of a test. [11]
  - **Test procedure**: A sequence of test cases in execution order, and any associated actions that may be required to set up the initial preconditions and any wrap up activities post execution. [11]

Assertions (1/1)

- **Assertions** help us specify what should/must hold
  - Expressive assertions make tests more ***readable***

JUnit 5 `Assertions`

```java
import static org.junit.jupiter.api.Assertions.*;

public class TestAssertions {
    @Test
    public void test_assertions(){
        assertTrue(true);
        assertEquals(7., 5f+2);
        assertEquals(7.2, 7.0, 0.5); // tolerance = 0.5
        assertNotEquals(7, 5+3);
        assertArrayEquals(new int[]{ 1, 2, 3},
                          List.of(1, 2, 3).stream().mapToInt(i->i).toArray());
        assertTimeout(Duration.ofMillis(500), () -> Thread.sleep(100));
        assertTimeoutPreemptively(Duration.ofMillis(500), () -> Thread.sleep(100));
        assertAll(() -> assertTrue(true),
                  () -> assertFalse(false));
        assertDoesNotThrow(() -> { });
        assertThrows(RuntimeException.class,
                     () -> { throw new RuntimeException("!!!"); } );
        assertNotSame(new int[]{ 1 }, new int[] { 1 });
        assertSame(this, this);
        assertNull(null);
        assertNotNull(this);
        // Descriptive messages for assertXXX()
        assertTrue(true, "an assertion");
        assertTrue(true, () -> "an assertion");
    }
```

Assertions (2/1)

xUnit.NET (**xunit.assert** library's **Assert** class)

```
public class TestAssertions
{
    [Fact]
    public void Test_Assertions() {
        Assert.True(true);
        Assert.Equal(7.0, 5+2);
        Assert.Equal(7.2, 5+2, 0); // tolerance up to 0 decimal digits
        Assert.StrictEqual(7.0, 5+2);
        Assert.Contains(new int[]{ 1,2,3 }, x => x == 3);
        Assert.DoesNotContain("foobar", "xxx");
        Assert.StartsWith("foo", "foobar");
        Assert.InRange(77, 0, 100);
        Assert.ThrowsAsync<Exception>(() => throw new Exception("!!!"));
        Assert.Superset(new HashSet<int> { 2, 3 }, new HashSet<int>{ 5, 2, 3
    });
    }
```

Assumptions and skipping/ignoring tests

- Sometimes, we may want to **run tests conditionally**
  - e.g., depending on the test environment
  - (Recall) **Precondition**: required state of test item/environment prior to test case execution [11]

JUnit 5

```
@Test
public void test_with_assumptions(){
  Assumptions.assumeTrue(System.getProperty("os.name").equals(...));
  // ...
}
```

xUnit.NET

```
public class CustomIgnoreFactAttribute : FactAttribute {
  public CustomIgnoreFactAttribute() {
    if(/* some logic */) Skip = "Ignored because ...";
  }
}

[CustomIgnoreFact]
public void Test_With_Assumptions(){ /* ... */ }
```

- Sometimes, we may want to ***temporarily*** **skip** execution of certain tests
  - e.g., because a test turned out to be ***wrong*** and needs amendments

JUnit 5

```
@Test @Disabled
public void test_to_ignore(){ /* .. */ }
```

xUnit.NET

```
[Fact (Skip = "specific reason")]
public void Test_To_Skip(){ /* .. */ }
```

Test lifecycle and test fixture (1/1)

- Tests should run **isolated** (should not affect each other)
- On the other hand, it is common for test classes to share setup/cleanup logic for preparing a common test context for multiple test cases
- **Fixture**: fixed environment used to consistently run tests
  - It includes the SUT + dependencies (e.g., input data, collaborators, a DB)

Test lifecycle and test fixture (2/2)

JUnit 5

- By default, JUnit creates **a new instance of the test class for each test case**
  - ⚠ so, you **can't share instance variables across test cases**
  - unless you annotate the test class with `TestInstance(Lifecycle.PER_CLASS)`

```
public class TestLifecycle {
    private int k = 0;

    @ParameterizedTest @ValueSource(strings = { "", "foo", "bar", "x" })
    public void test_string_size_non_negative(String s){
        System.out.println(k++); // will print "0" for every test case
        Assertions.assertTrue(s.length() >= 0);
    }
```

- **Lifecycle methods:** `@BeforeAll, @AfterAll, @BeforeEach, @AfterEach`

```
public class TestSuite {
  @BeforeAll  public static void beforeAll() { /* ... */ }
  @AfterAll   public static void afterAll()  { /* ... */ }
  @BeforeEach public void beforeEach(){ /* ... */ }
  @AfterEach  public void afterEach() { /* ... */ }
```

Test lifecycle and test fixture (3/2)

xUnit.NET

- xUnit also creates **a new instance of the test class for each test case**
  - setup code can go in the **constructor**
  - cleanup code by implementing **`IDisposable`** (method **`Dispose()`**)
  - class-wide setup/cleanup by using a class fixture (**`IClassFixture<C>`**)
  - fixture spanning multiple test classes using a collection fixture

```
public class TestLifecycle : IDisposable, IClassFixture<MyClassFixture> {
    public TestLifecycle(){ Console.WriteLine("Before Each"); }

    public void Dispose(){ Console.WriteLine("After Each"); }

    [Theory] [InlineData("")] [InlineData("foo")]
    public void Test_Strlen(string s) { Assert.True(s.Length >= 0); }
}

public class MyClassFixture : IDisposable {
    public MyClassFixture(){ Console.WriteLine("Before All"); }
    public void Dispose(){ Console.WriteLine("After All"); }
}

[CollectionDefinition("GlobalFixture")]
public class GlobalFixture : ICollectionFixture<SomeFixture> { }

[Collection("GlobalFixture")]
public class SomeTestClass {
    SomeFixture fixture;
    public DatabaseTestClass1(SomeFixture f) { this.fixture = f; }
}
```

3A (Arrange – Act – Assert)

- **3A (Arrange – Act – Assert)**: a common pattern for ***structuring the code*** of test cases

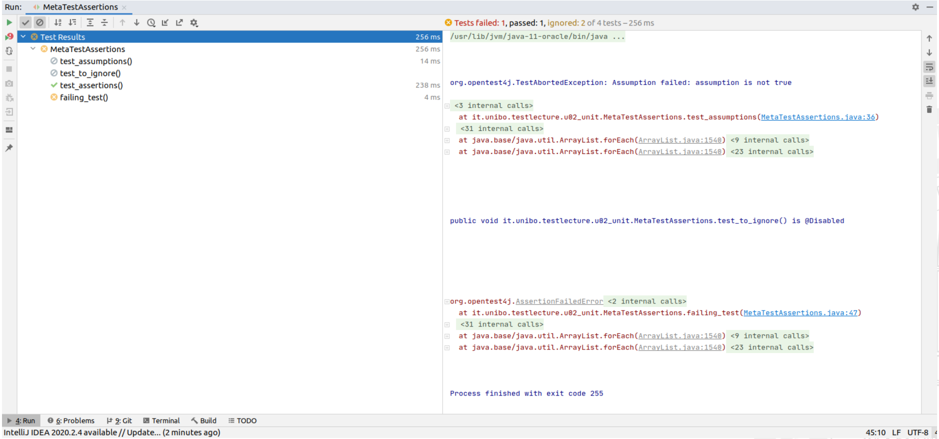
```
@Test
public void test_turning_off_when_on() {
    // Arrange
    device.on();
    assumeTrue(device.isOn());

    // Act
    device.on();

    // Assert
    assertTrue(device.isOn());
}
```

- Sometimes, the “arrange” part is (partially) built in setup code (cf. lifecycle methods)
- **Style advice:** avoid comments; use blank lines to separate the AAA sections

```
@Test
public void test_turning_off_when_on() {
    device.on();
    assumeTrue(device.isOn());

    device.on();

    assertTrue(device.isOn());
}
```

Test execution & reporting (1/1)

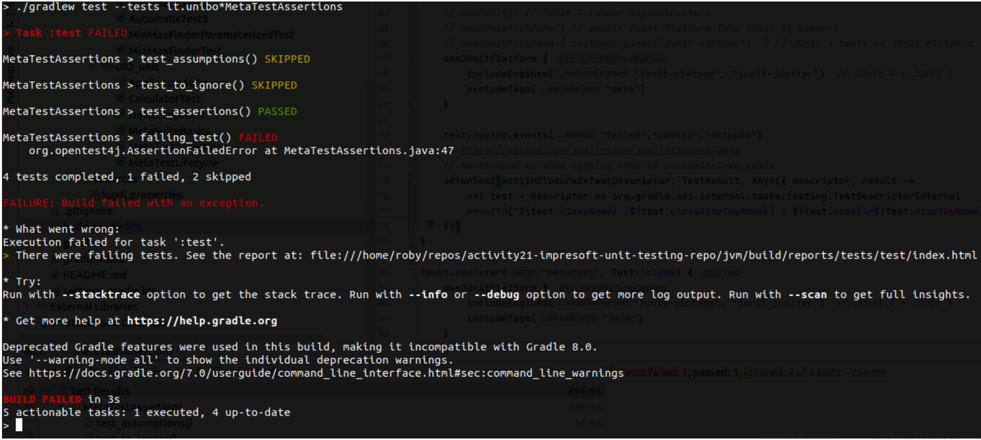
a) From IDE (e.g., IntelliJ)
- using built-in support / plugins
- leveraging integration with build tools
  - **IntelliJ**: File → Settings → Build, Execution, and Deployment → Gradle → Run Tests With ..



Test execution & reporting (2/1)

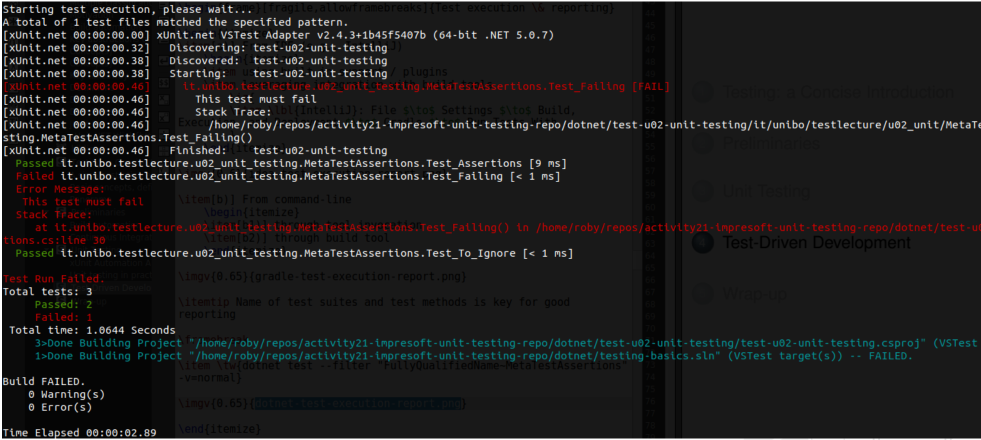
b) From command-line
   - b1) through tool invocation
   - b2) through build tool

- `./gradlew test --tests it.unibo*MetaTestAssertions`



Test execution & reporting (3/2)

- `dotnet test --filter "FullyQualifiedName~MetaTestAssertions" -v=normal`



- Name of test suites and test methods is key for good reporting

Test execution & reporting (4/2)

More on **reports**

- Gradle
  - generates XML reports under `build/test-results/<test-task>/TEST-<testclass>.xml`
  - generates an HTML report under `build/reports/tests/<test-task>/index.html`
- `dotnet`
  - using ***loggers*** (e.g., **xunit.testlogger** which yields **xUnit v2 XML reports**)

```
dotnet add package XunitXml.TestLogger --version 3.0.66
dotnet test --logger:"xunit;LogFilePath=test_result.xml"
```

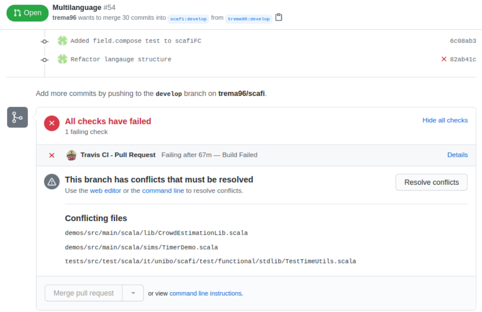
Comparison of unit testing frameworks [12]

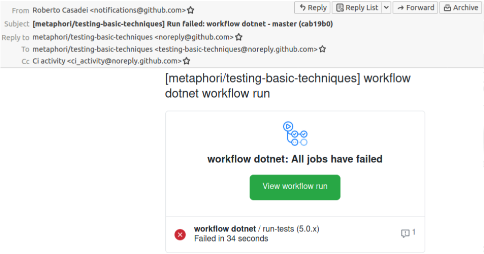
	JUnit 4	JUnit 5	xUnit	NUnit	MSTest
Test method	@Test	@Test	[Fact]	[Test]	[TestMethod]
Test class	n/a	n/a	n/a	[TestFixture] (opt.)	[TestClass]
Before each (setup)	@Before	@BeforeEach	Constructor	[SetUp]	[TestInitialize]
After each (teardown)	@After	@AfterEach	IDisposable.Dispose	[TearDown]	[TestCleanup]
Before all	@BeforeClass	@BeforeAll	IClassFixture<T>	[OneTimeSetUp]	[ClassInitialize]
After all	@AfterClass	@AfterAll	IClassFixture<T>	[OneTimeTearDown]	[ClassCleanup]
Disabling tests	@Ignore	@Disabled	[Fact(Skip=".")]	[Ignore]	[Ignore]
Tagging	@Category	@Tag	[Trait]	[Category]	[TestCategory] (VS)
Data-driven tests	@RunWith(Parameterized.class)	@ParameterizedTest + @ValueSource etc.	[Theory] + [InlineData] etc.	[TestCase]	[DataTestMethod] + [DataRow]

- Mostly a matter of platform / support / taste
- JVM
  - **JUnit 5** is a strong choice
  - Another strong choice is **TestNG**
- .NET
  - **xUnit** cleaner and more concise than nUnit (it uses fewer attributes) [12]

Regression testing

- **Regression testing** = **repeat testing**, carried out using previously used test cases
  - **Goal: preventing regressions**: i.e., changes in code leading to failing tests that were before successful
- **Regression tests** are important in a software development process
  - **Block** integration of changes into the some branch if regressions exist
  - **Notify** developers in case of regressions





Outline

Outline

Code/Test coverage

Coverage

- **Coverage** is a **property** of a **set of tests** and **target system**
  - Several definitions/metrics (and inconsistent terminology)
  - Many "things" can be "covered", and at many levels
- **Code coverage**: which/how much code is executed during testing
  - Mainly used for finding untested code[a]
  - Note: 100% coverage doesn't mean that tests cover all the scenarios
- **Test coverage**: how much of the ***behaviour*** is tested
  - Another meaning: how much tests have been executed in some context

[a] https://martinfowler.com/bliki/TestCoverage.html

How much coverage?

- **Coverage metrics** are a **good negative indicator** but a **bad positive one** [12]
  - ⚠ Imposing a certain "coverage target" creates a perverse incentive.
- The goal is not 100% coverage [13] → testing the "right things"
  - considering effort vs. risk

Different kinds of “coverage”

```
A;
if(C1 && C2) B; else C;
D;
if(C3  C4) E; else F;
G;
if(C5) H;
```

- **Statement coverage**—How many “statements” are covered:
  `A,B,C,D,E`, ...
- **Branch/Decision coverage**—How many “branches” are covered:
  `C1 && C2==true`, `C1 && C2==false`, ...
- **Condition coverage**—How many “individual conditions” affecting “decisions” are covered
  `C1=true`, `C1=false`, ...
  - ⚠ **Condition coverage does not guarantee Decision coverage**

$D = C_1 \vee C_2$	C_1	C_2
T	F	T
T	T	F

- **Decision-condition coverage**: decision coverage & condition coverage
- **Path coverage**—How many “execution paths” are covered:
  `ABDEG`, `ACDEGH`, ...
- **Increasing strenght of coverage**: statement $\rightarrow$ decision $\rightarrow$ condition $\rightarrow$ decision/condition $\rightarrow$ path

Comparison of coverage metrics

In a set of test cases designated for	Each statement is executed at least once	Each decision takes on all possible outcomes at least once	Each condition in a decision takes on all possible outcomes at least once	All possible combinations of condition outcomes occur at least once
Statement Coverage	Y	**N**[2]	N	N
Decision Coverage	Y	Y	**N**[3]	N
Condition Coverage	**N**[4]	**N**[5]	Y	N
Decision-Condition Coverage	Y	Y	Y	**N**[6]

[2] Cf. `if`s without `else` branch

[3] Cf. $C_1 \vee C_2$: we can decision cover by choosing $\{[\top, \bot], [\bot, \bot]\}$

[4] Since you do not necessarily have decision coverage..

[5] Cf. $C_1 \wedge C_2$: we can condition but not decision cover by choosing $\{[\top, \bot], [\bot, \top]\}$

[6] Cf. $C_1 \vee C_2$: we can condition-decision cover by choosing $\{[\top, \top], [\bot, \bot]\}$

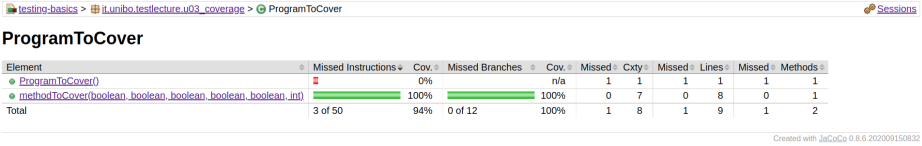
Coverage on the JVM: JaCoCo

- **Mission:** standard for code coverage analysis on the JVM
  - **Other coverage tools:** **EMMA**, **Cobertura**
- **Focus:** lightweightness, flexibility, documentation, ***integration with build/dev tools***
- **Integrations:** Ant, Maven, **Gradle**, **IntelliJ IDEA**, Eclipse, Codacy, Codecov..
- **How it works:** JaCoCo runs as a Java agent that instruments (via ASM) the bytecode while running tests

JaCoCo plugin in Gradle

- **`jacoco`** plugin provides a task **`jacocoTestReport`**
- By default, a HTML report is generated at `build/reports/jacoco/test`

```
plugins {
    jacoco
}
```



testing-basics > it.unibo.testlecture.u03_coverage > ProgramToCover Sessions

ProgramToCover

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
ProgramToCover()		0%		n/a	1	1	1	1	1	1
methodToCover(boolean, boolean, boolean, boolean, boolean, int)		100%		100%	0	7	0	8	0	1
Total	3 of 50	94%	0 of 12	100%	1	8	1	9	1	2

Created with JaCoCo 0.8.6.202009150832

Coverage in .NET 5 (1/1)

- The xUnit test project template integrates with **coverlet.collector** by default

```
dotnet test --collect:"XPlat Code Coverage"
```

  - The `"XPlat Code Coverage"` argument is a friendly name that corresponds to the data collectors from Coverlet. This name is required (case insensitive).

- **Output:** **TestResults/{guid}/coverage.cobertura.xml**
- **Generate reports:** using **ReportGenerator**

```
dotnet tool install -g dotnet-reportgenerator-globaltool
reportgenerator
"-reports:Path\To\TestProject\TestResults\{guid}\coverage.cobertura.xml"
"-targetdir:coveragereport"
-reporttypes:Html
```

Coverage in .NET 5 (2/3)

< Summary

Class:	it.unibo.testlecture.u03_coverage.ProgramToCover
Assembly:	u02-unit-testing
File(s):	File 1: /home/roby/repos/activity21-impresoft-unit-testing-repo/dotnet/u02-unit-testing/it/unibo/testlecture/u03_coverage/ProgramToCover.cs
Covered lines:	9
Uncovered lines:	0
Coverable lines:	9
Total lines:	16
Line coverage:	100% (9 of 9)
Covered branches:	4
Total branches:	6
Branch coverage:	66.6% (4 of 6)

Methods/Properties

methodToCover(...)

Metrics

Method	Branch coverage	Cyclomatic complexity	Line coverage
File 1: methodToCover(...)	66.66%	6	100%

File(s)

/home/roby/repos/activity21-impresoft-unit-testing-repo/dotnet/u02-unit-testing/it/unibo/testlecture/u03_coverage/ProgramToCover.cs

#	Line	Line coverage
	1	`using System;`
	2	
	3	`namespace it.unibo.testlecture.u03_coverage {`
	4	`    public class ProgramToCover {`
2	5	`        public static void methodToCover(bool c1, bool c2, bool c3, bool c4, bool c5, int k) {`
2	6	`            Console.Write("A");`
4	7	`            if(c1 && c2) Console.Write("B"); else Console.Write("C");`

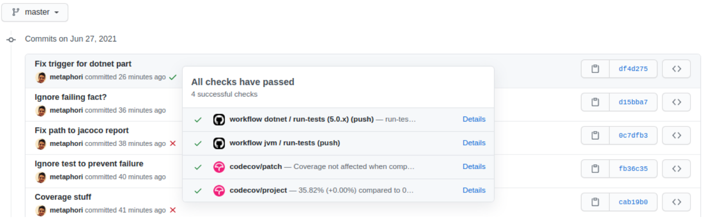
Publish coverage results on the cloud

- You can use services like **CodeCov**
  - Generate a secret token to be added to you're repository environment
  - Define a workflow step handling the publishing
    - CodeCov GH action

.github/workflows/workflow-jvm.yaml

```
  - name: Upload coverage to Codecov
    uses: codecov/codecov-action@v1
    with:
      token: ${{ secrets.CODECOV_TOKEN }}
      files: ./jvm/build/reports/jacoco/test/jacocoTestReport.xml
```

- Top features
  - **Pull Request Comments**: provides an overview of how a PR affects coverage
  - **Commit Status**: e.g. blocking PRs that don't meet a certain coverage threshold
  - **Merging Reports**: e.g. generated through multiple tools



Outline

Test case design and documentation

- **Different types of systems need different approaches to test case design**
  1) **Transaction-based systems**: tend to have complex, structured inputs;
  2) **Control systems**: tend to have a relatively large number of fairly simple inputs (e.g., signals from sensors); outputs are also simple (signals send to actuators); ***I/O relationship NOT simple*** (depending on history of inputs leading to a certain ***state***)
  3) **I/O transformers**: e.g. calculators, compilers, translators..; ***precise rules governing I/O relationship***
- **Test case documentation (IEEE-829):**
  1) test case ID;
  2) test items
  3) input specifications (test data);
  4) output specifications (expected result);
  5) environmental needs;
  6) special procedural requirements;
  7) inter-test case dependencies (what other TCs must be executed before this one, and why)
- **Issue: test outcomes have to be predicted in advance**; **how?**
  1) Define them yourself / look them up
  2) **Generate through a test oracle**

Test oracle

- **Test oracle**: any program, process, or body of data that specifies the **expected outcome** in a test case
- **Test oracle problem**: ***determining the correct output for a given input***
- **A classification for test oracles:**
  1) **Specified**: based on formal specifications
  2) **Derived**: distinguishes in/correct behaviour by using info derived from artifacts of the system (e.g., docs, logs, previous tests)
  3) **Implicit**: relies on implied info and assumptions (e.g. program crash)
  4) **Human**: human input is used to determine the test oracles
- **Common oracles include:** specs and docs; purchased test suites; other products; heuristics (providing approx results or exact results on few inputs); statistical oracles; consistency oracles (derived oracle comparison results of other test executions); model-based oracles (use same model to generate and verify system behaviour); human oracle (manually analyse correctness of the SUT)

Example: here, we “programmed” an oracle “2+3”

```
assertEquals(2 + 3, SUT.sum(2, 3), "2 + 3 is 5");
```

- Some **types of test** are more suited to use of **automatic test oracles**
  - e.g., what test oracles for usability?

Structural testing (white-box testing)

- Best applied at level of **unit testing** (small programs)
- Best carried out by the **author** of the unit under test
- **Goal:** detect and fix **structural bugs**
  - Control bugs; sequence bugs
  - Missing paths; unreachable code
  - Logic bugs, initialisation bugs
- **Main approaches:**

1) **Control FlowGraph (CFG)**: **static**; technique for constructing a graph-based representation of a program's control flow
2) **Path testing techniques**: **dynamic**; family of techniques based on judiciously selecting a set of test paths through a program
   - full path coverage typically impossible in practice
3) **Data flow testing**: **dynamic**; focuses on tracing a sequence of events of individual variables in code (e.g., **D**ef/Decl, **U**se, **C**omputation, **T**ermination)

⚠ These are best used in combination but, still, they cannot achieve complete testing; so WBT should extend info to other techniques

WBT » Control Flow Graph (CFG)

- **Control Flow Graph (CFG)**: **graphical representation of the control structure of SW**
  - A CFG ***represents all the paths that MIGHT be taken during execution*** of a program

CFG structure

- CFG ≈ **flowchart**. **Differentia:** in CFG, the **nodes** are sequences of statements called **basic blocks**
- **Basic block**: sequence of statements that has **one entry point** and **one exit point**
- **Entry block**: basic block through which all control flow ***enters*** the CFG
- **Exit block**: basic block through which all control flow ***leaves*** the CFG
- The **edges** of a CFG represent **decisions, cases, junctions** in the **control flow**

Building CFGs

- **Manual construction:** number basic blocks; identify entry/exit blocks; add edges
- **Goal:** prepare for **path testing** or support **static analysis** / program visualisation

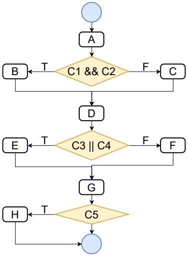
- **Path**: sequence of statements from an entry block to an exit block through a certain number of control flow decisions/junctions
- **CFG and coverage ➔:** statement coverage (basic blocks visited at least once); decision coverage (branches traversed at least once); path coverage (paths traversed at least once)

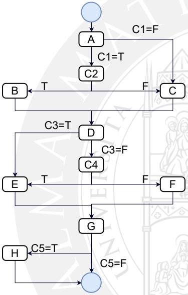
Cyclomatic complexity [14]

- The **cyclomatic complexity (CC)** of a section of source code is the number of linearly independent paths within it
  - "linearly independent": each path has at least one edge that is not in one of the other paths
  - So, the CC is the minimum number of paths that may generate, when combined, all possible paths
- $M = E - N + 2C$
  - M: complexity
  - E: num of edges
  - N: num of nodes
  - C: num of connected components ($C = 1$ for a single method)
- For a program with 1 entry point and 1 exit point, it is the number of decision points + 1 (where decision points with compound predicates should be split)

```
A;
if(C1 && C2) B; else C;
D;
if(C3  C4) E; else F;
G;
if(C5) H;
```

- $M = 16 - 12 + 2 = 6 = 5 + 1$





[14] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, 1976

Functional/black-box testing (BBT): intro

- **Functional testing**: treat the program as a **black box**
  - While WBT ➔ starts with source code, **BBT** starts with the **specification**
- **Main functional testing techniques:**

1) **Equivalence Partitioning (EP)**
2) **Boundary Value Analaysis (BVA)**
   - Both EP and BVA are systematic **domain testing** techniques
   - BVA is more effective than EP at finding bugs

- **Two approaches for automated, script-driven testing for BBT of components:**

1) **Data-driven testing**: divides scripting from test data—test data is read from a table/spreadsheet and a generic script performs the same test with different data
   - cf. parameterized tests in JUnit; theories in xUnit.NET
2) **Keyword-driven testing**: the table contains **keywords and test data**; the keywords are specific to the system and describe actions to be taken by the program.

BBT » Equivalence Partitioning

- **Recall: completeness problem** – often impractical to run all possible test cases
- Equivalence Relation (RST): ***Reflexive, Symmetric, Transitive***
- Equivalence Class: set of objects in an equivalence relation
- As far as an equivalence relation is concerned, all the objects in an equivalence class are the same ➔ **one object can represent the entire class!**
  - For the **Economics of testing** (cf. Myers), we do not want **redundant test cases**. E.g., in path testing, we do not want to test the same path multiple times.
- ***How to select equivalence classes?*** With **equivalence partitioning**
  - ***The equivalence classes partition the input domain***
  - Partitioning is really a matter of **classification**: when a program receives an input, it classifies the input and acts accordingly.
- **Rules of thumb**
  - N inputs ➔ 1 valid class (N inputs); 2 invalid (no inputs and $N+1$ inputs)
  - Input is a range $[A, B]$ ➔ 1 valid class; 2 invalid ($[A-1]$ and $[B+1]$)
  - "X must be $= Y$" ➔ 1 valid class ($X = Y$); 1 invalid ($X = Z \neq Y$)
- **Specifying equivalence classes:** class ID; class name; description; valid/invalid; example value
- **Using equivalence classes for BBT:** (1) start from the specs; (2) identify valid and invalid classes for each input; (3) create TCs covering as many valid equivalence classes as possible; (4) **use only one invalid class per TC (do not combine two invalid inputs on one TC)**

Equivalence Partitioning: Example

- **Example:** a "word count" program which accepts:
  - a string `w` for the word to count;
  - a string filename `f` for the file to read.

Class	`w`	`f`	Example	N/Valid
E1	$\neg$ (valid word)	(valid filename) $\wedge$ exists	`w=null, f="somefile"`	N
E2	$\neg$ (valid word)	(valid filename) $\wedge$ $\neg$ exists	`w="", f="nonexistent"`	N
E3	$\neg$ (valid word)	$\neg$ (valid filename)	`w="", f="/"`	N
E4	(valid word)	(valid filename) $\wedge$ exists	`w="foo", f="somefile"`	V
E5	(valid word)	(valid filename) $\wedge$ $\neg$ exists	`w="foo", f="nonexistent"`	N
E6	(valid word)	$\neg$ (valid filename)	`w="foo", f="/"`	N

  - for some notion of "valid word", "valid filename", "exists"
  - the partitions may be chosen based on the expected behaviour of the SUT

BBT » Boundary Value Analysis (BVA)

- **Boundary Value Analysis (BVA)** builds on Equivalence Partitioning
- **Idea:** **values at the boundary (i.e., on the min and max edges) of an equivalence partition are tested**
- **Motivation:** boundaries are common locations for errors often resulting in faults
- Missing/empty values are considered as boundary values

E.g.: Year of birth `yyyy`: 4 digits (valid), not only digits, less digits, more digits

Decision tables

- Test case design can focus on specific types of inputs, but often it is the specific **combination of inputs (conditions)** to determine the expected outcome (action)
- **Decision table**:
  - **Left:** Stub – **Right:** Entries – **Top:** Conditions – **Bottom:** Actions
  - Each **column** in the entries is a rule. **Each rule is the basis for a test case**!

c_1	F	F	F	F	T	T	T	T
c_2	F	F	T	T	F	F	T	T
c_3	F	T	F	T	F	T	F	T
a_1	X	X						
a_2							X	
a_3	X		X					
a_4		X				X	X	

- **Check for completeness**: calculate the total num of rules required (2^c rules for c conditions—cf. truth tables)
- **Check for ambiguity**: look for two rules with ***same conditions*** leading to ***different actions*** (multiple Xs in a column)
- **Check for redundancy**: by looking for two rules with ***different conditions*** leading to ***same action*** (multiple Xs in a row)

Decision table: example

- **Example:** login functionality

c_1: valid ID	F	F	F	F	T	T	T	T
c_2: valid passw	F	F	T	T	F	F	T	T
c_3: 3 bad attempts	F	T	F	T	F	T	F	T
a_0: success							X	
a_2: "bad ID" msg	X	X	X	X				
a_3: "retry passw" msg					X			
a_4: account locked						X		
impossible								X

- 4 test cases are sufficient
  - TC1: $\neg c_1$
  - TC2: $c_1 \wedge \neg c_2 \wedge \neg c_3$
  - TC3: $c_1 \wedge \neg c_2 \wedge c_3$
  - TC4: $c_1 \wedge c_2 \wedge \neg c_3$

> **collapsed/limited decision table**

c_1: valid ID	F	T	T	T
c_2: valid passw	-	F	F	T
c_3: 3 bad attempts	-	F	T	F
a_0: success				X
a_2: "bad ID" msg	X			
a_3: "retry passw" msg		X		
a_4: account locked			X	

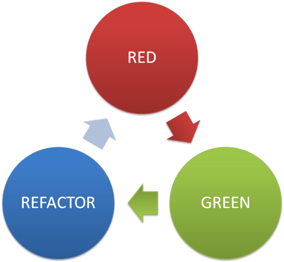
Outline

Test-Driven Development (TDD) [5]

- A.k.a. **test-first development** (cf. **test-last development**)
- A core technique of **eXtreme Programming** [4]

What

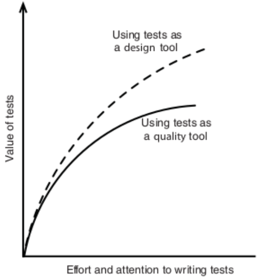
- “Development” is guided (“driven”) by “tests”
  - I.e., it is not “really” a testing activity; tests as (useful) side-effect
- Exercise your code before even writing it
  - Makes you **explore** the problem before attempting a solution
  - Forces you to think as the **user** of your code
  - Makes you **focus** on what is important right now (cf., KISS)
- **Red – Green – Refactor** cycle



TDD: why

Benefits

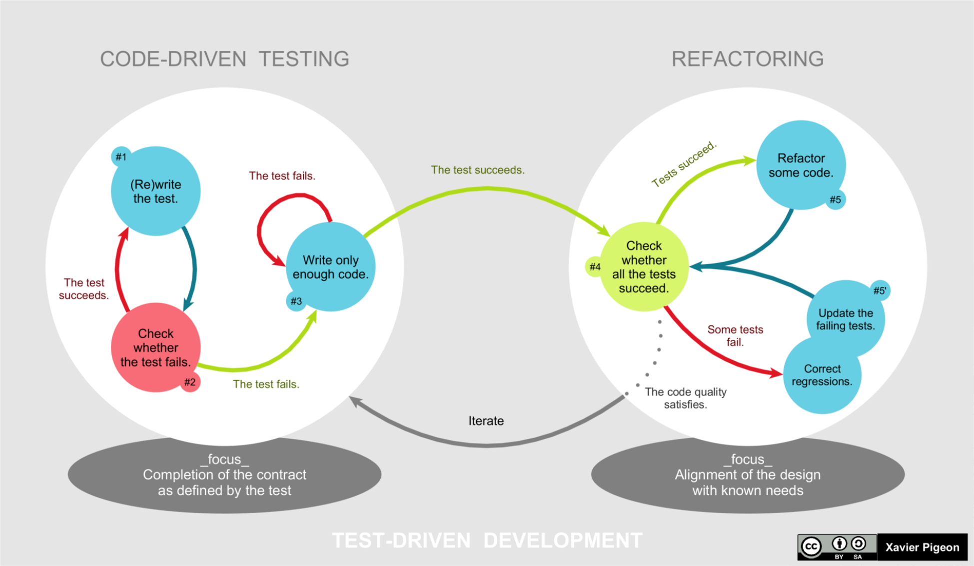
- **Specification** before implementation
- **Guides** the (detailed) **design** process
- You end up with **regression tests** (i.e., tests to catch regressions)



"The biggest value of writing a test lies not in the resulting test but in what we learn from writing it." [13]

[13] L. Koskela, *Effective Unit Testing: A Guide for Java Developers*, ser. Running Series. Manning, 2013

TDD: a detailed view



Example: DeviceManager (1) RED

Phase #1a (RED): write your test

```
public class DeviceManagerTest {
    @Test
    public void add_two_devices_to_manager(){
        DeviceManager dm = new DeviceManagerImpl();
        Device d1 = new DeviceImpl();
        Device d2 = new DeviceImpl();

        dm.add(d1);
        dm.add(d2);
        List<Device> devices = dm.managedDevices();

        assertTrue(devices.contains(d1));
        assertTrue(devices.contains(d2));
    }
}
```

Phase #1b (RED): make it compile

```
public interface DeviceManager {
    void add(Device d);
    List<Device> managedDevices();
}

public class DeviceManagerImpl implements DeviceManager {
    @Override public void add(Device d) { }
    @Override public List<Device> managedDevices() { return null; }
}
```

Phase #2c: see it RED!

```
./gradlew test
```

Example: DeviceManager (2) GREEN

Phase #2a (GREEN): write “just enough” code to make the test pass

```
public class DeviceManagerImpl implements DeviceManager {
    private List<Device> devices;

    public DeviceManagerImpl(){
        devices = new ArrayList<>();
    }

    @Override
    public void add(Device d) {
        devices.add(d);
    }

    @Override
    public List<Device> managedDevices() {
        return devices;
    }
}
```

Phase #2b: see it GREEN!

```
./gradlew test
```

Example: `DeviceManager` (3) REFACTOR

Phase #3a (REFACTOR): improve your code

```
public interface DeviceManager {
    void addDevice(Device d);
    List<Device> managedDevices();
}

public class DeviceManagerImpl implements DeviceManager {
    private List<Device> devices;

    public DeviceManagerImpl(){ devices = new ArrayList<>(); }

    @Override public void addDevice(Device d) { devices.add(d); }

    @Override public List<Device> managedDevices() {
      return Collections.unmodifiableList(devices);
}   }
```

Phase #3a (REFACTOR): adjust your test if needed

```
public class DeviceManagerTest {
    @Test
    public void add_two_devices_to_manager(){
        DeviceManager dm = new DeviceManagerImpl();
        // ...
        dm.addDevice(d1);
        dm.addDevice(d2);
        // ...
}   }
```

Phase #3b: see it **still** GREEN!

```
./gradlew test
```

Outline

Summary

- Testing as a **process** for **quality** and **risk management**
- **Test automation** to effectively sustain software development
  - Strictly related with build automation and CI/CD
- **Test automation frameworks** for effective test automation
  - JUnit 5
  - xUnit.NET
- Basic notions and terminology about **automated (unit) testing**
  - Test suite, test cases, assertions, fixture, 3A, test lifecycle...
- **Coverage** as a key negative indicator
- **Test case design**
  - Oracle problem
  - **Black-box testing** (EP, BVA, decision tables)
  - **White-box testing** (CFGs, path testing, cyclomatic complexity)
- **Test-Driven Development (TDD)**: tests to guide design of software
  - Red - Green - Refactor

Bibliography (1/1)

[1] A. P. Mathur, *Foundations of Software Testing*, 1st. Addison-Wesley Professional, 2008, ISBN: 8131716600.

[2] C. Tudose, *JUnit in Action, Third Edition*. Manning Publications, 2020, ISBN: 9781617297045. [Online]. Available: https://books.google.it/books?id=JShVzQEACAAJ.

[3] G. Meszaros, *XUnit Test Patterns: Refactoring Test Code*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006, ISBN: 0131495054.

[4] K. Beck, *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[5] ——, *Test Driven Development. By Example (Addison-Wesley Signature)*. Addison-Wesley Longman, Amsterdam, 2002, ISBN: 0321146530.

[6] L. Crispin and J. Gregory, *Agile testing: A practical guide for testers and agile teams*. Pearson Education, 2009.

[7] W. D. Young, W. E. Boebert, and R. Y. Kain, "Proving a computer system secure," *Scientific Honeyweller*, vol. 6, no. 2, pp. 18–27, 1985, Reprinted in Tutorial: Computer and Network Security, M. D. Abrams and H. J. Podell, editors, IEEE Computer Society Press, 1987, pp. 142–157.

[8] R. Patton, *Software Testing*. Sams, 2006, ISBN: 9780672327988. [Online]. Available: https://books.google.it/books?id=MTEiAQAAIAAJ.

[9] A. Spillner, T. Linz, and H. Schaefer, *Software Testing Foundations: A Study Guide for the Certified Tester Exam : Foundation Level, ISTQB Compliant*. Rocky Nook, 2011. [Online]. Available: https://books.google.it/books?id=33dCnQAACAAJ.

Bibliography (2/1)

[10] T. Koomen and M. Pol, *Test Process Improvement: A Practical Step-by-step Guide to Structured Testing*, ser. ACM Press books series. Addison-Wesley, 1999, ISBN: 9780201596243. [Online]. Available: https://books.google.it/books?id=AHMxngEACAAJ.

[11] I. I. S. T. Q. Board), *Standard glossary of terms used in software testing*, E. van Veenendaal, Ed., https://glossary.istqb.org/, 2021.

[12] V. Khorikov, *Unit Testing: Principles, Practices and Patterns*. Manning Publications, 2020.

[13] L. Koskela, *Effective Unit Testing: A Guide for Java Developers*, ser. Running Series. Manning, 2013, ISBN: 9781935182573.

[14] T. J. McCabe, “A complexity measure,” *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.