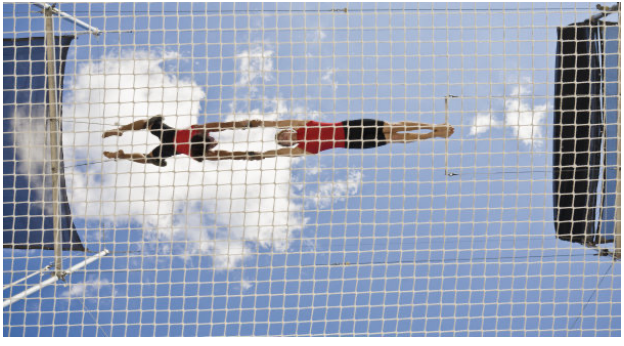
Testing Basics



Agenda

- Whose responsibility is testing?
- What is testing?
- What testing is not
- Why should we test?
- What should we test?
- When to test?
- Testing terminology
- Test workflow
- Test automation
- Test strategies

Whose responsibility is testing?

- ❑ Software engineer/developer
- ❑ Test engineer
- ❑ Quality assurance team/specialist
- ❑ Architect
- ❑ Designer
- ❑ Requirements/business analyst
- ❑ Customer
- ❑ End user

Whose responsibility is testing?

Testing is a whole-team responsibility

Developers
unit, integration, system, acceptance

Designers
integration

Customers
acceptance

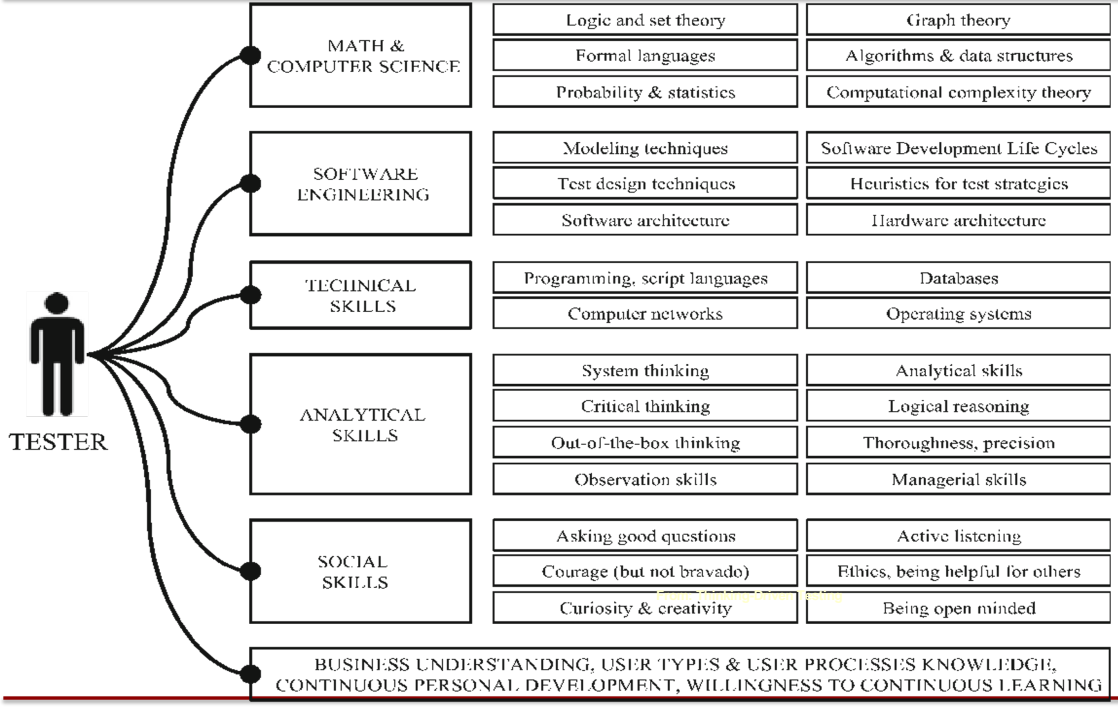
Analysts
acceptance

Test engineers & QA team
system, acceptance, exploratory

End Users
beta

Architects
integration, system

Testing is easy, right?



Skills needed for testing

<= Domain knowledge

Courtesy of Jeff Gennari: adapted from “Thinking-Driven Testing,” Roman, 2017

What is Testing?

Category of software V&V techniques that rely on execution of code to reveal faults and errors

- Executes code ➔ dynamic
- Reveals faults ➔ but only if faulty behavior gets executed

What is the biggest disadvantage of testing?

Testing cannot reveal all faults…

- Not an exhaustive technique that can prove software is fault-free

Testing shows the presence, not the absence of bugs.

Dijkstra (1969)

It can produce false negatives!

- Not a substitute for all types of quality assurance

Testing attempts to cause failures & errors… to reveal faults

…programming mistake
…wrong assumptions
… incompatibility
… poor design

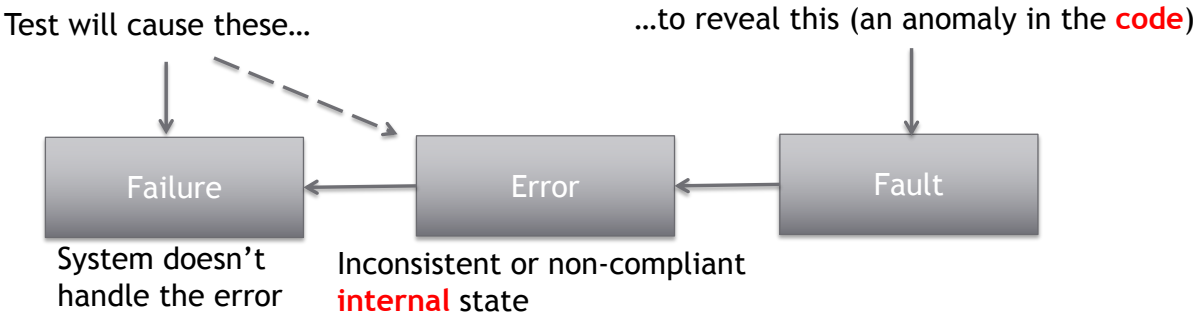
Observed:
Unresponsive server

Not directly observed:
Repeated client timeouts re-issuing requests

Caused by:
Poor design decision

Tests will catch these

Failure is any externally observable inability of a system to perform as expected, to the point of becoming useless, or dangerous, or confusing*

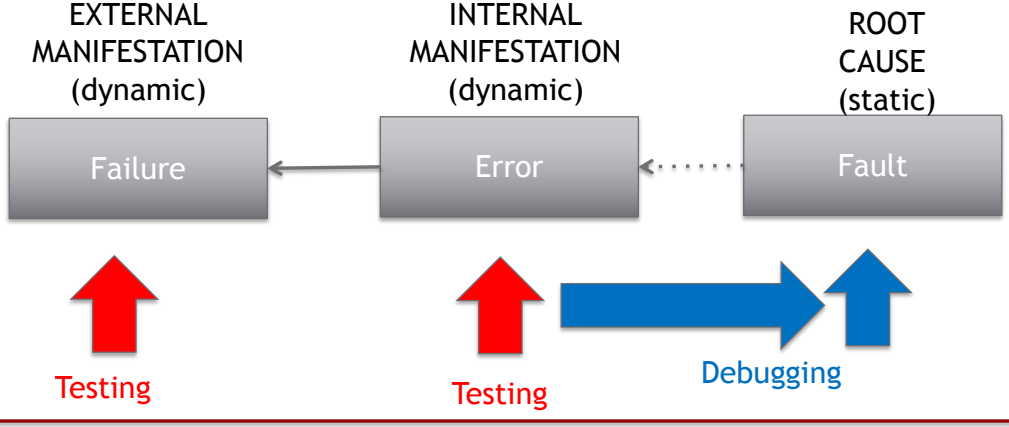


*Ammann & Offutt 2008

Debugging is sometimes confused with testing

Testing ≠ Debugging

Hopefully, testing will give enough diagnostic info to easily identify the fault and fix it (feedback principle)



Tests and faults

GPS Example

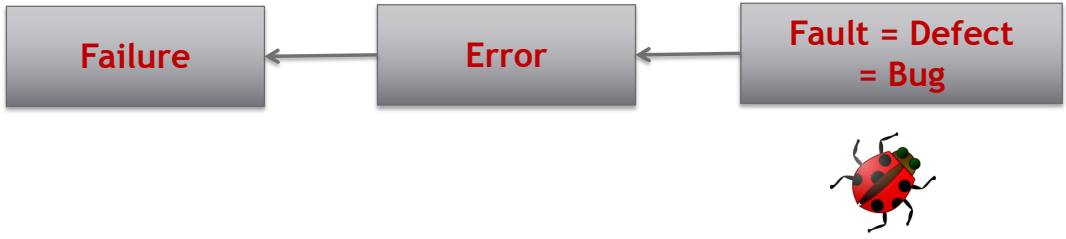
The user is confused.

Estimated travel distance between A and B is shorter than the shortest theoretical distance

Linear distance between two points is calculated using the wrong function:

$d := \text{sqrt}(|x_2 - x_1|^2 + |y_2 - x_1|^2)$

Terminology



Virtues (benefits and secondary benefits) of testing?

Testing has many virtues even though it cannot find all faults

- ✓ reduces the number and probability of faults
- ✓ increases confidence in software quality

But there is more to virtues of testing

- ✓ reduces the number and probability of faults
- ✓ increases confidence in software quality
- ✓ scopes work (visibility/repeatability principles)
  "I'm done when these tests pass"
- ✓ makes work tangible (visibility/repeatability principles)
  "I can show this functionality is working: all tests pass"
- ✓ gives feedback, helps with diagnosis (feedback principle)
  "I can figure out where the bug is: test fails with this message"
- ✓ helps us understand behavior (visibility principle) **documentation**
  "By looking at these tests, I can see what this method is doing"

What Should We Test?

Most parts of new software at different granularity levels (redundancy principle)

Third-party software and software that are unfamiliar to **understand its behavior**

We test all things at all levels

- Individual units

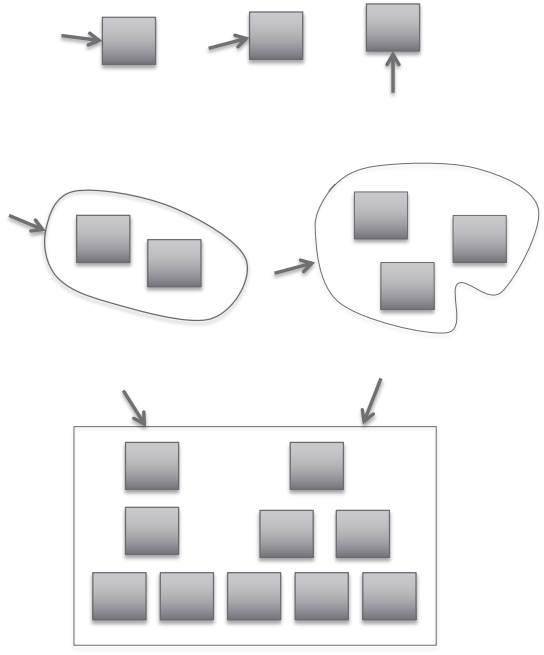
  Unit Testing

- Groups of related units

  Integration Testing
  ~~Component Testing~~
  ~~Functional Testing~~

- System as whole

  System Testing
  Acceptance Testing



Unit: smallest fragment of code abstraction that can be assigned to a single developer

Which aspects to test

- Functionality (correctness)
  - Flows, input-output relationships, and behaviors
- Quality attributes
  - Performance, Reliability, Security, Scalability, Usability, …
- Change/Regression
  - Re-testing of something that was already working, usually with existing test cases, to provide confidence that the system still functions correctly following modification or extension

When to test?

- Just before product is released?

OR

- Throughout the software lifecycle?

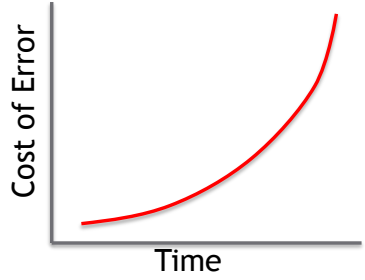
When to test?

- Just before product is released? ✗

OR

- Throughout the software lifecycle? ✓

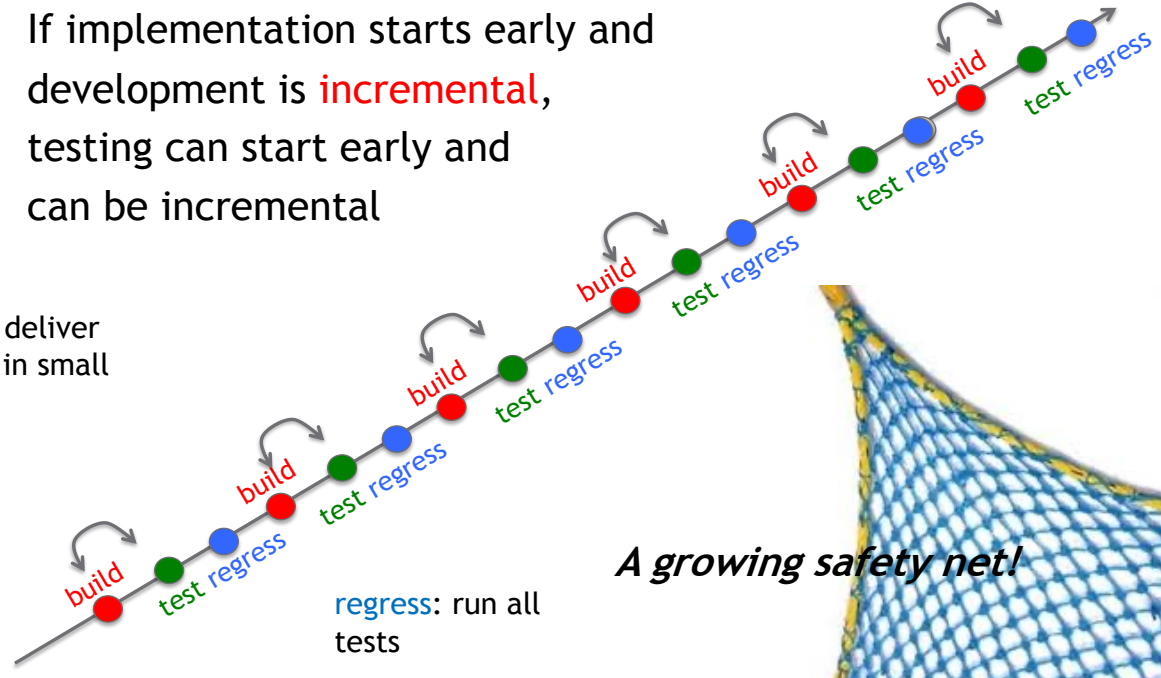
Test early and test often!



Test early, test often: In-process testing / Continuous testing

If implementation starts early and development is incremental, testing can start early and can be incremental

incremental: deliver functionality in small chunks



regress: run all tests

A growing safety net!

How to test software

- Identify part of software whose behavior you wish to verify
- Figure out how to access that part
- Determine the appropriate triggers — Inputs or Stimuli
- Determine what the expected behavior is — Part of Oracle
- Exercise the software with chosen triggers and observe the resulting behavior — Results or Outputs
- Compare observed behavior with expected behavior — Assessment
- Pass or fail the test depending on the comparison — Verdict
  + give info on possible cause

Test Case

Test Structure and Workflow

Test case…

…is a schema that has a specific
diagnostic purpose (focusing on a possible failure)

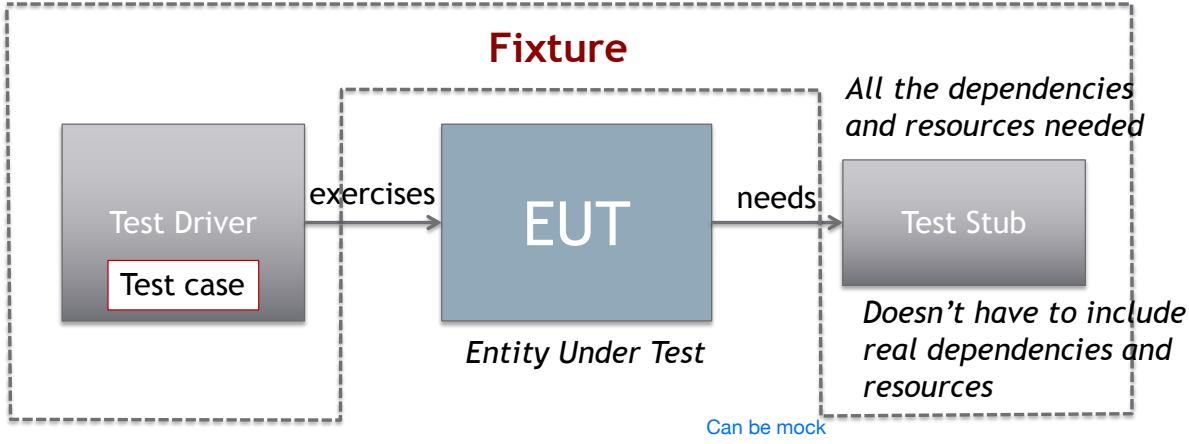
Test case ≈ Test

implemented & executed as test code

```
@Test public void testFifteen() {
       p.addPoint();
       assertEquals("Fifteen", p.getScore());
}
```

// Do something;
// Check condition;

Test cases need extra plumbing

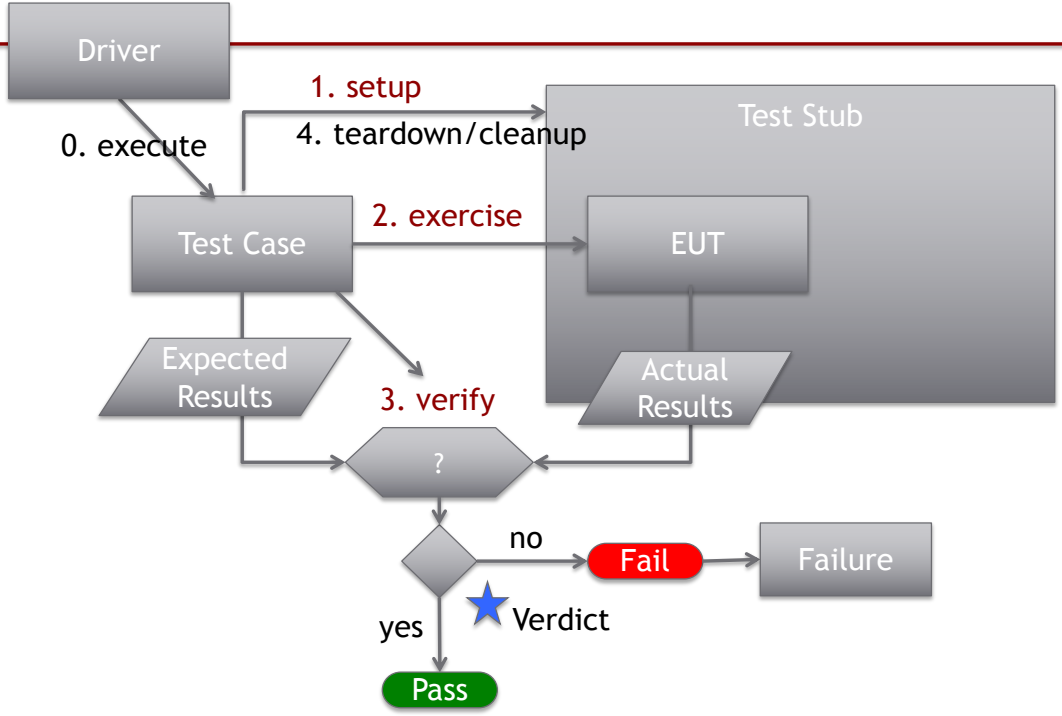


Test Stub: EUT dependencies and resources that are simulated/included in the fixture
(things that are not the focus of testing but need to be present for the exercised behavior in EUT to work)

In unit testing, we generalize the concept of a simulated "Stub" as a "Test Double"

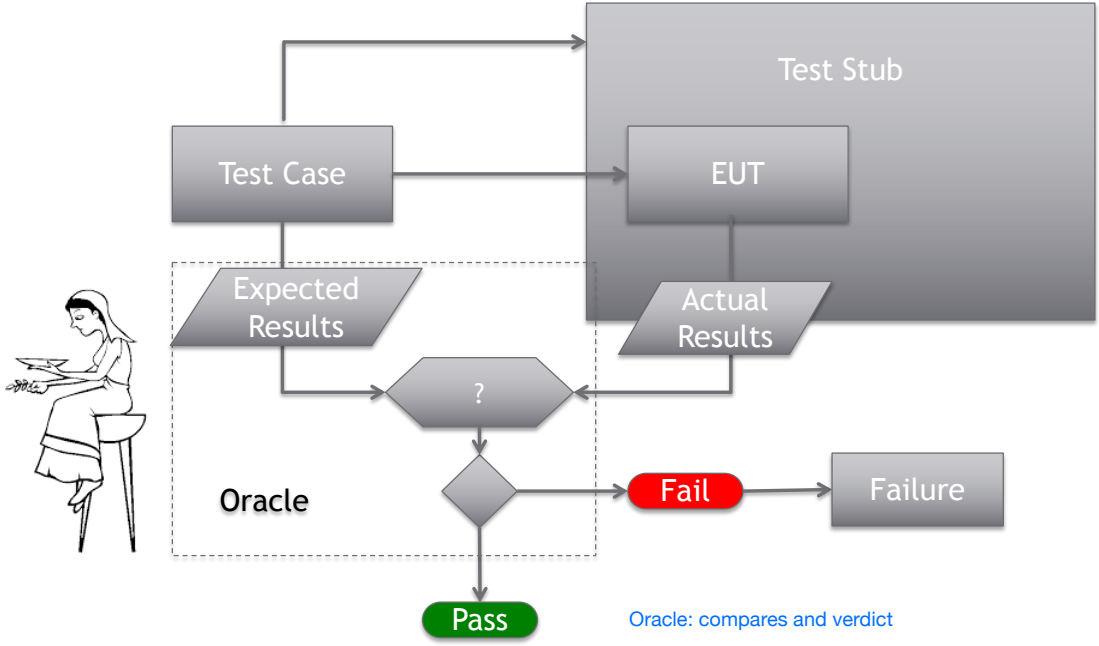
Workflow: Arrange, Act, Assert

Life of a test case



BRIEFLY: **ARRANGE → ACT → ASSERT (→ TEAR-DOWN / CLEANUP)**

Oracle is the procedure for deciding whether a test passes or fails



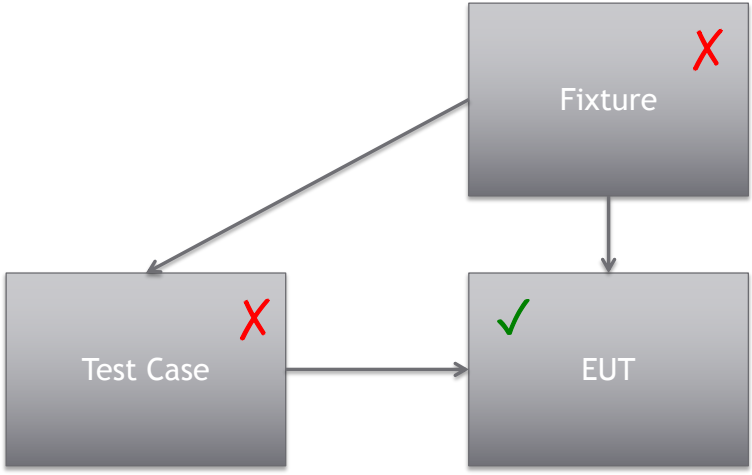
Alternative causes of test case failure

Suppose we are certain the EUT implements the behavior we are testing correctly, but the associated test case is failing.

What could possibly cause this?

Alternative causes of test case failure

Could the test case or fixture be wrong?



The Oracle

- Oracle: the mechanism that determines whether a test (test case) passes
  - Includes expected results if applicable
  - A comparison of expected and observed results
  - An assertion that verifies whether a condition holds
  - A human verdict

  or
  - Simply lack of a failure (successful execution with no crashes)

The Oracle Problem

… is determining what the oracle should be

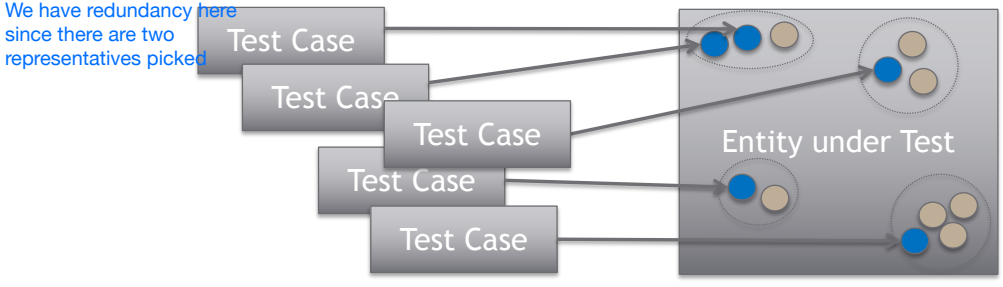
Often reduces to determining what the expected results are for a given set of inputs or initial states

Not always easy!

How much testing to do?
(and how do we know we are done?)

We test EUT by…

writing a representative test case for every “behavior of interest”

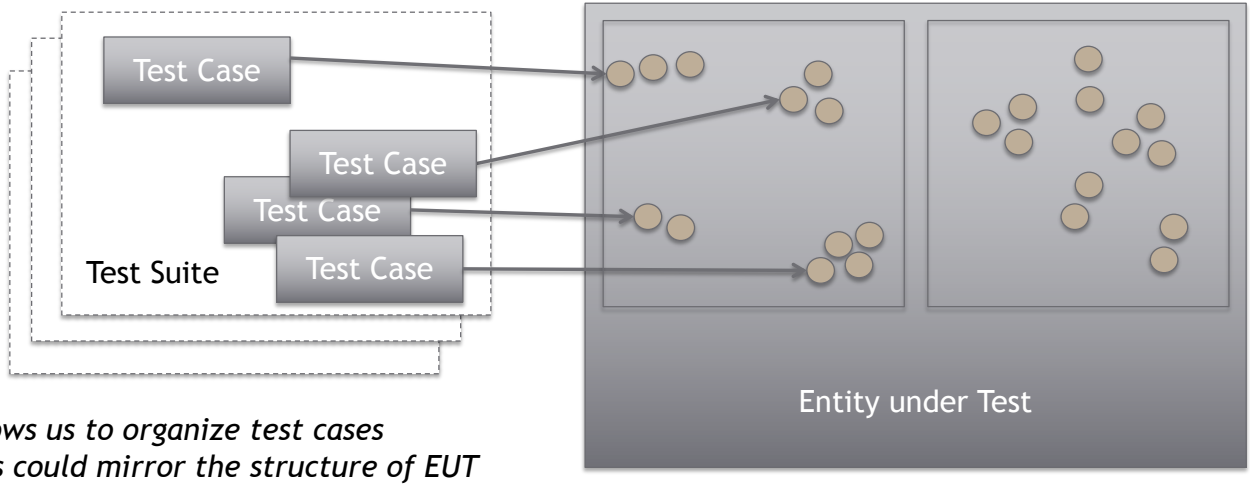


related behaviors form clusters: exercise the same logic

Organizing tests

Collection of related test cases: test suite

Basically one test suite for each production class

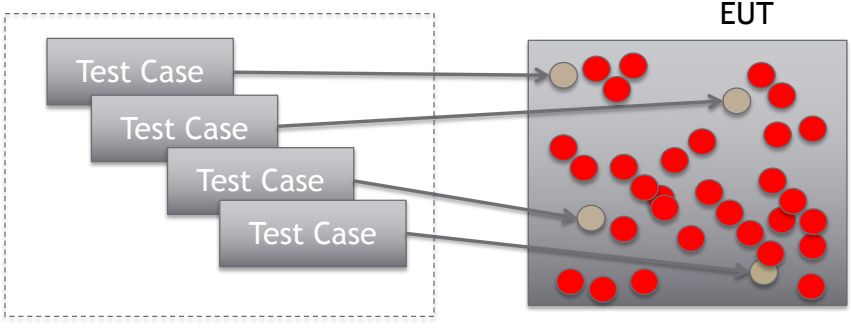


- *Allows us to organize test cases*
- *This could mirror the structure of EUT*

Can’t cover the whole universe

But, we cannot cover all possible behaviors of EUT

Even a small program may have infinite number of possible behaviors!



When can we stop testing?

- Five criteria from Ryber
  - We have achieved the coverage aims we defined in the strategy (adequacy criteria)
  - The number defects discovered is lower than the target value we had established based on experience
  - The cost of detecting more defects is larger than the estimated loss arising from remaining defects. May not be worthwhile
  - The project team draws the collective conclusion that the product is ready to be released
  - The decision maker gives the order to go to production

Courtesy of Jeff Gennari "Essential Software Test Design, Ryber, 2007

We want to automate tests as much as possible to make testing…

- repeatable
- efficient
- scalable
- ***practical***

Automated tests facilitate future changes (via regression)

How to automate tests

Harness the power of numerous testing frameworks to automate your test cases

A testing framework for every type of testing and every programming environment imaginable!

Can all kinds of testing be completely automated?

Can all types of testing be automated?

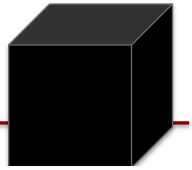
How easy is it to automate the testing of a utility class?

How easy is it to automate the usability testing of a whole application?

How easy is it to automate testing of embedded software?

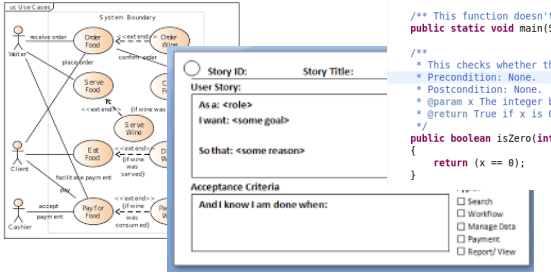
Testing approaches (wrt intrusiveness)

Black-box testing (spec-based)



Relies only on specification of EUT's external behavior

- Requirements specifications
  - Use case (elaborated)
  - User stories (acceptance criteria)
  - Formal specifications
- Interface specifications
  - API documentation
- Quality attribute specifications
- Inferred informal requirements: textual or oral or implied




```
public class HelloWorld {

    /** This function doesn't do anything yet.
    public static void main(String[] args) {

    /**
     * This checks whether the passed integer is zero.
     * Precondition: None.
     * Postcondition: None.
     * @param x The integer being checked.
     * @return True if x is 0, false otherwise.
     */
    public boolean isZero(int x)
    {
        return (x == 0);
    }
```

Use Case: Specialty_Unit

Basic Flow → The Use Case starts when a user, Citizen or Attendant, desires to register information or get information.

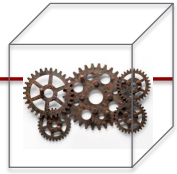
1. User gets in the system.
2. User indicates his or her operation.
3. Use Case *finishes*.

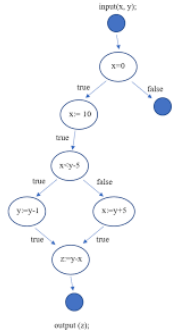
Extension Points:

E1. User Gets in. This extension point occurs on the step 1.
E2. Attendant Chooses Unit. This extension point occurs on the step 2.
E3. Attendant Chooses Specialty. This extension point occurs on the step 2.
E4. Citizen Choose Specialty_Query
E5. Citizen Choose Unit_Query. Th

- Treat the EUT as a black-box
- Exercise the EUT using its outward-facing interface and check the results using the oracles derived from the underlying specifications

White-box testing (structural)





Relies on knowledge of EUT's implementation:

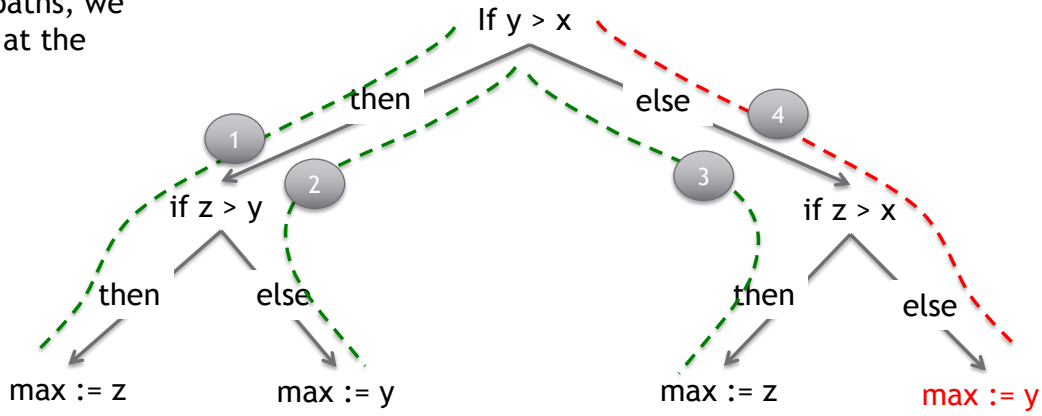
- Code itself
- Bytecode
- Another abstraction (model) of implementation

- Peek inside the EUT implementation/model you want to test
- Analyze it to figure out how you'll exercise the implemented/modeled behaviors
- Reverse-engineer the appropriate inputs that will trigger the execution of those behaviors (*generating test cases*)
- Add oracles, exercise the EUT using the selected inputs

White-box testing requires us to examine the EUT's internals

To cover all paths, we need to look at the code!

Find the max of 3 integers…



Find input triplets (x, y, z) to cover all paths

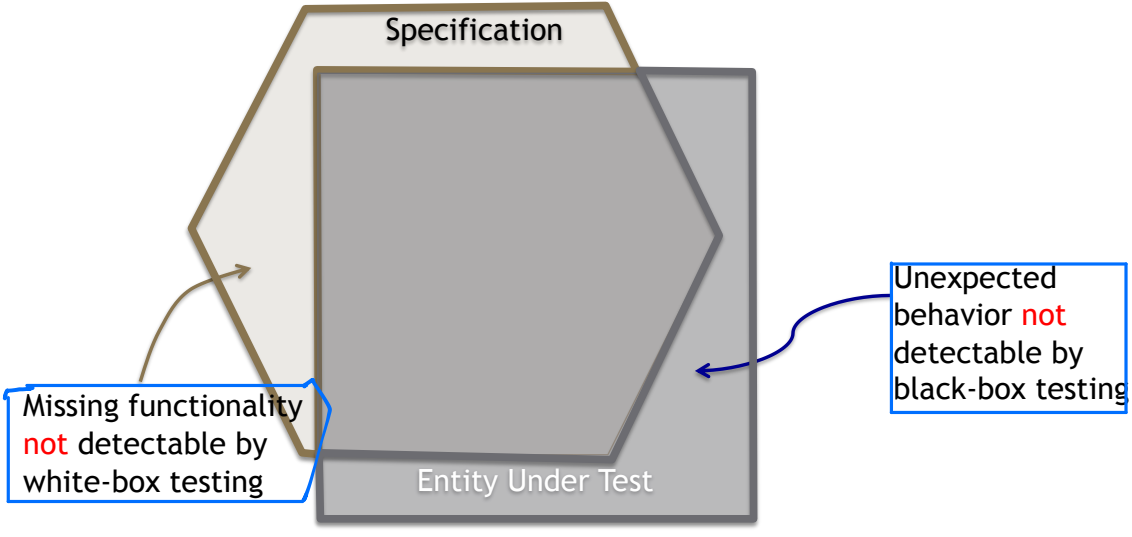
(3, 4, 5), (3, 5, 4), (4, 3, 5), (3, 3, 5), (5, 3, 4), (5, 5, 3)

1 2 3 3 4 4

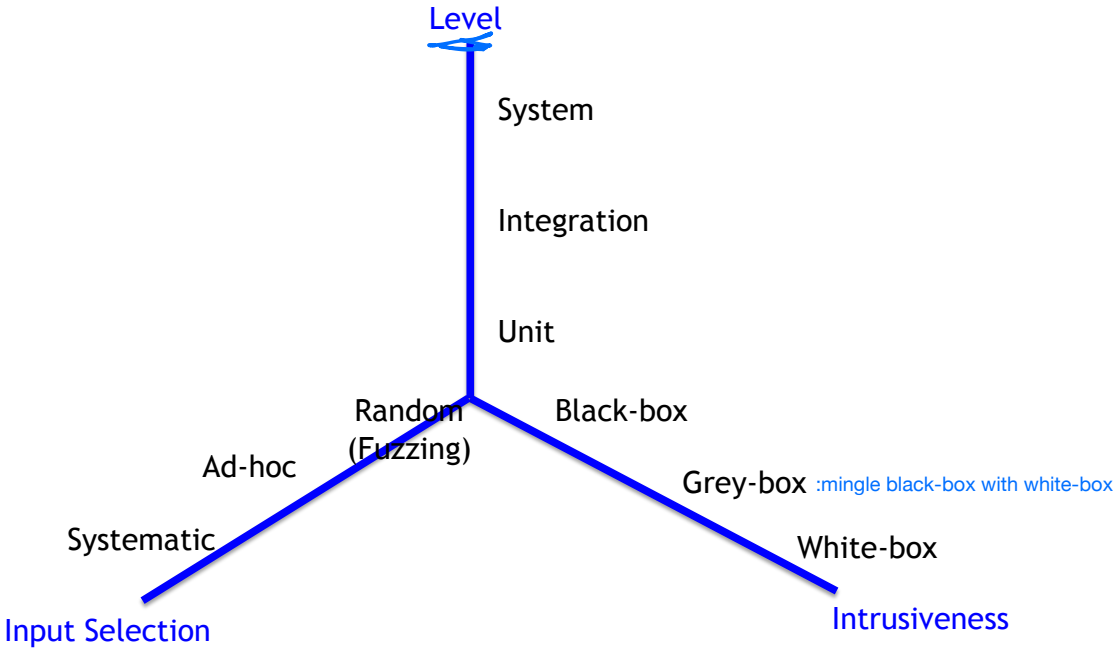
redundant and not useful for this oracle, why?

Black and white-box testing approaches focus on different faults

Spec and EUT overlap, but don't exactly match each other: different techniques are suitable for covering the unmatched regions



Testing approaches can be represented in three dimensions: Level, Input Selection, and Intrusiveness



Some things we need to know for all types of testing and testing approaches

- Designing tests
- Selecting test inputs
- Determining oracles
- Setting up test fixture
- Measuring test adequacy (e.g., coverage)

All the things we will cover!