CSE 4283 / 6283
Software Testing and QA

Dr. Tanmay Bhowmik
tbhowmik@cse.msstate.edu

Special thanks to Dr. Nan Niu & Dr. Byron Williams

Agenda

Last Class:
Testability

This Class:
Unit Testing

Next Class:
Boundary Testing

Defects / Bugs (recap)

- **Error**: A mistake made by a programmer or software engineer which caused the fault, which in turn may cause a failure
  - conceptual mistakes
  - human misunderstanding
  - e.g., when the class is full, student can still enroll if the instructor permits

Picture source: Internet

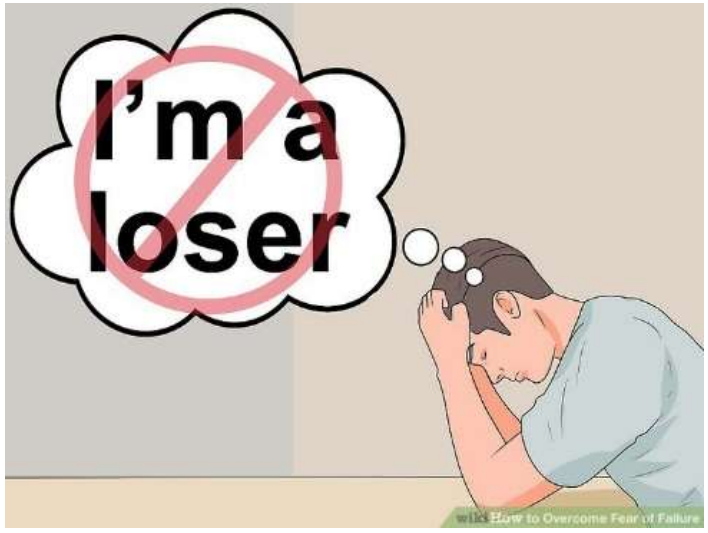
Defects / Bugs (recap)

- **Fault**: Condition / internal characteristic that *may* cause a failure in the system
  - a mistake written down in code and/or document
  - e.g., if(current_enrol=max_enrol) {//cannot enroll any more}
  - SHOULD BE if(current_enrol==max_enrol) {//cannot enroll any more}

Picture source: Internet

Defects / Bugs (recap)

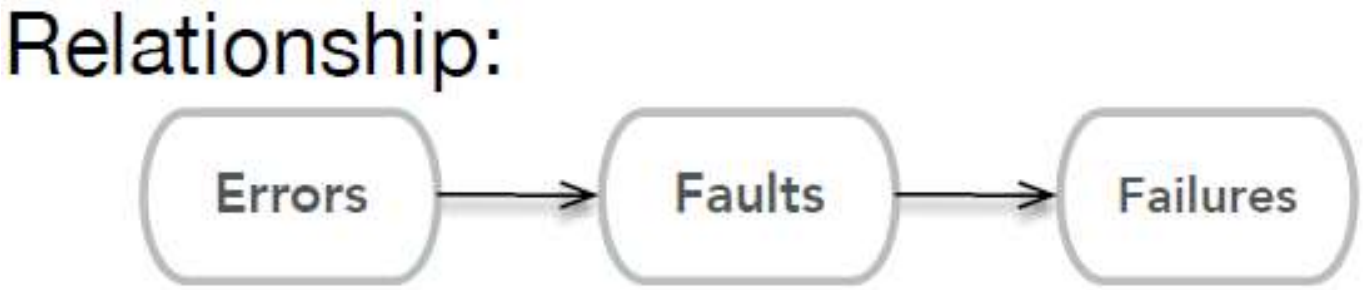
- **Failure**: Inability of system to perform a function according to its specification due to some fault
  - deviation from expected behavior
  - something goes wrong at execution
  - e.g., student cannot enroll in a course even if nobody is currently enrolled



Picture source: Internet

Defects / Bugs (recap)

- **Bug**: An abstract way of describing the above - problematic terms; avoid



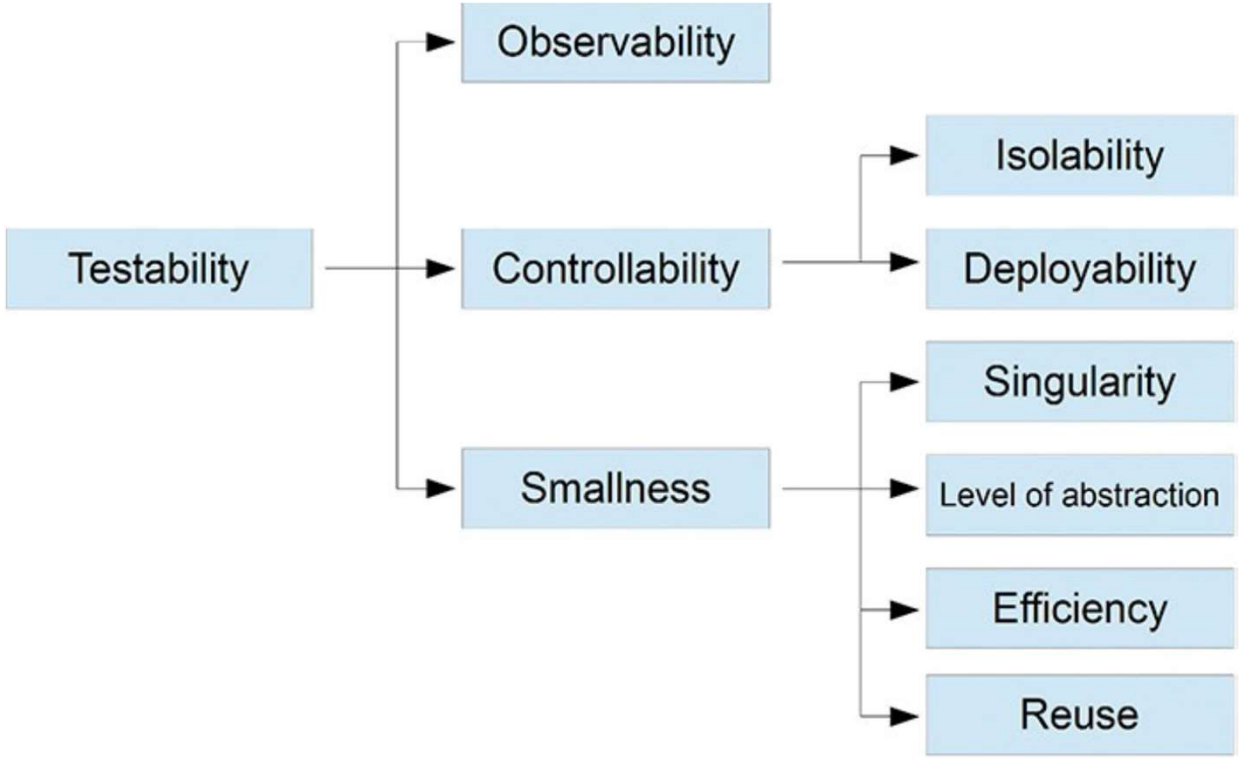
Picture source: Internet

Defects (recap)

Error, Fault, or Failure in the system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways

(think specifications / stories / expectations)

Testability Quality Decomposed (recap)

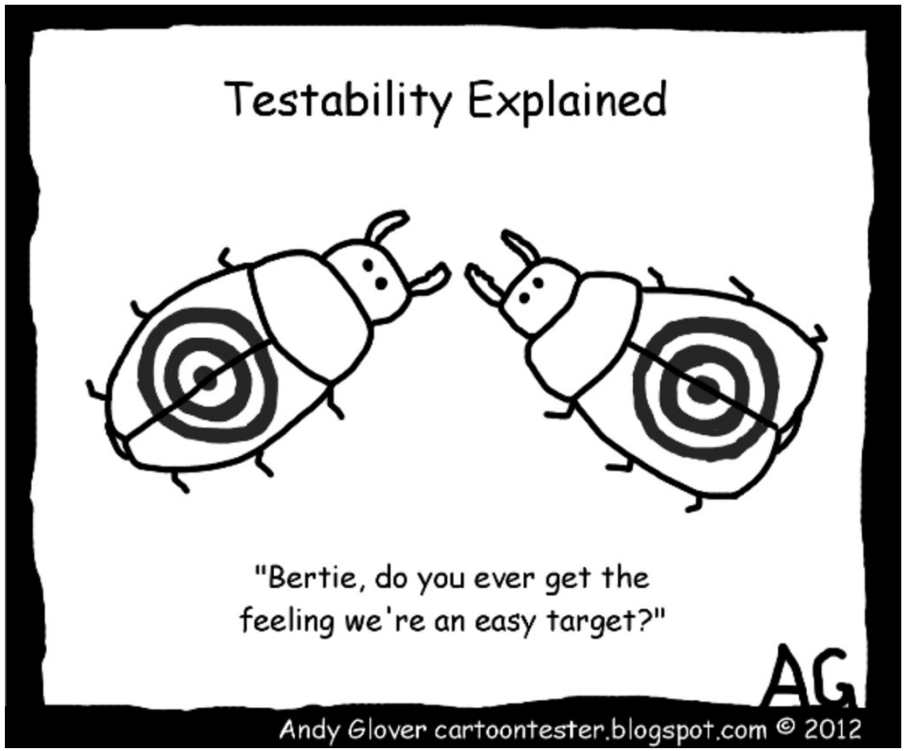


- **NOTE:** When a ***program element*** is testable, it means that it can be put in a **known state**, **acted on,** and then **observed**.
- Further, it means that this can be done **without affecting any other program elements** and **without them interfering**

Picture source: Internet

Code-level Testability

- Some constructs and behaviors in code have great impact on its testability



Picture source: Internet

Direct Input / Output

- Program element's behavior affected solely by values that have been passed in via its public interface — ***direct input***
- Reliance on only direct input is quite a desirable property
  - largest concern is to find relevant inputs to pass as arguments to the tested method
  - not caring about other actors or circumstances that may affect its behavior
- ***direct output*** - return value - observable through public interface

Indirect Input / Output

- Input is considered ***indirect*** if it isn't supplied using the program element's public interface
  - "Would I be able to test this without having access to the source code?" … "no," → indirect input
  - e.g., static variables/methods, system properties, files, databases, queues, etc.

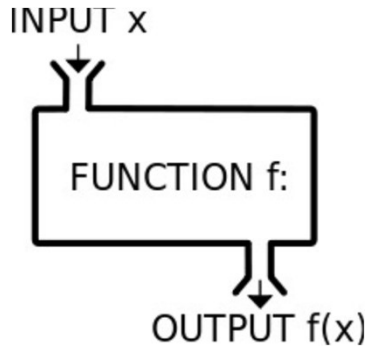
- ***indirect output*** - no return value or return plus other output - not observable through public interface
  - reliance on system for output verification

Direct Input / Output

1. **It's consistent**—Given the same set of input data, it always returns the same output value, which doesn't depend on any hidden information, state, or external input.
2. **It has no side effects**—The function doesn't change any variables or data of any type outside of the function. This includes output to I/O devices.

Indirect Input / Output

- Changing the value of a variable outside the scope of the function
- Modifying data referenced by a parameter (call by reference)
- Throwing an exception
- Doing some I/O



Picture source: Internet

State

```
public void dispatchInvoice(Invoice invoice) {
    TransactionId transactionId = transactionIdGenerator.generateId();
    invoice.setTransactionId(transactionId);
    invoiceRepository.save(invoice);
    invoiceQueue.enqueue(invoice);
    processedInvoices++;
}
```

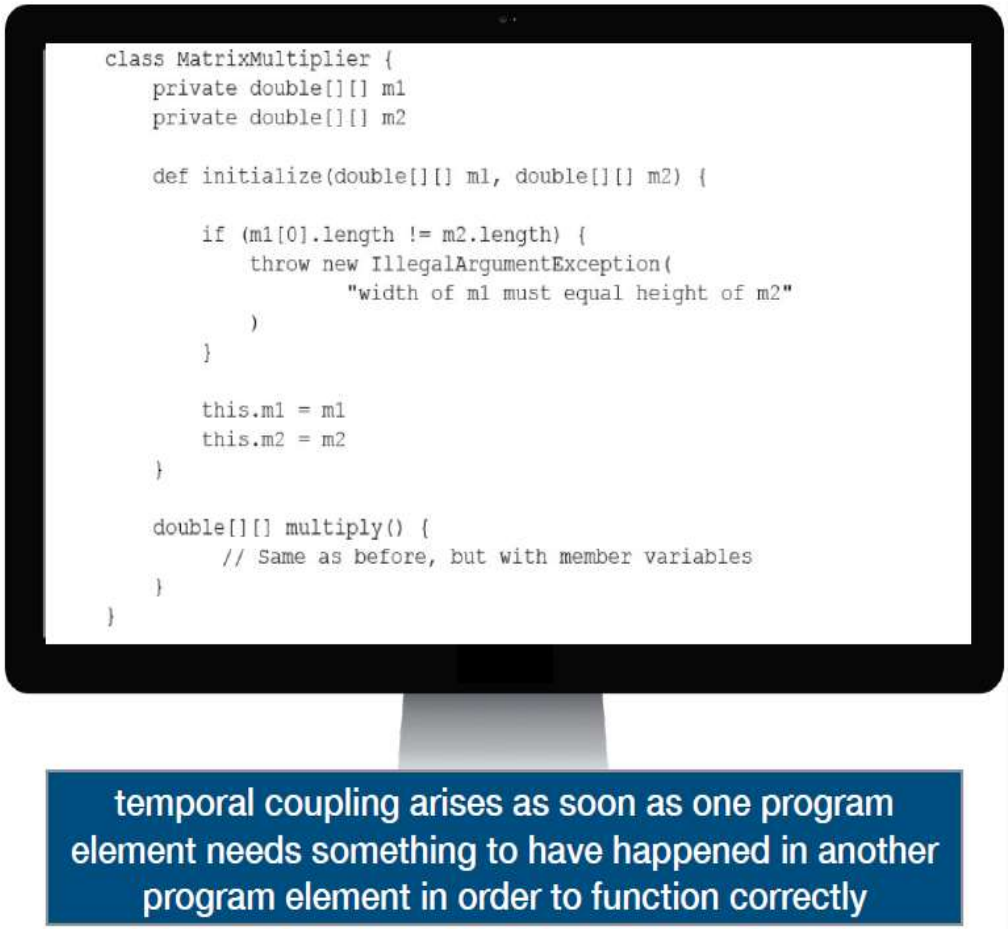
a program is described as **stateful** if it is designed to remember preceding events or user interactions; the remembered information is called the **state** of the system

```
if (++processedInvoices == BATCH_SIZE) {
    invoiceRepository.archiveOldInvoices();
    invoiceQueue.ensureEmptied();
}
```

"How do I set up a test so that I reach the correct state prior to verifying the expected behavior?"

Temporal Coupling

- Temporal coupling is a close cousin of state
- The order of invocation
- Given a program element with functions f1 and f2, there exists a temporal coupling between them if, when f2 is called, it expects that f1 has been called first—that is, it relies on state set up by f1



Picture source: Internet

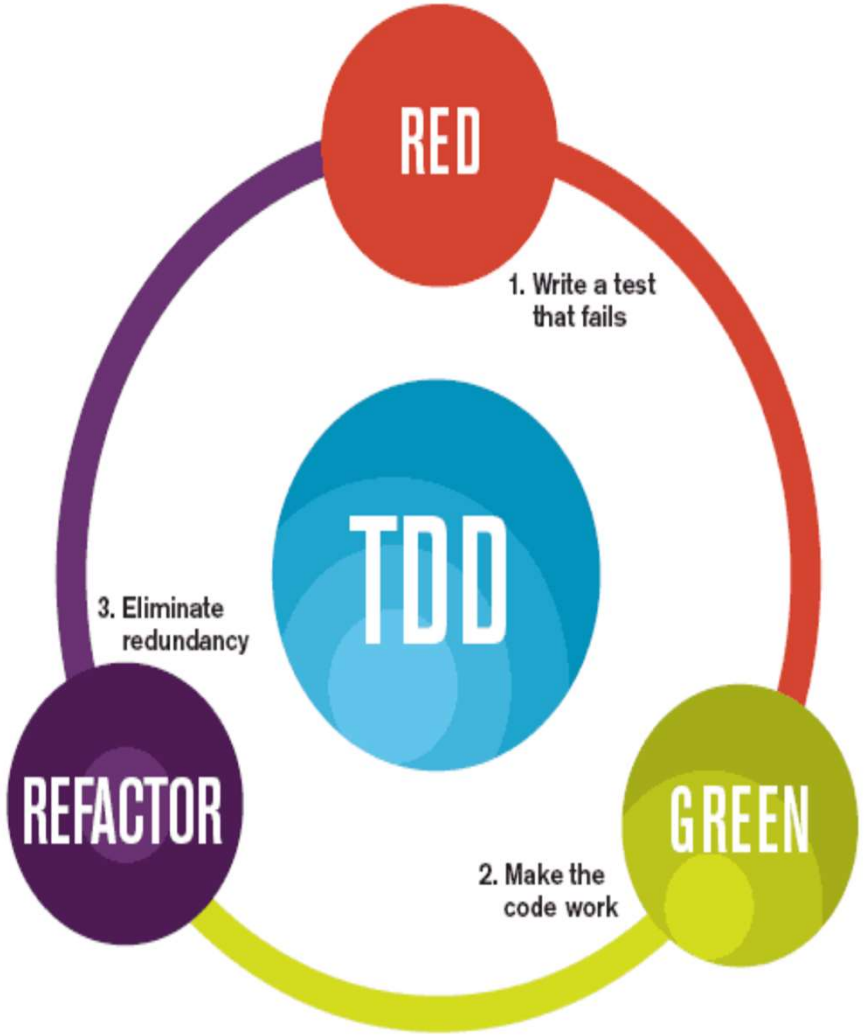
Unit Testing & TDD

What is TDD?

- A software development **technique** that uses **short development iterations**
- Based on pre-written test cases that define desired improvements or new functions
- Each iteration **produces code necessary to pass that iteration's tests**
- Then the programmer or team **refactors** the code to accommodate changes
- Preparing tests before coding **facilitates rapid feedback** changes
  - TDD **is a software design method**, not just a method of testing

Test-Driven Development (TDD)

- A software development process where a unit's ***tests are written before*** the unit's implementation and guide the unit's development as the tests are ***executed repeatedly*** until they all succeed, signaling complete functionality.
- The TDD process steps are commonly shortened to "Red, Green, Refactor"
- Each time a new function, feature, object, class, or other software unit will be developed.



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

Refactoring - process of restructuring existing computer code without changing its external behavior - refactoring improves nonfunctional attributes of the software

Picture source: Internet

Test-Driven Development (TDD)

1. **Red**
   - Write a new test for a section of code (the “unit”)
   - Verify failure of the new test and the success of existing tests. If the new test passes, verify that it is not redundant, then start composing the next test (step 1a).
2. **Green**
   - Write some code to implement, modify, or develop the unit
   - Repeat until the tests pass. If one or more tests fail, continue coding until all tests pass. If the tests pass, the developer can be confident that the new/modified code works as specified in the test. Stop coding immediately once all tests pass.
3. **Refactor**
   - Refactor the code to improve non-functional code structure, style, and quality
   - Confirm tests pass. If one or more tests fail, the refactor caused problems; edit until all tests pass. If the tests all pass, the developer can be confident that the refactor did not effect any tested functionality.
4. **[Repeat]**

TDD — Design Methodology

- Test-Driven Development (or test driven design) is a methodology

- Common points about TDD:
  - TDD is not (just) about testing
  - TDD is about design and development
  - By testing first you design your code

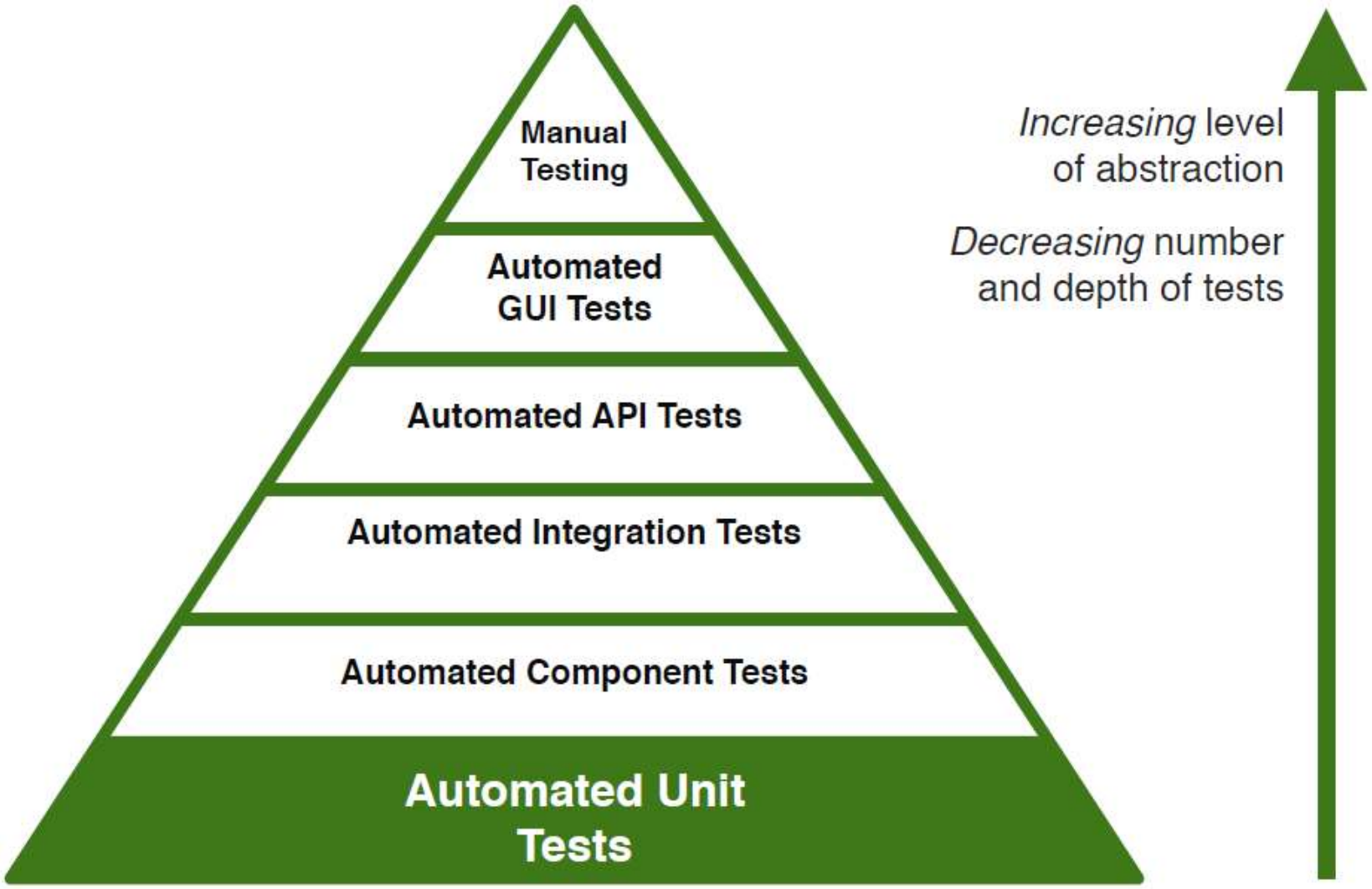
Unit Testing

- A test that invokes a **small**, **testable unit** of work in a software system and then **checks a single assumption** about the resulting output or behavior
- Key concept: **Isolation** from other software components or units of code
- Low-level and focused on a tiny part or “unit” of a software system
- Usually written by the **programmers themselves** using common tools
- Can be written to be **fast** and run along with other unit tests in **automation**

Unit Testing

- Form of **white-box testing** that focuses on the implementation details
- Typically uses **coverage criteria** as the exit criteria
- Definition of a "unit" is sometimes ambiguous
  - Commonly considered to be the **"smallest testable unit"** of a system
  - OOP languages: treat each **object** as a unit
  - Functional or procedural languages: treat each **function** as a unit
  - Many testing frameworks allow sets of unit tests to be **grouped**
    - allows tests at the function level and grouped by their parent object

Software Testing “Pyramid”



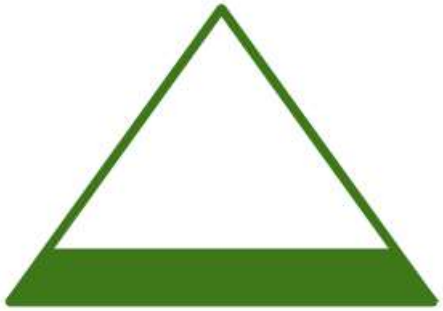
Picture source: Internet

Unit Testing vs TDD

- Unit testing and TDD are distinct concepts
- While closely related and often used together, they could be used separately
- The following slides and demos will present the two concepts combined, as they are frequently used together

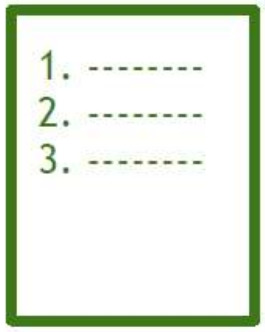
Unit Testing

Level of testing



Test-Driven Development

Development *process*



The Problem

```
def add_one(x):
    return x + 1
```

How can we test this one function before it is used elsewhere in a program?

What if it was more complex?

What if it was in extremely large system?

What if we wanted to test it automatically so when it's modified, we can easily make sure it still works?

Answer: "Unit Testing"

Source Code

Unit Tests

```
import pytest

def test_example_1():
    assert add_one(3) == 4

def test_example_2():
    assert add_one(-3) == -2

def test_example_3():
    assert add_one(5) == 20
```

Will Pass

A single "assertion"

Will Pass

Will Fail
(Need to fix this test…)

Example: Unit Tests (JavaScript + Jasmine)

```
function isPositive(x) {
    return x >= 1;
}
```

The unit tests revealed one or more defects in this code, hopefully before other functions relied on it

Source Code

Results	Unit Tests
✓	expect(isPositive(5)).toEqual(true);
✓	expect(isPositive(-5)).toEqual(false);
✓	expect(isPositive(1)).toEqual(true);
✓	expect(isPositive(-1)).toEqual(false);
✗	expect(isPositive(0.5)).toEqual(true);
✓	expect(isPositive(-0.5)).toEqual(false);
✗	expect(isPositive(0)).toEqual(true);

Unit Testing Tools

- Test Framework
  - Defines the test writing syntax
  - Likely language-specific because it hooks into the system's execution
  - *Examples*: Jasmine, Mocha, Jest (for JavaScript); PyTest (for Python); JUnit (for Java)
- Test Runner
  - Executes all (or a specific subset) of the system's unit tests and presents, displays, or otherwise outputs the results
    - a local test runner on a developer's computer or
    - run on a server (e.g. a Continuous Integration (CI) server)
  - Might also spin up mocks, a virtual environment, or any other resources the tests require
  - Often a basic test runner is built into the test framework
    - likely run via the command line
  - Example: Karma (for web application testing)

More Tools

- Mocks
  - Provides a “mock” or **simulated implementation** of each external dependency or resource required by the methods being tested - aka. **stub**
  - May return random, dummy, or cached data
  - The need for mocks and their implementations varies between systems
  - *Example*: EasyMock
    - provides dynamically generated Mock objects (at runtime), without having to implement them

Picture source: Internet

More Tools

- **Coverage Reporter**
  - Determines and provides a report on the test coverage metrics of a set of code
  - May generate **metrics such as statement, branch, function, executions per line, and line coverage**
    - grouped by file, class, component, or for the entire system
  - Might run independently or during each test executed by a test runner
  - *Example*: Istanbul (for JavaScript), Coverage.py (python) → Tools like coveralls (https://coveralls.io/)

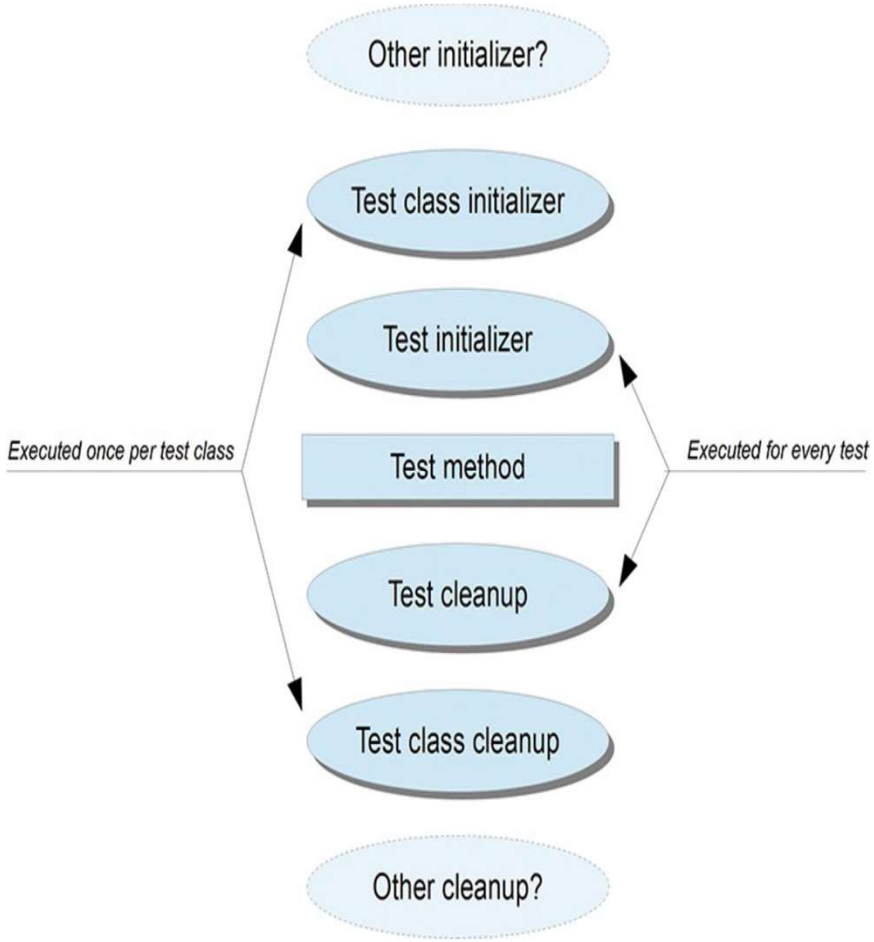
Picture source: Internet

What is JUnit?

- Open source Java testing framework used to write and run repeatable automated tests (junit.org)
- A structure for writing test drivers adopted by many ***"xUnit"*** testing frameworks (.NET framework)
- JUnit features include:
  - **Assertions** for testing expected results
  - **Test features** for sharing common test data
  - **Test suites** for easily organizing and running tests
  - Graphical and textual **test runners**
- JUnit can be used as stand alone Java programs (from the command line) or within an IDE

xUnit Tests

- xUnit can be used to test …
  - … an entire object or part of an object, i.e., a method or some interacting methods
  - … interaction between several objects
- It is primarily intended for unit and integration testing, not system testing
- Each test is embedded into one test method
- A test class contains one or more test methods
- Test classes include :
  - A collection of test methods
  - Methods to set up the state before and update the state after each test and before and after all tests



JUnit Assertions

- `void` **`assertEquals`**`(boolean expected, boolean actual)` - Checks that two primitives/objects are equal
- `void` **`assertTrue`**`(boolean expected, boolean actual)` - Checks that a condition is true
- `void` **`assertFalse`**`(boolean condition)` - Checks that a condition is false
- `void` **`assertNotNull`**`(Object object)` - Checks that an object isn't null
- `void` **`assertNull`**`(Object object)` - Checks that an object is null
- `void` **`assertSame`**`(boolean condition)` - The assertSame() method tests if two object references point to the same object
- `void` **`assertNotSame`**`(boolean condition)` - The assertNotSame() method tests if two object references do not point to the same object
- `void` **`assertArrayEquals`**`(expectedArray, resultArray)` - The assertArrayEquals() method will test whether two arrays are equal to each other

Summary

- Unit Testing
  - Code-level testability
  - Temporal coupling
  - TDD
  - Unit Testing
  - Unit Testing tools

- Next topic
  - Boundary Testing

THANK YOU