Testing (Part II)

Week 8

Blame doesn't fix bugs.

— Anonymous

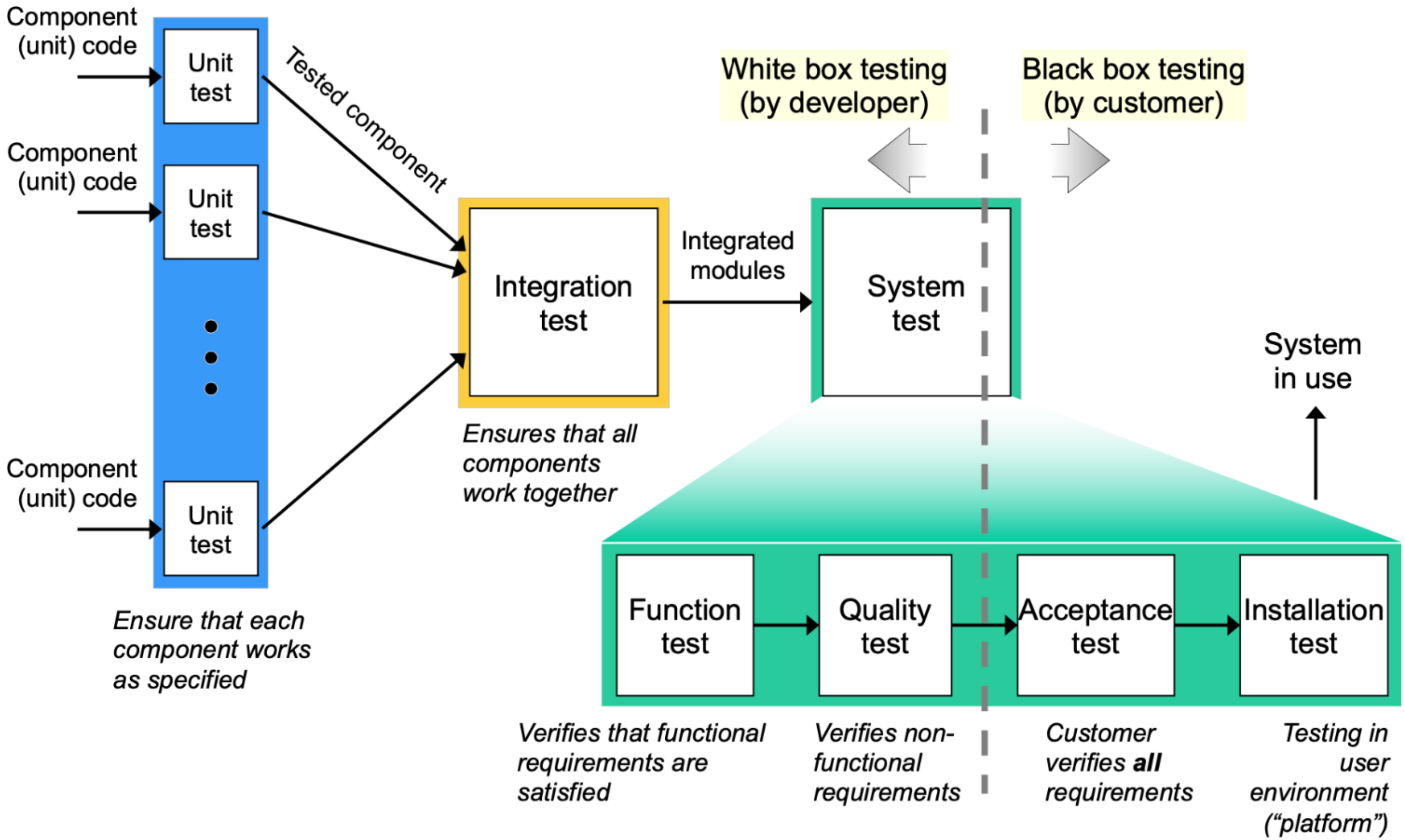
Objectives

- Understand integration testing
- Learn how much testing is sufficient
- Learn how to incorporate testings in development process

Content

- Integration testing
- Test coverage and code quality metrics
- Testing in development process
    - Test-Driven Development (TDD)
    - Continuous Integration (CI) and Continuous Development (CD)

Organization of Testing

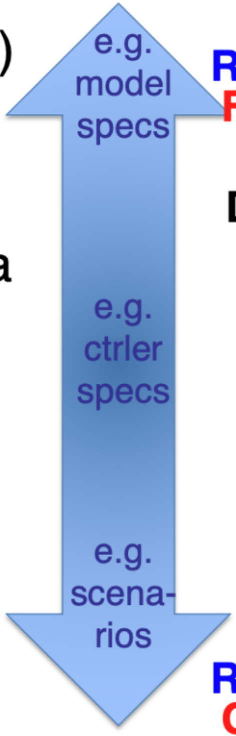


Tradeoff (1/2)

Unit (one method/class)

Functional or module (a few methods/classes)

Integration/system



Runs fast High coverage
Fine resolution
Many mocks;
Doesn't test interfaces

Few mocks;
tests interfaces
Runs slow Low coverage
Coarse resolution

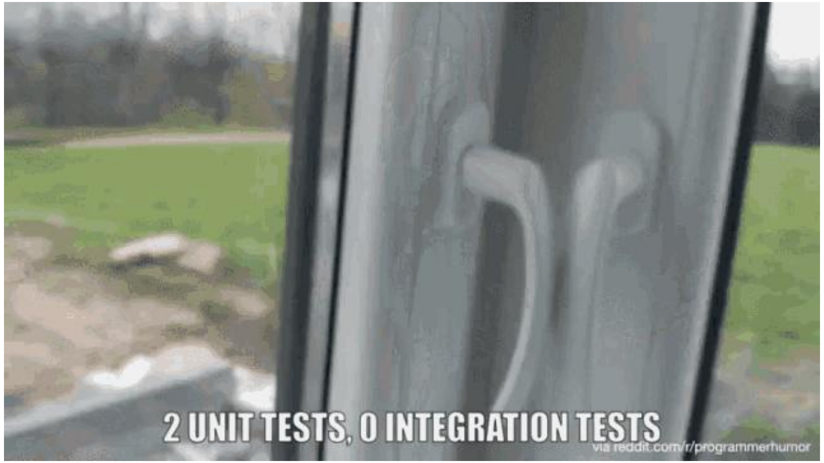
Tradeoff (2/2)

- “Focus on unit tests, they’re more thorough”
- “Focus on integration tests, they’re more realistic”

⇒ each finds bugs the other misses

Why Integration Testing?

- Unit tests are not enough
- Must check different parts work well **together**



What is Integration Testing?

- Checks how different parts of your app interact together
  - Verify interaction and integration points
  - Check data and control flow between multiple parts

- Usually done after unit tests and before E2E tests
  - Unit tests: tests small parts of your app
  - Integration tests: tests integration of two or more parts of app
  - E2E tests: tests large parts of app (e.g. user flow)

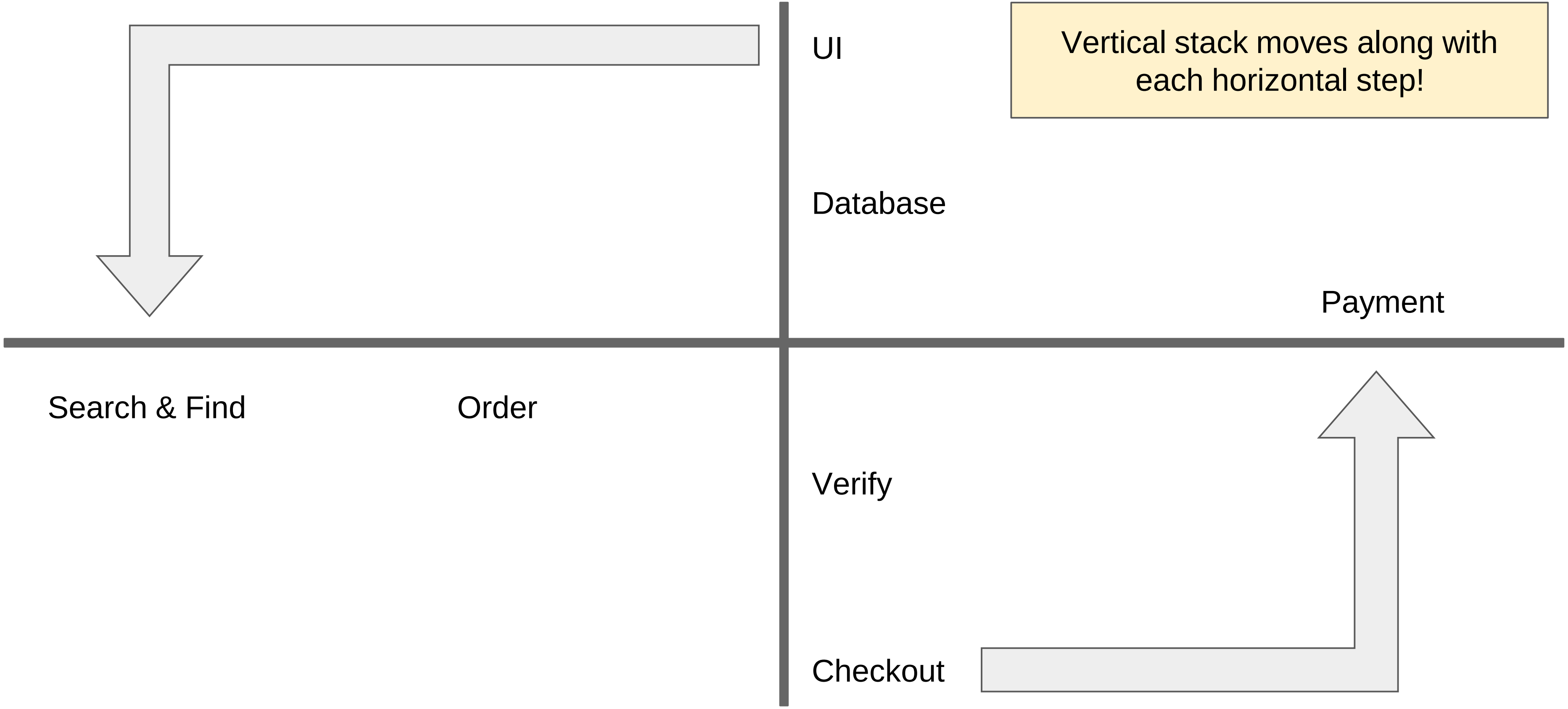
Unit Test vs. Integration Test

	Unit Test	Integration Test
Scope	Small	Medium
Speed	Fast	Slow
Fidelity	Low fidelity	High fidelity
Device	Local tests	Instrumented tests (usually)

Integration Test Approach

- Horizontal integration tests
  - Test modules of the same level in the app's architecture
  - Useful for testing similar modules, identifying issues in parallel integration of components
  - Search, find, order, and purchase tested together for e-commerce
- Vertical integration tests
  - Vertically slice the app by use cases
  - Test integration of all modules within the same slice
  - Useful for testing data and control flow of modules at different levels
  - UI, algorithm, database tested together for search functionality

Horizontal & Vertical Testing Example



In-Class Activities

- We will learn how to do UI-driven vertical integration testing using a “Espresso” tool
    - How to use Espresso
    - Using Espresso to write UI tests for Fragments
    - Using Espresso & Mockito to write UI tests with the Navigation component

Code Quality and Test Coverage

How Much Code Quality?

- Bad: “LGTM - Looks Good To Me”
- A bit better: continuous review
  - Pair programming
  - Code review over pull requests
  - Check “code smells”
- Better question: “how to objectively measure code quality?”
  - Use various metrics such as ABC score or cyclomatic complexity

Code Quality Metrics

- Use measurable and objective number to identify the complexity of code
- Indicate the simplicity and modularity of the code
- Over-complication lowers code quality and increase cost

Metric	Target Score
Assignment-Branch-Condition score (ABC score)	<20 per method
Cyclomatic complexity	<10 per method

ABC Score (1/2)

- A vector <Assignments, Branches, Conditions>
  - A: # of assignments or variables changed (state change)
  - B: # of branches (different execution paths)
  - C: # of conditions (different execution paths)
- The score can be represented as the magnitude:

$$|< ABCvector >| = \sqrt{(A^2 + B^2 + C^2)}$$

- High ABC score may mean hard-to-manage, complex code

ABC Score (2/2)

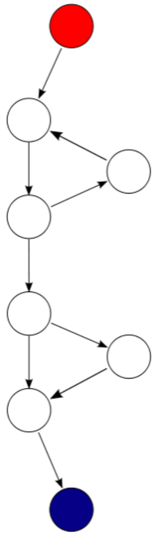
- C, C++, Java has specific rules
- |ABC vector| = |<3, 2, 4>| = 5.38

```
public class Example {
        public void ABC_method(int a){
 a              int b = 0;
 c              if(a > b){
 b                      System.out.println("Your input is positive.");
 c              } else if (a == b){
 aca                    for(int i=0; i<10; i++);
 c              } else {
 b                      System.out.println("Your input is negative.");
                }
        }
}
```

Source: https://wstomv.win.tue.nl/edu/2ip30/references/ABCmetric.pdf

Cyclomatic Complexity

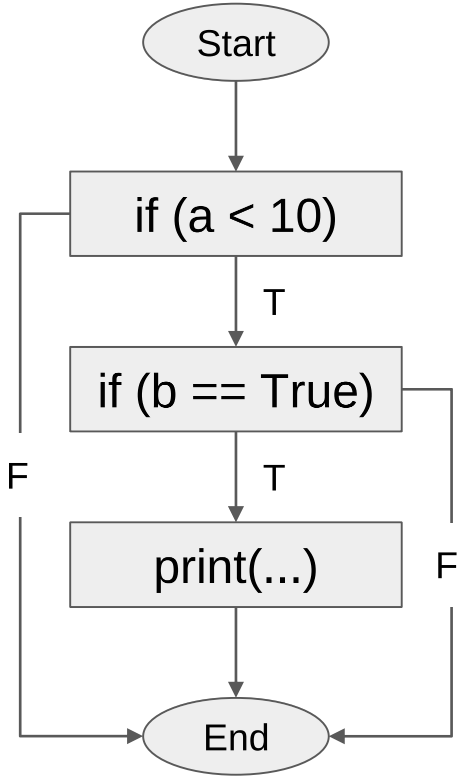
- First introduced by McCabe, 1976
- Uses a control-flow graph showing all executable paths
- Measures the # of linearly independent paths in code
  - Each path has at least one edge not included in another path
- CC = E - N + 2P
  - E = # of edges: between 2 consecutive commands
  - N = # of nodes: indivisible groups of commands
  - P = # of connected components:
    connected subgraph not part of another subgraph
    (usually 1)



Cyclomatic Complexity: Example 1

- CC = E - N + 2P = 6 - 5 + 2*1 = 3

```
void function (int a, boolean b){
    if (a < 10){
        if (b == True)
            System.out.println("Hello World!");
    }
    return;
}
```

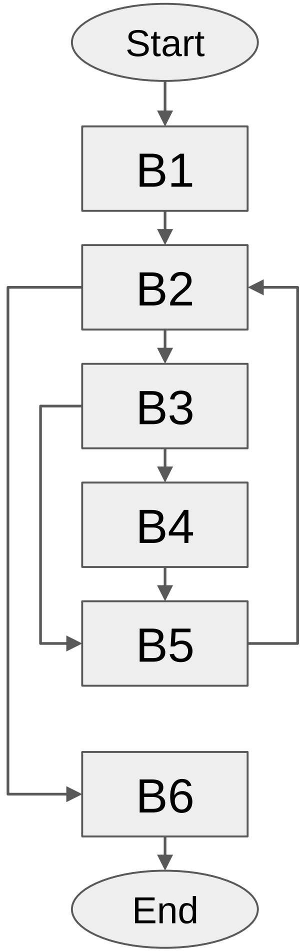


Source: https://blog.naver.com/PostView.naver?blogId=suresofttech&logNo=220811189835

Cyclomatic Complexity: Example 2

- CC = E - N + 2P = 9 - 8 + 2*1 = 3

```
int oddSum(int n) {
B1    int total = 0;
      for (int i = 1;
B2          i<=n;
B5          i++)
      {
B3        if (i % 2 == 1)
B4            total += i;
      }
B6    return total;
}
```



Source: https://velog.io/@suuzin/소프트웨어공학-순환복잡도Cyclomatic-Complexity

How Much Testing?

- Bad: “Until time to ship”
- A bit better: code-to-test ratio
  - 1.2 ~ 1.5: reasonable
  - Often much higher for production systems
- Better question: “How thorough is my testing?”
  - Formal methods
  - Coverage measurement

Test Coverage Metrics

- Measurable and objective number to identify the coverage of testing

Metric	**Target Score**
Code-to-test ratio	<= 1:2
Statement coverage	90% or more

Code-to-Test Ratio

- Formula
  - lines of test / lines of code
- The very basic metric metric for testing
- Indirectly indicate coverage of testing
- Usually not enough to indicate code quality
- 2 or higher for many production systems

Statement Coverage

- Formula:
  - (# statements executed by tests) / (# total statements)
- Miller and Maloney, 1963
- One of commonly used test coverage metric
- Reports execution footprint of tests

Test Coverage

- Coverage = % of covered code of total available code
- Considered “covered” if the target code is executed at least once by at least one test
- Types based on coverage unit
    - C0: every statement (line of code)
    - C1: every branch
    - C1+ decision coverage: every subexpression in conditional
    - C2: every path

statement coverage

branch coverage

path coverage

Test Coverage Example

- Calculate C0, C1, C1+ and C2

```
def function(x, y)
    if x==0 or y>0:
        do_something1
    else:
        do_something2

    end
```

```
// Test cases

function(1, 5);

function(0, -1);
```

Test Coverage Example

- C0: 4/5 = 80%
  - start and end are also counted
  - do_something2 isn't executed
- C1: 1/2 = 50%
  - The other side of condition isn't taken
- C1+: 100%
  - x==0 is F in func(1, 5) and T in func(0, -1)
  - y>0 is T in func(1, 5) and F in func(0, -1)
- C2: 1/2 = 50%
  - Condition = true is executed
  - Condition = false isn't executed

```
def function(x, y)
    if x==0 or y>0:
        do_something1
    else:
        do_something2

    end
```

Testing Methods for High Coverage

- Equivalence testing
- Boundary testing

Enhance input space coverage ("black-box" testing)

- Control-flow testing
- State-based testing

Enhance code coverage ("white-box" testing)

Source: https://www.ece.rutgers.edu/~marsic/books/SE/instructor/slides/lec-11%20Testing.ppt

Equivalence Testing (1/2)

- Divides the space of all possible inputs into equivalence groups such that the program is expected to “behave the same” on each input from the same group
- Assumptions
  - A well-intentioned developer may have made mistakes that affect a whole class of input values
  - We do not have any reason to believe that the developer intentionally programmed special behavior for any input combinations that belong to a single class of input values

Equivalence Testing (2/2)

- Step 1. Partition the values of input parameters into equivalence groups
- Step 2. Choosing the test input values from each group

```
Step 1.
Valid X is in range (0, 100)
    Equivalence class #1: 0 < X < 100
    Equivalence class #2: X <= 0
    Equivalence class #3: X >= 100

Step 2.
Test at least one value in each class
```

Boundary Testing (1/2)

- Boundary testing is a special case of equivalence testing that focuses on the boundary values of input parameters
  - Assumption: developers often overlook special cases at the boundary of equivalence classes (e.g., Microsoft Zune bug)
- Selects elements from the “edges” of each equivalence class, or outliers
  - zero, min/max values
  - empty set, empty string, and null
  - confusion between > and >=

Boundary Testing (2/2)

- Important because programs fail mostly at input boundaries

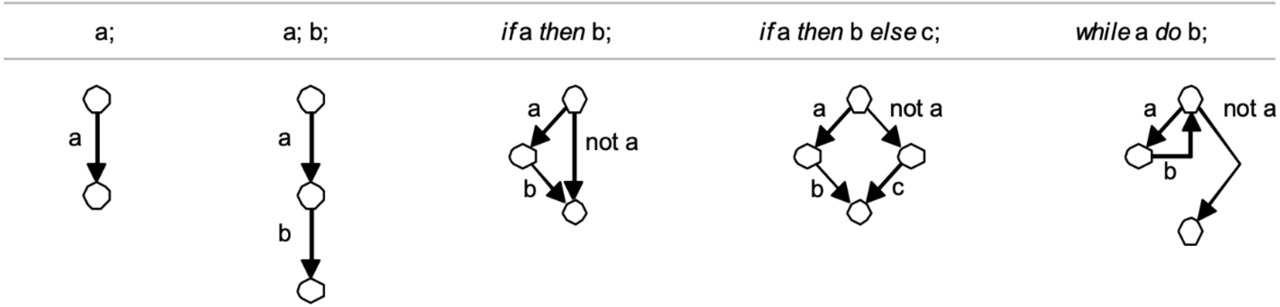
```
Example 1.
Valid input X in [MIN, MAX]. Test with:
- X = MIN, X = MAX
- X < MIN, X > MAX

Example 2.
Test a function findMax that returns the max value of a given array.
Test with:
- An empty array.
- An array with one element.
- An array with many elements.
- An array with a large number of elements.
```

Control-Flow Testing (1/2)

- Statement coverage
  - Each statement executed at least once by some test cases
- Edge coverage
  - Every edge (branch) of the control flow is traversed at least once by some test cases

Constructing the control graph of a program for Edge Coverage:



* Exceptions are also a form of control flow, as is concurrency or multithreading

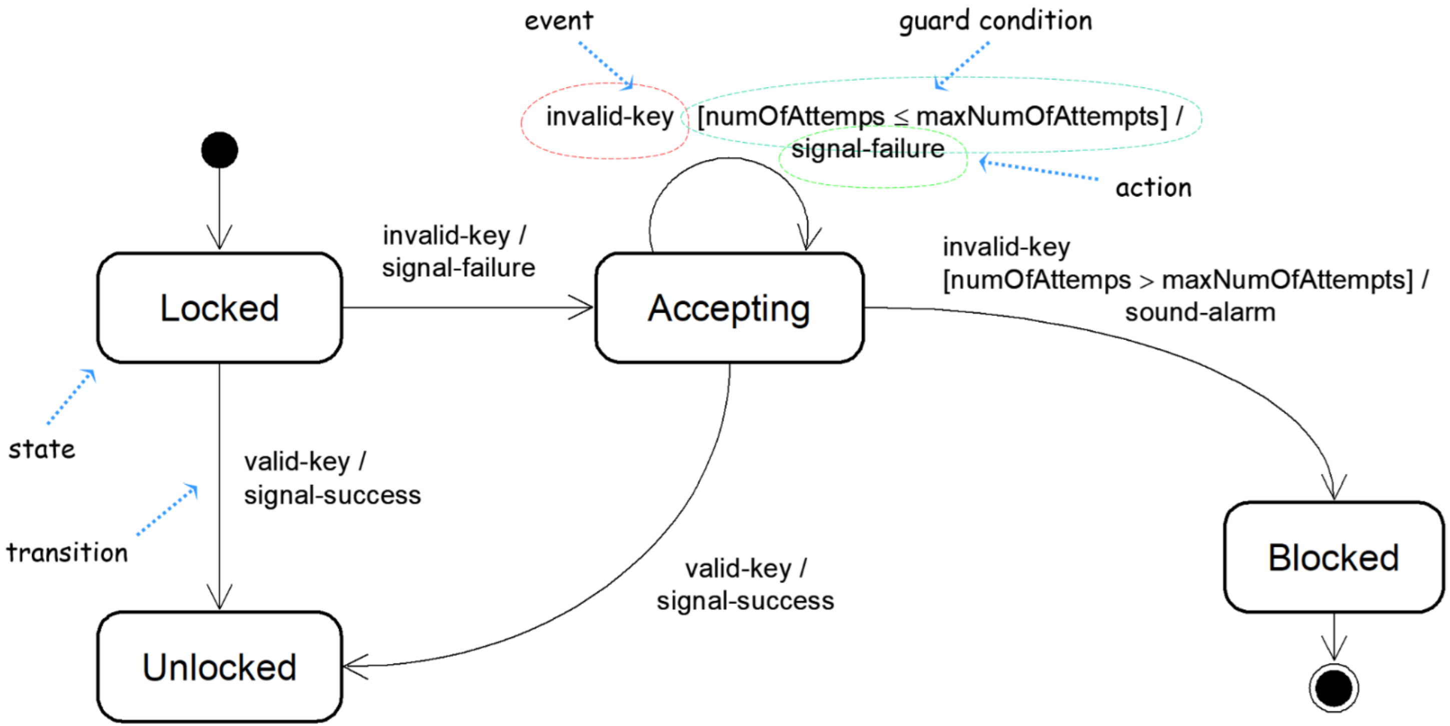
Control-Flow Testing (2/2)

- Condition coverage
  - Every condition takes TRUE and FALSE outcomes at least once in some test case
- Path coverage
  - Finds the number of distinct paths through the program to be traversed at least once

State-based Testing (1/2)

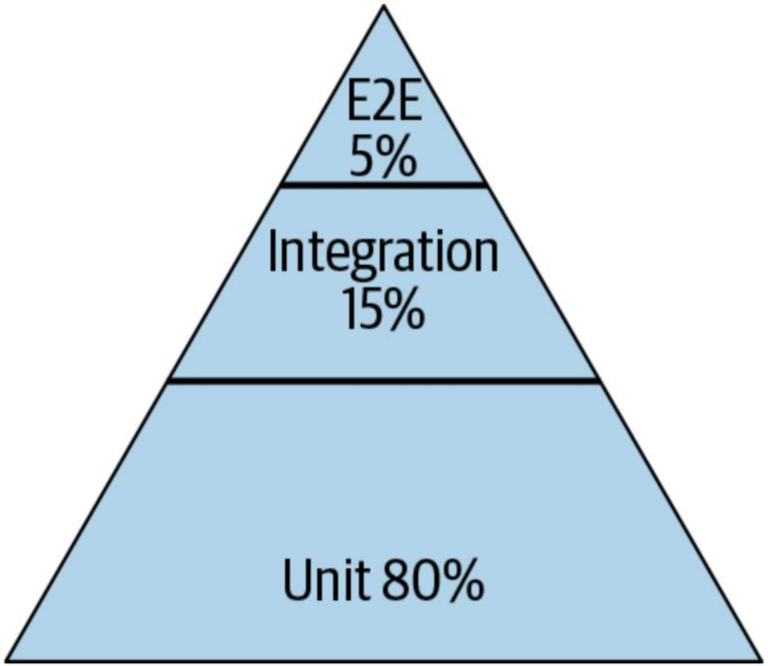
- Software can be understood as a state machine
- Defines a set of abstract states that a software unit (object) can take and tests the unit’s behavior by comparing its actual states to the expected states
- This approach is popular with object-oriented systems

State-based Testing (2/2)

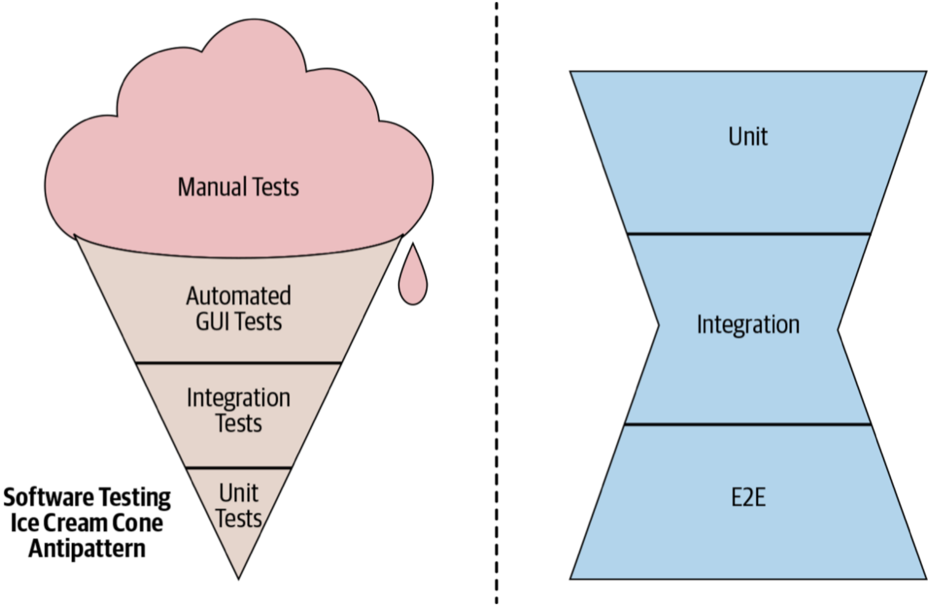


Testing in Development Process

Testing in Development Process



Google's version of Mike Cohn's test pyramid; percentages are by test case count, and every team's mix will be a little different

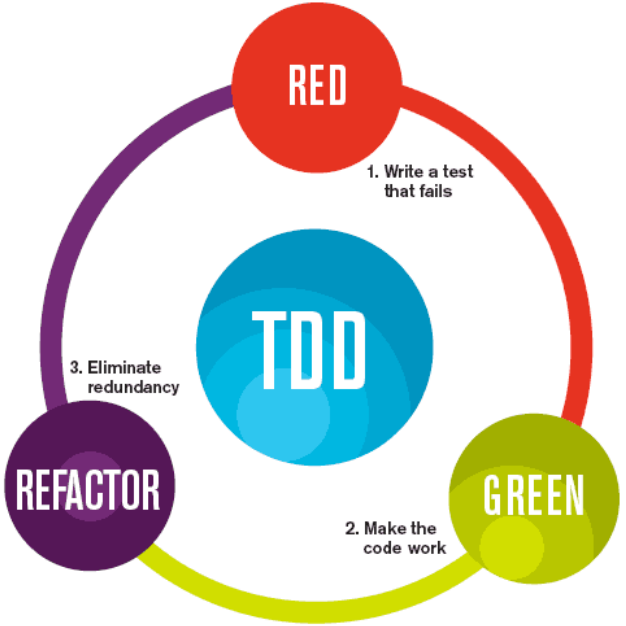


Test suite antipatterns

Source: https://abseil.io/resources/swe-book/html/ch11.html

Test-Driven Development (TDD)

- Tests drive the design and implementation
  - You write tests first, and then code in order to make tests pass
- Process
  - Write test that specifies desired behavior
  - Run test (test will fail)
  - Write code that makes the test pass
  - Refactor code to improve readability and structure (test must always pass)



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

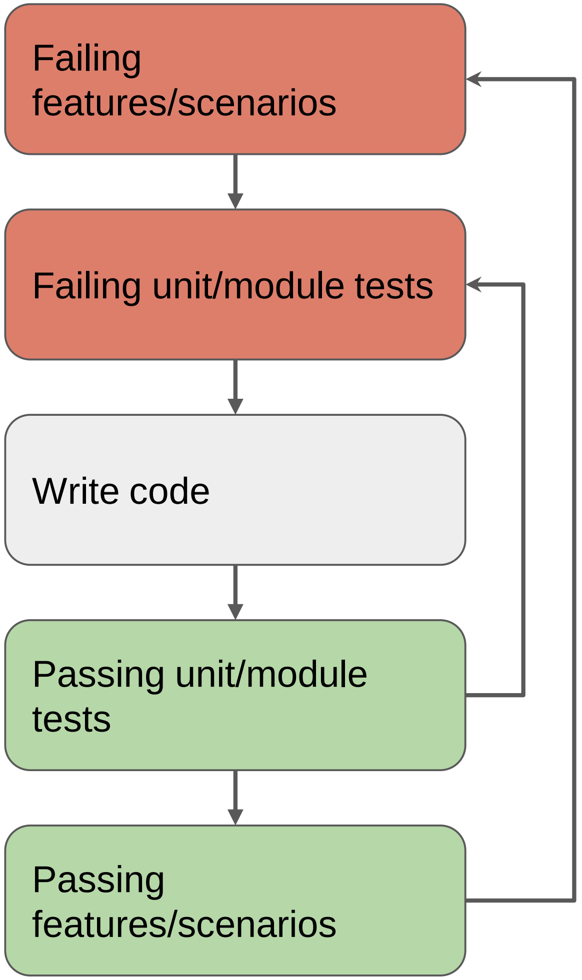
TDD in Industry

Metric description	IBM: Drivers	Microsoft: Windows	Microsoft: MSN	Microsoft: VS
Defect density of comparable team in organization but not using TDD	W	X	Y	Z
Defect density of team using TDD	0.61W	0.38X	0.24Y	0.09Z
Increase in time taken to code the feature because of TDD (%) [Management estimates]	15 – 20%	25-35%	15%	20-25%

Source.: Nagappan, Nachiappan, et al. "Realizing quality improvement through test driven development: results and experiences of four industrial teams." Empirical Software Engineering 13 (2008): 289-302

Ideal Development Process

- Two core questions in development
  - Building it right
    - Ensuring the implementation meets the specification
    - **Verification** by unit/module tests
  - Building the right thing
    - Ensuring the product satisfies user needs
    - **Validation** by feature/scenario tests



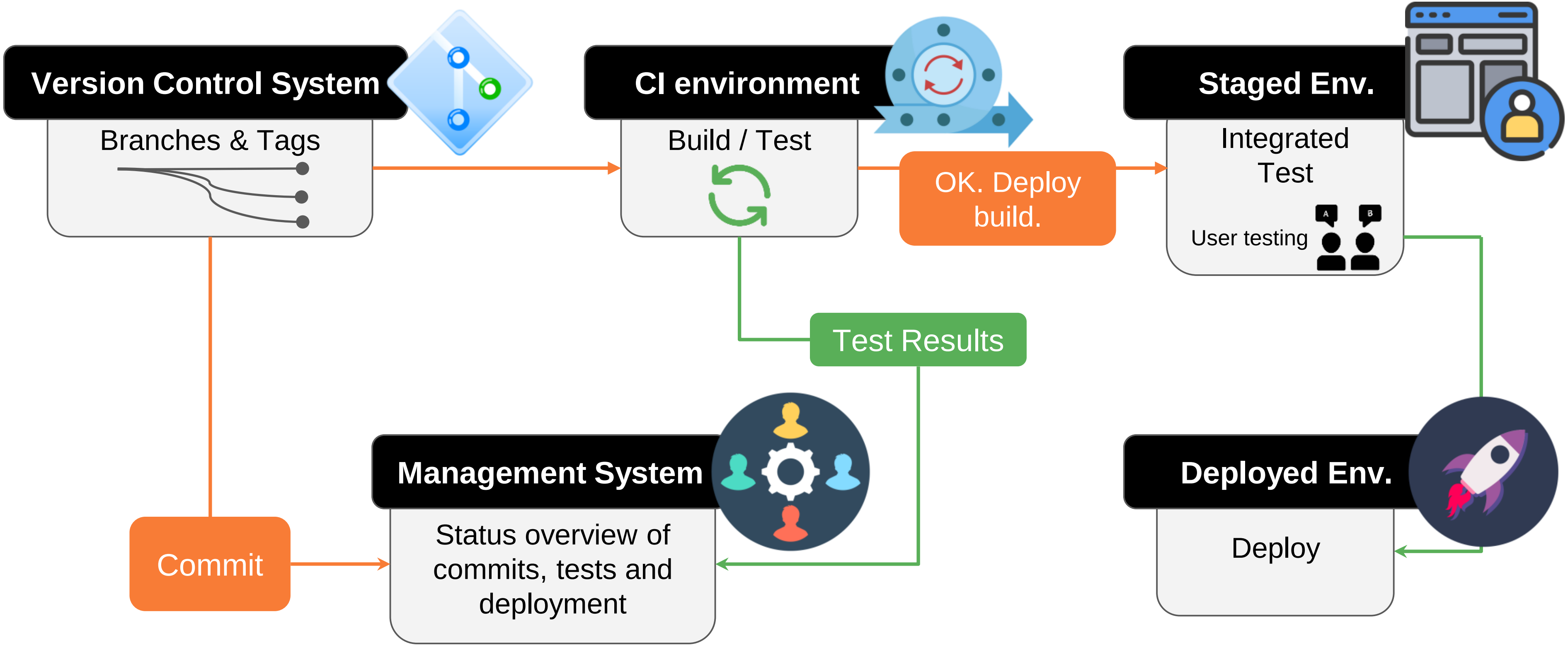
Continuous Integration (CI)

- Developers work on separate branches
- Individual work is integrated into a single main branch
- CI system automatically builds and tests the entire system
  - Continuously monitor readiness of code (ensure code can be built)
  - Discover issues earlier
  - Reduce integration pain through automation and isolation of issues
  - Test beyond single developer's resources
  - Eliminate reliance on developers' discipline

Continuous Delivery / Deployment (CD)

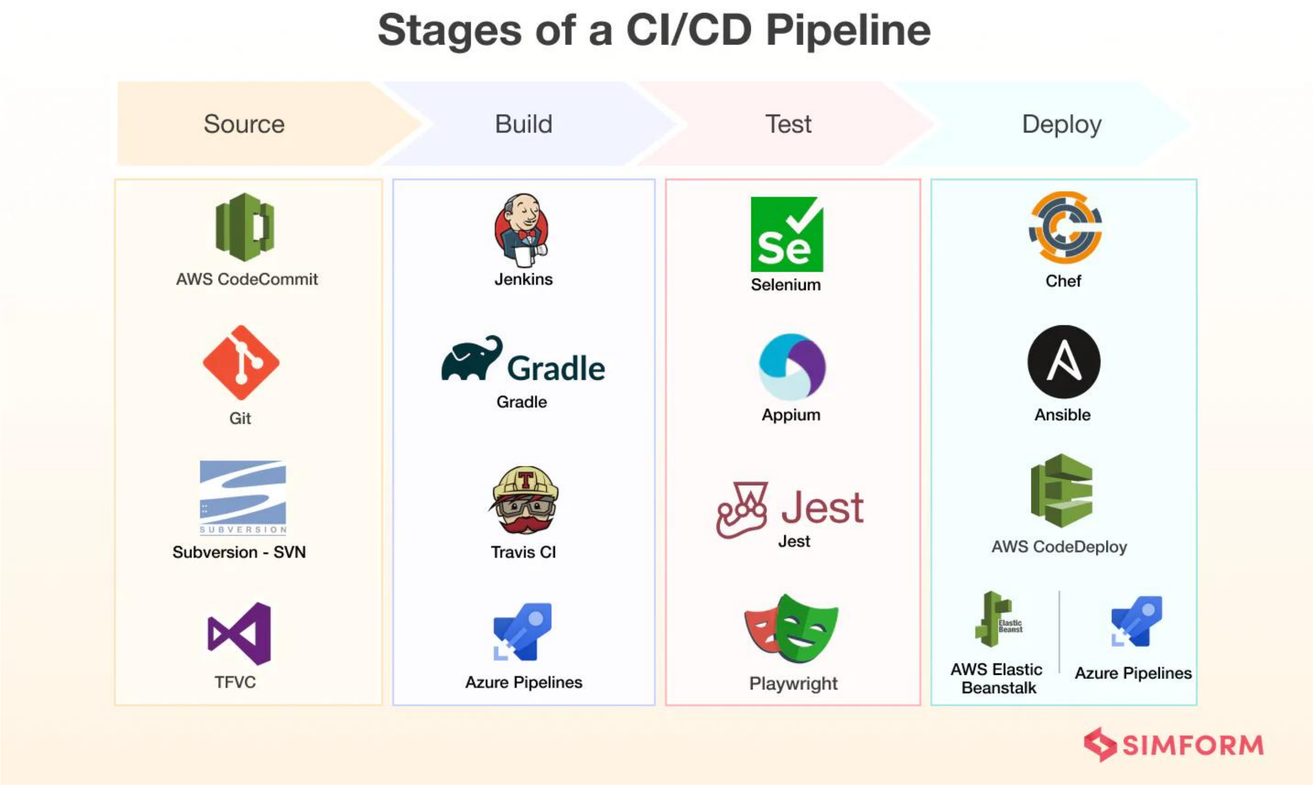
- Extension of Continuous Integration
- Automate the delivery of applications to selected infrastructure environments (development, testing and production) as soon as they pass tests and build checks

CI/CD Pipeline



Source (translated): https://techblog.woowahan.com/2579/

CI/CD Pipeline



Source: https://www.simform.com/blog/scalable-ci-cd-pipeline-examples/

Sources

- Rutgers Uni. - Software Engineering lec 11
- Cornell Univ 18, 19장
- Software Engineering at Google book (16,17,18)
- ESaaS chapter 8 youtube playlist ← Recommend!!
- ESaaS slide chapter 8
- EPFL (for exercises) ← Recommend!!