Formality of Java Programming Part 2

Steven Castellucci

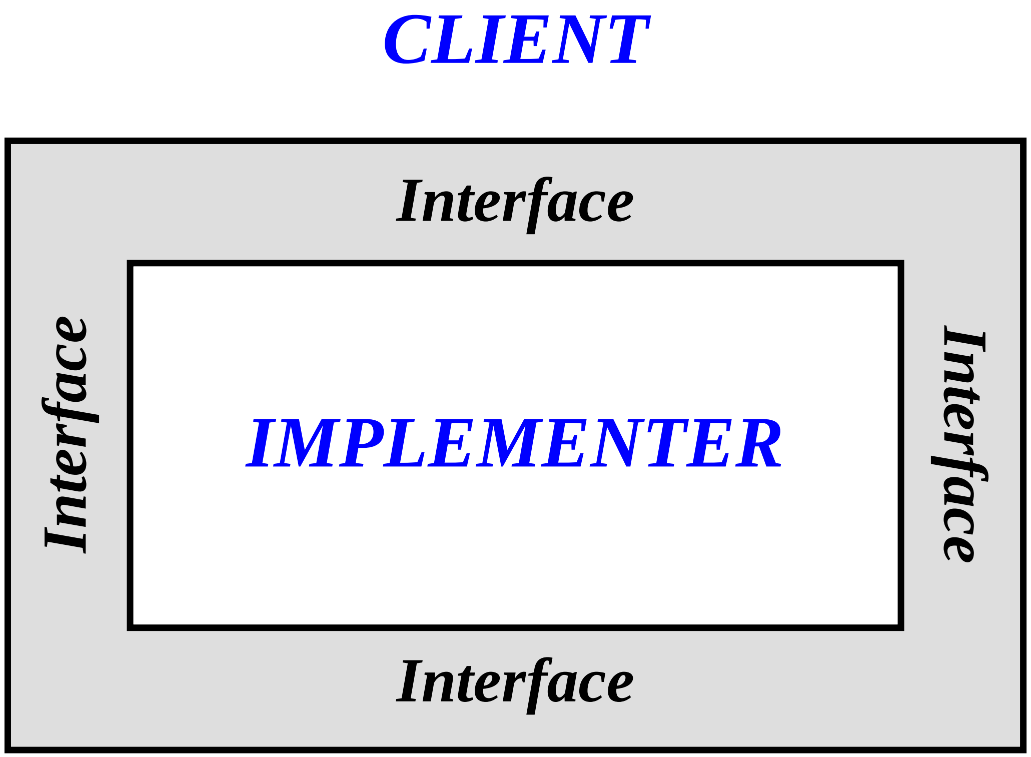
Division of Responsibilities

- Often programming in teams or large groups
- Need efficient technique to…
    - Describe who does what
    - What classes/methods are needed
    - What methods will take as arguments
    - What methods will return as results
    - What methods will throw if there is an error

The Client-Implementer View

- The **client** develops the main class
  - Understands the big picture, the purpose of the application
  - Knows what each component does but not how it does it
- The **implementer** develops a component
  - Focuses only on the inner details of one component
- Client and Implementer share info on a need-to-know basis

The Client–Implementer View



- The "interface" is the application programming interface (API)

Contracts

- Guarantee between client and implementer
- Precondition
  - What the client must satisfy
- Postcondition
  - What the implementer must deliver
- Liability
  - Pre. is satisfied and post. is satisfied → Good
  - Pre. is satisfied and post. is not satisfied → Implementer at fault
  - Pre. is not satisfied → Client at fault
  - If no precondition stated, then client need not satisfy anything

Contracts in Java

- Methods in the Java specify contracts as follows:
  - Precondition is always true unless stated otherwise
  - Postcondition is specified under Returns and Throws
- Example:

`double squareRoot(double x)`
Returns the square root of the given argument.

Parameters:
  `x` - an argument.
Returns:
  the positive square root of `x`.
Throws:
  an exception if `x` < `0`.

Testing

- Imperative to test all classes for correctness
- Compare calculated output with expected output
  - Identical result → test passed
  - Different result → test failed
- Testing requires multiple test cases to ensure correct operation under various condition with various inputs
- Example: Test kilometresToMiles method

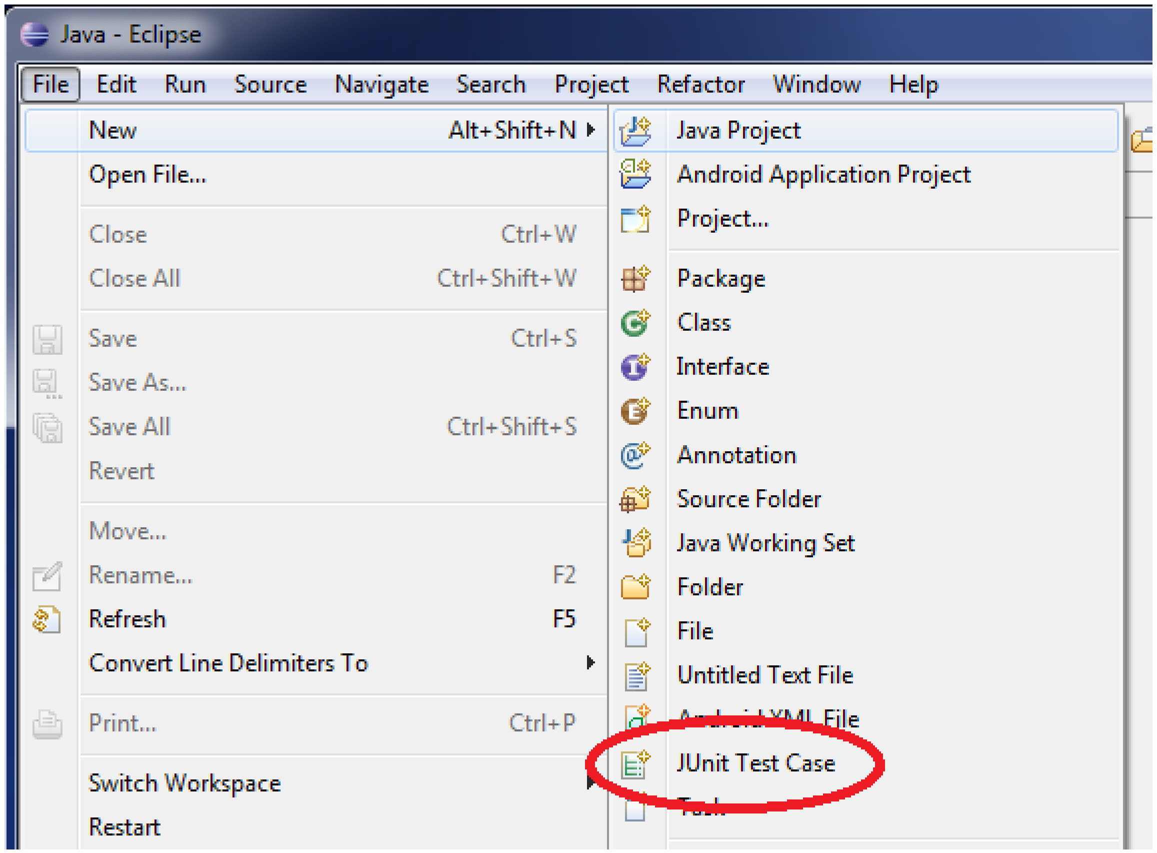
Testing (Implemented Code)

```
public class DistanceUtility
{
      public static final double MI_PER_KM = 0.621371;

      private DistanceUtility() {}

      public static double kilometresToMiles(double km)
      {
            return km * MI_PER_KM;
      }

}
```

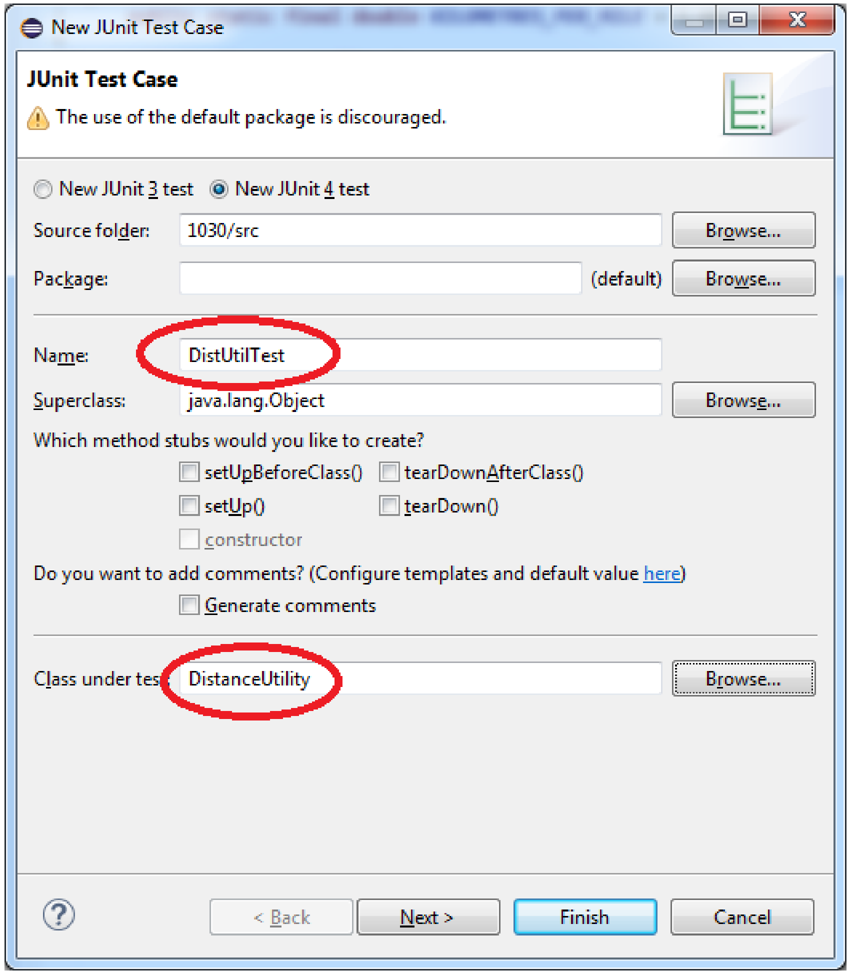
Testing (Testing Class)

```
public class DistUtilTester
{
  public static void main(String[] args)
  {
    double input = 2;
    double expected = 1.24274238; // used calculator as oracle
    double actual = DistanceUtility.kilometresToMiles(input);
    double epsilon = 0.000001;
    if (Math.abs(actual – expected) < epsilon)
    {
      System.out.println("passed");
    }
    else
    {
      System.out.println("failed");
    }
}
```

Testing (JUnit in Eclipse)



Testing (JUnit in Eclipse) (2)



New JUnit Test Case

JUnit Test Case

The use of the default package is discouraged.

( ) New JUnit 3 test [✓ New JUnit 4 test]

Source folder: 1030/src Browse...

Package: (default) Browse...

Name: DistUtilTest

Superclass: java.lang.Object Browse...

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()

☐ setUp() ☐ tearDown()

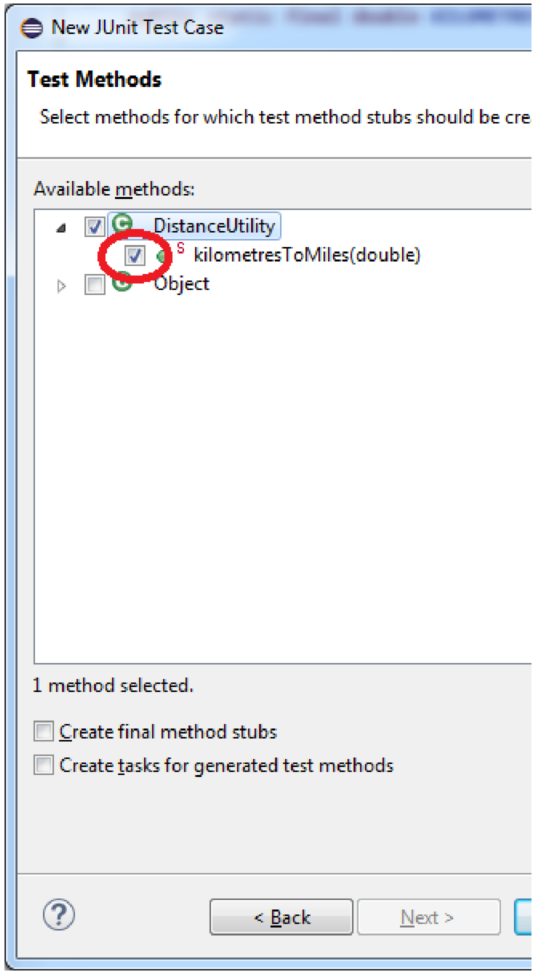
☐ constructor

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Class under test: DistanceUtility Browse...

< Back | Next > | Finish | Cancel



New JUnit Test Case

Test Methods

Select methods for which test method stubs should be cre

Available methods:

- ☑ DistanceUtility
  - ☑ kilometresToMiles(double)
- ☐ Object

1 method selected.

☐ Create final method stubs

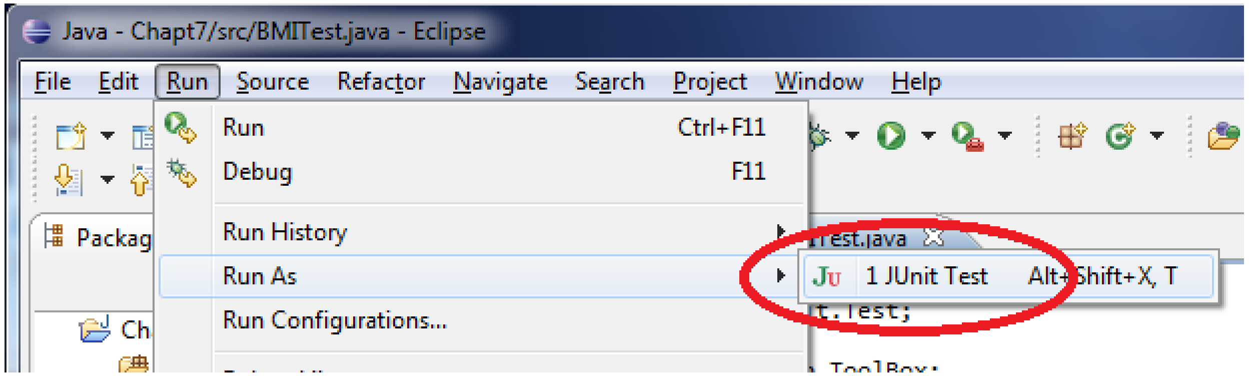
☐ Create tasks for generated test methods

< Back | Next >

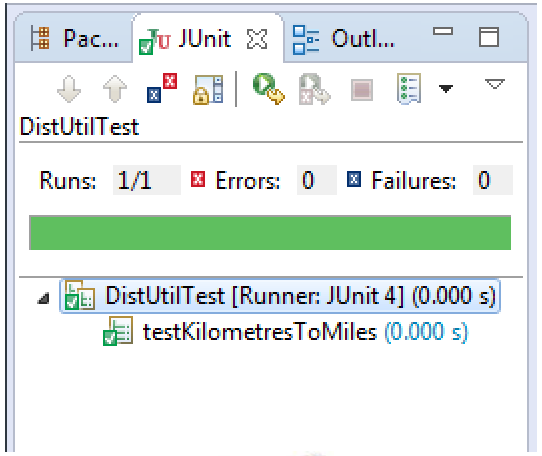
Testing (JUnit in Eclipse) (3)

```
@Test
public void testKilometresToMiles()
{
    double input = 2;
    double expected = 1.24274238; // calculator as oracle
    double actual = DistanceUtility.kilometresToMiles(input);
    double epsilon = 0.000001;
    assertEquals("Actual and expected values exceed epsilon!",
        expected, actual, epsilon);
}
```

Testing (JUnit in Eclipse) (4)



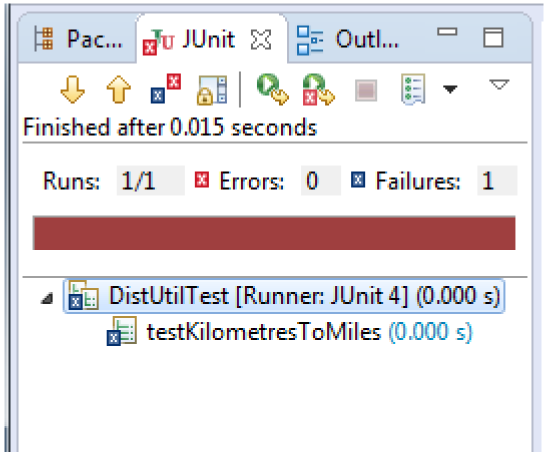
Java - Chapt7/src/BMITest.java - Eclipse



DistUtilTest

Runs: 1/1 Errors: 0 Failures: 0

DistUtilTest [Runner: JUnit 4] (0.000 s)
testKilometresToMiles (0.000 s)



Finished after 0.015 seconds

Runs: 1/1 Errors: 0 Failures: 1

DistUtilTest [Runner: JUnit 4] (0.000 s)
testKilometresToMiles (0.000 s)

Choosing Test Cases

- Test cases should represent valid and invalid inputs to test correctness and robustness
- Boundary cases often described by
  - If-statements
  - Loop conditions
- But what if you don't have access to the code?

Black-Box Testing

- Testing a program, class, or module without having access to its code
- Choose test cases based solely on contract information provided by API