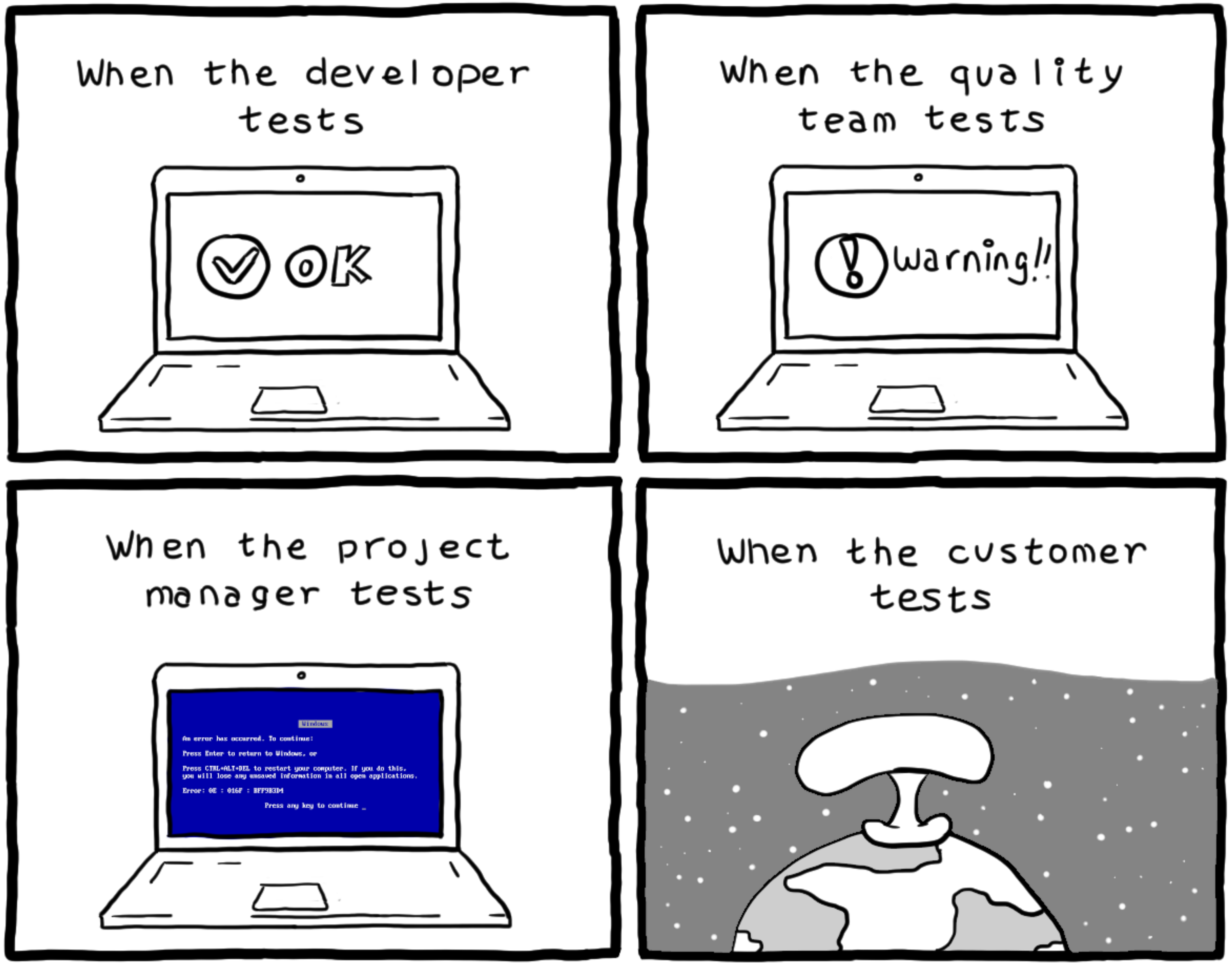

When the developer tests
OK
When the quality team tests
warning!!
When the project manager tests
Windows
An error has occurred. To continue:
Press Enter to return to Windows, or
Press CTRL+ALT+DEL to restart your computer. If you do this, you will lose any unsaved information in all open applications.
Error: 0E : 016F : BFF9B3D4
Press any key to continue _
When the customer tests

Software Testing
Why, What, Who, and When

CSCI 2134: Software Development

Agenda

- Lecture Contents
  - Motivation
  - What is testing
  - Types of testing
  - When to test
  - Scaffolding
- Brightspace Quiz
- Readings:
  - This Lecture: Chapter 22
  - Next Lecture: Chapter 23

Detecting Software Defects (Bugs)

We have many techniques for detecting defects: **Which one do we use?**

- **Informal design reviews** 🔵🟢: Informal team discussion of major design decisions
- **Formal design inspections** 🔵🟢: A review of each design decision, where each person has a specific role
- **Informal code reviews** 🟡: Informal team look-over of the one or more pieces of code
- **Formal code inspection** 🟡: A line-by-line review of code, where each person has a specific role
- **Personal desk-checking of code** 🟡: Individual review of personal code
- **Modeling or prototyping** 🟢🟡: Creation of proof-of-concept or simplified versions of the system to verify design decisions
- **Unit test** 🟡🟠🔴🟣: Execution of a complete class, routine, or small program, which is tested in isolation from the more complete system
- **New function (component) test** 🟡🟠🔴🟣: Execution of a class, package, small program, or other program element that involves the work of multiple programmers or programming teams, which is tested in isolation from the more complete system
- **Integration test** 🟡🟠🔴🟣: Combined execution of two or more classes, packages, components, or subsystems that have been created by multiple programmers or programming teams
- **System test** 🟡🟠🔴🟣: Execution of the software in its final configuration, including integration with other software and hardware systems.

Use Combinations of Techniques

- Different techniques are useful for finding different defects in different stages of development
- None of these techniques are 100% effective on their own
- In combination, these techniques are 95% effective (McConnel)

Table 20-2 Defect-Detection Rates

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%
Formal code inspections	45%	60%	70%
Modeling or prototyping	35%	65%	80%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Low-volume beta test (<10 sites)	25%	35%	40%
High-volume beta test (>1,000 sites)	60%	75%	85%

Source: Adapted from *Programming Productivity* (Jones 1986a), "Software Defect-Removal Efficiency" (Jones 1996), and "What We Have Learned About Fighting Defects" (Shull et al. 2002).

(McConnell, "Code Complete 2nd Edition, 2004, pg 470)

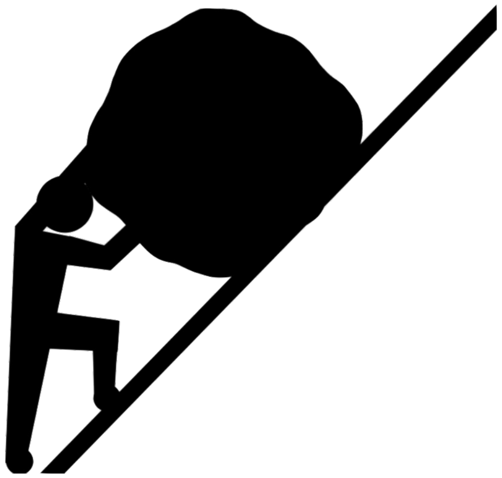
Testing vs Debugging

- Testing is the process of detecting defects
- Debugging is the process of
  - **Diagnosing** (find) the cause of a defect
  - **Correcting** (remove) defects
- Before we can debug, we need to know that there is a bug
- Testing provides an automated way of finding bugs

Testing So Far

- In CSCI 1110
  - Programs were small (a couple classes)
  - Test suites were provided
  - If code passed test suites, it was assumed to be working
- Going forward
  - Programs get bigger, much bigger
  - Test suites may not be provided (you will need to create your own)
  - You would like to be sure that your code is working
- **We need to learn how to test**

Challenges of Testing



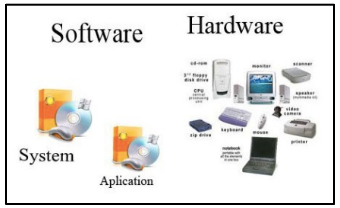
- **Different mindset**: You're trying to break it rather than create it
- **No guarantee**: Can't prove the absence of defects
  - Well, not economically in just about all cases
- **No improvement in quality**: Does not (on its own) improve software
  - It shows when quality failed
- **Counter intuitive**: Asked to find errors in code that you thought to be correct
  - If you knew there was an error there, you would have fixed it already
  - You (typically) look for errors where you spent your most time, which may well have fewer errors because of that attention
- **Variety of tests:**
  - Different kinds of testing techniques at different stages of development
  - Need to know when to use what tests

Types of Testing

- Unit testing
  Code of one developer
  - Blackbox testing
  - Graybox testing
  - Whitebox testing

- Functional / component testing
  code for one class or package from multiple developers
- Integration testing
  code for 2+ classes coming together

- Regression testing
  Re-doing past tests
- System testing
  Software in its final configuration

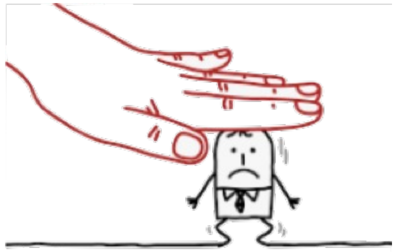


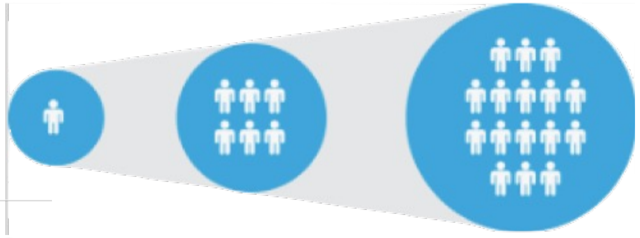
- Alpha testing
- Beta testing

- Acceptance testing

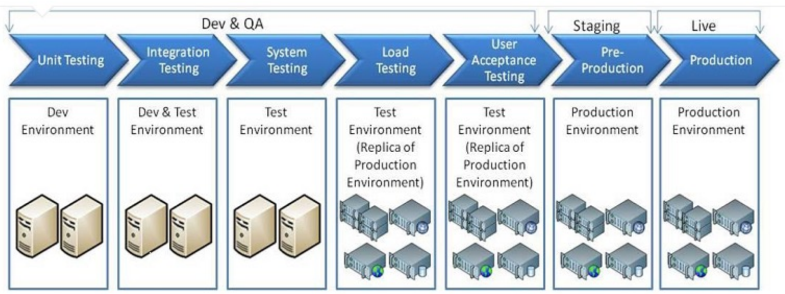
Other Kinds of Testing

- Performance testing
  - Load testing
  - Stress testing
- Scalability testing
- Usability testing
- Security testing
- Reliability testing
- Recovery testing
- Compatibility testing





Testing Environments



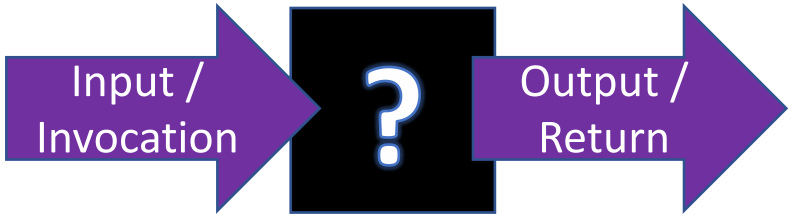
Testing / Environment	Unit Testing	Component Testing	Integration Testing	Regression Testing	System Testing	Deployment
Personal workstation • Personal development environment	✅	✅	✅ Depending on size	✅		
Integration server • Standard development environment	✅	✅	✅	✅		
Test server • Similar to production environment • Fixed dummy data for regression testing				✅	✅	
Preproduction server • Mirror of live data					✅	✅
Production server • Live data						✅

Unit Testing

- **Scope**: Small units of code
  - Methods and functions
  - Simple classes
  - Typically written by a single developer
- **Goal**: Ensure the small simple building blocks work
  - Easiest to do because they focus on small chunks of code
- **Principle**: In most cases smaller pieces of code are easier to test
- Three types, depending on how much is known about the implementation

  Tests are based on:
  - **Black Box**: Strictly on specification or interface
  - **White Box**: Specification or interface, and the implementation
  - **Grey Box**: Specification or interface, and partial knowledge of the implementation

Black Box Testing



- Black box testing tests the code strictly based on what it is supposed to do
- This typically happens when:
  - Tests are created before the code is written
  - Someone other than the developer creates the tests
- Example test cases:
  - Insert into empty list
  - Insert into half-full list
  - Insert into full list
  - Other?
- Notice: The test cases are quite generic

Example of a method specification

- Signature:

```
boolean offerFirst(E e)
```

- Description:

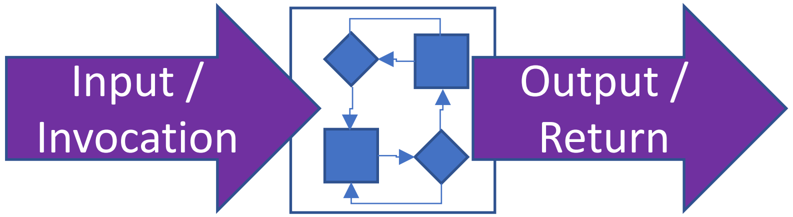
  Inserts the specified element at the front of this list unless it would violate capacity restrictions.

- Parameters:

  e - the element to add

- Returns: true if the element was added to this list, else false

White Box Testing

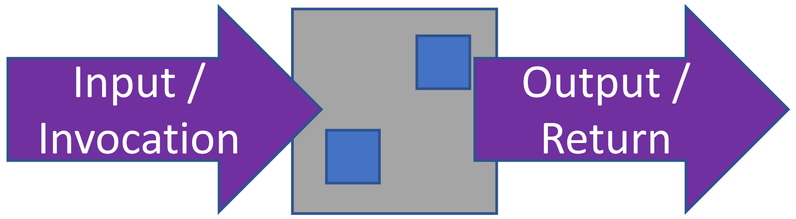


- White box testing tests the code with knowledge about the implementation
- This typically happens when:
  - Tests are created during or after the code is written
  - The developer creates the tests
- Example test cases:
  - Insert into empty list
  - Insert into half-full list
  - Insert into list with MAX elements
  - Insert into list whose current array is full
  - Other?
- Notice: The test cases are specific to this implementation

Example of a known implementation

```
boolean offerFirst(E e) {
  if (arr.length == num) {
    if (num == MAX) {
      return false;
    }
    int size = min(MAX, num*2);
    arr = duplicate(arr, size);
  }
  array[num++] = e;
  return true;
}
```

Grey Box Testing



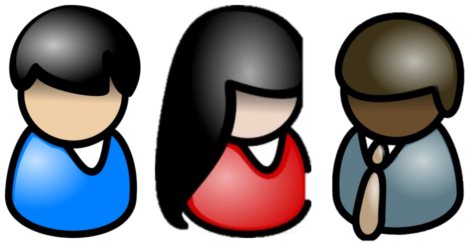
- Grey box testing tests the code with partial knowledge about the implementation
- This typically happens when:
  - Tests are created when the code is written
  - The implementation has changed or evolved
  - Someone other than the developer created the test.
- Example test cases:
  - Insert into empty list
  - Insert into half-full list
  - Insert into full list
  - Insert into a sufficiently large list to cause the implementation to grow the array
  - Other?
- Notice: The test cases make some assumptions about the implementation

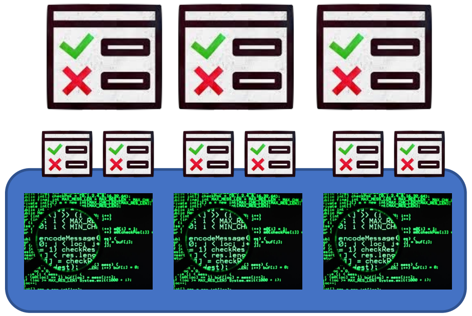
Example of a partially known implementation

```
interface ArrayBasedList {
  boolean offerFirst(E e);
}
```

Component Testing

- Component testing involves code that was worked on by multiple developers
  - All developers are not equally familiar with all the code
    grey box testing is more common
  - Different developers contribute different tests
  - Amount of code is larger
  - More interacting within the code to consider
  - More test cases are required
- The fundamental difference between unit testing and component testing is scale



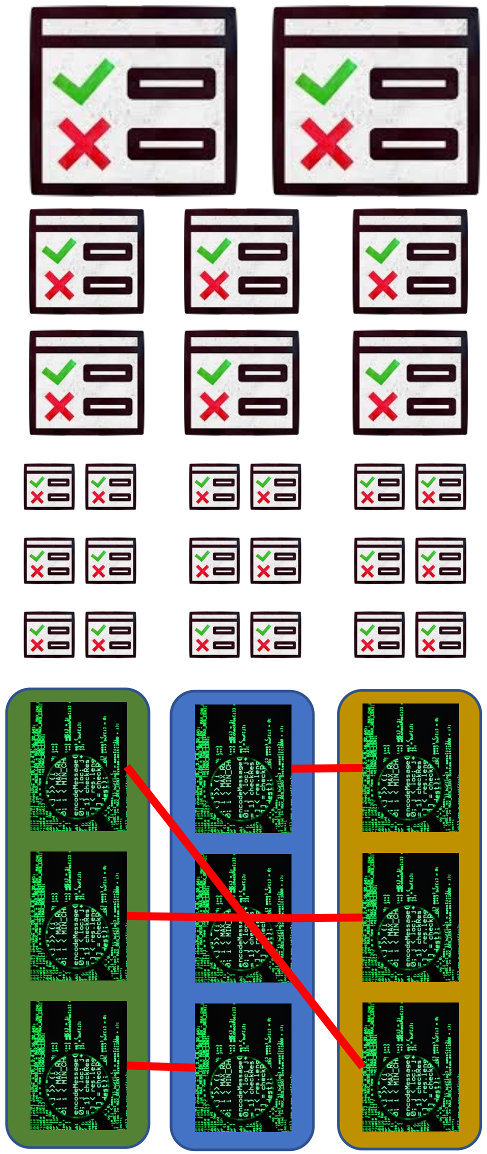


Integration Testing

- Integration testing involves multiple components and the interactions between them
  - All developers are not equally familiar with all the code

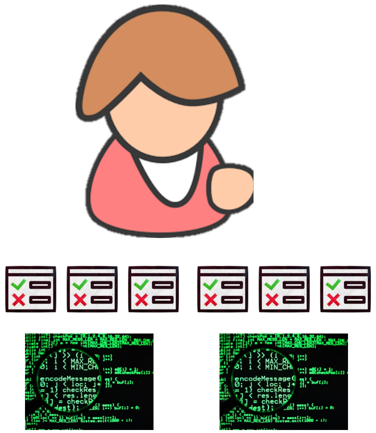
    black box and grey box testing is common
  - All unit and component tests are performed as part of the integration tests

    Components and units must work individually as well as together
  - Focus is on interactions between components
  - Testing is typically done in a ”standard” environment on a server
    - Depending on size of project, some integration testing can be done on developers’ workstations
- At this stage *some* of the system components are tested together
- **Note:** Unit, component, and integration testing all fall on a continuum of complexity but essentially use the same techniques for creating and deploying tests

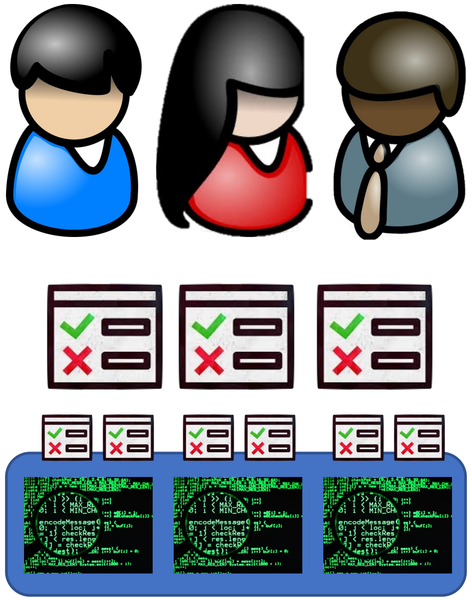


The Testing Continuum

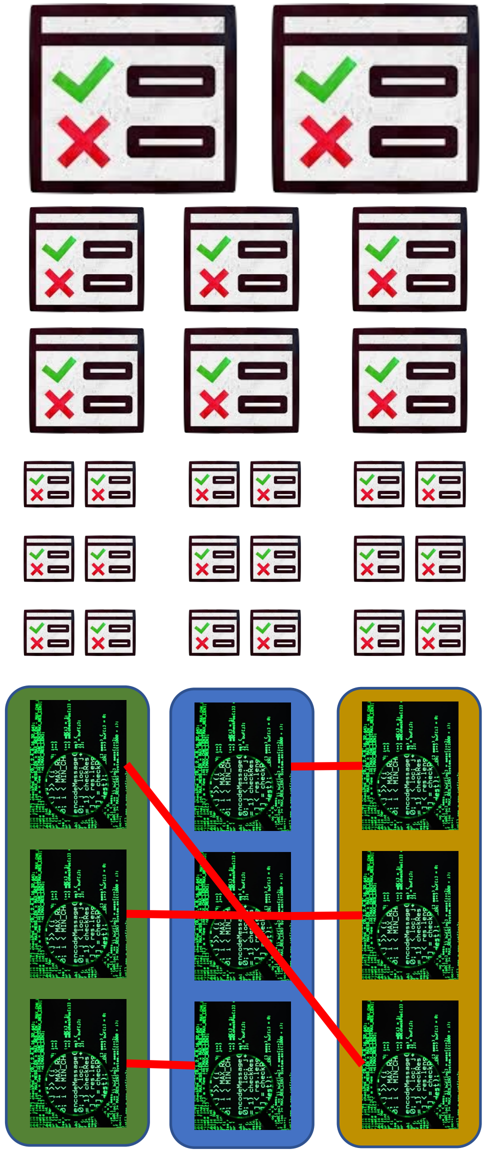
- The type of testing being done is the same in all three cases.
- The scale of the tests differs
- All are a mix of black box, grey box, and white box testing



Unit Testing



Component Testing

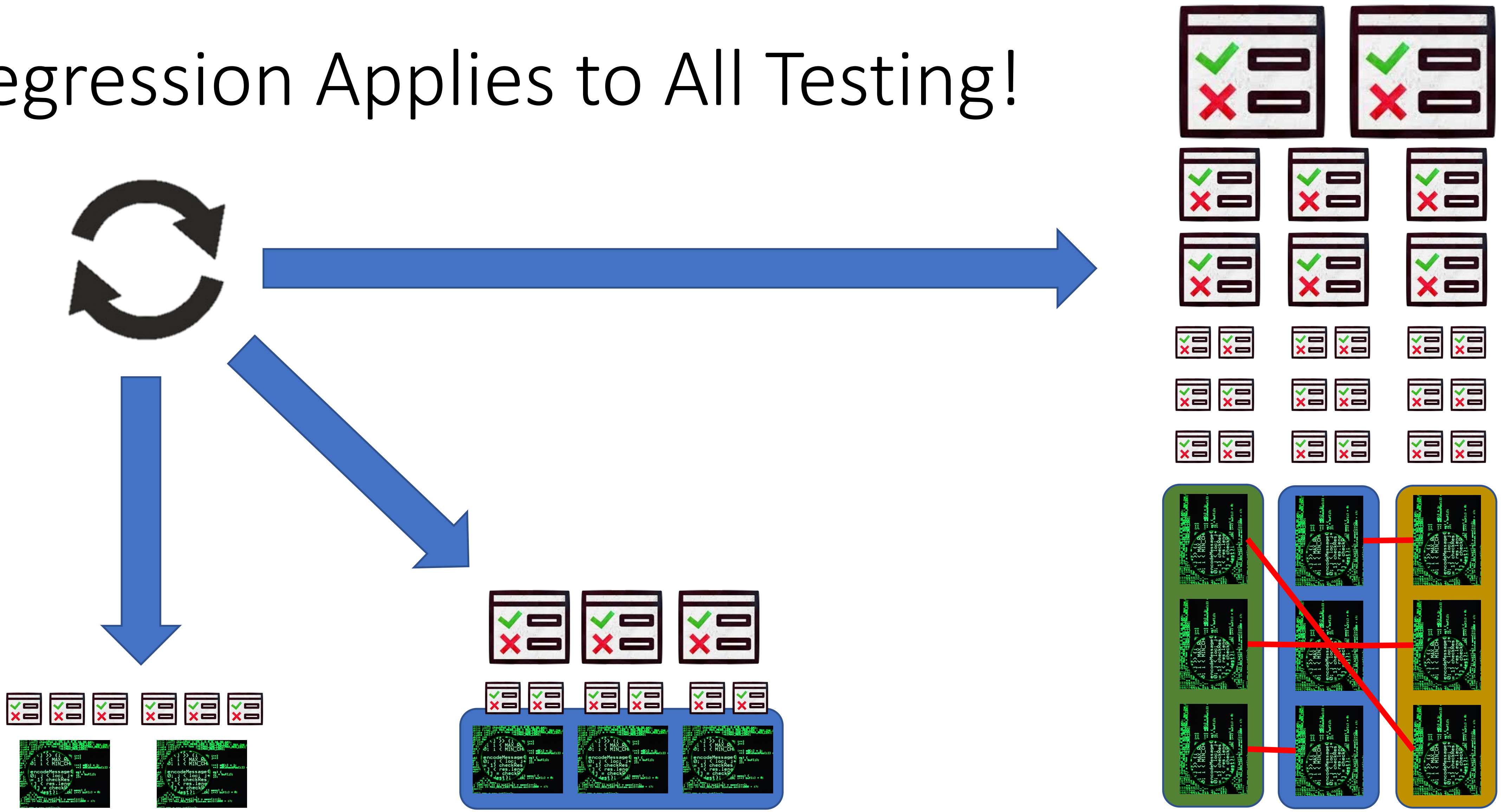


Integration Testing

Regression Testing

- **Definition:** *Regression testing* is the repetition of previously executed test cases for the purpose of finding defects in software that previously passed the same set of tests. (McConnel, “CC2”, 2004)
- **Key Idea:** Every time we make a change to the software we rerun all tests (unit, component, integration) to ensure no new defects were introduced
- Regression testing assures that we have not broken anything!
  - **Don’t commit broken code!**
- **Implementation**: Typically implemented as a set of scripts that can be run automatically by the developer and on the integration server
- **Key Idea**: Regression testing uses existing tests, not new tests

Regression Applies to All Testing!



Test Timing

- Two fundamental questions:
  - When should tests be created?
  - When should tests be performed?
- Observations:
  - We cannot perform tests before creating them
  - We cannot perform tests until we have something to test
- When should testing be performed?

  As early as possible!
- Why?

  The sooner the bugs are detected, the easier it is to fix!

Aside: Bug Density and Debugging Time

- The number of bugs is proportional to the size of the code base
  - After implementation there are typically 50 – 70 bugs per 1000 lines of code (yikes)
  - This can be described by the formula $\boldsymbol{B(n) = b \cdot n}$, where
    - $\boldsymbol{b}$ is a constant, e.g., 0.07
    - $\boldsymbol{n}$ is the size of the code base

    E.g., for a 2000 line program, we expect approximately $B(2000) = 0.07 \cdot 2000 = 140$ bugs
- The time to fix a bug is proportional to the size of the code base
  - Especially true for smaller programs
  - This can be described by the formula $\boldsymbol{T(n) = t \cdot n}$, where
    - $\boldsymbol{t}$ is a constant, e.g., 0.05
    - $\boldsymbol{n}$ is the size of the code base

    E.g., for a 2000 line program, we expect that it takes approximately $T(2000) = 0.05 \cdot 2000 = 100$ minutes to fix a bug
- **Total Debugging Time**: $\boldsymbol{D(n) = B(n) \cdot T(n) = b \cdot t \cdot n^2}$

Testing in CSCI 1110 and Other Courses

- Workflow
  - Write entire program
  - Test entire program
  - Debug entire program
  - Fix entire program

- Problems:
  - Bigger programs have more bugs
  - Debugging a larger chunk of code takes much more time than the cumulative time of debugging several small chunks

Testing Approaches

Test After Coding Entire Program

Codebase
n lines of code

Total debugging time: $D(n) = t \cdot b \cdot n^2$

D(1000) = 0.05 * 0.07 * 1000^2
= 3500

Test During Implementation (As you go)

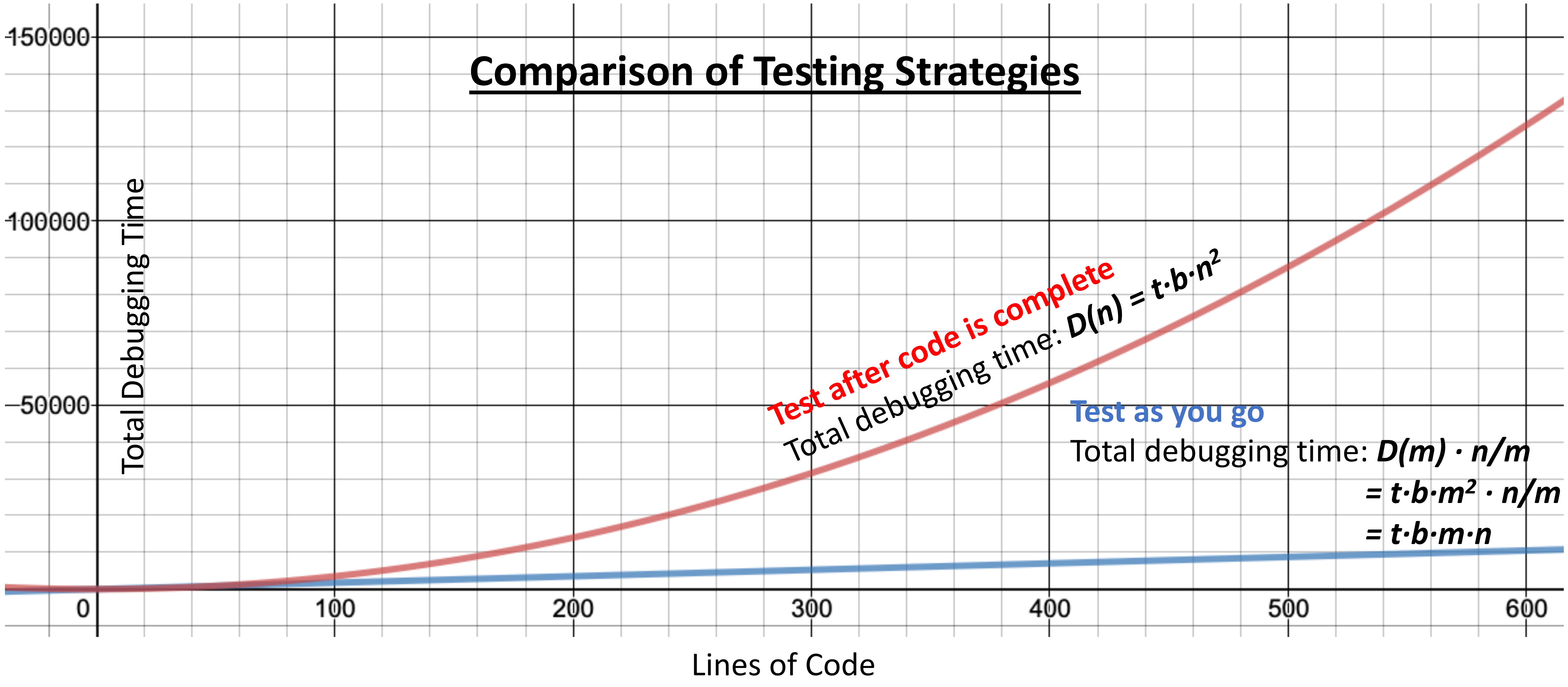
m lines of code (m < 50)
m lines of code (m < 50)
m lines of code (m < 50)
m lines of code (m < 50)
m lines of code (m < 50)
m lines of code (m < 50
m lines of code (m < 50)
m lines of code (m < 50)

n / m

Total debugging time: $D(m) \cdot n/m$
$= t \cdot b \cdot m^2 \cdot n/m$
$= t \cdot b \cdot m \cdot n$

D(1000) = 0.05 * 0.07 * 1000 * 50
= **175**

Testing Approaches (continued)



Test Early, Test Often

- Testing should bracket implementation
  - Develop test cases before you start coding based on unit specification (black/grey box)
  - Write the unit (method or small class) of code
  - Execute tests and debug unit
  - Add more whitebox tests if needed
- This is called **Test-Driven Development**:
  - Tests are based on the specifications
  - Dictate the development of the code
- Note:
  - Some testing will have to happen after the entire component is complete
  - This is why we differentiate between unit testing and component testing
- In the ideal world:
  - We can (and should) develop component tests prior to starting on the component
  - We can (and should) develop integration tests prior to starting on the integration

Why Test-Driven Development?

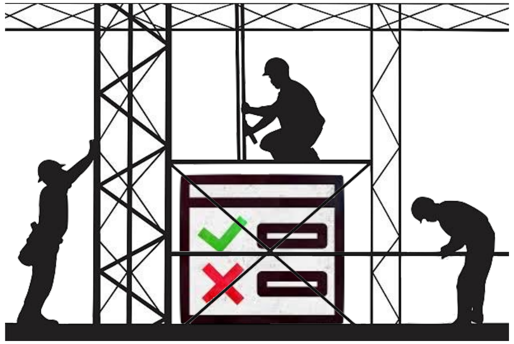
- Tend to detect defects earlier because you have identified the difficult cases in advance of writing the code
- Forces you to think a bit about the requirements and design early
- Exposes problems or ambiguities in the requirements earlier
- Doesn’t take any more effort before coding than after

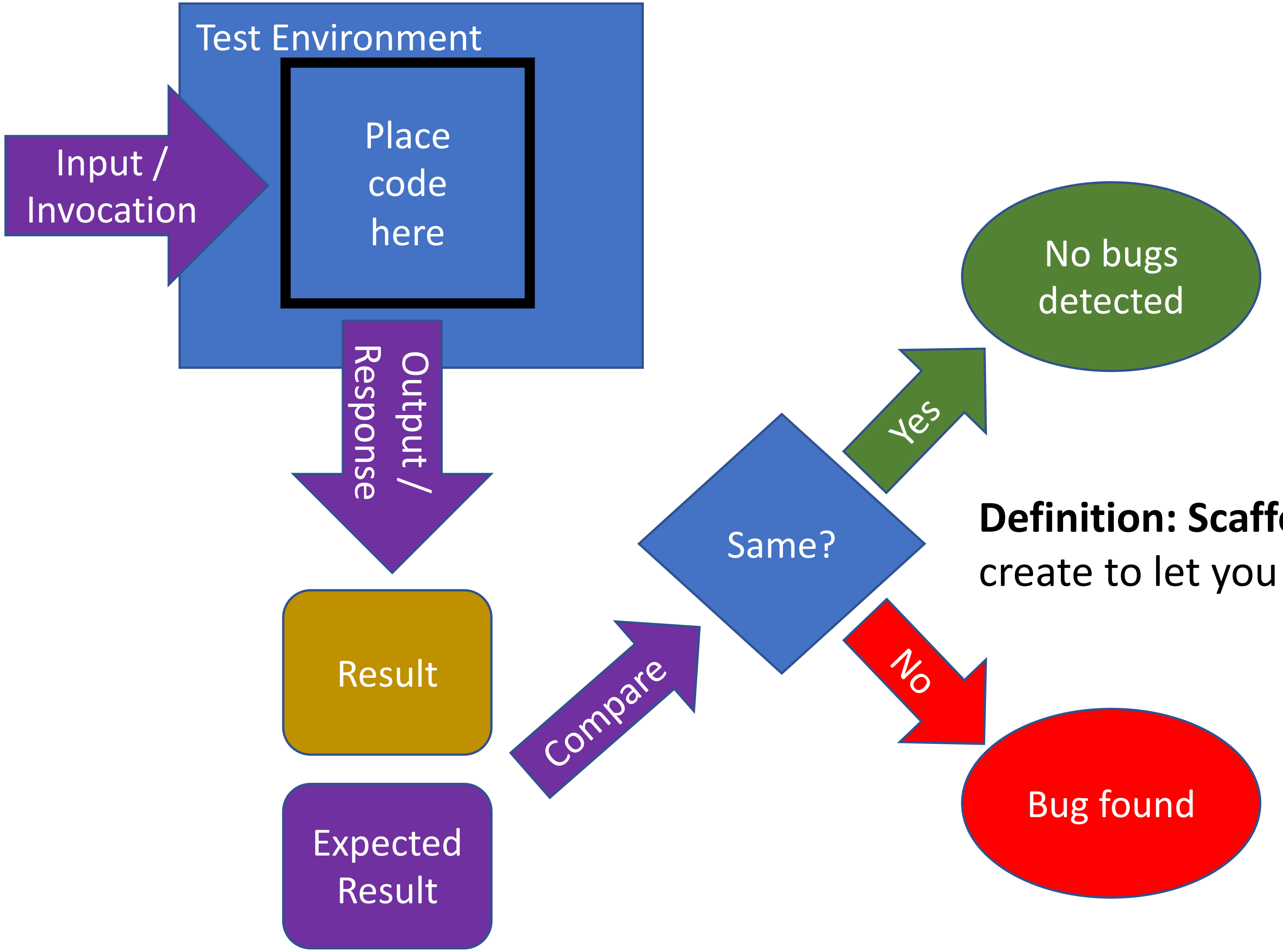
The Alternative

- Testing begins after coding completes
  - Approaching development sequentially
  - Want to get to the code first! (rightly or wrongly)

- **There is a reason why 75% of developer time is spent debugging.** ☹

Scaffolding





Definition: Scaffolding is the structure you create to let you run/test parts of your code

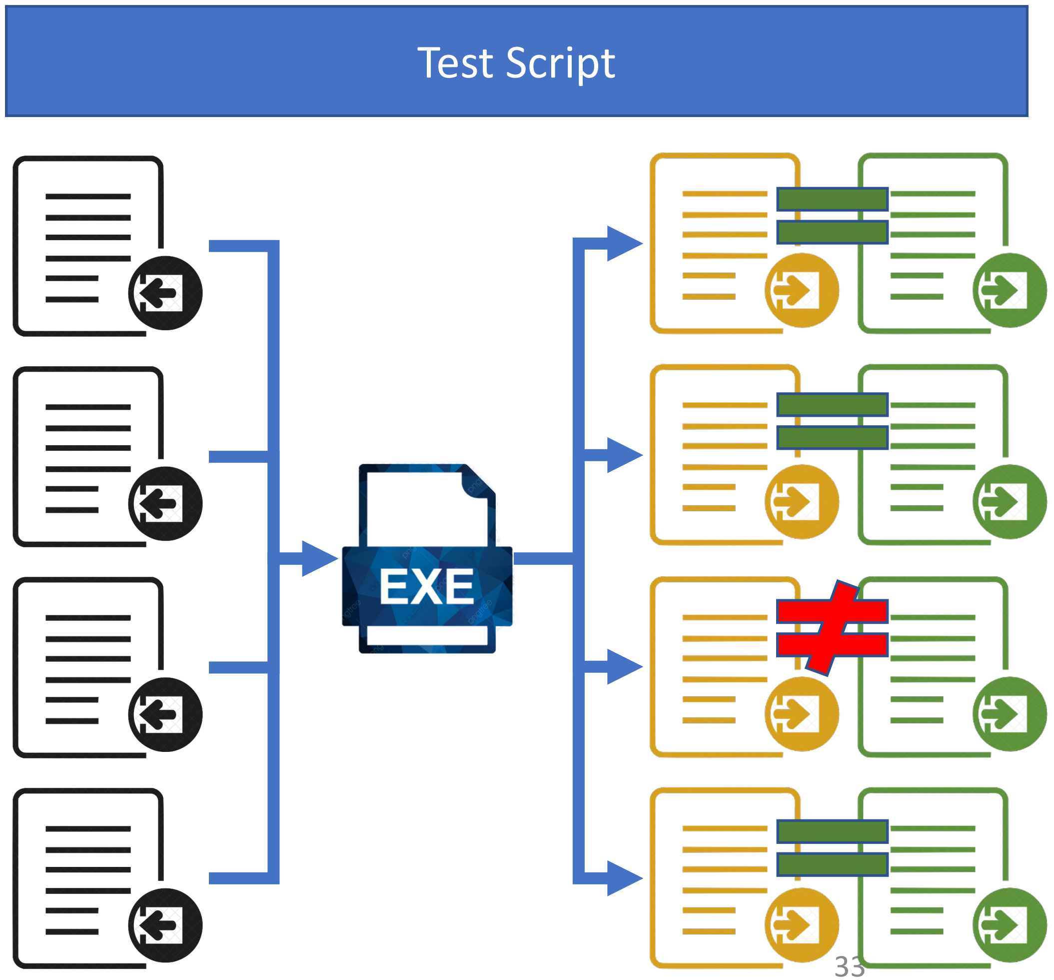
Types of Scaffolding for Testing

- Scripts
  - Shell scripts written to execute an entire program
  - E.g., Mimir does this in CSCI 1110
- Test harness / stubs
  - Additional code to run and support running of your code
  - Used to test classes and components that do not run on their own
  - E.g., The “Runner” or ”Demo” programs that you may have written in CSCI 1110
- Frameworks
  - Generalized test harnesses that can be integrated with various systems
  - E.g., JUnit

Testing Scripts

- Scripts are used when a full executable is being tested

  Either
  - The entire program
  - An executable consisting of
    - a test harness
    - the code to be tested
- A script-based test consists of:
  - Shell script (typically)
  - Executable
  - Input file
  - Expected output file



Example of a Test Script

```
#!/bin/sh
TESTS='00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19'

for T in $TESTS; do
    echo ==================================
    echo Test file: test.$T.in
    ./myprog < test.$T.in > test.$T.out
    if diff test.$T.out test.$T.gold ; then
        echo " " PASSED
    else
        echo " " FAILED
    fi
done
```

Loop through all tests

Run a test

Output file

Expected output file

Compare outputs

Input file

Test Harnesses and Stubs

- Test harnesses and stubs provide a way to test a component of a piece of software

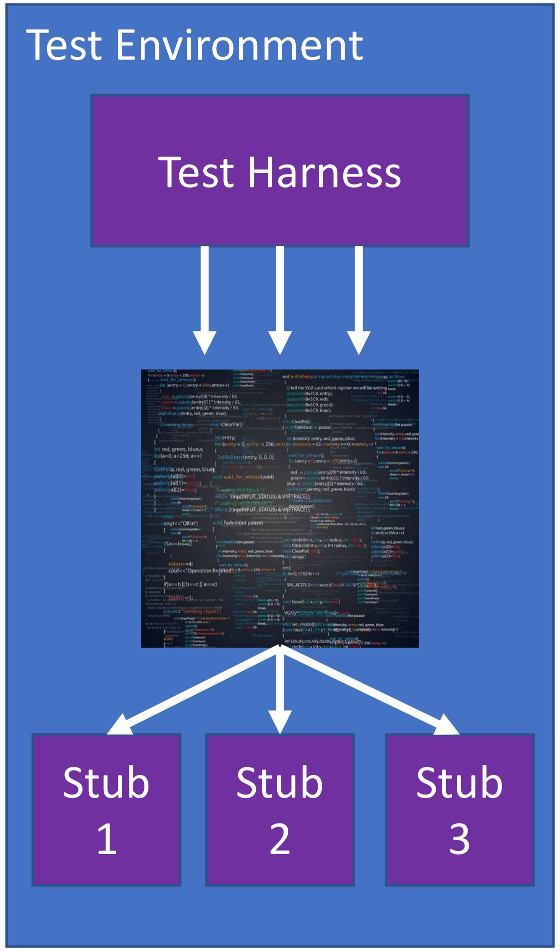
- Test harness
  - Makes calls to the code being tested
  - Typically has a main-line program
  - May or may not be driven by external input

  E.g., a Runner program that you may have implemented in CSCI 1110

- Stub
  - Small pieces of code that simulate code that your component may call while it is running
  - Typically used when the actual code is too complex, unfinished, or unpredictable to be provide known return values

  E.g., Using empty methods when implementing a class

- In many cases a combination of test harness, stubs, and scripts are used



Example of a Test Harness

MatrixTester.java

Matrix.java | MatrixTest.java | MatrixTester.java

```java
// Decode the command and execute it
switch( cmd ) {
case "new":   // new <h> <m>              Create a new zero matrix of
  h = s.nextInt();                     // height h and
  w = s.nextInt();                     // width w and
  matrices[mIdx] = new Matrix( h, w ); // assign to matrix <M>
  break;
case "clone": // clone <M> <N>         // Create a clone of
  matN = matrices[getMatrixIdx( s )];  // matrix <N> and
  if( matN != null ) {
    matrices[mIdx] = new Matrix( matN );  // Assign to matrix <M>
  } else {
    status = ERR;
  }
  break;
case "load": // load <M> ...           // Load the following matrix
  matrices[mIdx] = new Matrix( s );    // Into matrix <M>
  break;
case "add": // add <M> <N> <P>            <M> = <N> +<P>
  matN = matrices[getMatrixIdx( s )];  // Matrix <N>
  matP = matrices[getMatrixIdx( s )];  // Matrix <P>

  // If matrices are not allocated, or addition fails, set status to ERR
  if( ( matM == null ) || ( matN == null ) || ( matP == null ) ||
      ( matN.add( matP, matM ) == null ) ) {
    status = ERR;
  }
  break;
case "scale": // scale <M> s <N>          <M> = s <N>
  value = s.nextDouble();              // read in the scalar and
  matN = matrices[getMatrixIdx( s )];  // matrix <N>

  // If matrices not allocated, or multiplication fails, set status to ERR
  if( ( matM == null ) || ( matN == null ) ||
      ( matN.multiplyWithScalar( value, matM ) == null ) ) {
    status = ERR;
  }
  break;
case "mult":  // mult <M> <N> <P>         <M> = <N> x <P>
  matN = matrices[getMatrixIdx( s )];  // matrix <N>
  matP = matrices[getMatrixIdx( s )];  // matrix <P>

  // If matrices not allocated, or multiplication fails, set status to ERR
```

MatrixTester › main()

Matrix.java

Matrix.java

~/Teach/2134/Labs/Lab4/src/Matrix.java

```java
private double [][] matrix;   // 2D array stores matrix of size height x width
int height;                   // number of rows in the matrix
int width;                    // number of columns in the matrix

/**
 * Constructor creates a zero matrix of size m x n
 * Parameters:  m: height of the matrix
 *              n: width of the matrix
 */
public Matrix(int m, int n) {
    // Instantiate 2D array and initialize height and width fields.
    matrix = new double[m][n];
    height = m;
    width = n;
}

/**
 * Constructor duplicates the passed matrix
 * Parameters:  mtx: matrix to be cloned
 */
public Matrix(Matrix mtx) {
    // Get dimensions of matrix to be cloned and instantiate array.
    height = mtx.getHeight();
    width = mtx.getWidth();
    matrix = new double[height][width];
```

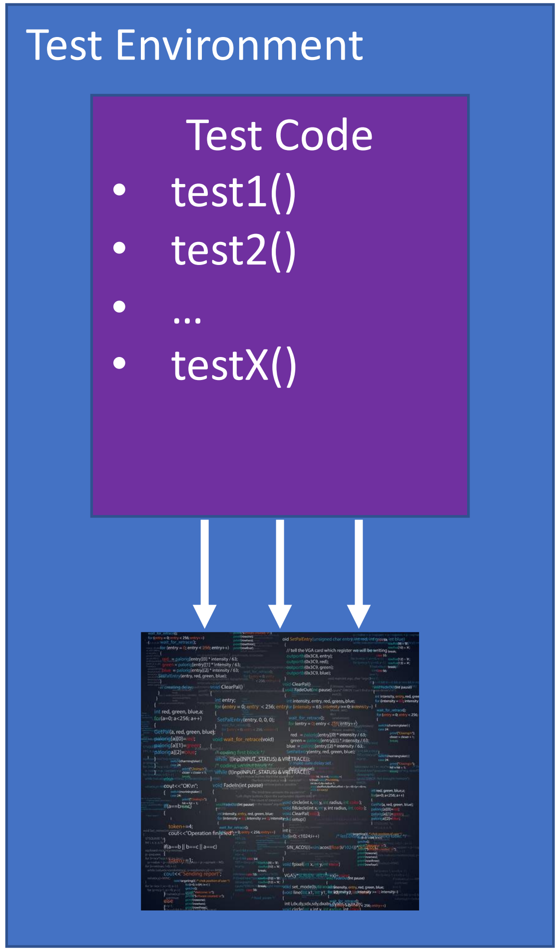
Matrix

Testing Frameworks

- The test-harness approach is so common that special libraries exist to provide test harnesses
- The programmer provides methods, each of which perform a test on the target code
- This requires the programmer to focus on the tests instead of writing code to support the tests
- Examples of Testing Frameworks

  Artos Arquillian AssertJ beanSpec BeanTest Cactus Concordion Concutest Cucumber-JVM Cuppa DbUnit EasyMock EtlUnit EvoSuite GrandTestAuto GroboUtils HavaRunner Instinct Java Server-Side Testing framework (JSST) JBehave JDave JExample JGiven JMock JMockit Jnario Jtest Jukito **JUnit** JUnitEE JWalk Mockito Mockrunner Needle NUTester OpenPojo PowerMock Randoop Spock SpryTest SureAssert Tacinga TestNG Unitils XMLUnit

  https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#Java



Example of JUnit Test for a *Matrix* class

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
import java.util.Scanner;

class MatrixTest {
    private final static String simpleMatrix = "2 2 1 2 3 4";    // [ 1 2 ]
                                                                   // [ 3 4 ]
    @Test
    void getElem() {
      Matrix m = new Matrix(new Scanner(simpleMatrix));
      assertEquals(2, m.getElem(1,2), "getElem() did not return correct value");
    }

    @Test
    void setElem() {
        Matrix m = new Matrix(new Scanner(simpleMatrix));
        m.setElem(2, 1, 5);
        assertEquals(5, m.getElem(2,1), "setElem() may not have set correct value");
    }
```

Regular Java class

Input for the tests

Each method is a test

Test the `getElem()` method

Create a *Matrix* object

Test `getElem()`

Test `setElem()`

Key Points

- Testing is a set of techniques that should be used in combination to detect defects
- Testing is challenging, requiring a different mindset but cannot verify that a piece of software is defect free
- Testing is performed at various granularities such as unit, component, integration, and system
- Blackbox testing is strictly based on the specification while whitebox testing incorporates knowledge of the implementation
- Regression testing reruns past tests to confirm the code was not broken
- It is beneficial to create and test in the course of development to catch defects as quickly as possible

Image References

Retrieved January 8, 2020

- http://pengetouristboard.co.uk/vote-best-takeaway-se20/
- https://i.pinimg.com/originals/b5/22/38/b52238fad11b0a3ecac36fa176041d98.jpg
- https://www.nbs-system.com/wp-content/uploads/2016/05/160503_Tests_boites-788x433.jpg
- https://www.pinclipart.com/picdir/middle/29-293393_challenges-peace-first-clip-art-work-teamwork-clip.png
- https://miro.medium.com/max/10000/1*6HUPtGBOSdERQkCOpL9QEg.png
- https://pngtree.com/freepng/black-repeat-icon_4841007.html

Retrieved September 2, 2020

- https://dzone.com/articles/software-testing-comic