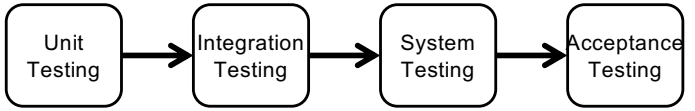
Software validation: unit testing

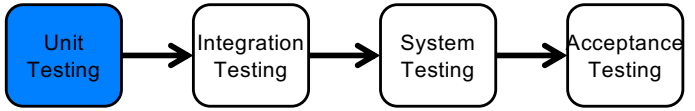
- The difference between Unit and Integration testing
- Static and Dynamic unit testing
- Black and White box testing approaches
- Code-coverage tests (White-box)
- Unit testing
  - Selecting test cases
  - Components of unit testing and examples: Test driver, tested unit, stubs, oracle
  - JUNIT
- Test Driven Development approach

CE202 Software Engineering, Autumn term
Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex

Reminder: Types of Testing

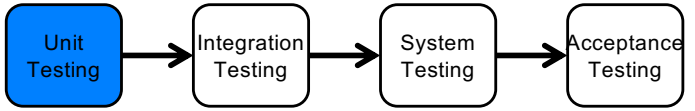


- Unit Testing
  - Individual component (class or subsystem)
  - Carried out by developers
  - Goal: Confirm that the component or subsystem is correctly coded and carries out the intended functionality
- Integration Testing
  - Groups of subsystems (collection of subsystems) and eventually the entire system
  - Carried out by developers
  - Goal: Test the interfaces among the subsystems.



Unit Testing

- Static Testing (at compile time)
  - Static Analysis
  - Review
    - Walk-through (informal)
    - Code inspection (formal)
- Dynamic Testing (at run time)
  - Black-box testing (Test the input/output behavior)
  - White-box testing (Test the internal logic of the subsystem or object)
  - Data-structure based testing (Data types determine test cases)



Black-box testing

- Focus: I/O behavior
  - If for any given input, we can predict the output, then the component passes the test
  - Requires a test oracle (to see if a test passes or not)
  - Almost always impossible to generate all possible inputs ("test cases")
- Goal: Reduce number of test cases by equivalence partitioning (see week 10 class):
  - Divide input conditions into equivalence classes
  - Choose test cases for each equivalence class.
  - (Example: If an object is supposed to accept a negative number, testing one negative number is enough)

Black-box testing: Test case selection

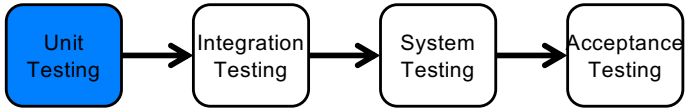
a) Input is valid across range of values

- Developer selects test cases from 3 equivalence classes:
  - Below the range
  - Within the range
  - Above the range

b) Input is only valid, if it is a member of a discrete set

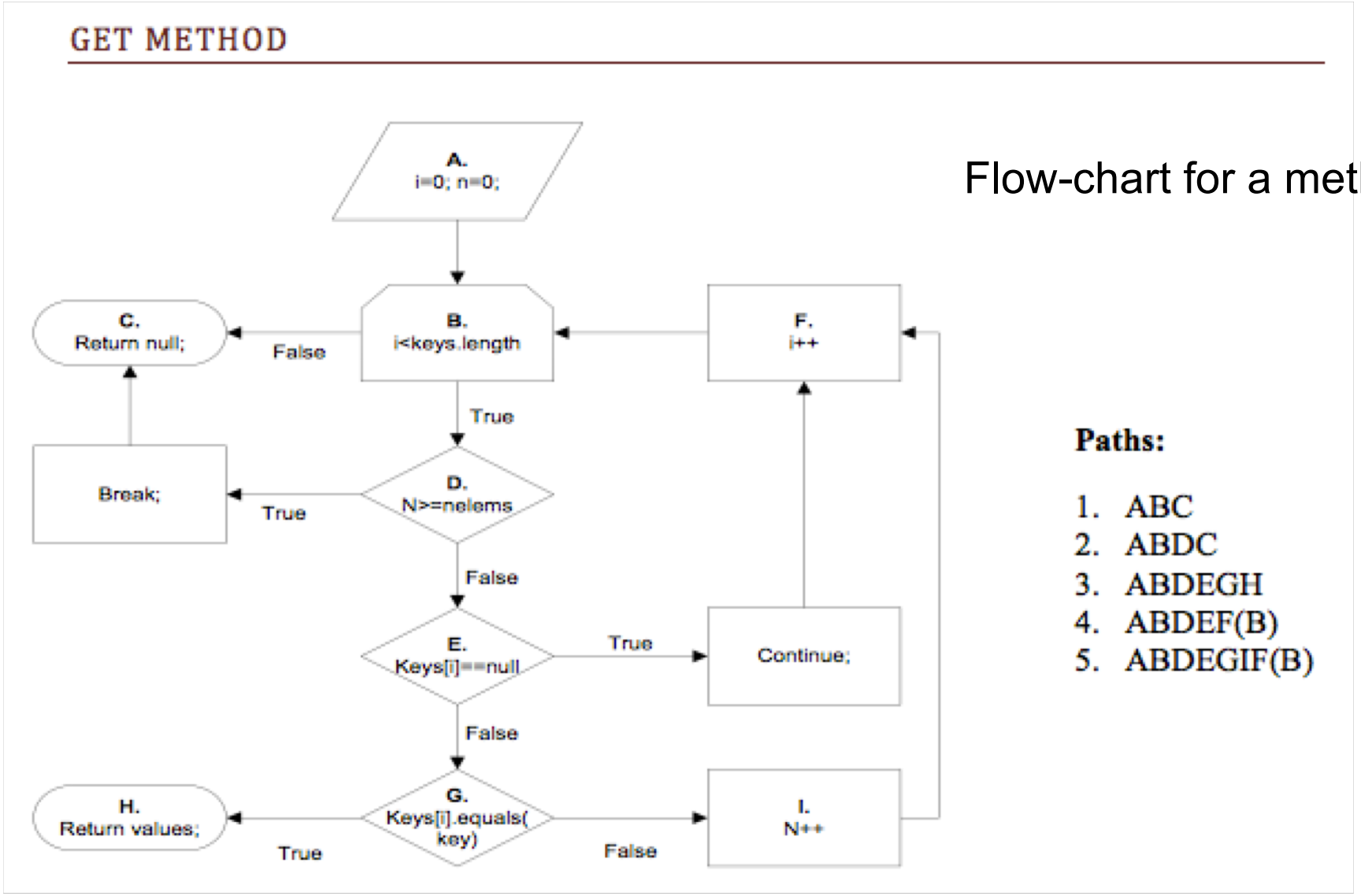
- Developer selects test cases from 2 equivalence classes:
  - Valid discrete values
  - Invalid discrete values
- No rules, only guidelines.

White-box testing overview



- Code coverage
- Branch coverage
- Condition/statement coverage
- Path coverage
- The differences between these approaches is discussed in the week 10 class

Code coverage test selection



Flow-chart for a method

Paths:

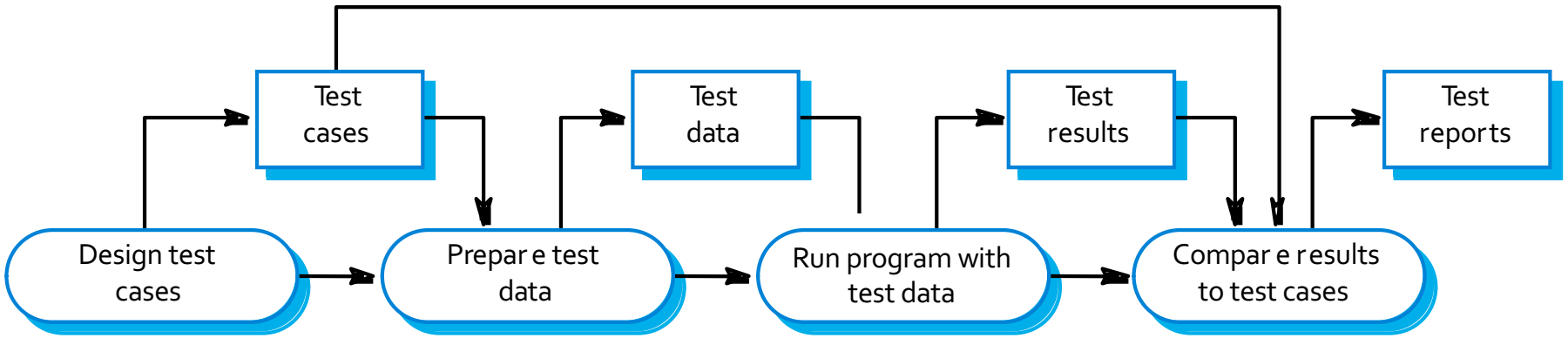
1. ABC
2. ABDC
3. ABDEGH
4. ABDEF(B)
5. ABDEGIF(B)

Unit Testing Heuristics

1. Create unit tests when object design is completed
   - Black-box test: Test the functional model
   - White-box test: Test the dynamic model
2. Develop the test cases
   - Goal: Find effective number of test cases
3. Cross-check the test cases to eliminate duplicates
   - Don't waste your time!
4. Desk check your source code
   - Sometimes reduces testing time
5. Create a test harness
   - Test drivers and test stubs are needed for integration testing
6. Describe the test oracle
   - Often the result of the first successfully executed test
7. Execute the test cases
   - Re-execute test whenever a change is made ("regression testing")
8. Compare the results of the test with the test oracle
   - Automate this if possible.

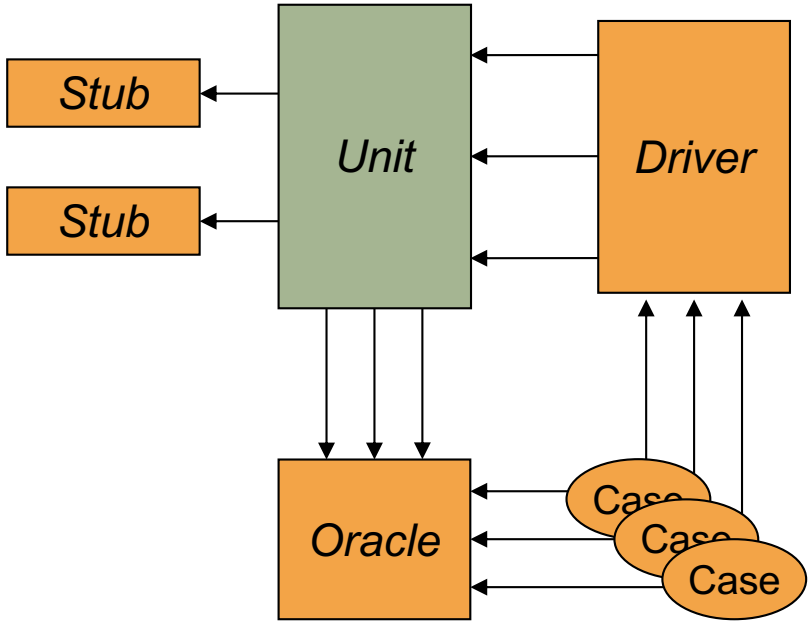
Don't forget regression testing - Re-execute test cases every time a change is made.

Designing and using test cases



Elements in unit testing

- Cases
- Driver
- Oracle
- Stubs

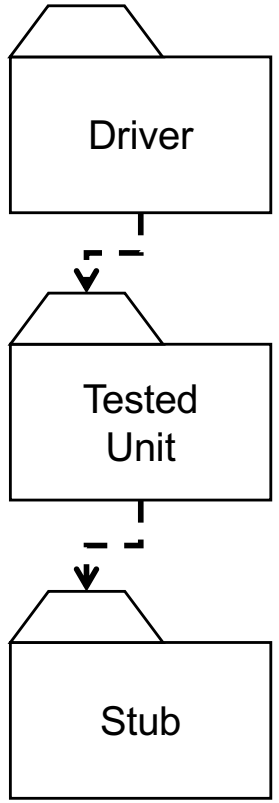


Terminology

- **Test case**: A set of input data and expected results
  - Exercises a component with the purpose of causing failures and detecting faults
- **Test driver**: A program that simulates the part of the system calling the unit under test ("subject")
  - Needed for all tests except for tests for the complete system (e.g., *integration test*)
- **Test oracle**: Declares either that the system passed or failed
  - WRT a collection of test cases
- **Test stub:** isolates the component under test from the rest of the program

Stubs and drivers

- Driver:
  - A component, that calls the `TestedUnit`
  - Controls the test cases
- Stub:
  - A component, the `TestedUnit` depends on
  - Partial implementation
  - Returns fake values.

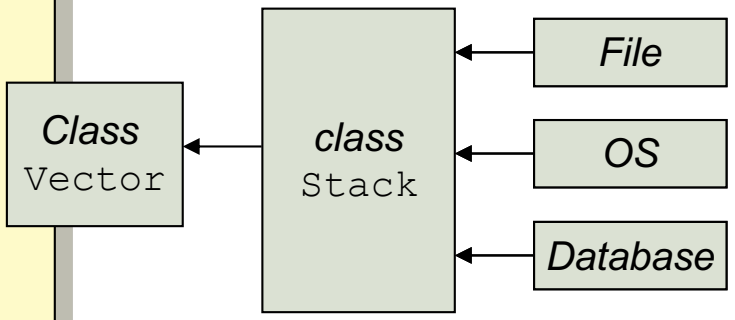


Example: Unit testing for `Stack`

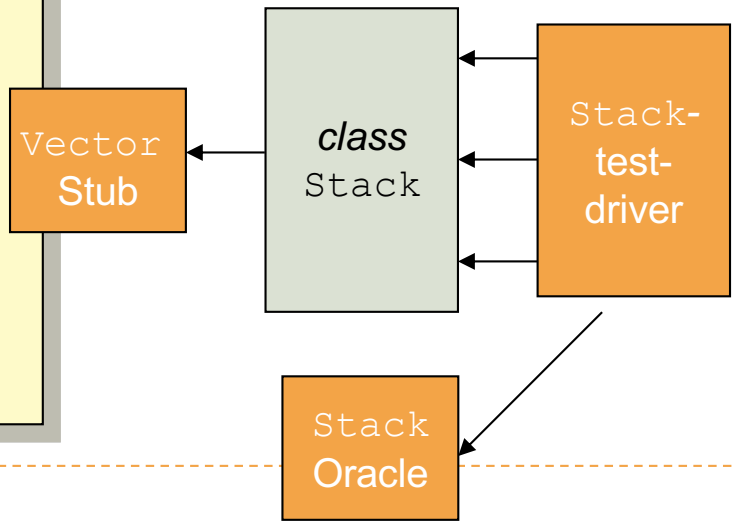
- Isolate class `Stack`

```
public class Stack {
   public void push(Object obj)
      throws Exception {
   {  if (! imp.add(obj))
         throw new Exception; }
…
}

 public Object pop()
      throws Exception {
      if (imp.size()==0)
         throw new Exception;
      return imp.removeElementAt(size-1);}

   ...

   private Vector imp;
}
```

Planned deployment of `Stack`:



Unit testing of `Stack`:



Example: Test driver for `Stack`

- A trivial example
- Missing:
  - Use larger input/output sets
  - Select inputs wisely
  - Need to stub Vector

```
package StackTest;

public class Driver {

   public static boolean test_case1() {
      Stack s = new Stack();
      String input = new String("testing");
      s.push(input);
      String output = s.pop();
      return
        oracle.report_case1(input,output);
   }

   public static boolean test_case2()
        {...}

   public static StackTestOracle oracle;
}
```

Example: Test Oracle for `Stack`

- A trivial example
  - Suitable only for `test_case1`
  - Returns **`true`** if fault occurred (test failed)
- Missing:
  - Report where fault occurred

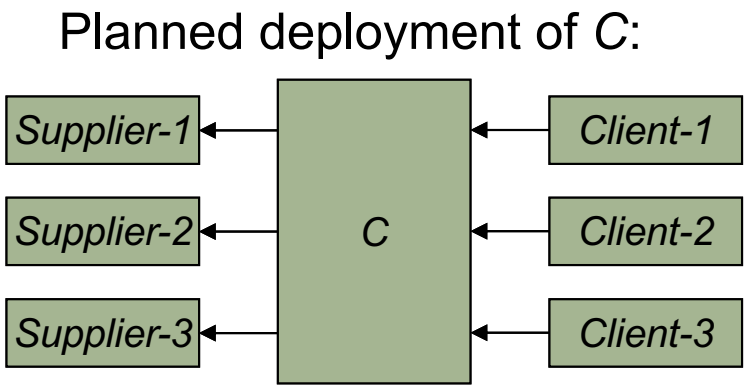
```
package StackTest;

public class StackTestOracle {

   public static boolean report_case1(
      String input, String output) {
      // Return false if no fault detected
      return (input != output);
   }
   ...
}
```

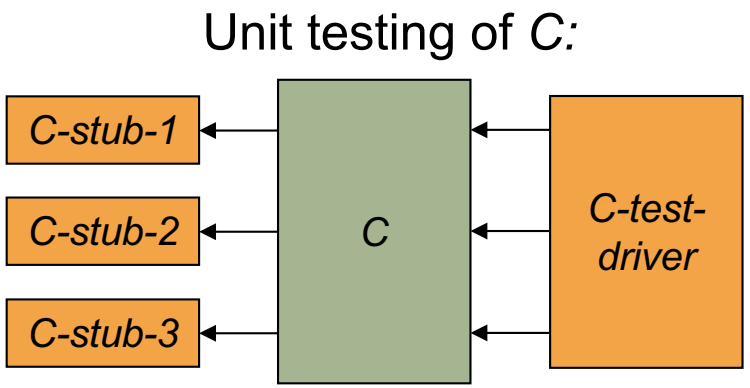
How to write a good test driver/oracle

- Driver: Read input from a file
  - E.g.: 4, 9, 16, 25, …
- Oracle: Read expected output from another file and compare with input
  - E.g.: 2, 3, 4, 5, …

Test stubs

- In unit testing we isolate the component under test *C* from the rest of the program:
  - *Test driver* calls *C*
  - *Test stubs* whenever *C* calls other components
- **A test stub** simulates units that are called by the subject
  - Needed for all tests except for tests for the complete system (e.g., *integration test*)





How to write a stub?

- Stubs should be kept as simple as possible
  - Otherwise: They need to be tested themselves…
- Possible alternatives:
  - Write a trivial stub
  - Use code from a reliable source
    - For example: Replace `StackStubs/Vector` with `java/util/LinkedList`
  - Replace one implementation with another

```
package StackTest;

// an Adapter for LinkedList
public class Vector extends java.util.LinkedList {
  // public void size()        —same as in LinkedList
  // public boolean add(obj)   —same as in LinkedList
  public void removeElementAt(int loc) {
    return this.remove();
  }
}
```

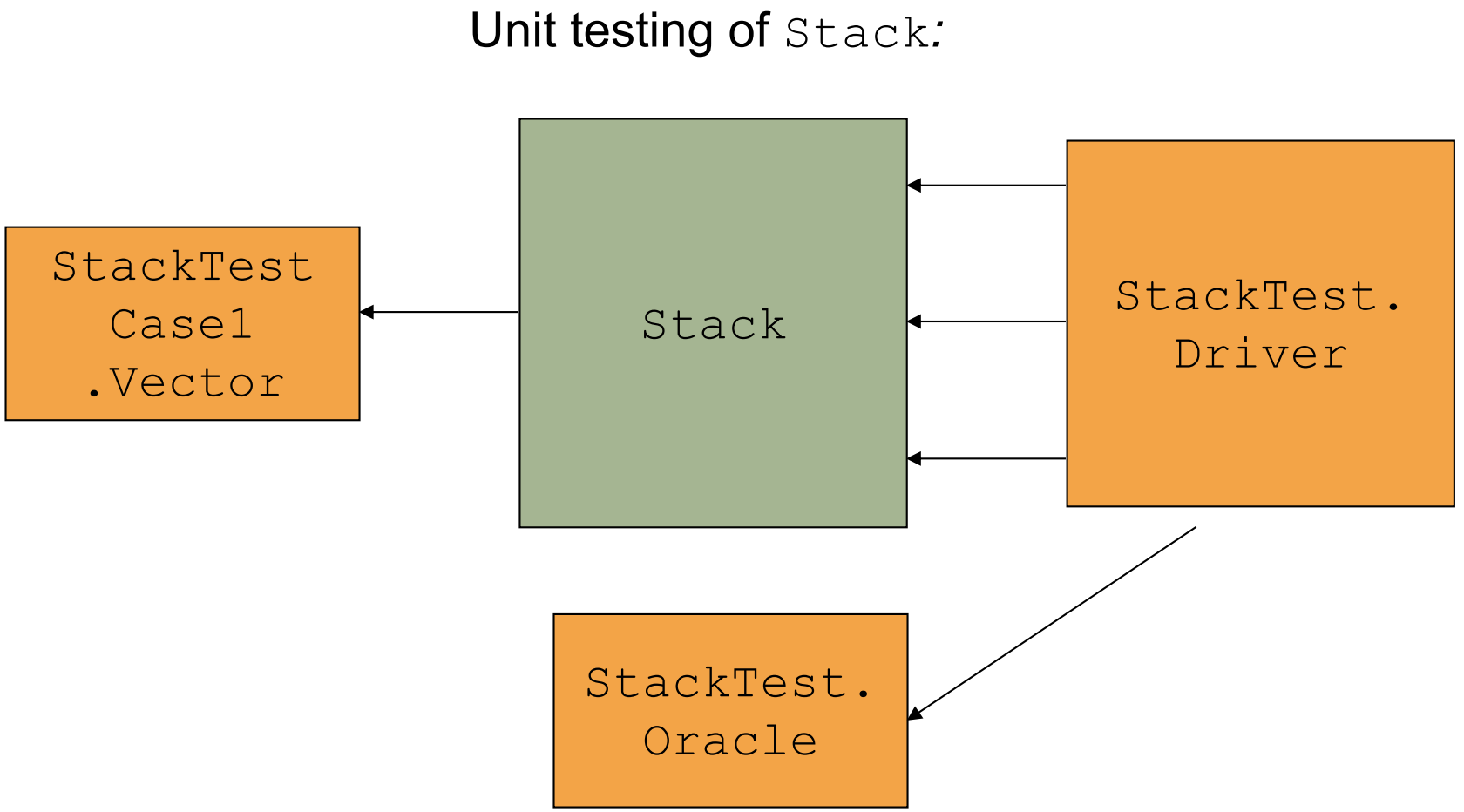
Example: Test stub for `Stack`

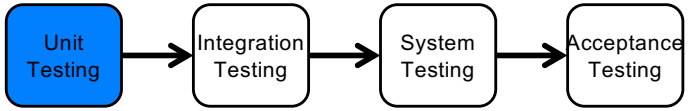
- Isolate `Stack`: Replace service suppliers with stubs!
  - Otherwise: Failures can occur in the suppliers!
- Replace `Vector` with a "Stub"
  - Can be trivial

```
public class Stack {
  public void push(Object obj)
    throws Exception {
    if (! imp.add(obj))
      throw new Exception; }

  public Object pop() throws Exception {
    if (imp.size()==0)
      throw new Exception;
    return imp.removeElementAt(size-1);
}

   ...

   private Vector imp;
}
```

```
package StackTestCase1;

public class Vector {
   public void size() { return 1; }
   public boolean add(obj) { return true; } // do nothing
   public String removeElementAt(int loc) {
      return new String("testing"); }
}
```

Summary: Example





JUnit: Overview

- A Java framework for writing and running unit tests
  - Test cases and fixtures
  - Test suites
  - Test runner
- Implements the same principles eg. provides test driver and ability to know if a test has passed or failed
- Written by Kent Beck and Erich Gamma
- Written with “test first” and pattern-based development in mind
  - Tests written before code
  - Allows for regression testing
  - Facilitates refactoring
- JUnit is Open Source
  - www.junit.org
  - JUnit Version 5, released September 2017

JUnit testing

- A JUnit *test* is a method contained in a class which is only used for testing. This is called a *Test class*. To define that a certain method is a test method, annotate it with the @Test annotation.
  - Essentially this is the **Test Driver**
- This method executes the code under test. You use an *assert* method, provided by JUnit or another assert framework, to check an expected result versus the actual result. These method calls are typically called *asserts* or *assert statements*.
  - This carries out the functionality of the **Oracle**
  - There are various types of assertions like Boolean, Null, Identical etc.

- Still may need to isolate your unit under test using **Stubs**

JUnit (simple) example

- This test assumes that the MyClass class exists and has a multiply(int, int) method.

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;
public class MyTests {
        @Test
        public void multiplicationOfZeroIntegersShouldReturnZero() {
          MyClass tester = new MyClass(); // MyClass is tested

          // assert statements
          assertEquals(0, tester.multiply(10, 0), "10 x 0 must be 0");
          assertEquals(0, tester.multiply(0, 10), "0 x 10 must be 0");
          assertEquals(0, tester.multiply(0, 0), "0 x 0 must be 0");
        }
}
```

Other JUnit features

- Textual and GUI interface
  - Displays status of tests
  - Displays stack trace when tests fail
- Integrated with Maven and Continuous Integration
  - http://maven.apache.org
    - Build and Release Management Tool
  - https://continuum.apache.org
    - Continous integration server for Java programs
  - All tests are run before release (regression tests)
  - Test results are advertised as a project report
- Many specialized variants
  - Unit testing of web applications
  - J2EE applications
  - Etc

The *Test Driven Development* philosophy

- The basic tenants are developing and implementing unit tests before writing a line of code
- Unit tests will and must fail up front
  Code is developed after the test is developed.
- A unique idea that is still foreign to many developers

TDD steps

- Quickly add a test just enough code to fail test
- Run testsuite to ensure test fails (may choose to run a subset of suite)
- Update your functional code to ensure new test passes
- Rerun test suite and keep updating functional code until test passes
- Refactor and move on

Benefits of TDD

- Shortens the programming feedback
- Provides detailed (executable) specifications
- Promotes development of highquality code
- Provides concrete evidence that your code works
- Requires developers to prove it with code
- Provides finelygrained, concrete feedback (in mins)
- Ensures that your design is clean by focusing on creation of operations that are callable and testable
- Supports evolutionary development

TDD and Agile

- TDD implies agile
- Strong emphasis on testing
- Tests should span entire breadth of codebase
- Once all software is ready for delivery, all tests should pass
- Seen as a unique way to address modern challenges in software development

Principles of Agile development (a reminder)

- Continuous delivery
- Welcome changing requirements
- Deliver working software frequently
- Involve the business and developers throughout the project
- Build projects around motivated people
- Communication should be facetoface
- Primary metric of progress is working software
- All participants should maintain a constant pace
- Continuous attention to technical excellence & good design
- Simplicity is essential
- Self organizing teams
- Periodic retrospective reviews

Testing in a more traditional (waterfall) environemnt

- Mostly implies a waterfall/bigbang process
- Should follow a structured approach ie.
  - unit -> integration -> system -> acceptance testing
- Very little emphasis on unit testing by developers
- Tests are sometimes developed as an afterthought
- Tests are mostly manual ie. not TDD
- Huge emphasis on QA team
- One view is that delivering quality software on time and within budget is almost accidental

Summary

- The difference between Unit and Integration testing
- Static and Dynamic unit testing
- Black and White box testing approaches
- Code-coverage tests (White-box)
- Unit testing
  - Selecting test cases
  - Components of unit testing and examples: Test driver, tested unit, stubs, oracle
  - JUNIT
- Test Driven Development approach
- Agile and waterfall approaches

Further reading

- Chapter 11 – Bruegge – Testing
- Chapter 8 – Pfleeger – Testing the Programs
- Introduction to Test Driven Development by Scott Ambler (http://agiledata.org/essays/tdd.html)