Testing (Theory)

Deadline-Driven Development™

- Mar **16**: Object-Oriented Design Artifacts v1: High-Level Design using CRC Cards
- Mar **18**: Object-Oriented Design Artifacts v2
  - Fixed & Improved CRC Cards
  - Detailed Design (**First Version**), using UML
    - Structure, *e.g.* using UML Class Diagram
    - Behavior, *e.g.* using UML Sequence and/or State Machine/Activity Diagrams
- Mar **25**: Final Object-Oriented Design Artifacts (**both** CRC Cards and UML!)
- **Apr 08**: 1st Release
  - Local Maven or Gradle build. **Must** compile and run!
  - **Should** demonstrate at least one basic (happy path) User Story
  - **Unit Tests** would be good (and become **required** in 2nd Release)

Test Effectiveness in Agile (XP)

(McConnell, Code Complete 2ed., *Table 20-3*)

Practice	Min	Typical (Median)	Max
Architecture Review — Pair Programming	25%	35%	40%
Informal Code Review — Pair Programming	20%	25%	35%
Self Code Review	20%	40%	60%
Unit Testing	15%	30%	50%
Integration Testing	25%	35%	40%
Regression Testing	15%	25%	30%
Total	**≈74%**	**≈90%**	**≈97%**

Testing is Easy, Right? **(xUnit)**

```
public class HelloWorldTest {
    @Test
    public void print_hello_world() {
        var printer = new Printer(...);
        var helloWorld = new HelloWorld(printer);

        helloWorld.run();

        assertThat(printer.messages()).containsOnly("Hello, world!");
        assertThatIllegalStateException()
            .isThrownBy(helloWorld::run)
            .withMessage("Cannot run twice");

        printer.closeQuietly();
    }
}
```

Setup *System Under Test (SUT)*

Excercise

Assertion

Verify

Tear Down [optional]

Testing is Easy, Right? (xUnit)

```java
public class EnterpriseHelloWorldTest {
    private LogPrinter printer;

    @Before
    public void setUp() {
        var logPrinter = new LogPrinter(...);
        LogManager.register(logPrinter);
        this.printer = logPrinter;
    }

    @After
    public void tearDown() {
        if (logPrinter == null) return;
        LogManager.unregister(logPrinter);
        this.printer = null;
    }

    @Test
    public void print_hello_world_enterprise() {
        var helloWorld = new HelloWorld(...);

        helloWorld.run();

        assertThat(...);
        assertThatIllegalStateException()
            .isThrownBy(helloWorld::run)
            .withMessage("Cannot run twice");
    }
```

Testing Strategies

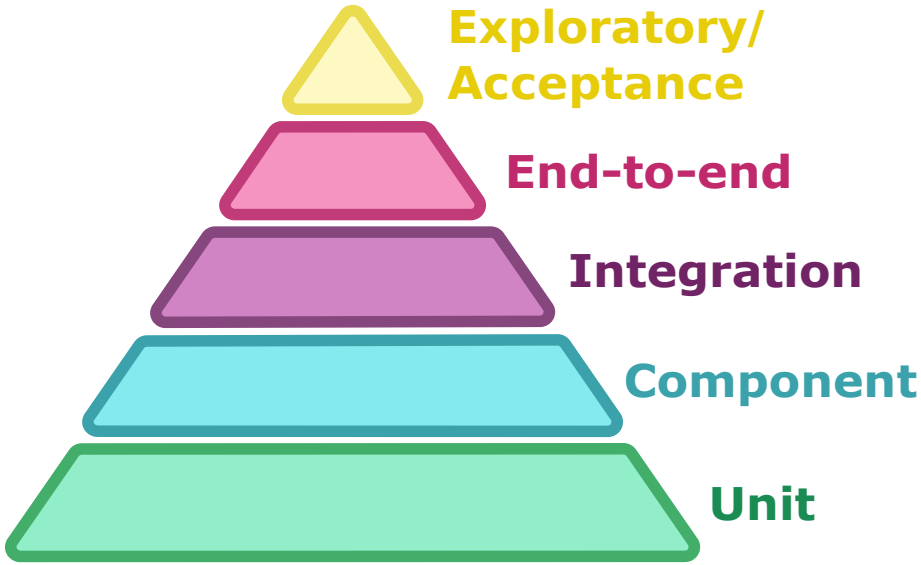
- White-box (Cover Structure and Algorithms)
- Black-box (Cover Functionality, *e.g.* Public API)
- But Encapsulation!
  - White-box goals (Coverage) but black-box methods (Public API)
- Gray-box: Overarching System Design, Structure and Algorithms are Partially Known. Test via Public APIs
  - *E.g.* Test backend via Web services and/or Web interface

Developer Intent

- Assume Clean: *It works on my machine*™
- **Assume Dirty:** If it exists, it has bugs
- Typical Ratio **5 : 1** → Mature **1 : 5**
  (McConnell Code Complete 2ed., 22.2)
  - ...no less Clean Tests. **25 times** more Dirty Tests

Test [Scenario] Pyramid

https://martinfowler.com/bliki/TestPyramid.html



- **Unit: https://martinfowler.com/bliki/UnitTest.html**
- Integration, *e.g.* with Real DB, HTTP Server, …
- **Component**
  - Test an (almost) self-contained, easily extractable part of the system
  - Can be executed in-process with *e.g.* in-memory DB
  - Esp. suitable for testing adapters/aggregators
- End-to-End (e2e)/System: Verify that System as a Whole Works
  - Have as few of them as possible
  - …while still covering as many essential User Stories/Journeys as possible
- Acceptance/Exploratory [not by devs, typically manual]

Specialized Test Strategies

- Contract Testing: Check that **Interfaces** match the spec
  - **Interface** is meant broadly: what each system exposes to the outside world, what inputs it expects, what outputs it produces, pre- and post-conditions, invariants
  - This is essentially an Integration Test with different focus
- Regression Testing: To prevent past problems from reoccurring
  - Any kind of test can be a regression test
- Fuzzing: Feed random values to the program and learn from it
  - Mostly for Security
- Mutation Testing: Randomly change (mutate) the program to see if test outcomes change
- Model Checking: Check concurrency invariants

Program Testing can be used to show the **presence** of bugs, but **never** to show their **absence**!

E.W. Dijkstra

(http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF)

Naive: Test All Values

- `parseFloat: String → float`. 32 Bits: 2^{32} Values
  - That's only 4 billion. We can test all of them:

```
for bits in 0..int.MAX_VALUE:
    f ← float.fromIntBits(bits)
    g ← float.parseFloat(f.toString())
    assert f == g
```

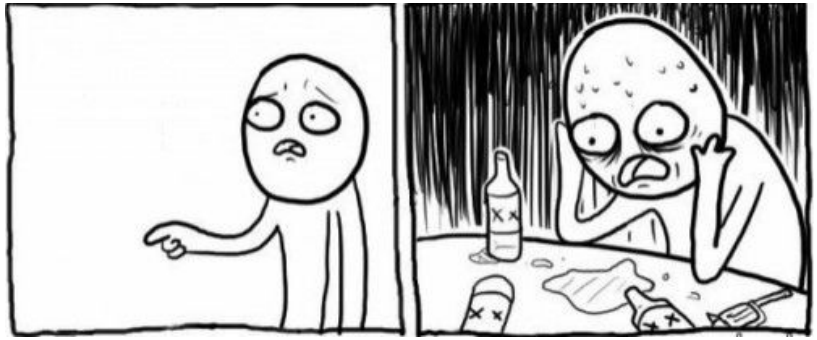
- `parseDouble`. 64 Bits: 2^{64} Values
  - Whoops!
  - But if we carefully generalize `parseFloat` to 64 bits...

Combinatorial Explosion

- `BigInteger`: Represents a Natural Number
  - Cannot test all values, value set is Infinite (but Countable)
- Finite Alphabet does not help!

  `name: String`, matches `[a-z]{1,16}` $\longrightarrow 2^{75}$

  `record` `Person(name: String, surname: String)` $\longrightarrow 2^{150}$
- Real-World:
  - UTF-8 text < 1000 chars $\rightarrow \left(2^{32}\right)^{1000}$
  - DEFLATE-compressed byte stream
  - …



Formal Approaches

- Flow Control (**if**, **for**, **while**/**do**, **try-catch**)
  - Try to walk every execution path
  - Verify with Code Coverage (JaCoCo, Cobertura, IntelliJ Code Coverage)
    - Class
    - Line
    - Statement Coverage
- Data Flow
  - Can detect suspicious data flows, *e.g.* assigning to the same variable over and over again

Practical Approaches

- Requirements: All User Stories/Use Cases. User Journeys
- Corner Cases, *e.g.* (max+1), (min−1), …
  - `Math.multiplyExact(int.MAX, int.MAX)`
  - `Arrays.copyOfRange(arr, 0, 7) // arr.length==6`
- Especially off-by-one errors
  - **Surprisingly common**!
  - Even considered the mark of an experienced developer

Practical Approaches *(Contd.)*

- Bad Data
  - No data (*e.g.* empty array, **`null`**)
  - Insufficient data (*e.g.* stream has only 6 bytes but we request to read 8)
  - Malformed data (*e.g.* missing file header, bad block size, …)
  - Incomplete data, *e.g.* forward reference to nowhere
  - Wrong data type

Practical Approaches *(Contd.)*

- Good Data
  - Expected
    - Average or median values
    - Reasonable input sizes
  - Boundary:
    - Maximum
    - Minimum

General Advice

- Test Systems/Components/Classes/Methods **one at a time**
  - **Much** easier to debug this way
- Design for (moderate) **Testability**
  - Use **Test Doubles** (Dummies, Stubs, Mocks, Fakes).
    Esp. in Unit and Component Tests.
    @*see* https://martinfowler.com/bliki/TestDouble.html
- Use simple values which are **easy to verify by hand**
- **Tests can have bugs, too!**

TDD

- **Test-Driven Development**

  Write a test →
  → See it fail →
  → Refactor or implement functionality →
  → See test pass

  (**Red-Green-Refactor**)

- **Leads to better APIs** because you have to think from the viewpoint of your Classes/Components' Users

BDD

- Behavior-Driven Development
- Business Analyst-friendly rebranding of xUnit
  - given — when — then
  - *vs* setup — excercise — verify
- Improves readability
  - Gherkin Language
  - Table approach to testing, *e.g.* **Spock**, FITnesse, ...

```
def "HashMap accepts null key"() {
  given:
  def map = new HashMap()
  when:
  map.put(null, "elem")
  then:
  notThrown(NullPointerException)
}
```