UNIT AND INTEGRATION TESTING

CSC Software Testing Materials (Unit and Integration Testing)

@ Dr. Jessica Young Schmidt and NCSU Computer Science Faculty

WHAT IS UNIT & INTEGRATION TESTING?

- Code testing is known!
- We can use our code to guide our tests
- Exercises:
  - Independent paths within the source code
  - Logical decisions as both true and false
  - Loops at their boundaries
  - Internal data structures
- Unit Testing → testing the functionality of individual methods
- Integration testing: → testing how units of code work together

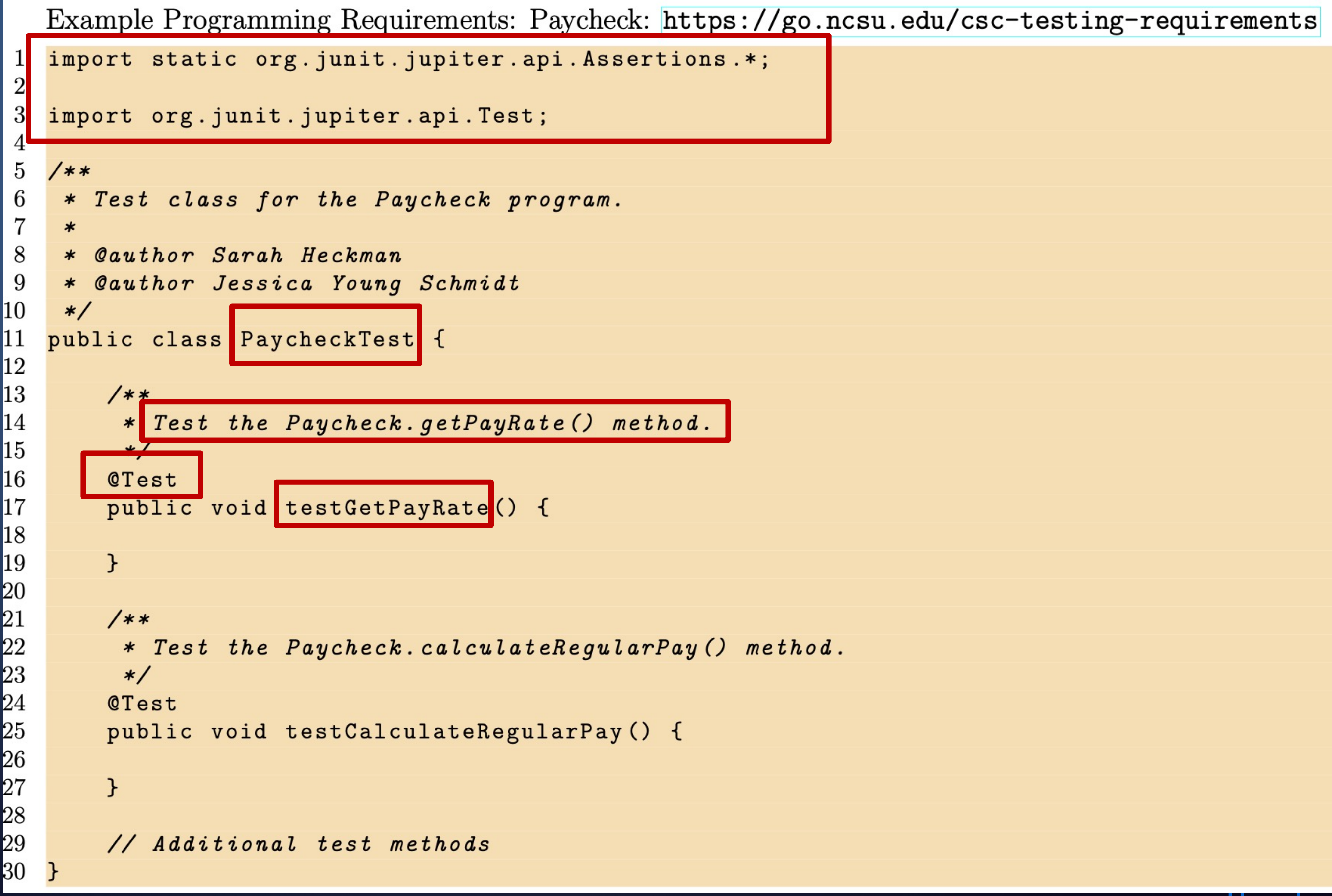
WRITING UNIT TESTS

- Focus on methods – how can we test? – Automation!
- Create a separate test class to exercise all of your program's methods (except main) – You will want to move most of your functionality out of main so you can test it
- Name your test class <NameOfSourceClass>Test.java
- Test class should be in the test directory
- Test methods
  - Test cases can be broken out into methods
  - One or more test method for each method under test.
  - Naming convention: test<MethodName><DescriptionOfTest>
  - Each test method will have one or more JUnit assert statements.

JUNIT

- Software testing framework for the Java programming language that reduces the complexity of implementing unit test cases for your code
- JUnit is not provided in the default Java libraries
- Download the JUnit libraries to lib (Junit5)
- JUnit Annotations
  - @Before is used to identify a method that executes before each of your individual test methods. This is useful for constructing new objects and ensures that each test executes with the same initial starting conditions.
  - @Test is used to identify each test method in your test class.

TEST CLASS SKELETON



Example Programming Requirements: Paycheck: https://go.ncsu.edu/csc-testing-requirements

```
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

/**
 * Test class for the Paycheck program.
 *
 * @author Sarah Heckman
 * @author Jessica Young Schmidt
 */
public class PaycheckTest {

    /**
     * Test the Paycheck.getPayRate() method.
     */
    @Test
    public void testGetPayRate() {

    }

    /**
     * Test the Paycheck.calculateRegularPay() method.
     */
    @Test
    public void testCalculateRegularPay() {

    }

    // Additional test methods
}
```

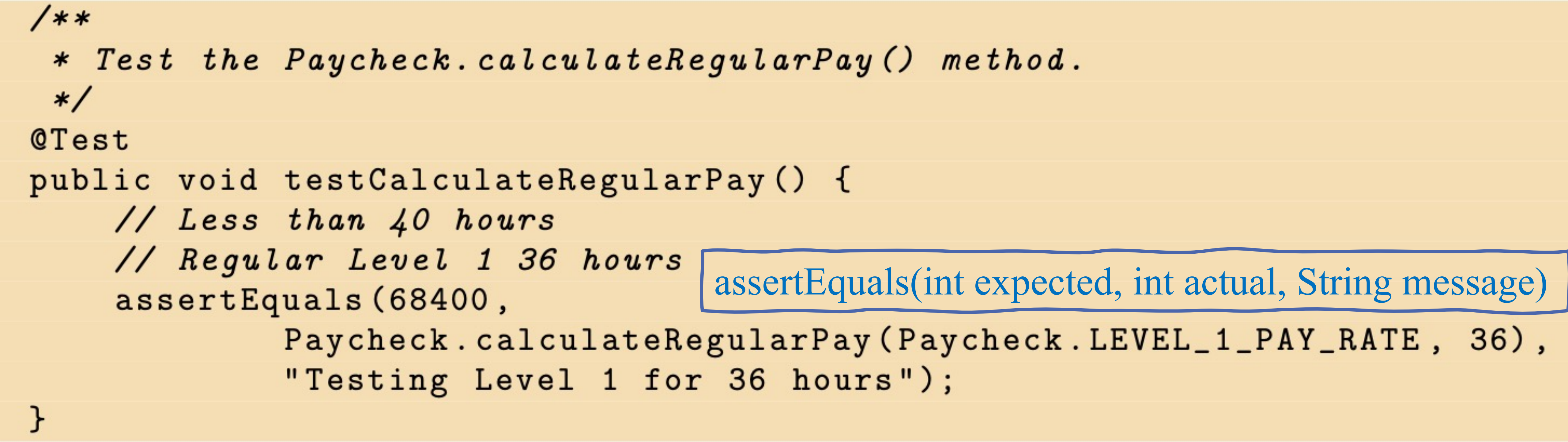
DIRECTORY STRUCTURE

Paycheck

- → src
    - → Paycheck.java
- → test
    - → PaycheckTest.java
- → lib
    - → junit-platform-console-standalone-1.6.2.jar
- → bin
    - → Paycheck.class
    - → PaycheckTest.class

ASSERT STATEMENTS

- assertTrue(boolean condition, String message)
- assertFalse(boolean condition, String message)
- assertEquals(int expected, int actual, String message)
- assertEquals(char expected, char actual, String message)
- assertEquals(Object expected, Object actual, String message)
- assertEquals(double expected, double actual, double delta, String message)



```
/**
 * Test the Paycheck.calculateRegularPay() method.
 */
@Test
public void testCalculateRegularPay() {
    // Less than 40 hours
    // Regular Level 1 36 hours
    assertEquals(68400,
            Paycheck.calculateRegularPay(Paycheck.LEVEL_1_PAY_RATE, 36),
            "Testing Level 1 for 36 hours");
}
```

assertEquals(int expected, int actual, String message)

COMPILE & EXECUTE TEST CASES

Compiling Source Code

```
javac -d bin -cp bin src/Paycheck.java
```

Executing Source Code

```
java -cp bin Paycheck
```

Compiling Test Cases

→ Mac/Linux → `javac -d bin -cp "bin:lib/*" test/PaycheckTest.java`

→ Windows → `javac -d bin -cp "bin;lib/*" test/PaycheckTest.java`

Executing Test Cases

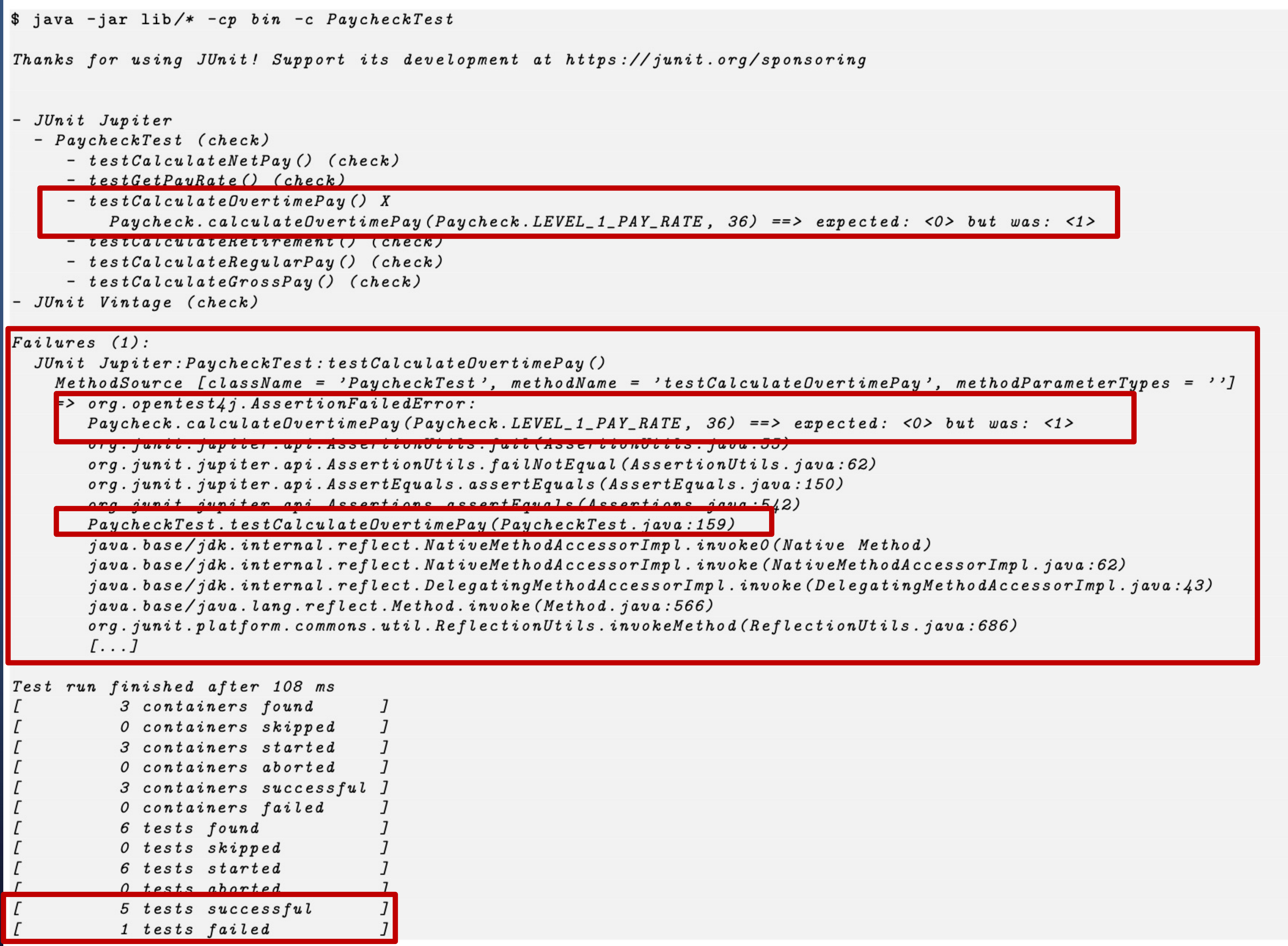
```
java -jar lib/* -cp bin -c PaycheckTest
```

DIRECTORY STRUCTURE

Paycheck

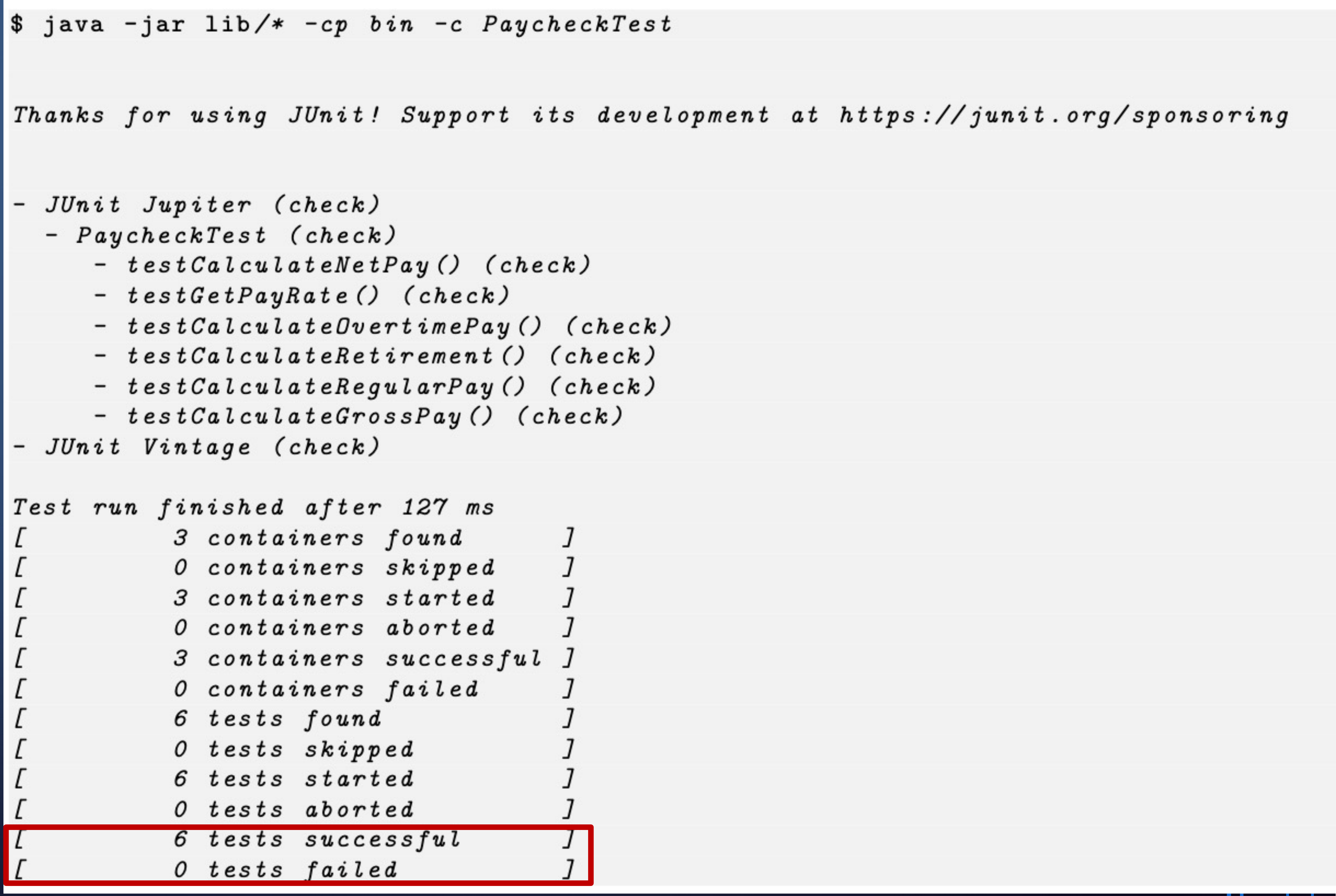
- → src
  - → Paycheck.java
- → test
  - → PaycheckTest.java
- → lib
  - → junit-platform-console-standalone-1.6.2.jar
- → bin
  - → Paycheck.class
  - → PaycheckTest.class

INTERPRETING THE RESULTS



```
$ java -jar lib/* -cp bin -c PaycheckTest

Thanks for using JUnit! Support its development at https://junit.org/sponsoring


- JUnit Jupiter
  - PaycheckTest (check)
     - testCalculateNetPay() (check)
     - testGetPayRate() (check)
     - testCalculateOvertimePay() X
         Paycheck.calculateOvertimePay(Paycheck.LEVEL_1_PAY_RATE, 36) ==> expected: <0> but was: <1>
     - testCalculateRetirement() (check)
     - testCalculateRegularPay() (check)
     - testCalculateGrossPay() (check)
- JUnit Vintage (check)

Failures (1):
  JUnit Jupiter:PaycheckTest:testCalculateOvertimePay()
    MethodSource [className = 'PaycheckTest', methodName = 'testCalculateOvertimePay', methodParameterTypes = '']
    => org.opentest4j.AssertionFailedError:
       Paycheck.calculateOvertimePay(Paycheck.LEVEL_1_PAY_RATE, 36) ==> expected: <0> but was: <1>
       org.junit.jupiter.api.AssertionUtils.fail(AssertionUtils.java:55)
       org.junit.jupiter.api.AssertionUtils.failNotEqual(AssertionUtils.java:62)
       org.junit.jupiter.api.AssertEquals.assertEquals(AssertEquals.java:150)
       org.junit.jupiter.api.Assertions.assertEquals(Assertions.java:542)
       PaycheckTest.testCalculateOvertimePay(PaycheckTest.java:159)
       java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
       java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
       java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
       java.base/java.lang.reflect.Method.invoke(Method.java:566)
       org.junit.platform.commons.util.ReflectionUtils.invokeMethod(ReflectionUtils.java:686)
       [...]

Test run finished after 108 ms
[         3 containers found      ]
[         0 containers skipped    ]
[         3 containers started    ]
[         0 containers aborted    ]
[         3 containers successful ]
[         0 containers failed     ]
[         6 tests found           ]
[         0 tests skipped         ]
[         6 tests started         ]
[         0 tests aborted         ]
[         5 tests successful      ]
[         1 tests failed          ]
```

INTERPRETING THE RESULTS



```
$ java -jar lib/* -cp bin -c PaycheckTest


Thanks for using JUnit! Support its development at https://junit.org/sponsoring


- JUnit Jupiter (check)
  - PaycheckTest (check)
     - testCalculateNetPay() (check)
     - testGetPayRate() (check)
     - testCalculateOvertimePay() (check)
     - testCalculateRetirement() (check)
     - testCalculateRegularPay() (check)
     - testCalculateGrossPay() (check)
- JUnit Vintage (check)

Test run finished after 127 ms
[         3 containers found      ]
[         0 containers skipped    ]
[         3 containers started    ]
[         0 containers aborted    ]
[         3 containers successful ]
[         0 containers failed     ]
[         6 tests found           ]
[         0 tests skipped         ]
[         6 tests started         ]
[         0 tests aborted         ]
[         6 tests successful      ]
[         0 tests failed          ]
```

TESTING STRATEGIES

- Test Requirements
- Test Equivalence Classes
- Test Boundary Values
- Test All Paths
- Test Exceptions

CONTROL FLOW DIAGRAM

- Pictorial description of the flow of program control
  - Diamonds → represent decisions
  - Rectangles → represent program statements
- Break apart compound conditionals
- Loops have one decision (the continuation test)
  - Unless the while loop has compound conditional tests – those should be broken up

CYCLOMATIC COMPLEXITY

- Measure of a method’s complexity
- Number of independent paths in the basis set of a method
  - Basis set → minimum number of paths that can be combined to generate every possible path paths may not be possible
- Use to estimate number of tests to write
  - Upper bound for the number of tests that must be conducted to cover the paths of a method
- Cyclomatic Complexity = number of decisions diamonds + 1

KEY POINTS

- With unit and integration testing, the code we are testing is known. We should use the code to guide our tests
  - One testing strategy is to ensure that every path in the method has been executed at least once.
  - We can determine all of the valid paths, called the basis set, through a method and write a test for each one.
  - Using equivalence classes to drive unit testing should identify most of the possible paths through a method.
- System testing should be completed along with unit and integration testing!