G52APR
Application programming

Introduction to Software Testing

Colin Higgins – based on material from various sources

General Testing

- Definitions and objectives.
- Software testing strategies.
- Software test classifications.
- White box testing
  - Data processing and calculation correctness tests
  - Correctness tests and path coverage
  - Correctness tests and line coverage
  - McCabe's cyclomatic complexity metrics
  - Software qualification and reusability testing
  - Advantages and disadvantages of white box testing.

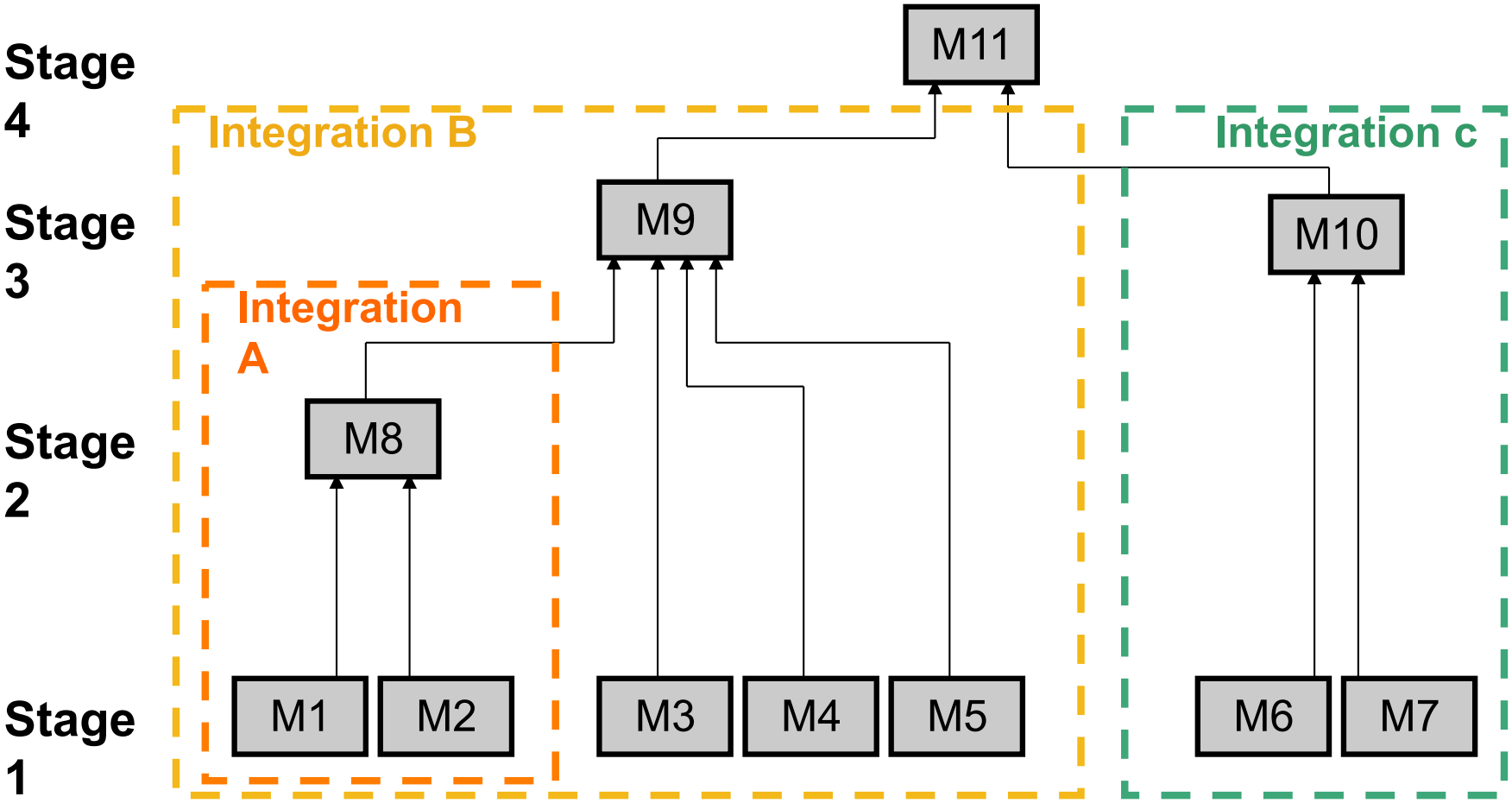
“Laws” of Testing

- The best person to test your code is someone else.
- A good test is one that finds an error.
- Testing can not prove the absence of errors.
- Complete test coverage is impossible, so concentrate on problem areas.
- It cost a lot less to remove bugs early.
- Code Testing ≠ Code Walkthrough.

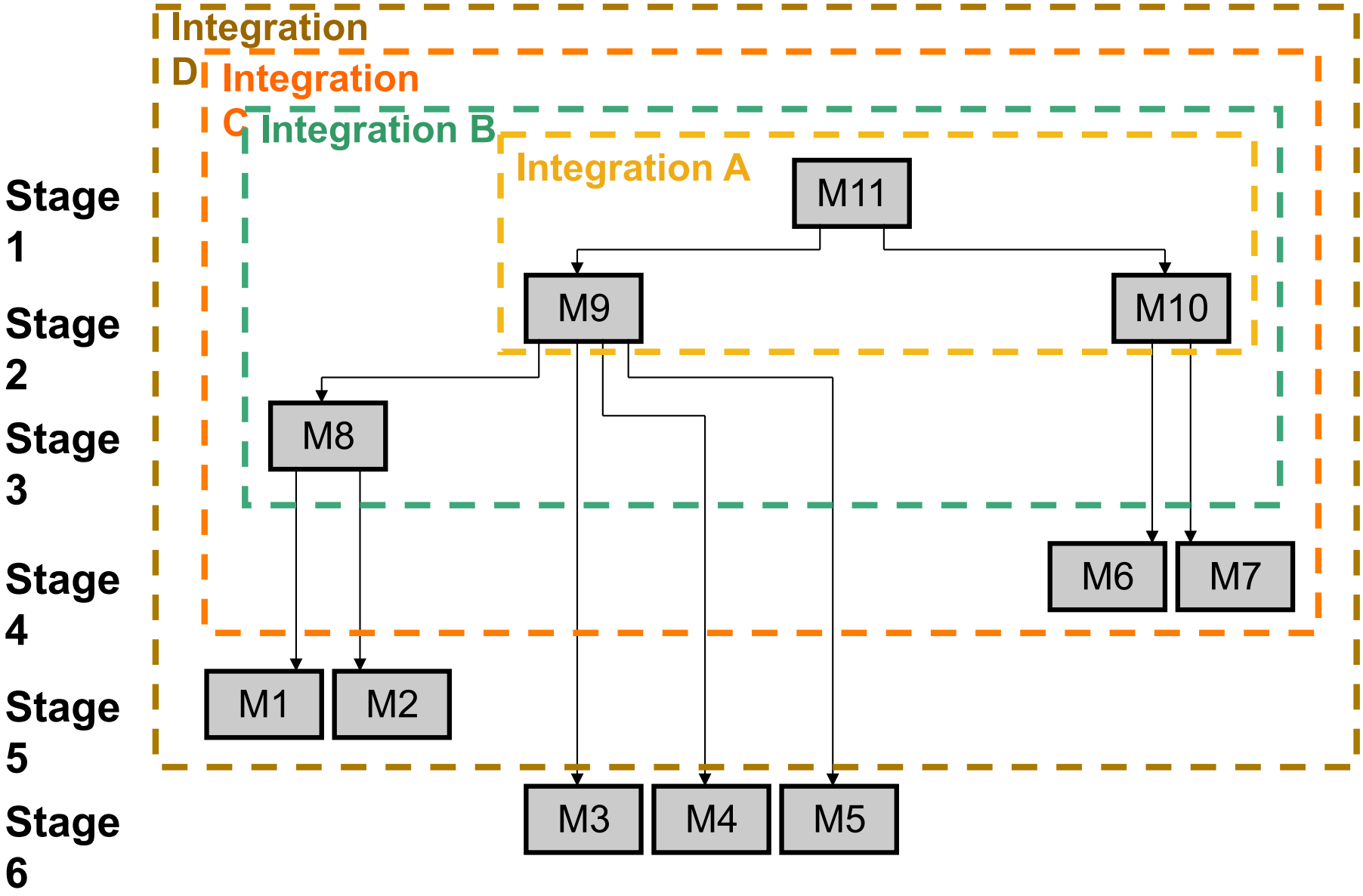
Testing Stages

- Unit Testing
  - modules of code
- Integration Testing
  - design
- Validation Testing
  - requirements
- System Testing
  - system engineering

Bottom-up testing

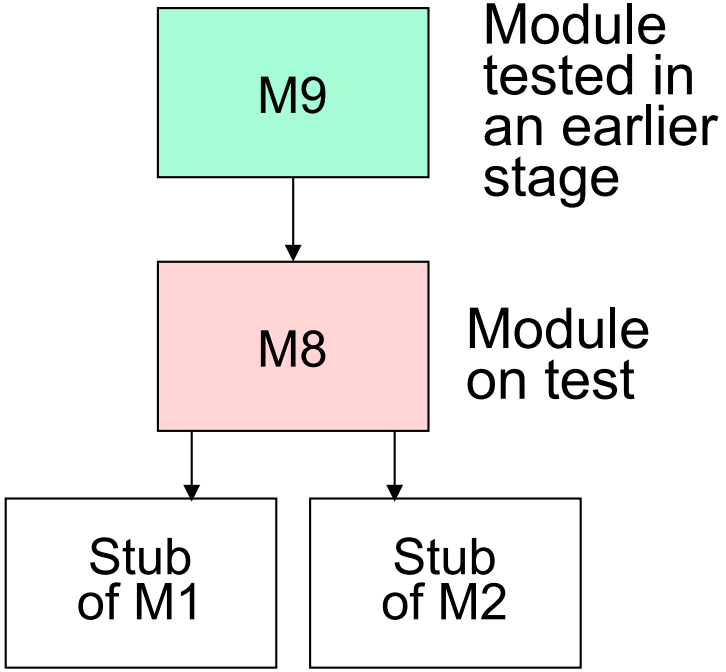


Top-down testing

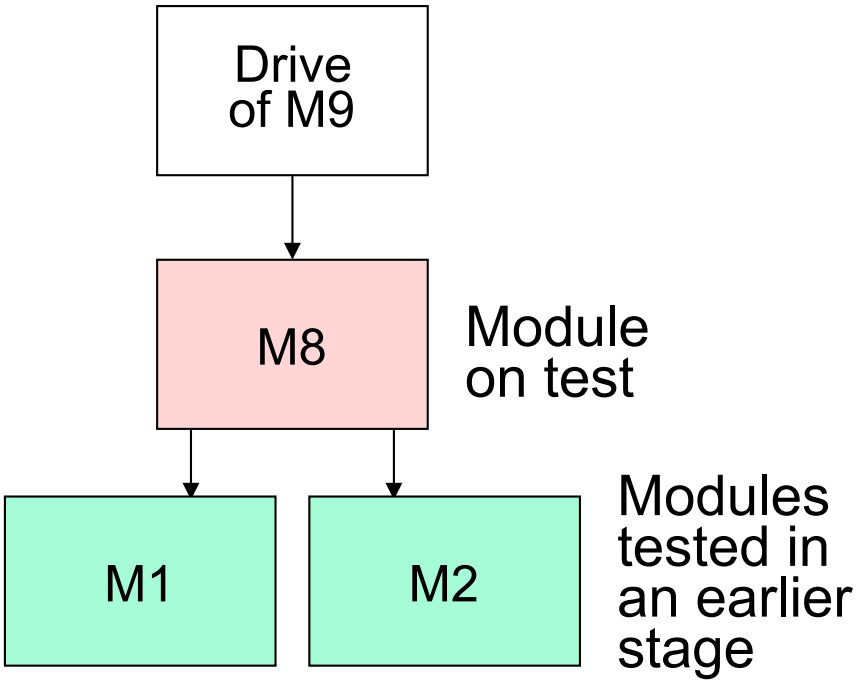


Use of stubs and drivers

Top-down testing of module M8



Bottom-up testing of module M8



Types of code coverage

- Line Coverage
  - Has every possible line of code been executed
- Function coverage
  - Has each function in the program been executed?
- Statement coverage
  - Has each line of the source code been executed?
- Condition coverage
  - Has each evaluation point (such as a true/false decision) been executed?
- Path coverage
  - Has every possible route through a given part of the code been executed?
- Entry/exit coverage
  - Has every possible call and return of the function been executed?

Example

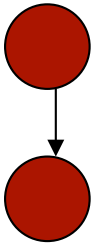
```
int example1 (int value, boolean cond1, boolean cond2)
{
   if ( cond1 )
      value ++;
   if ( cond2 )
      value --;
   return value;
}
```

- Total **Statement Coverage** with one case - True True.
- Total **Path Coverage** with four paths - TT TF FT FF.
- But, total path coverage is usually impractical, so Basis Path Testing is usually better.

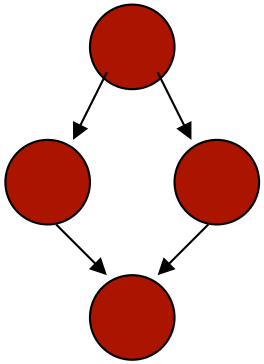
Objective is to test each conditional statement as both true and false

1. Draw a Flow Graph
2. Determine the Cyclomatic Complexity
   - CC = number of regions
   - CC = E - N + 2
3. Max Number of tests = CC
4. Derive a basis set of independent paths
5. Generate data to drive each path

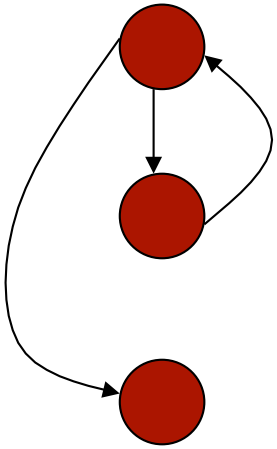
Flow Graphs



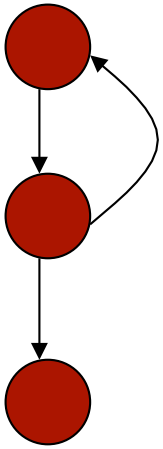
simple sequence



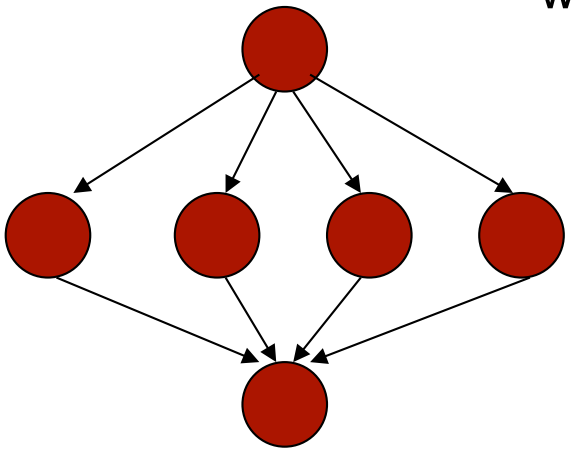
if then else



while



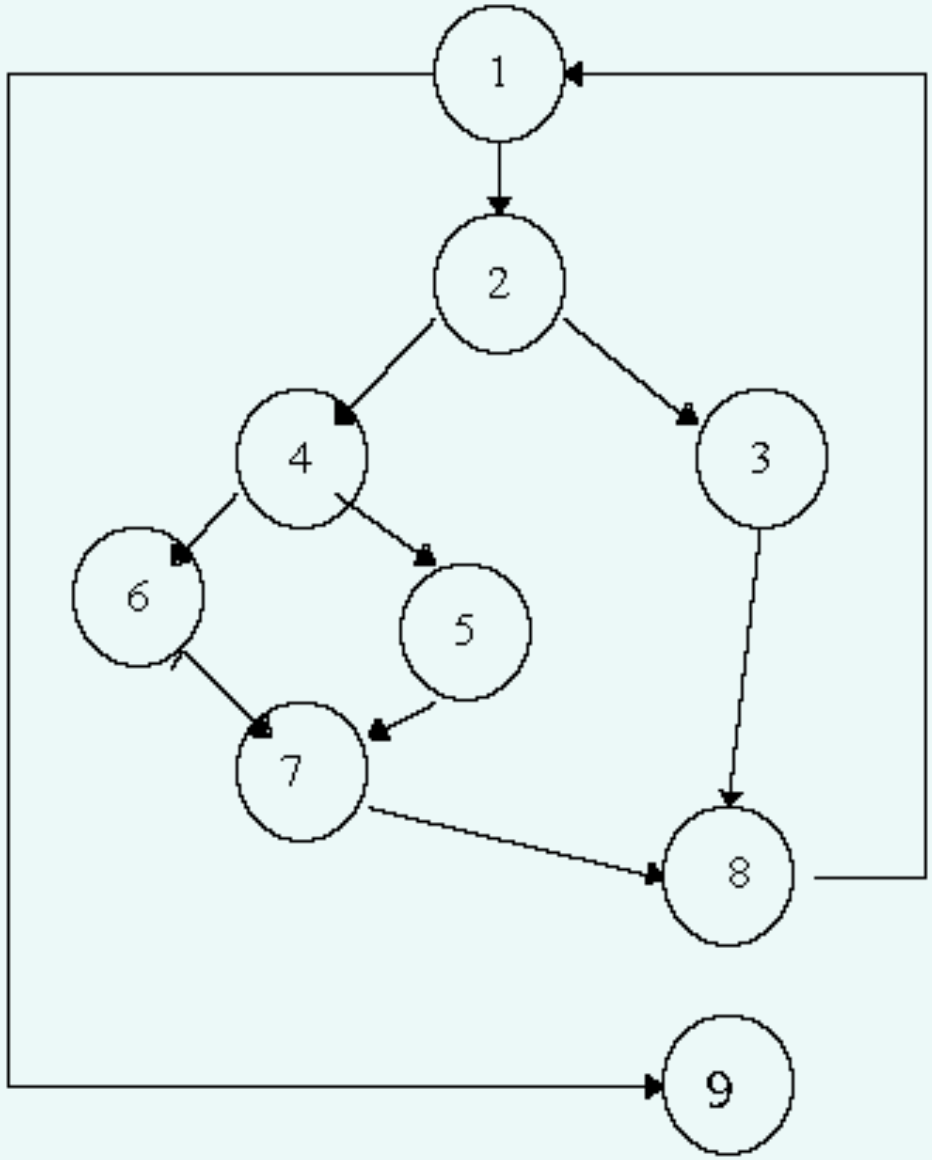
do until



case statement

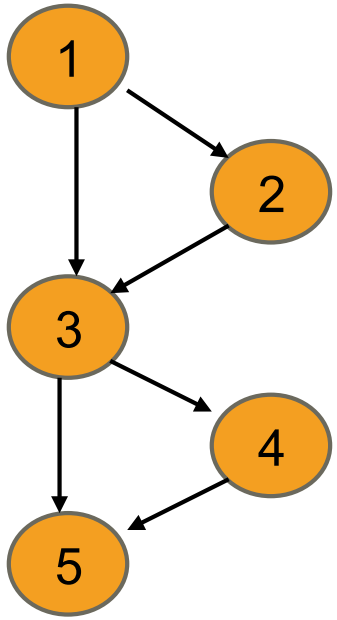
Cyclomatic Complexity 4 example

```
1: WHILE NOT EOF LOOP
2:      Read Record;
2:      IF field1 equals 0 THEN
3:              Add field1 to Total
3:              Increment Counter
4:      ELSE
4:              IF field2 equals 0 THEN
5:                      Print Total, Counter
5:                      Reset Counter
6:              ELSE
6:                      Subtract field 2 from Total
7:              END IF
8:      END IF
8:      Print "End Record”
9: END LOOP
9: Print Counter
```



Using basis path

```
   int example1 (int value, boolean cond1, boolean
   cond2)
   {
1     if ( cond1 )
2        value ++;
3     if ( cond2 )
4        value --;
5     return value;
   }
```



Complexity = 3

Basis Paths	Test Data
1 3 5	false false
1 2 3 5	true false
1 2 3 4 5	true true

Test suites

- To test your code
  - You can do *ad hoc* testing (testing whatever occurs to you at the moment), or
  - You can build a test suite (a thorough set of tests that can be run at any time)
- Disadvantages of writing a test suite
  - It's a lot of extra programming
    - *True*—but use of a good test framework can help quite a bit
  - You don't have time to do all that extra work
    - *False*—Experiments repeatedly show that test suites reduce debugging time more than the amount spent building the test suite
- Advantages of having a test suite
  - Your program will have many fewer bugs
  - It will be a ***lot*** easier to maintain and modify your program
    - This is a *huge* win for programs that, unlike class assignments, get actual use!

XP approach to testing

- In the Extreme Programming approach,
  - Tests are written before the code itself
  - If code has no automated test case, it is *assumed not to work*
  - A test framework is used so that automated testing can be done after every small change to the code
    - This may be as often as every 5 or 10 minutes
  - If a bug is found after development, a test is created to keep the bug from coming back
- Consequences
  - Fewer bugs
  - More maintainable code
  - Continuous integration—During development, the program *always works*—it may not do everything required, but what it does, it does right

Testing in Java - JUnit

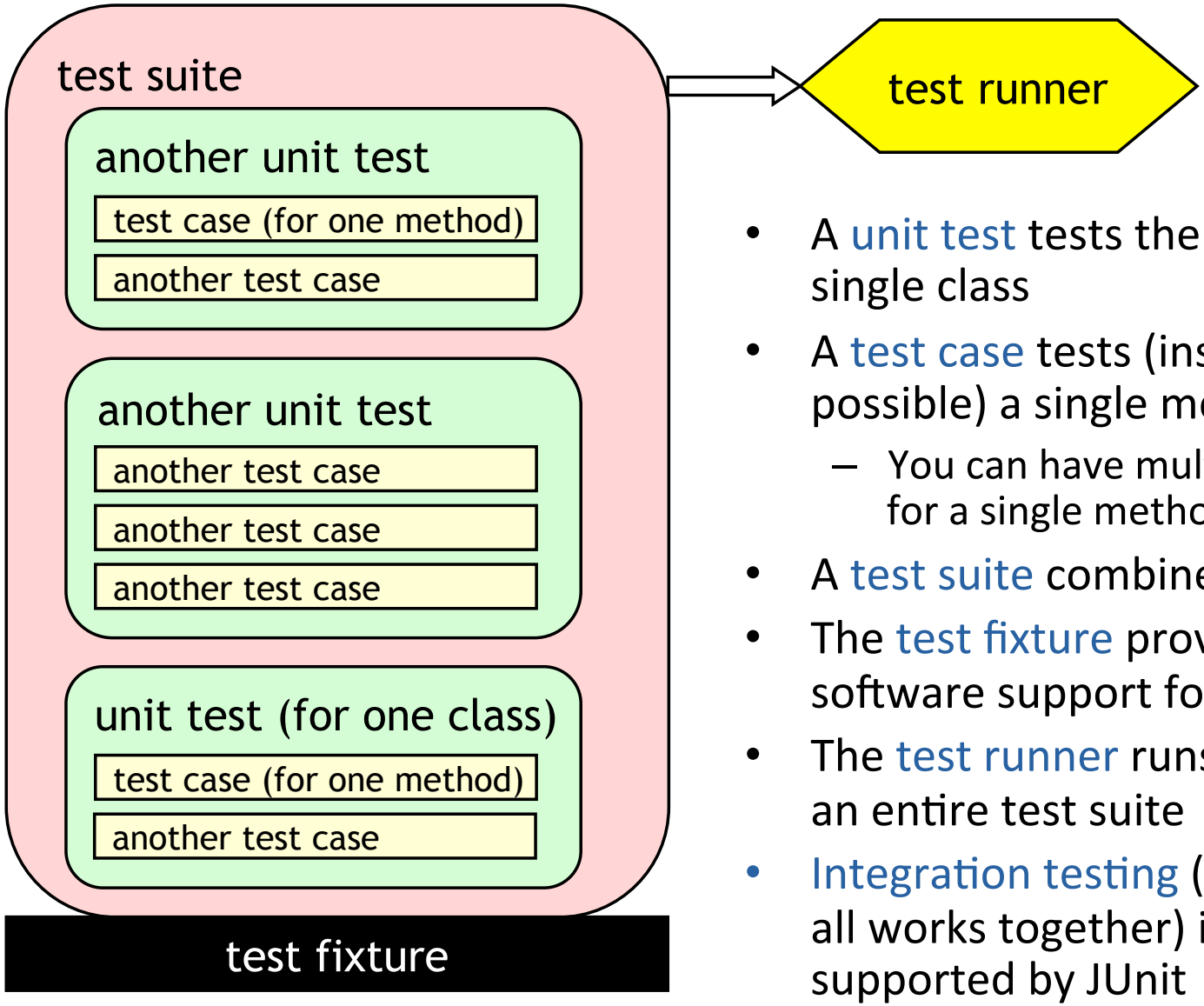
- JUnit is a framework for writing tests
  - JUnit was written by Erich Gamma (of *Design Patterns* fame) and Kent Beck (creator of XP methodology)
  - JUnit uses Java's reflection capabilities (Java programs can examine their own code)
  - Uses annotations as control (@...)
  - JUnit helps the programmer:
    - define and execute tests and test suites
    - formalize requirements and clarify architecture
    - write and debug code
    - integrate code and always be ready to release a working version

Example: Old way vs. new way

- ```
  int max(int a, int b) {
      if (a > b) {
          return a;
      } else {
          return b;
      }
  }
  ```
- ```
  void testMax() {
      int x = max(3, 7);
      if (x != 7) {
          System.out.println("max(3, 7) gives " + x);
      }
      x = max(3, -7);
      if (x != 3) {
          System.out.println("max(3, -7) gives " + x);
      }
  }
  ```
- ```
  public static void main(String[] args) {
      new MyClass().testMax();
  }
  ```

- ```
  @Test
  void testMax() {
      assertEquals(7, max(3, 7));
      assertEquals(3, max(3, -7));
  }
  ```

Terminology



- A unit test tests the methods in a single class
- A test case tests (insofar as possible) a single method
  - You can have multiple test cases for a single method
- A test suite combines unit tests
- The test fixture provides software support for all this
- The test runner runs unit tests or an entire test suite
- Integration testing (testing that it all works together) is not well supported by JUnit

Writing a JUnit test class I

– Start by importing these JUnit 4 classes:

- import org.junit.*;
  import static org.junit.Assert.*; // note static import
  – Declare your test class in the usual way
- public class MyProgramTest {
  – Declare an instance of the class being tested
  – You can declare other variables, but *don't* give them initial values here
- public class MyProgramTest {
  MyProgram program;
  int someVariable;

Writing a JUnit test class II

- Define a method (or several methods) to be executed *before each test*
- Initialize your variables in this method, so that each test starts with a fresh set of values

```
@Before
public void setUp() {
    program = new MyProgram();
    someVariable = 1000;
}
```

- You can define one or more methods to be executed after each test
- Typically such methods release resources, such as files
- Usually there is no need to bother with this method

```
@After
public void tearDown() {
}
```

A simple example

- Suppose you have a class Arithmetic with methods int multiply(int x, int y), and boolean isPositive(int x)

- import org.junit.*;
  import static org.junit.Assert.*;

- public class ArithmeticTest {

```
    @Test
    public void testMultiply() {
        assertEquals(4, Arithmetic.multiply(2, 2));
        assertEquals(-15, Arithmetic.multiply(3, -5));
    }

    @Test
    public void testIsPositive() {
        assertTrue(Arithmetic.isPositive(5));
        assertFalse(Arithmetic.isPositive(-5));
        assertFalse(Arithmetic.isPositive(0));
    }


}
```

Assert methods I

- Within a test,
  - Call the method being tested and get the actual result
  - **Assert** what the correct result should be with one of the assert methods
  - These steps can be repeated as many times as necessary

- An assert method is a JUnit method that performs a test, and throws an AssertionError if the test fails
  - JUnit catches these Errors and shows you the result

- static void assertTrue(boolean ***test***)
  static void assertTrue(String ***message***, boolean ***test***)
  - Throws an AssertionError if the test fails
  - The optional *message* is included in the Error

- static void assertFalse(boolean ***test***)
  static void assertFalse(String ***message***, boolean ***test***)
  - Throws an AssertionError if the test fails

Assert methods II

- assertEquals(***expected***, ***actual***)
  assertEquals(String ***message***, ***expected***, ***actual***)
  - ***expected*** and ***actual*** must be both objects *or* the same primitive type
  - For objects, uses your equals method, ***if*** you have defined it properly, as described on the previous slide

- assertSame(Object ***expected***, Object ***actual***)
  assertSame(String ***message***, Object ***expected***, Object ***actual***)
  - Asserts that two arguments refer to the *same* object

- assertNotSame(Object ***expected***, Object ***actual***)
  assertNotSame(String ***message***, Object ***expected***, Object ***actual***)
  - Asserts that two objects do not refer to the same object

Assert methods III

- assertNull(Object ***object***)
  assertNull(String ***message***, Object ***object***)
  - Asserts that the object is null (undefined)

- assertNotNull(Object ***object***)
  assertNotNull(String ***message***, Object ***object***)
  - Asserts that the object is null

- fail()
  fail(String ***message***)
  - Causes the test to fail and throw an AssertionFailedError
  - Useful as a result of a complex test, when the other assert methods aren't quite what you want

Writing a JUnit test class, III

- **This page is really only for expensive setup, such as when you need to connect to a database to do your testing**
  - If you wish, you can declare *one* method to be executed *just once,* when the class is first loaded
- ```
  @BeforeClass
  public static void setUpClass() throws Exception {
        // one-time initialization code
  }
  ```
  - If you wish, you can declare *one* method to be executed *just once,* to do cleanup after all the tests have been completed
- ```
  @AfterClass
  public static void tearDownClass() throws Exception {
        // one-time cleanup code
  }
  ```

Special features of @Test

- You can limit how long a method is allowed to take
- This is good protection against infinite loops
- The time limit is specified in milliseconds
- The test fails if the method takes too long

```
@Test (timeout=10)
 public void greatBig() {
    assertTrue(program.ackerman(5, 5) > 10e12);
}
```

- Some method calls should throw an exception
- You can specify that a particular exception is expected
- The test will pass if the expected exception is thrown, and fail otherwise

```
@Test (expected=IllegalArgumentException.class)
public void factorial() {
    program.factorial(-5);
}
```

Test-Driven Development (TDD)

- It is difficult to add JUnit tests to an existing program
  - The program probably wasn't written with testing in mind
- It's actually better to write the tests *before* writing the code you want to test
- This seems backward, but it really does work better:
  - When tests are written first, you have a clearer idea what to do when you write the methods
  - Because the tests are written first, the methods are necessarily written to be testable
  - Writing tests first encourages you to write simpler, single-purpose methods
  - Because the methods will be called from more than one environment (the "real" one, plus your test class), they tend to be more independent of the environment

Stubs

- In order to run our tests, the methods we are testing have to exist, but they don't have to be right
- Instead of starting with "real" code, we start with stubs—minimal methods that always return the same values
  - A stub that returns void can be written with an empty body
  - A stub that returns a number can return 0 or -1 or 666, or whatever number is most likely to be *wrong*
  - A stub that returns a boolean value should usually return false
  - A stub that returns an object of any kind (including a String or an array) should return null
- When we run our test methods with these stubs, we want the test methods to *fail!*
  - This helps "test the tests"—to help make sure that an incorrect method does not pass the tests

Test suites

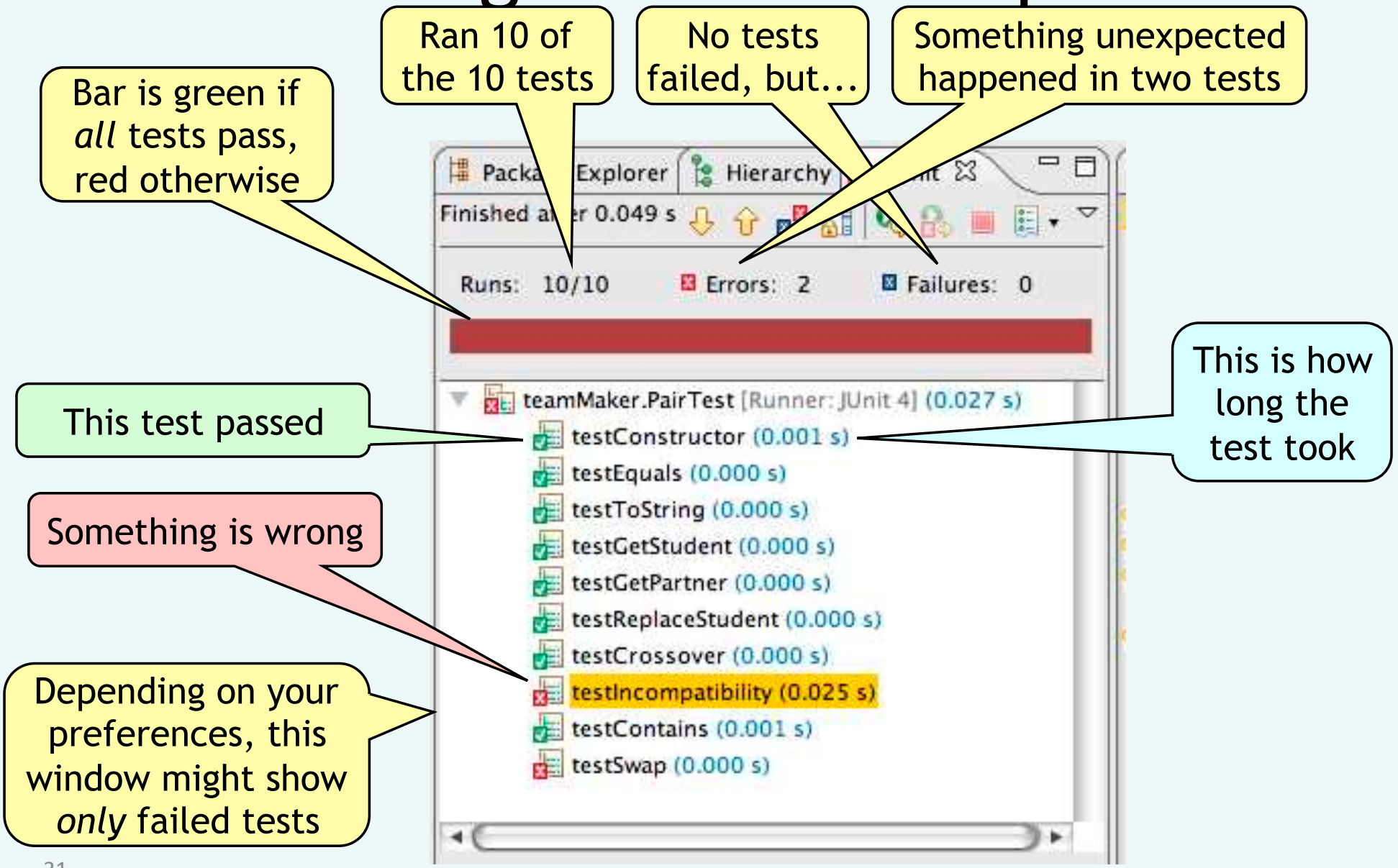
- You can define a suite of tests

```
@RunWith(value=Suite.class)
@SuiteClasses(value={
                        MyProgramTest.class,
                        AnotherTest.class,
                        YetAnotherTest.class
                    })
public class AllTests { }
```

JUnit in Eclipse

- If you write your method stubs first (as on the previous slide), Eclipse will generate test method stubs for you
- To add JUnit 4 to your project:
  - Select a class in Eclipse
  - Go to File → New… → JUnit Test Case
  - Make sure New JUnit 4 test is selected
  - Click where it says "Click here to add JUnit 4…"
  - Close the window that appears
- To create a JUnit test class:
  - Do steps 1 and 2 above, if you haven't already
  - Click Next>
  - Use the checkboxes to decide which methods you want test cases for; don't select Object or anything under it
    - I like to check "create tasks," but that's up to you
  - Click Finish
- To run the tests:
  - Choose Run → Run As → JUnit Test

Viewing results in Eclipse



Recommended approach

- Write a test for some method you intend to write
  - If the method is fairly complex, test only the simplest case
- Write a stub for the method
- Run the test and make sure it fails
- Replace the stub with code
  - Write just enough code to pass the tests
- Run the test
  - If it fails, debug the method (or maybe debug the test); repeat until the test passes
- If the method needs to do more, or handle more complex situations, add the tests for these first, and go back to step 3

The End

If you don't unit test then you aren't a software engineer, you are a typist who understands a programming language.

--Moses Jones

1. Never underestimate the power of one little test.
2. There is no such thing as a dumb test.
3. Your tests can often find problems where you're not expecting them.
4. Test that everything you say happens actually does happen.
5. If it's worth documenting, it's worth testing.

--Andy Lester