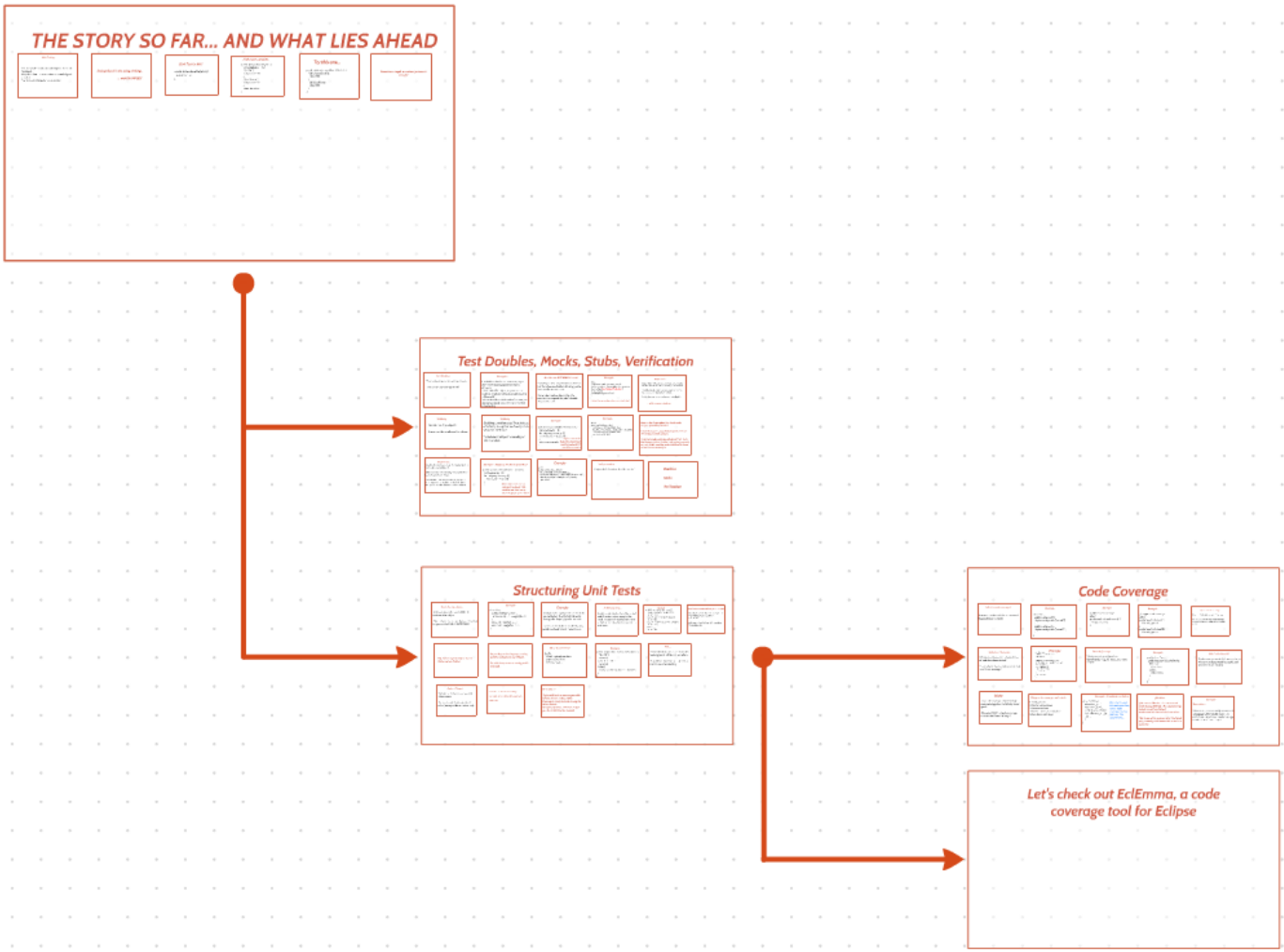
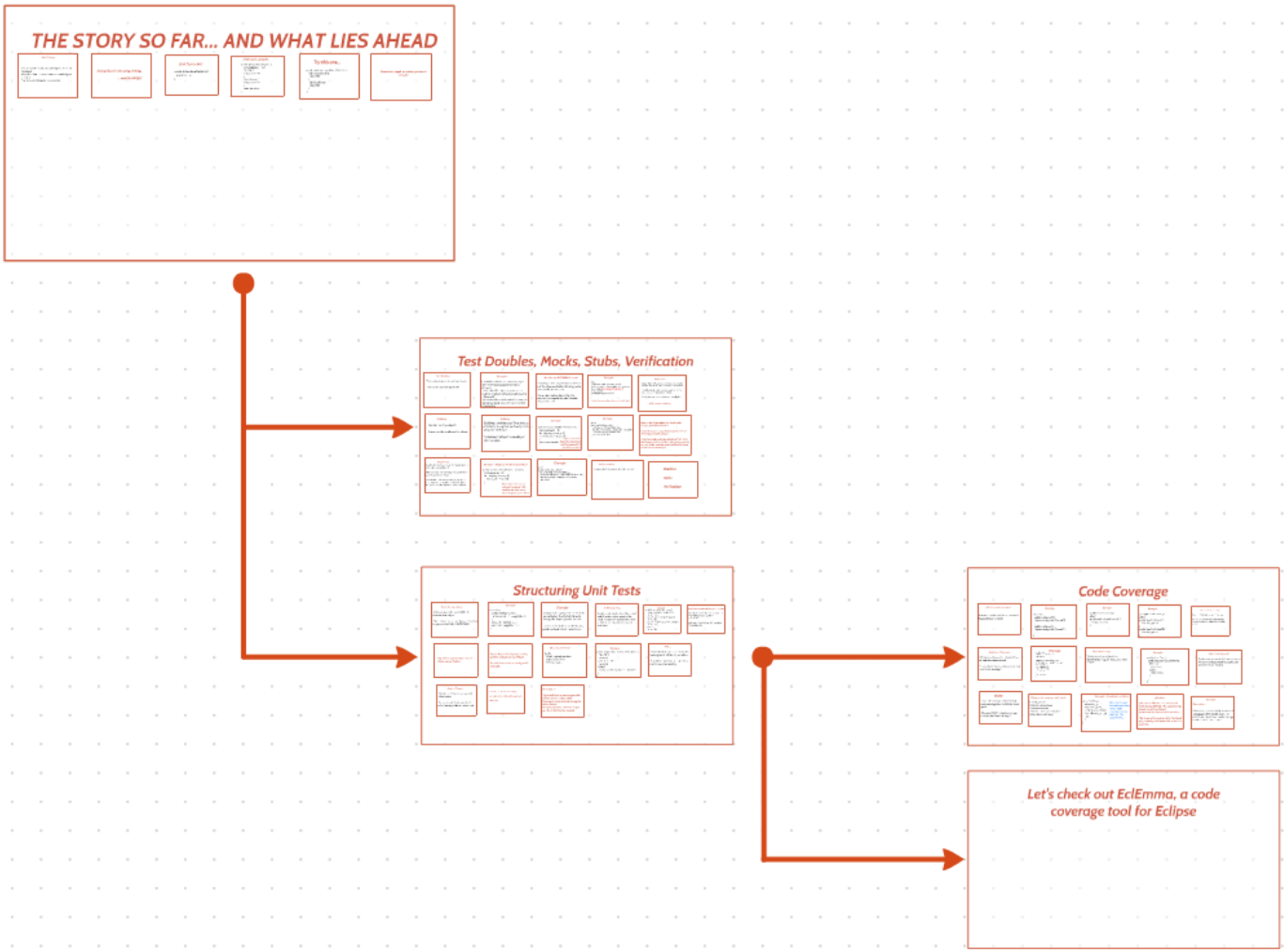
CS1699 - Lecture 8 - Advanced Unit Testing



CS1699 - Lecture 8 - Advanced Unit Testing



THE STORY SO FAR... AND WHAT LIES AHEAD

Unit Testing

Test the smallest units of code (objects, methods, functions).
Very white-box - requires intimate knowledge of codebase.
The first tests that code hits, normally.

Let's get back in the swing of things..

... with EXAMPLES!

JUnit Test for this?

```
public int doubleMe(int n) {
  return n * 2;
}
```

A bit more complex...

```
public String laboonify(int s) {
  String toReturn = "a";
  if (s < 10) {
    toReturn += "b";
  }
  if (s % 2 == 0) {
    toReturn += "c"
  }
  return toReturn;
}
```

Try this one...

```
public static void multiQuack(Duck d) {
  if (d.isAGoodDuck()) {
    d.quack();
  }
  if (d.isDuckling()) {
    d.quack();
  }
}
```

Sometimes simple assertions just aren't enough!

Unit Testing

Test the smallest units of code (objects, methods, functions).
Very white-box - requires intimate knowledge of codebase.
The first tests that code hits, normally.

Let's get back in the swing of things..

... with EXAMPLES!

JUnit Test for this?

```
public int doubleMe(int n) {
   return n * 2;
}
```

A bit more complex...

```
public String laboonify(int s) {
   String toReturn = ''a'';
   if (s < 10) {
     toReturn += ''b'';
   }
   if (s % 2 == 0) {
     toReturn += ''c''
   }
   return toReturn;
}
```

Try this one...

```
public static void multiQuack(Duck d) {
   if (d.isAGoodDuck()) {
      d.quack();
   }
   if (d.isDuckling()) {
      d.quack();
   }
}
```

Sometimes simple assertions just aren't enough!

Test Doubles, Mocks, Stubs, Verification

Test Doubles

"Fake" objects you can use in your tests.

They can act any way you want!

Examples

1. A doubled database connection, so you don't need to actually connect to the database
2. A doubled File object, so you can test read/write failures without actually making a file on disk
3. A doubled RandomNumberGenerator, so you can always produce the same number when testing

Doubles are EXTREMELY useful

They let you test only the item under test, not the whole application, allowing you to focus on the current item.

Remember, double objects that the current class depends on; don't double the current class!

Example

```
@Test
public void testDeleteFrontOneItem() {
 LinkedList<Integer> ll = new LinkedList<Integer>();
 ll.addToFront(Mockito.mock(Node.class));
 ll.deleteFront();
 assertEquals(ll.getFront(), null);
}

// Now there are no dependencies on Node class!
```

Keep in mind...

Depending on the testing software used, if you call a method on a mocked object, it will by default either:

1. Call the method on the regular object (which may not have been initialized by the mock!)

2. Not allow you to do so unless you've stubbed it

... which brings us to stubbing.

Stubbing

Doubles are fake objects.

Stubs are fake methods/functions.

Stubbing

Stubbing a method says "hey, instead of actually doing that method, just do whatever I tell you."

"Whatever I tell you" is usually just return a value.

Example

```
public int quackAlot(Duck d, int num) {
 int numQuacks = 0;
 for (int j=0; j < num; j++) {
  numQuacks += d.quack();
 }
 return numQuacks;
}
```

Right now, we're dependent on Duck working correctly if we want to test it

Example

```
@Test
public void testQuackAlot() {
   Duck mockDuck = mock(Duck.class);
   mockDuck.when(mockDuck.quack()).thenReturn(1);
   int val = quackAlot(mockDuck, 100);
   assertEquals(val, 100);
}
```

Now, we don't care about how Duck works, only our quackAlot() method.

If something goes wrong in Duck.quack, tests on THAT class will fail, not here.

Tests that break easily are called BRITTLE. Tests that depend on lots of other code working correctly are very brittle, and also make it difficult to know where the error actually is.

Verification

Note that this is different from the "verification" in "verification and validation".

In this case, it means "verifying that a method has been called 0, 1, or n times."

A test double which uses verification is called a Mock. However, many frameworks don't have a strong differentiation between doubles ad mcks

Example - A Slightly Modified quackAlot()

```
public void quackAlot(Duck d, int num) {
 int throwaway = 0;
 for (int j=0; j < num; j++) {
  throwaway = d.quack();
 }
}
```

Note that there are no numbers to check. We need to see how many times d.quack was called.

Example

```
@Test
public void testQuackAlot() {
   Duck mockDuck = mock(Duck.class);
   mockDuck.when(mockDuck.quack()).thenReturn(1);
   Mockito.verify(mockDuck, times(5)).quack();
   quackAlot();
}
```

Verify vs Assertion

Verify is a kind of assertion about the code itself.

Doubles

Stubs

Verification

Test Doubles

"Fake" objects you can use in your tests.

They can act any way you want!

Examples

1. A doubled database connection, so you don't need to actually connect to the database
2. A doubled File object, so you can test read/write failures without actually making a file on disk
3. A doubled RandomNumberGenerator, so you can always produce the same number when testing

Doubles are EXTREMELY useful

They let you test only the item under test, not the whole application, allowing you to focus on the current item.

Remember, double objects that the current class depends on; don't double the current class!

Example

```
@Test
public void testDeleteFrontOneItem() {
 LinkedList<Integer> ll = new LinkedList<Integer>();
 ll.addToFront(Mockito.mock(Node.class));
 ll.deleteFront();
 assertEquals(ll.getFront(), null);
}
```

// Now there are no dependencies on Node class!

Keep in mind...

Depending on the testing software used, if you call a method on a mocked object, it will by default either:

1. Call the method on the regular object (which may not have been initialized by the mock!)

2. Not allow you to do so unless you've stubbed it

... which brings us to stubbing.

Stubbing

Doubles are fake objects.

Stubs are fake methods/functions.

Stubbing

Stubbing a method says "hey, instead of actually doing that method, just do whatever I tell you."

"Whatever I tell you" is usually just return a value.

Example

```
public int quackAlot(Duck d, int num) {
 int numQuacks = 0;
 for (int j=0; j < num; j++) {
   numQuacks += d.quack();
 }
 return numQuacks;
}
```

Right now, we're dependent on Duck working correctly if we want to test it

Example

```
@Test
public void testQuackAlot() {
    Duck mockDuck = mock(Duck.class);
    mockDuck.when(mockDuck.quack()).thenReturn(1);
    int val = quackAlot(mockDuck, 100);
    assertEquals(val, 100);
}
```

Now, we don't care about how Duck works, only our quackAlot() method.

If something goes wrong in Duck.quack, tests on THAT class will fail, not here.

Tests that break easily are called BRITTLE. Tests that depend on lots of other code working correctly are very brittle, and also make it difficult to know where the error actually is.

Verification

Note that this is different from the "verification" in "verification and validation".

In this case, it means "verifying that a method has been called 0, 1, or n times."

A test double which uses verification is called a Mock. However, many frameworks don't have a strong differentiation between doubles ad mcks

Example - A Slightly Modified quackAlot()

```
public void quackAlot(Duck d, int num) {
  int throwaway = 0;
  for (int j=0; j < num; j++) {
    throwaway = d.quack();
  }
}
```

Note that there are no numbers to check. We need to see how many times d.quack was called.

Example

```
@Test
public void testQuackAlot() {
    Duck mockDuck = mock(Duck.class);
    mockDuck.when(mockDuck.quack()).thenReturn(1);
    Mockito.verify(mockDuck, times(5)).quack();
    quackAlot();
}
```

Verify vs Assertion

Verify is a kind of assertion about the code itself.

Doubles

Stubs

Verification

Structuring Unit Tests

From the top-down...

Unit tests should hit every PUBLIC method of an object.

Private methods are usually tested by their impact on the PUBLIC INTERFACE.

Example

```
class Bird {
  public int chirpify(int n) {
    return nirpify(n) + noogiefy(n + 1);
  }
  private int nirpify(n) { ... }
  private int noogiefy(n) { ... }
}
```

Example

nirpify() and noogiefy() private methods are not tested directly, but indirectly through the chirpify() public method.

If a private method is not called by any public methods, why is it even there?

A differing view...

Code is code. Code should be tested, and it's easier to test lower in the stack. Imagine if chirpify() were very complex and called numerous private methods.

Example

```
public boolean foo(boolean n) {
  if (bar(n) && baz(n) && quux(n)) {
    return true;
  } else if (baz(n) ^ (thud(n) || baa(n)) {
    return false;
  } else if (meow(n) || chew(n) || chirp(n)) {
    return true;
  } else {
    return false;
}
```

That's Nine Private Methods, or 2 ^ 9 Tests!

If each of those methods are complex, it may make sense to test them individually.

Otherwise, it can be hard to know where the problem lies.

P.S. This is a great way to start a flame war on Twitter.

Depending on the language, testing private methods can be difficult.

For this class, focus on testing public methods.

What should I test?

Ideally...

1. Each equivalence class
2. Boundary values
3. Failure modes

Example

```
public int quack(int n) throws Exception {
  if (n < 10) {
    return 1;
  } else if (n < 20) {
    return 2;
  } else {
    throw new Exception("too many!");
  }
}
```

But....

Remember that a test suite that takes too long to run will be run less often.

Depending on your project, you may want more or less testing.

Rule of Thumb

Try to hit common use cases and failure cases.

Focus on code that is executed often (use a profiler or similar tool).

Sometime it's difficult to test things.

Especially when working with legacy code.

Such is life.

Best advice -

Try to add tests as soon as possible. Ideally, before coding (TDD).
Develop in a way to make it easy for others to test.
In legacy systems, add tests as you go. Don't fall into the morass!

From the top-down...

Unit tests should hit every PUBLIC method of an object.

Private methods are usually tested by their impact on the PUBLIC INTERFACE.

Example

```
class Bird {
  public int chirpify(int n) {
    return nirpify(n) + noogiefy(n + 1);
  }
  private int nirpify(n) { ... }
  private int noogiefy(n) { ... }
}
```

Example

nirpify() and noogiefy() private methods are not tested directly, but indirectly through the chirpify() public method.

If a private method is not called by any public methods, why is it even there?

A differing view...

Code is code. Code should be tested, and it's easier to test lower in the stack. Imagine if chirpify() were very complex and called numerous private methods.

Example

```
public boolean foo(boolean n) {
  if (bar(n) && baz(n) && quux(n)) {
    return true;
  } else if (baz(n) ^ (thud(n) || baa(n)) {
    return false;
  } else if (meow(n) || chew(n) || chirp(n)) {
    return true;
  } else {
    return false;
}
```

That's Nine Private Methods, or 2 ^ 9 Tests!

If each of those methods are complex, it may make sense to test them individually.

Otherwise, it can be hard to know where the problem lies.

P.S. This is a great way to start a flame war on Twitter.

Depending on the language, testing private methods can be difficult.

For this class, focus on testing public methods.

What should I test?

Ideally...

1. Each equivalence class
2. Boundary values
3. Failure modes

Example

```
public int quack(int n) throws Exception {
   if (n < 10) {
     return 1;
   } else if (n < 20) {
     return 2;
   } else {
     throw new Exception("too many!");
   }
}
```

But....

Remember that a test suite that takes too long to run will be run less often.

Depending on your project, you may want more or less testing.

Rule of Thumb

Try to hit common use cases and failure cases.

Focus on code that is executed often (use a profiler or similar tool).

Sometime it's difficult to test things.

Especially when working with legacy code.

Such is life.

Best advice -

Try to add tests as soon as possible.
Ideally, before coding (TDD).
Develop in a way to make it easy for others to test.
In legacy systems, add tests as you go. Don't fall into the morass!

Code Coverage

What is code coverage?

How much of the codebase is covered by a particular test suite.

Example

```
class Duck
  public void quack() {
    System.out.println("Quack!");
  }
  public void quock() {
    System.out.println("Quock!");
  }
}
```

Example

```
// 50 % code coverage
@Test
public void testDuckQuack() {
  // quack testing
}
```

Example

```
// 100% code coverage
@Test
public void testQuack() {
  // testing quack
}
public void testQuock() {
  // testing quock
}
```

Aspects of Code Coverage

Function/Method/Subroutine Coverage -

What percentage of functions/methods/subroutines been called by the unit testing code?

Statement Coverage

What percentage of statements/lines of code have been tested?

This is what is most commonly referred to as "code coverage".

Example

```
class Raccoon
  public boolean hasMask() {
    return true;
  }
  public int solvePuzzle(Puzzle p) {
    PuzzleSolver ps = new PuzzleSolver();
    int attempt=0;
    while (!p.solved) {
      ps.solve(attempt);
      attempt++;
    }
    return attempt;
  }
}
```

Branch Coverage

What percentage of branches (conditionals - e.g., ifs, cases, etc.) were tested?

Example

```
public class Duck {
  public boolean isDucky(int n) {
    if (n < 5) {
      return true;
    } else {
      return false;
    }
  }
}
```

What's the Benefit?

Code coverage metrics lets you easily see where more tests would be useful and where tests are missing.

Note

Low code coverage is bad, but high code coverage does not always mean good.

Also, even 100% of code coverage cannot catch 100% of bugs!

Things code coverage can't catch..

1. Timing issues
2. Performance issues
3. Race conditions
4. Permutations of a statement
5. Combinatoric issues

Example - Combinatoric Failure

```
class BadTime {
  private int _a = 1;
  private int _b = 1;
  public float getDiv() {
    int toReturn _a / _b;
    _b--;
  }
}
```

Running through this code one time, 100% code coverage and no failures. The second time...

Side Note

Side effects (like the _b--; statement) make testing difficult. The easiest things to test are self-contained, mathematical, deterministic functions.

This is one of the reasons why functional programming with immutable state is so powerful.

Example

Thread fun...

Whenever you have multiple threads in a language with mutable state, you could have a Bad Time. Code coverage numbers won't protect you.

What is code coverage?

How much of the codebase is covered by a particular test suite.

Example

```
class Duck
  public void quack() {
    System.out.println(''Quack!'');
  }
  public void quock() {
    System.out.println(''Quock!'');
  }
}
```

Example

```
// 50 % code coverage
@Test
public void testDuckQuack() {
  // quack testing
}
```

Example

```
// 100% code coverage
@Test
public void testQuack() {
    // testing quack
}
public void testQuock() {
    // testing quock
}
```

Aspects of Code Coverage

Function/Method/Subroutine Coverage -

What percentage of functions/methods/ subroutines been called by the unit testing code?

Statement Coverage

What percentage of statements/lines of code have been tested?

This is what is most commonly referred to as "code coverage".

Example

```
class Raccoon
  public boolean hasMask() {
    return true;
  }
  public int solvePuzzle(Puzzle p) {
    PuzzleSolver ps = new PuzzleSolver();
    int attempt=0;
    while (!p.solved) {
      ps.solve(attempt);
      attempt++;
    }
    return attempt;
  }
}
```

Branch Coverage

What percentage of branches (conditionals - e.g., ifs, cases, etc.) were tested?

Example

```
public class Duck {
  public boolean isDucky(int n) {
    if (n < 5) {
      return true;
    } else {
       return false;
    }
  }
}
```

What's the Benefit?

Code coverage metrics lets you easily see where more tests would be useful and where tests are missing,

Note

Low code coverage is bad, but high code coverage does not always mean good.

Also, even 100% of code coverage cannot catch 100% of bugs!

Things code coverage can't catch..

1. Timing issues
2. Performance issues
3. Race conditions
4. Permutations of a statement
5. Combinatoric issues

Example - Combinatoric Failure

```
class BadTime {
   private int _a = 1;
   private int _b = 1;
   public float getDiv() {
     int toReturn _a / _b;
     _b--;
   }
}
```

Running through this code one time, 100% code coverage and no failures. The second time...

Side Note

Side effects (like the _b--; statement) make testing difficult. The easiest things to test are self-contained, mathematical, deterministic functions.

This is one of the reasons why functional programming with immutable state is so powerful.

Example

Thread fun...

Whenever you have multiple threads in a language with mutable state, you could have a Bad Time. Code coverage numbers won't protect you.

Let's check out EclEmma, a code coverage tool for Eclipse

CS1699 - Lecture 8 - Advanced Unit Testing