Test Execution and Automation

CSCE 747 - Lecture 17 - 03/14/2017

Executing Tests

- We've covered many techniques to derive test cases.
- How do you run them on the program?
  - You could run the code and check results by hand.
  - **Please don't do this.**
    - Humans are slow, expensive, and error-prone.
  - Test design requires effort and creativity.
  - Test execution should not.

Test Automation

- **Test Automation** is the development of software to separate repetitive tasks from the creative aspects of testing.
- Automation allows control over *how* and *when* tests are executed.
    - Control the environment and preconditions.
    - Automatic comparison of predicted and actual output.
    - Automatic hands-free reexecution of tests.

Testing Requires Writing Code

- Testing cannot wait for the system to be complete.
  - The component to be tested must be isolated from the rest of the system, instantiated, and *driven* using method invocations.
  - Untested dependencies must be *stubbed out* with reliable substitutions.
  - The deployment environment must be simulated by a controllable *harness*.

Test Scaffolding

Test scaffolding is a set of programs written to support test automation.

- Not part of the product
- Often temporary

Allows for:

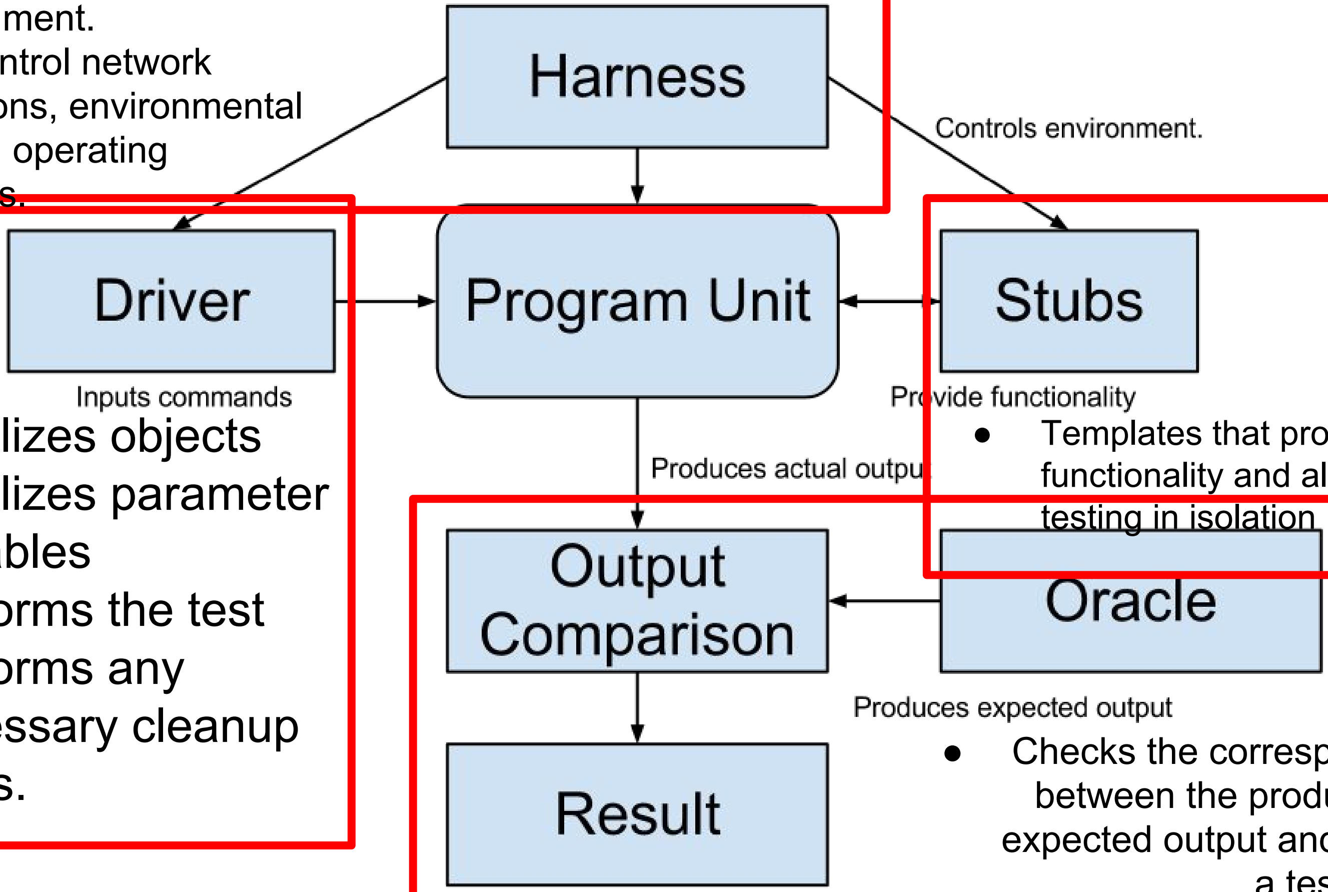
- Testing before all components complete.
- Testing independent components.
- Control over testing environment.

Test Scaffolding

- A **driver** is a substitute for a main or calling program.
  - Test cases are drivers.
- A **harness** is a substitute for all or part of the deployment environment.
- A **stub** (or **mock object**) is a substitute for system functionality that has not been completed.
- Support for recording and managing test execution.

Test Scaffolding

- Simulates the execution environment.
- Can control network conditions, environmental factors, operating systems.



- Initializes objects
- Initializes parameter variables
- Performs the test
- Performs any necessary cleanup steps.

- Templates that provide functionality and allow testing in isolation

- Checks the correspondence between the produced and expected output and renders a test verdict.

Writing an Executable Test Case

- Test Input
  - Any required input data.
- Expected Output (Test Oracle)
  - What *should* happen, i.e., values or exceptions.
- Initialization
  - Any steps that must be taken before test execution.
- Test Steps
  - Interactions with the system (such as method calls), and output comparisons.
- Tear Down
  - Any steps that must be taken after test execution to prepare for the next test.

Writing a Unit Test

JUnit is a Java-based toolkit for writing executable tests.

- Choose a target from the code base.
- Write a “testing class” containing a series of unit tests centered around testing that target.

```
public class Calculator {
  public int evaluate (String
                  expression) {
    int sum = 0;
    for (String summand:
                  expression.split("\\+"))
      sum += Integer.valueOf(summand);
    return sum;
  }
}
```

Writing a Unit Test

```
public class Calculator {
  public int evaluate (String
              expression) {
    int sum =
    for (Stri
              expression.split("\
      sum += Integer.valueOf(summ
    return sum;
  }
}
```

```
import static
org.junit
import or

public class CalculatorTest {
  @Test
  public void evaluatesExpression() {
    Calculator calculator =
          new Calculator();
    int sum =
          calculator.evaluate("1+2+3");
    assertEquals(6, sum);
     calculator = null;
  }
}
```

Convention - name the test class after the class it is testing or the functionality being tested.

Each test is denoted with keyword **@test**.

Initialization

Test Steps

Input

Oracle

Tear Down

Test Fixtures - Shared Initialization

@Before annotation defines a common test initialization method:

```
@Before
public void setUp() throws Exception
{
   this.registration = new Registration();
   this.registration.setUser(“ggay”);
}
```

Test Fixtures - Teardown Method

@After annotation defines a common test tear down method:

```
@After
public void tearDown() throws Exception
{
   this.registration.logout();
   this.registration = null;
}
```

More Test Fixtures

- @BeforeClass defines initialization to take place before any tests are run.
- @AfterClass defines tear down after all tests are done.

```
@BeforeClass
  public static void setUpClass() {
     myManagedResource = new
          ManagedResource();
  }

  @AfterClass
  public static void tearDownClass()
throws IOException {
    myManagedResource.close();
    myManagedResource = null;
  }
```

Test Skeleton

@Test annotation defines a single test:

```
@Test
public void test<MethodName><TestingContext>() {
   //Define Inputs
   try{ //Try to get output.
   }catch(Exception error){
      fail("Why did it fail?");
   }
   //Compare expected and actual values through
assertions or through if statements/fails
}
```

Assertions

Assertions are a "language" of testing - constraints that you place on the output.

- assertEquals, assertArrayEquals
- assertFalse, assertTrue
- assertNull, assertNotNull
- assertSame,assertNotSame
- assertThat

assertEquals

```
@Test
public void testAssertEquals() {
   assertEquals("failure - strings are not
equal", "text", "text");
}

@Test
public void testAssertArrayEquals() {
    byte[] expected = "trial".getBytes();
    byte[] actual = "trial".getBytes();
    assertArrayEquals("failure - byte arrays
not same", expected, actual);
}
```

- Compares two items for equality.
- For user-defined classes, relies on `.equals` method.
  - Compare field-by-field
  - `assertEquals(studentA.getName(), studentB.getName())` rather than `assertEquals(studentA, studentB)`
- assertArrayEquals compares arrays of items.

assertFalse, assertTrue

```
@Test
public void testAssertFalse() {
   assertFalse("failure - should be false",
(getGrade(studentA, “CSCE747”).equals(“A”));
}

@Test
public void testAssertTrue() {
     assertTrue("failure - should be true",
(getOwed(studentA) > 0));
}
```

- Take in a string and a boolean expression.
- Evaluates the expression and issues pass/fail based on outcome.
- Used to check conformance of solution to expected properties.

assertSame, assertNotSame

```
@Test
public void testAssertNotSame() {
   assertNotSame("should not be same Object",
studentA, new Object());
}

@Test
public void testAssertSame() {
    Student studentB = studentA;
   assertSame("should be same", studentA,
studentB);
}
```

- Checks whether two objects are clones.
- Are these variables aliases for the same object?
  - assertEquals uses .equals().
  - assertSame uses ==

assertNull, assertNotNull

```
@Test
public void testAssertNotNull() {
    assertNotNull("should not be null", new
Object());
}

@Test
public void testAssertNull() {
    assertNull("should be null", null);
}
```

- Take in an object and checks whether it is null/not null.
- Can be used to help diagnose and void null pointer exceptions.

assertThat

```
@Test
public void testAssertThat{
  assertThat("albumen", both(containsStr
  assertThat(Arrays.asList("one", "two",
  assertThat(Arrays.asList(new String[] { "fur                                    ));
  assertThat("good", allOf(equalTo("good")
  assertThat("good", not(allOf(equalTo("ba
  assertThat("good", anyOf(equalTo("bad"), e
  assertThat(7, not(CombinableMatcher.<Integer> either(equalTo(3)).or(equalTo(4))));
}
```

both - two properties must be met

has items - a list contains an indicated subset
ms.

everyItem - all items in list must
m

allOf - all listed properties must be true

not(allOf(...)) - if all of these properties

anyOf - at least one of the listed

either - pass if one of these properties is true.

Testing Exceptions

- When testing error handling, we expect exceptions to be thrown.
- In JUnit, we can ensure that the right exception is thrown.

```
@Test(expected = IndexOutOfBoundsException.class)
public void empty() {
     new ArrayList<Object>().get(0);
}
```

Testing Exceptions - Rules

- Rules can be used to encapsulate repeated test behavior.
  - Such as ensuring that the right exception is thrown.
- In the test, state which exception is expected and examine its stack trace.

```
@Rule
public ExpectedException thrown =
ExpectedException.none();

@Test
public void testExceptionMessage() throws
IndexOutOfBoundsException {
 List<Object> list = new ArrayList<Object>();
 thrown.expect
           (IndexOutOfBoundsException.class);
 thrown.expectMessage("Index: 0, Size: 0");
 list.get(0);
}
```

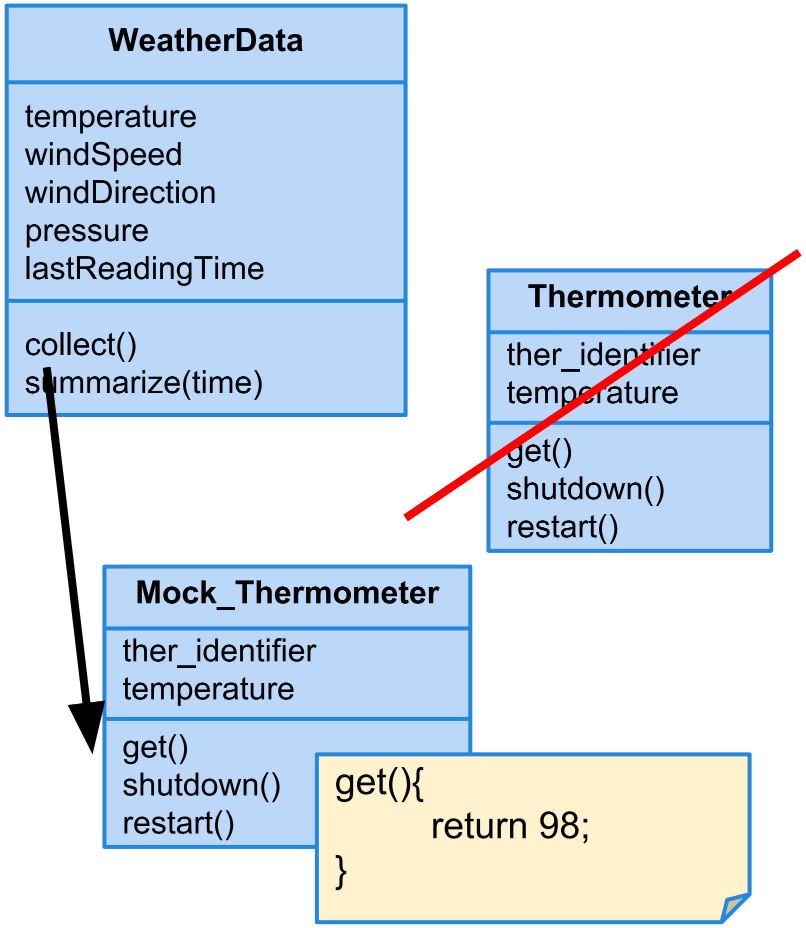
Scaffolding

- Stubs and drivers are code written as replacements other parts of the system.
  - May be required if pieces of the system do not exist.
- Scaffolding allows greater control over test execution and greater observability to judge test results.
  - Ability to simulate dependencies and test components in isolation.
  - Ability to set up specialized testing scenarios.
  - Ability to replace part of the program with a version more suited to testing.

Object Mocking

Components may depend on other, unfinished (or untested) components. You can **mock** those components.

- Mock objects have the same interface as the real component, but are hand-created to simulate the real component.
- Can also be used to simulate abnormal operation or rare events.



Replacing Interfaces

- Scaffolding can be complex - can replace any portion of the system.
- If an interface does not allow control or observability - write scaffolding to replace it.
  - Allow inspection of previously-private variables.
  - Replace a GUI with a machine-usable interface.
  - May be useful after testing.
    - Expose a command-line interface for scripting.

Generic vs Specific Scaffolding

- Simplest driver - one that runs a single specific test case.
- More complex:
  - Common scaffolding for a set of similar tests cases,
  - Scaffolding that can run multiple test suites for the same software (i.e., load a spreadsheet of inputs and run then).
  - Scaffolding that can vary a number of parameters (product family, OS, language).
- Balance of quality, scope, and cost.

Activity - Unit Testing

You are testing the following method:

```
public double max(double a, double b);
```

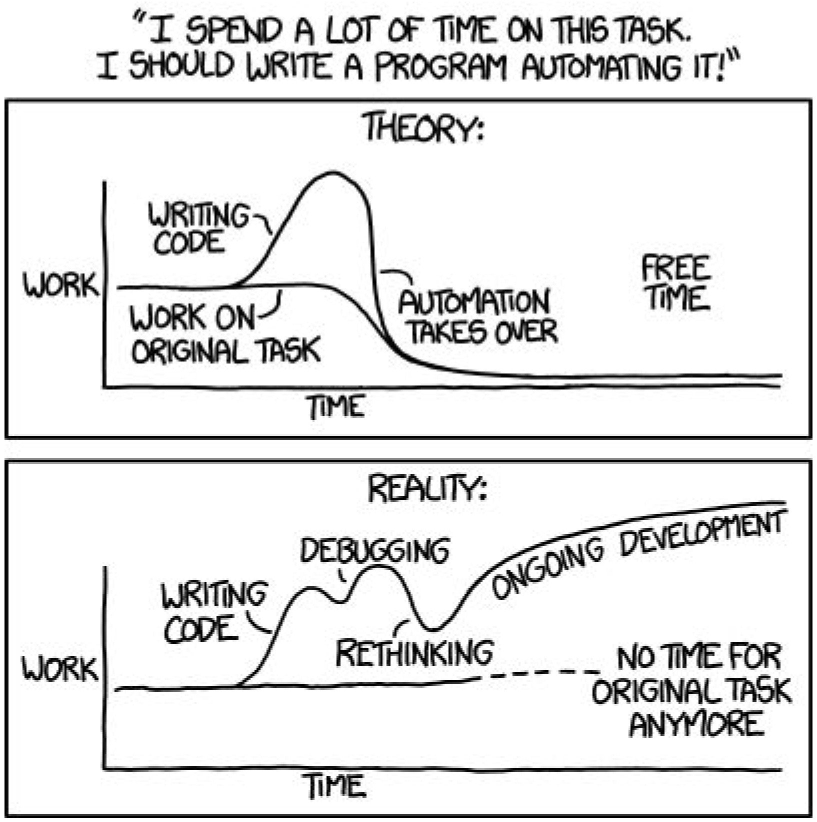
Devise three executable test cases for this method in the JUnit notation. See the attached handout for a refresher on the notation.

Activity Solution

```
@Test
  public void aLarger() {
    double a = 16.0;
    double b = 10.0;
    double expected = 16.0;
    double actual = max(a,b);
    assertTrue("should be larger", actual>b);
    assertEquals(expected, actual);
  }
@Test
  public void bLarger() {
    double a = 10.0;
    double b = 16.0;
    double expected = 16.0;
    double actual = max(a,b);
    assertThat("b should be larger", b>a);
    assertEquals(expected, actual);
  }
```

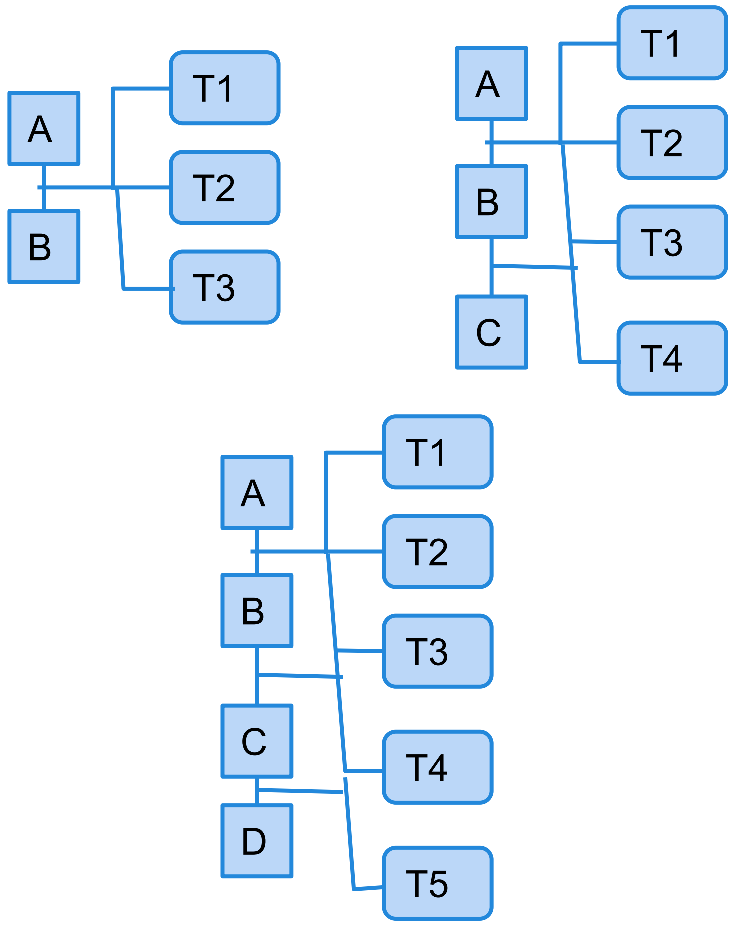
```
@Test
  public void bothEqual() {
    double a = 16.0;
    double b = 16.0;
    double expected = 16.0;
    double actual = max(a,b);
    assertEquals(a,b);
    assertEquals(expected, actual);
  }
@Test
  public void bothNegative() {
    double a = -2.0;
    double b = -1.0;
    double expected = -1.0;
    double actual = max(a,b);
    assertTrue("should be negative",actual<0);
    assertEquals(expected, actual);
  }
```

Automation Trade-Offs



Some common strategies help guide automation.

Incremental Testing



Test pieces of the system as they are completed. Use scaffolding (stubs, drivers) to test in isolation, then swap out for real components to test integration.

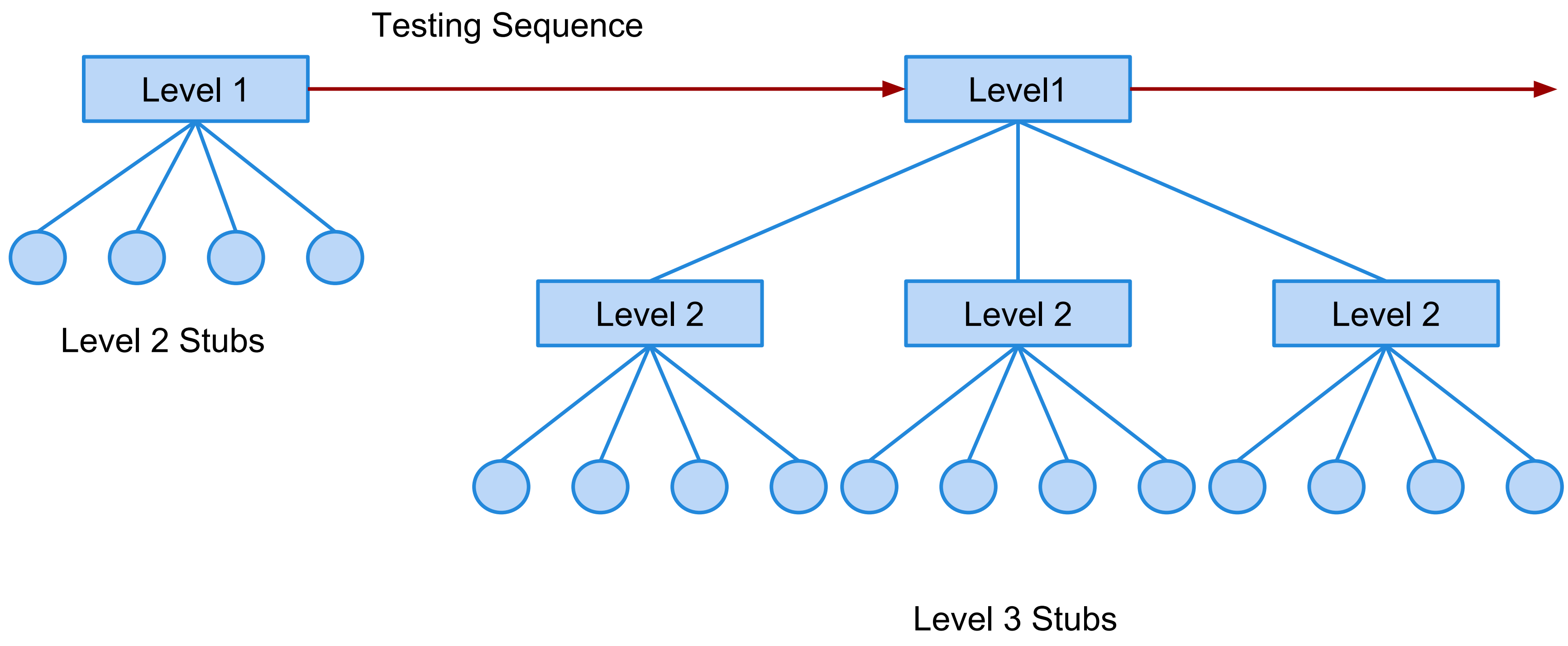
Advantages:

- Easily test components in isolation.
- Discover faults earlier.

Disadvantage:

- Expensive to develop scaffolding.

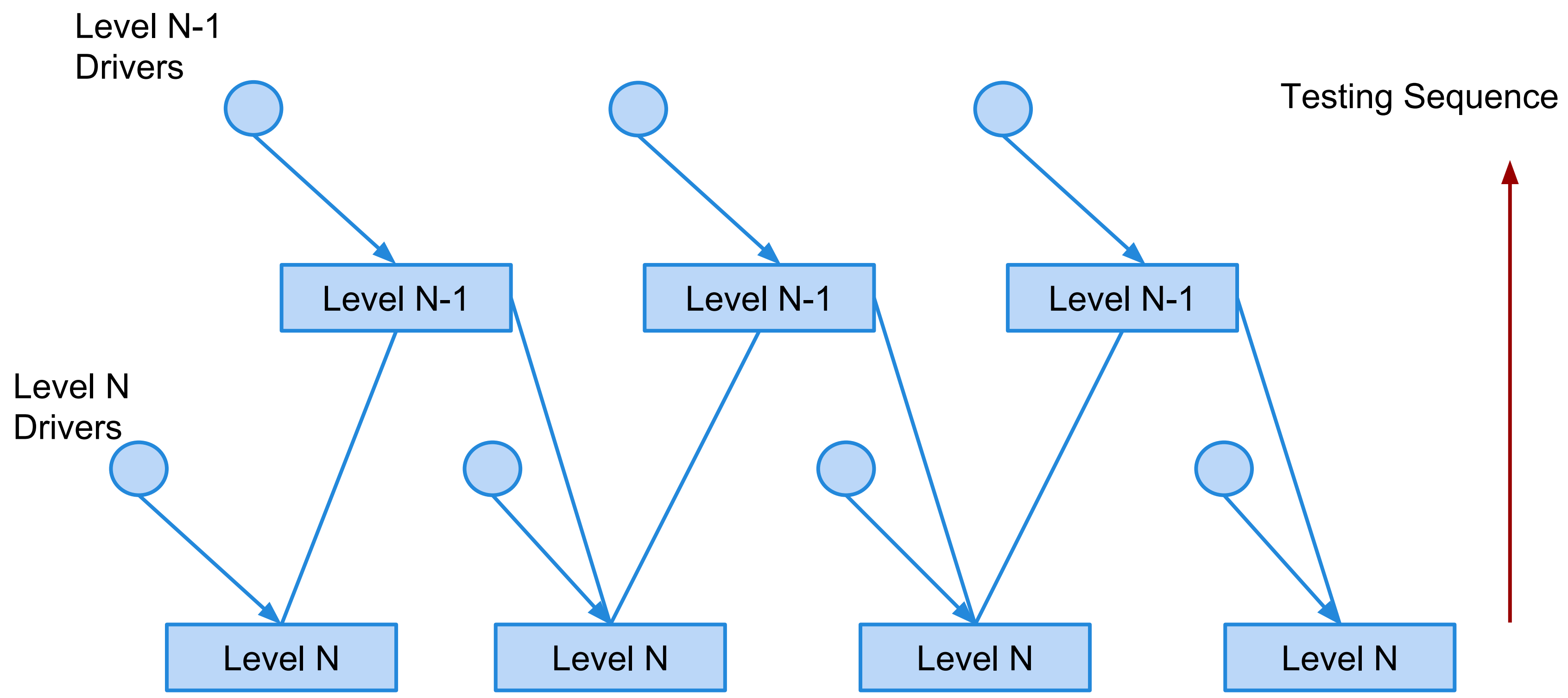
Top-Down Testing



Top-Down Testing

- Start with the high levels of a system (based on control-flow, data-flow, or architecture) and work your way downwards.
  - Use in conjunction with top-down development.
- Very good for finding architectural or integration errors.
- May need system infrastructure in place before testing is possible.
- Requires large effort in developing stubs.

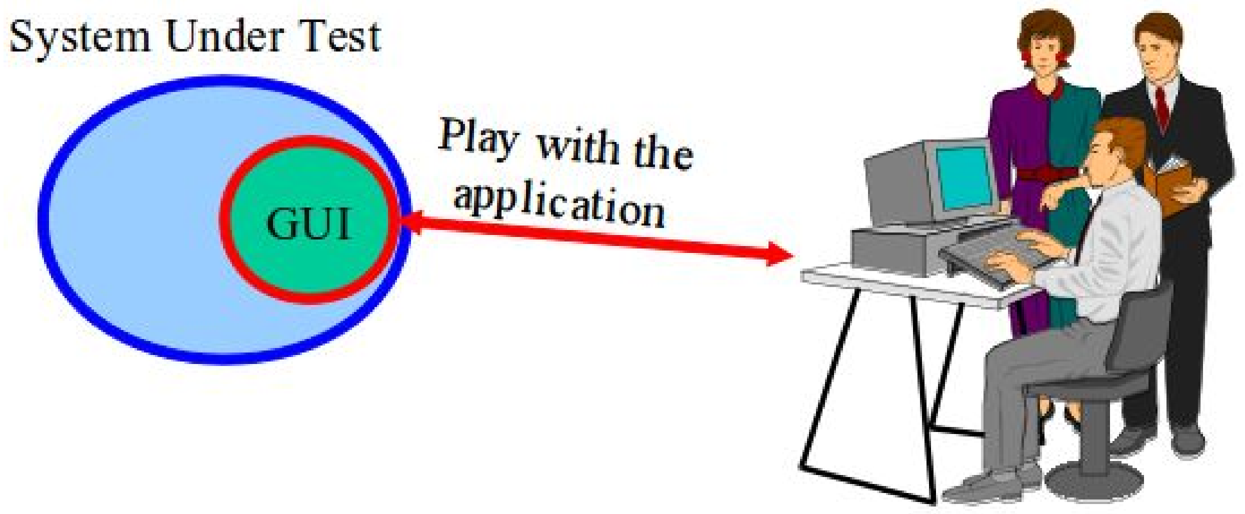
Bottom-Up Testing



Bottom-Up Testing

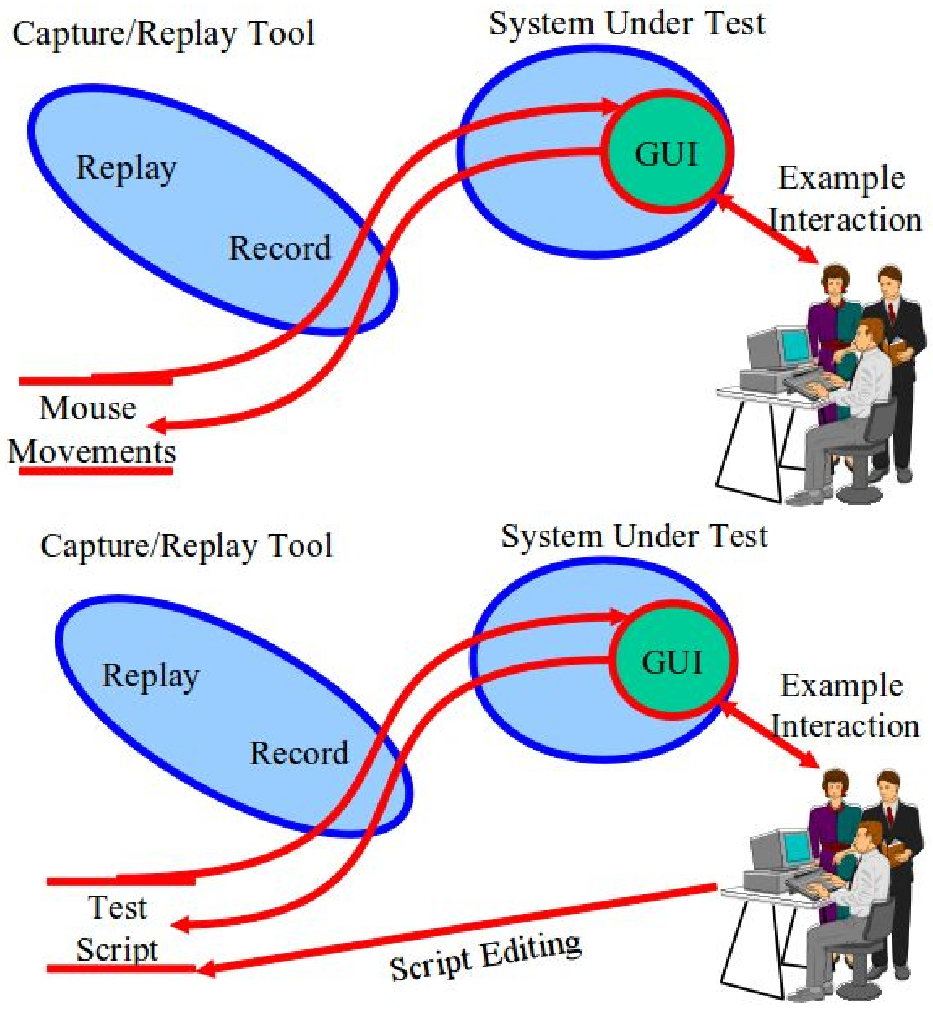
- Start with the lower levels of a system (based on control-flow, data-flow, or architecture) and work your way upwards.
  - Use in conjunction with bottom-up development.
- Appropriate for object-oriented systems.
- Necessary for testing critical infrastructure.
- Does not find major design problems, but very good at testing individual components.
- Requires high effort in developing drivers.

What About Graphical Interfaces?



- Graphical components of projects often tested manually by real users.
- Heavily tested during alpha/beta testing.

Capture and Replay



1. Have a human interact with the system, walking through several different scenarios.
2. Record their mouse motions and clicks during these scenarios.
3. Take these test cases and modify them to create additional tests.

Capture and Replay

- Common test automation method:
  - Have a human do something once.
  - Let the computer take the same actions.
    - Allows retesting without additional human involvement as long as interface is unchanged.
- Often can be used to create additional tests:
  - Transform their actions into a script.
  - Encode a series of *transformations* that can be automatically invoked on the script.
  - Requires an oracle, but can rely on generic oracles.

Continuous Integration

- Development practice that requires code be frequently checked into a shared repository.
- Each check-in is then verified by an automated build.
  - The system is compiled and subjected to an automated test suite, then packaged into a new executable.
- By integrating regularly, developers can detect errors quickly, and locate them more easily.

CI Practices

- Maintain a code repository.
- Automate the build.
- Make the build self-testing.
- Every commit should be built.
- Keep the build fast.
- Test in a clone of the production environment.
- Make it easy to get the latest executable.
- Everyone can see build results.
- Automate deployment.

How Integration is Performed

- Developers check out code to their machine.
- Changes are committed to the repository.
- The CI server:
  - Monitors the repository and checks out changes when they occur.
  - Builds the system and runs unit/integration tests.
  - Releases deployable artefacts for testing.
  - Assigns a build label to the version of the code.
  - Informs the team of the successful build.

How Integration is Performed

- If the build or tests fail, the CI server alerts the team.
  - The team fixes the issue at the earliest opportunity.
  - Developers are expected not to check in code they know is broken.
  - Developers are expected to write and run tests on all code before checking it in.
  - No one is allowed to check in while a build is broken.
- Continue to continually integrate and test throughout the project.

We Have Learned

- Test automation can be used to lower the cost and improve the quality of testing.
- Automation involves creating drivers, harnesses, stubs, and oracles.
- Systems can be tested in a top-down or bottom-up style.
- Automated testing enables continuous integration and deployment.

Next Time

- Unit testing lab - let's get our hands dirty.
  - You will need a laptop with a Java IDE installed (and jUnit).
  - Individual assignment - turn in by March 17, 11:59 PM.
- Assignment 3
  - Out now. Due March 28
  - Complete the unit testing lab first.