# 1. Introduction to Software Testing

Writing software is a process where developers take a vision of what an application should do in the form of requirements, design a solution, and then write code that implements the requirements. However, the code that is written may not exactly match the requirements of the system, resulting in lost revenue for a company (or for this class, lost points on an assignment).

Software testing is a process for identifying software faults. A software fault, or logic error, is defined as “an incorrect step, process, or data definition in a program” [1]. By systematically testing software, we can identify faults in our code and correct those faults as early as possible. The goal of testing is to remove as many faults as possible before releasing the application to our customer. The International Organization for Standardization (ISO) defines software testing as “the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite execution domains, against the expected behavior” [2]. The keywords here are:

- Dynamic: meaning we execute the code under test;
- Suitably selected: meaning we systematically and deliberately choose test inputs; and
- Expected behavior: meaning we know what we want the program to do, so that we will know when the program does not meet the requirements.

Software faults typically originate from a programmer mistake, which is “a human action that produces an incorrect result” [1]. A mistake in the requirements, design, or code, is a fault. Faults remains latent in the software until running the software produces a failure. A software failure is “an event in which a system or system component does not perform a required function within specified limits” [1]. Failures are when a program’s behavior deviates from the expected program behavior. Testing identifies failures in a program’s behavior. Investigation of the failure (guided by the failing test case) will lead to one or more underlying fault(s) in the software that must be fixed. This investigation and fixing of faults is called debugging.

Testing consists of many executions of a program and each execution has a different input with a different expected result. A distinct execution with a given input and expected output is a test case.
1.1. Test Cases

A test case contains (explicitly or implicitly) the following information:

1. A unique identifier;
2. A set of test inputs;
3. The expected results from executing the test inputs; and
4. The actual results of running the test case.

Unique Identifier: A unique identifier names a test case and the name should be descriptive of what the test verifies or what requirement(s) the test validates.

Test Inputs: A test case should have a set of one or more inputs. The inputs drive the test case and may be parameters to a method, user inputs to a user interface (UI), or files that an application uses. The key is that these inputs should be specific and repeatable. Both characteristics are required so that anyone can execute a test case and receive the expected results (assuming that the program has no faults). If specific inputs are not recorded, then more time may be spent trying to rediscover broken functionality, rather than fixing functionality.

Expected Results: The expected results describe the output from a correct execution of the program on a set of given inputs. By knowing what is correct, we can quickly identify when a program is not working. The expected results may be returned from a method or associated methods of a class, be a message or output to a UI, or information written to a file. All expected results should be specific values for clarity in determining if a test passes or fails. The expected results are determined by examining the requirements of the program.

Actual Results: The actual results record what happens when the test case is run. Any difference in expected and actual results represents a failing test case.

1.2. Testing Process

Program requirements document what the customer wants a program to do. The requirements define the expected results of a test case. If a test case fails, meaning the actual results of the program do not match the expected results, then we have not met our requirements. A test failure implies a debugging session to find the underlying fault is necessary. Figure 1 shows the flow for generating and running test cases on a program.
Developers start by writing their program and test cases. These tasks may be done simultaneously or asynchronously depending on personal preference; however, both the source code and the test cases are required before any tests can be run on a program.

A test case is run by executing the program with the test inputs specified by a test case. If the actual output does not match the expected output, the test fails. The failure helps guide developers to the underlying fault to fix the mistake.

*Passing tests increase confidence that a program meets the program requirements. If there is more time before a deadline, additional tests will increase confidence further. While software testing is useful for finding underlying faults, no amount of testing can unequivocally demonstrate the absence of faults in your code. Additional testing will provide some confidence that your program will perform as expected most of the time, at least for the cases you tested.*

Ultimately, your product must be delivered. Plan your time to allow for sufficient testing of the requirements to ensure delivery of a high quality product.

1.3. Testing Strategies

There are two basic testing techniques: Black Box testing (Section 2) and White Box testing (Section 3). The following sections contain basic testing techniques, including strategies for maximizing the effectiveness of testing. For you, the student, the cost of a failure (and the underlying fault) is a few points off of your programming assignment. For large corporations, the cost of releasing a failure to users can be in the millions, if not billions, of dollars. A mistake can also lead to the loss of customers’ confidence in the company.
1.4. Example Programming Scenario

In the following sections, we will use the following example to illustrate black box and white box testing of the Paycheck program:

Raleigh's Parks and Recreation Department hires landscapers to care for and maintain the city's parks.

An employee has one of three skill levels; each with a hourly pay rate:

Skill Level - Hourly Pay Rate ($)
-----------------------------------------
Level 1 - $19.00  
Level 2 - $22.50  
Level 3 - $25.75

All employees may opt in for insurance, which results in a deduction from their pay check.

Deduction - Weekly Cost ($)
---------------------------
Option 1 - Medical Insurance - $24.50  
Option 2 - Dental Insurance - $15.30  
Option 3 - Vision Insurance - $5.25

Employees at skill level 3 may also opt to place up to 6% of their gross pay into a retirement account.

The following information is printed about the employee's pay check: 1) employee's name 2) hours worked for a week 3) hourly pay rate 4) regular pay for up to 40 hours worked 5) overtime pay (1.5 pay rate) for hours over 40 worked 6) gross pay (regular + overtime) 7) total deductions 8) net pay (gross pay – total deductions). If the net pay is negative, meaning the deductions exceeds the gross pay, then an error is printed.

The Paycheck program prompts the user for information about the Employee, including the name, level (1, 2, or 3), hours worked, retirement percent, and whether he or she has medical, dental, and vision insurances. This program assumes a perfect user. There is no error checking for user input based on data type.

The following commands compile and run the Paycheck program:
$ javac Paycheck.java  
$ java Paycheck
2. Black Box Testing

Black box testing, which is also called functional or system testing, ignores the internals of the program and instead focuses on the inputs and the expected results of the program as defined by the requirements. The tester treats the program as a “black box”; the program implementation that generates the program output is unknown. The tester identifies the expected program output from the requirements and can compare the actual output with the expected output. Figure 2 presents a pictorial representation of black box test cases.

![Black Box Test Diagram]

Figure 2: Black box testing. The executable program, a black box, only considers the program’s input and output without regard to the internal workings (source code) of the program.

Black box testing can be used to find the following types of failures [3]:
- Incorrect or missing functions,
- Interface errors,
- Errors in data structures or external data base access,
- Behavior or performance errors, and
- Initialization and termination errors.

Typically, black box tests are run by an unbiased third-party, not the programmer who developed the code. However, in the absence of the third-party tester, a developer can test their own code by ignoring the internals. One good strategy for effective black box testing is to write the test cases before starting to write the code. Black box tests test what the customer wants the program to do and are obtained from the program requirements.

Like all test cases, black box test cases have an identifier, a set of specific and repeatable inputs, a set of expected results, and the actual results from executing the test case. The test information can be organized into a Black Box Test Plan document using the table in Table 1. The Black Box Test Plan should include an overview of the program, tests, and information about how to start testing the program. If the tests require any files, information about where those files are and the file contents should be included.
The test ID column of the test plan table contains a unique identifier for the test case. The unique identifier provides a common language for talking about test cases with developers and testers. The test ID could be a number, but a descriptive phrase about the test case is best. The test ID column can also contain additional information like the test author's name, the requirement under test, the type of test, etc.

The description column contains the steps and inputs the tester enters into the program. The description may also include preconditions for running the test. For example, the program must be started. The information that is recorded in the description should be specific and repeatable, which means that actual values are given. Instead of saying, “enter a dollar amount between 10 and 20” the description should say “enter 15.” For black box testing, the description provides all inputs a user must enter into a program.

The expected results column reports what the program should do as defined by the requirements. This means that the expected results are determined prior to running the program. Like the description column, the expected results should be specific. For black box tests cases, the expected results report what the program presents to the user through the user interface.

The actual results column is filled in when the test is executed. Ideally, the actual results should match the expected results. When they do not match, the test fails, and work to debug the program commences. Failing and passing test cases should be rerun after debugging. Passing tests should be rerun to ensure that any changes to the program under test did not cause a new fault.

We are covering three black box testing strategies: testing requirements, testing representative values of equivalence classes, and testing boundary values. Some of the test cases generated by one strategy may also be generated by another strategy.

---

### Table 1: Test case planning template

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Description</th>
<th>Expected Results</th>
<th>Actual Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.1. Test Requirements

When testing a program, you should ensure that all of the customer requirements are tested at least once. Our Paycheck program has the requirement that if the net pay is negative, meaning the deductions exceed the gross pay, then an error is printed. The following test case in Table 2 would test that requirement. The bold text in the description shows the values the tester would enter into the Paycheck program at the given prompts.

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Description</th>
<th>Expected Results</th>
<th>Actual Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boundary regular hours, All insurances, Level 1</td>
<td>Preconditions: Paycheck program started Employee Name: Alice Anderson Employee Level: 1 Hours Worked: 0 Medical Insurance (Y/N): Y Dental Insurance (Y/N): Y Vision Insurance (Y/N): Y</td>
<td>&lt;Error Message&gt;</td>
<td></td>
</tr>
</tbody>
</table>

2.2. Test Equivalence Classes

There is only a limited amount of time to test and an infinite number of possible test cases. We want to focus on test cases that will uncover new errors, not test cases that are essentially equivalent to other test cases. We can divide the input or output space into ranges or categories of data. We generate test cases by picking one representative value from each of the possible input ranges/categories or by picking input values that would get to a representative output value. If all of the values in an equivalence class are handled in a similar way, testing one representative value in each equivalence class increases confidence that the program would work with other inputs from the equivalence class. At a minimum, there are two equivalence classes for input: (1) valid or allowed input and (2) invalid or not allowed input.

The following guidelines are helpful for defining equivalence classes [3]:

- If the input condition is a range, you’ll create one valid and two invalid (above and below the valid range) equivalence classes.
- If input conditions are specific values, then the values are each a valid (for that value) equivalence class and all other values are in an invalid equivalence class.
- If input conditions are members of a set, then the members of the set are in the valid equivalence class and all other possibilities are in the invalid equivalence class.
- If an input condition is a boolean, there is one valid and one invalid equivalence class.

Representative values for an equivalence class are the specific valid/invalid values or a middle value in a range. If valid or invalid input is a range of numbers, choose a value that is in the middle of the range, rather than a value at range boundary. The boundary values are considered next (Section 2.3).
There are three main groups of input and output in the Paycheck program: skill levels, insurances, and hours worked. For employees of skill level 3, there is also the equivalence class for retirement. For skill levels, there equivalence classes for skill level 1, skill level 2, skill level 3, and invalid skill level. For each type of insurance, there are two equivalence classes: having insurance and not having insurance. For hours worked, there are three equivalence classes: regular hours, overtime hours, and invalid hours (negative).

Our test of the requirements may also satisfy equivalence classes. The test shown in Table 2 satisfies the equivalence class for a skill level 1. We can add other tests for skill levels as shown in Table 3. Note that to make a complete test execution, we are also providing input for all user input.

### Table 3: Equivalence class test for skill level

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Description</th>
<th>Expected Results¹</th>
<th>Actual Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mid-range regular hours, Two insurances, Level 1</td>
<td>Preconditions: Paycheck program started Employee Name: Carol Cole Employee Level: 1 Hours Worked: 10 Medical Insurance (Y/N): Y Dental Insurance (Y/N): N Vision Insurance (Y/N): Y</td>
<td>Name</td>
<td>OT</td>
</tr>
<tr>
<td>Carol Cole</td>
<td>0.00</td>
<td>190.00</td>
<td>29.75</td>
</tr>
<tr>
<td>Mid-range regular hours, Two insurances, Level 2</td>
<td>Preconditions: Paycheck program started Employee Name: Carol Cole Employee Level: 2 Hours Worked: 10 Medical Insurance (Y/N): Y Dental Insurance (Y/N): N Vision Insurance (Y/N): Y</td>
<td>Name</td>
<td>OT</td>
</tr>
<tr>
<td>Carol Cole</td>
<td>0.00</td>
<td>225.00</td>
<td>29.75</td>
</tr>
<tr>
<td>Mid-range regular hours, Two insurances, Level 3, Boundary retirement</td>
<td>Preconditions: Paycheck program started Employee Name: Carol Cole Employee Level: 3 Hours Worked: 10 Medical Insurance (Y/N): Y Dental Insurance (Y/N): N Vision Insurance (Y/N): Y</td>
<td>Name</td>
<td>OT</td>
</tr>
<tr>
<td>Carol Cole</td>
<td>0.00</td>
<td>257.50</td>
<td>32.32</td>
</tr>
</tbody>
</table>

In the above test cases, we consider yes and no for each of the insurance types. The two tests in Table 4 also test yes and no for each insurance.

¹ Due to space limitations in the tables (this table and following tables), the expected results are broken across six lines instead of two.
### Table 4: Equivalence class test for insurance

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Description</th>
<th>Expected Results</th>
<th>Actual Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mid-range regular hours, Two insurances, Level 1</td>
<td><strong>Preconditions:</strong> Paycheck program started &lt;br&gt; Employee Name: Danny D David &lt;br&gt; Employee Level: 1 &lt;br&gt; Hours Worked: 20 &lt;br&gt; Medical Insurance (Y/N): Y &lt;br&gt; Dental Insurance (Y/N): Y &lt;br&gt; Vision Insurance (Y/N): N</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Name&lt;br&gt; PayRate</td>
<td>Hours</td>
<td>Gross</td>
</tr>
<tr>
<td></td>
<td>Regular&lt;br&gt; Danny D David</td>
<td>20.00</td>
<td>380.00</td>
</tr>
<tr>
<td>Boundary regular hours, One insurance, Level 1</td>
<td><strong>Preconditions:</strong> Paycheck program started &lt;br&gt; Employee Name: Ellen Edwards &lt;br&gt; Employee Level: 1 &lt;br&gt; Hours Worked: 39 &lt;br&gt; Medical Insurance (Y/N): N &lt;br&gt; Dental Insurance (Y/N): N &lt;br&gt; Vision Insurance (Y/N): Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Name&lt;br&gt; PayRate</td>
<td>Hours</td>
<td>Gross</td>
</tr>
<tr>
<td></td>
<td>Regular&lt;br&gt; Ellen Edwards</td>
<td>39.00</td>
<td>741.00</td>
</tr>
</tbody>
</table>

Considering the possible equivalence classes for hours worked other tests of the requirements. For the Paycheck program, we would want at least one middle test that would result in each class. Tests for these equivalence classes are shown in Table 5.
### Table 5: Equivalence class tests for hours worked

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Description</th>
<th>Expected Results</th>
<th>Actual Results</th>
</tr>
</thead>
</table>
| Mid-range regular hours, Two insurances, Level 1 | **Preconditions:** Paycheck program started  
Employee Name: **Danny D David**  
Employee Level: 1  
Hours Worked: 20  
Medical Insurance (Y/N): Y  
Dental Insurance (Y/N): Y  
Vision Insurance (Y/N): N | Name | Hours | PayRate |
|         |             | Danny D David | | 20.00 | 19.00 | | 390.00 | 39.80 | 340.20 |

| Mid-range overtime hours, No insurance, Level 2 | **Preconditions:** Paycheck program started  
Employee Name: **Hilda Henderson**  
Employee Level: 2  
Hours Worked: 50  
Medical Insurance (Y/N): N  
Dental Insurance (Y/N): N  
Vision Insurance (Y/N): N | Name | Hours | PayRate |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Hilda Henderson</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 2.3. Test Boundary Values

Programmers tend to make mistakes at the boundaries of equivalence classes. Think of the mistakes you may have made when specifying the start and end points of a for loop. Boundary value analysis guides the creation of test cases at the boundaries or edges of a problem. When testing a boundary, you want to test the boundary itself. You also want to test the values immediately on either side of the boundary.

The input for hours worked for the Paycheck program has two boundaries as illustrated in Figure 3. We want to test the boundary of 0 and the values to either side. We also want to test the boundary of 40. Some tests for the boundary are shown in Table 6.

<table>
<thead>
<tr>
<th>hours worked &lt; 0</th>
<th>0 &lt;= hours worked &lt;= 40</th>
<th>hours worked &gt; 40</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑↑↑</td>
<td>↑↑↑</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3:** Boundary values for hours worked
### Table 6: Boundary value tests

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Description</th>
<th>Expected Results</th>
<th>Actual Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Boundary regular hours, Two insurances, Level 3, Mid-range retirement</strong></td>
<td>Preconditions: Paycheck program started Employee Name: Bob Baker Employee Level: 3 Hours Worked: 1 Medical Insurance (Y/N): N Dental Insurance (Y/N): Y Vision Insurance (Y/N): Y</td>
<td>Name</td>
<td>Hours</td>
</tr>
<tr>
<td>Bob Baker</td>
<td>Regular</td>
<td>1.00</td>
<td>25.75</td>
</tr>
<tr>
<td></td>
<td>OT</td>
<td>0.00</td>
<td>25.75</td>
</tr>
<tr>
<td></td>
<td>Gross</td>
<td>1004.25</td>
<td>55.46</td>
</tr>
<tr>
<td></td>
<td>Deduc.</td>
<td>948.79</td>
<td></td>
</tr>
<tr>
<td><strong>Boundary regular hours, One insurance, Level 3, Boundary retirement</strong></td>
<td>Preconditions: Paycheck program started Employee Name: Ellen Edwards Employee Level: 3 Hours Worked: 39 Medical Insurance (Y/N): N Dental Insurance (Y/N): N Vision Insurance (Y/N): Y</td>
<td>Name</td>
<td>Hours</td>
</tr>
<tr>
<td>Ellen Edwards</td>
<td>Regular</td>
<td>39.00</td>
<td>25.75</td>
</tr>
<tr>
<td></td>
<td>OT</td>
<td>0.00</td>
<td>25.75</td>
</tr>
<tr>
<td></td>
<td>Gross</td>
<td>1004.25</td>
<td>55.46</td>
</tr>
<tr>
<td></td>
<td>Deduc.</td>
<td>948.79</td>
<td></td>
</tr>
<tr>
<td><strong>Boundary regular hours, One insurance, Level 3, Boundary retirement</strong></td>
<td>Preconditions: Paycheck program started Employee Name: Frank Frankenstein Employee Level: 3 Hours Worked: 40 Medical Insurance (Y/N): Y Dental Insurance (Y/N): N Vision Insurance (Y/N): N</td>
<td>Name</td>
<td>Hours</td>
</tr>
<tr>
<td>Frank Frankenstein</td>
<td>Regular</td>
<td>40.00</td>
<td>25.75</td>
</tr>
<tr>
<td></td>
<td>OT</td>
<td>0.00</td>
<td>25.75</td>
</tr>
<tr>
<td></td>
<td>Gross</td>
<td>1030.00</td>
<td>86.30</td>
</tr>
<tr>
<td></td>
<td>Deduc.</td>
<td>943.70</td>
<td></td>
</tr>
<tr>
<td><strong>Boundary overtime hours, One insurance, Level 3, Mid-range retirement</strong></td>
<td>Preconditions: Paycheck program started Employee Name: George George Employee Level: 3 Hours Worked: 41 Medical Insurance (Y/N): N Dental Insurance (Y/N): Y Vision Insurance (Y/N): N</td>
<td>Name</td>
<td>Hours</td>
</tr>
<tr>
<td>George George</td>
<td>Regular</td>
<td>41.00</td>
<td>25.75</td>
</tr>
<tr>
<td></td>
<td>OT</td>
<td>0.00</td>
<td>25.75</td>
</tr>
<tr>
<td></td>
<td>Gross</td>
<td>1068.62</td>
<td>36.67</td>
</tr>
<tr>
<td></td>
<td>Deduc.</td>
<td>1031.95</td>
<td></td>
</tr>
</tbody>
</table>

### 3. White Box Testing

With white-box testing, the code under test is known, and we can use what we know about the code and the paths through the code to guide our tests. Additionally, our test cases and knowledge about the code can help guide us to the code locations likely to contain errors. White box testing techniques allow the tester to [3]: 1) exercise independent paths within the source code; 2) exercise logical decisions as both true and false; and 3) exercise loops at their boundaries. Later classes in the CSC curriculum at NCSU will provide instruction on using white-box testing to test internal data structures.

Typically, when performing white box testing, you want to focus on testing individual methods. This type of testing is called **unit testing**. Unit testing means that we are testing a specific unit of...
code, in our case, individual methods. When we start testing methods together, for example when one method calls another method, we are running integration tests. Integration testing means that we are testing how units of code work together. A combination of unit and integration testing can increase the confidence that small portions of our code work together. If the small portions work together, then it is more likely the full system will work correctly.

The test strategies discussed for black box testing (e.g., test requirements, test equivalence classes, and test boundaries) still apply for white box testing. The focus of the test is shifted from the entire program to a method or small unit of our code. With white box testing, we can consider another testing strategy: basis set testing. Since we know the code under test, we can write test cases to exercise all paths in the code. We will write white box tests using each of these strategies.

First, we will discuss how to structure our white box tests so that we may automate their execution.

### 3.1. Writing White Box Test Cases

For white box test cases, we can automate our test cases by creating a test class. The test class is another program that needs to be in the same directory as the program under test. To show the link between the test class and the code under test, the test class should be named `<ProgramUnderTest>Test.java`. Therefore, the test class for a program called Paycheck would be PaycheckTest.

As a test program, your test class will have a main method. Running the test class will also run the program under test because the test class will call the methods in the program under test. If our program under test contains static methods, we can call methods in the program under test similarly to how we call methods of the Math class:

```
<ProgramUnderTest>.<methodName>({parameters});
```

If our program under test contains instance methods (which we will discuss in Chapter 8), we will need to create an instance of the class under test before calling the methods.

When creating a test program, the test cases can be broken out into methods. You can have one or more test method for each method under test. A good naming convention is to call your test method `test<MethodName><DescriptionOfTest>`.

Each test method may contain many tests of the given method. The test must have the four standard test items: id, description, expected results, and actual results. All of these items may be printed to the console to show the tests in a table similar to the black box test plan. The `testResult()` method, shown in Figure 4 as part of skeleton test class for the Paycheck program, accepts a String for each of the test items and prints them in a nicely formatted row of a larger table.
/**
 * Test class for the Paycheck program.
 * @author Sarah Hecken
 */
public class PaycheckTest {

    /** Constants for passing and failing test output */
    public static final String PASS = "PASS";
    public static final String FAIL = "FAIL";

    /** Counters for test cases */
    public static int testCounter = 0;
    public static int passingTestCounter = 0;

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

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

    /**
     * Test the Paycheck.calculateGrossPay() method.
     */
    public static void testCalculateGrossPay() {
    }

    /**
     * Test the Paycheck.calculateTotalDeductions() method.
     */
    public static void testCalculateRetirement() {
    }

    /**
     * Test the Paycheck.calculateNetPay() method.
     */
    public static void testCalculateNetPay() {
    }

    /**
     * Prints the test information.
     * @param id id of the test
     * @param desc description of the test (e.g., method call)
     * @param exp expected result of the test
     */

private static void testResult(String id, String desc, String exp, String act) {
    testCounter++;
    String result = FAIL;
    if (exp.equals(act)) {
        result = PASS;
        passingTestCounter++;
    }
    System.out.printf("%30s%-65s%-15s%-15s%-7s\n", id, desc, exp, act, result);
}

// ** Starts the program.
// @param args command line arguments
// */
public static void main(String[] args) {
    testGetPayRate();
    testCalculateRegularPay();
    testCalculateOvertimePay();
    testCalculateGrossPay();
    testCalculateRetirement();
    testCalculateNetPay();
    System.out.printf("%4d / %4d passing tests", passingTestCounter, testCounter);
}

Figure 4: Skeleton of the Paycheck program.

3.2. Test Requirements

We start by testing the requirements of the method, which means we are testing the main functionality of the method. For Paycheck.calculateRegularPay(), the main functionality is that the method will return the employee's regular pay for the given pay rate and hours worked. The code for Paycheck.calculateRegularPay() is in Figure 5.
/**
  * Returns the employee's regular pay for the hours worked up to the first
  * REGULAR_PAY_MAX_HOURS hours worked.
  * @param payRate employee's pay rate
  * @param hoursWorked number of hours worked by the employee
  * @return employee's regular pay
  */
public static int calculateRegularPay(int payRate, double hoursWorked) {
    if (hoursWorked > REGULAR_PAY_MAX_HOURS) {
        return payRate * REGULAR_PAY_MAX_HOURS;
    }
    return (int)(payRate * hoursWorked);
}

Figure 5: Implementation of the calculateRegularPay method in the Paycheck program.

A simple test is shown in Figure 6, below. Note that each piece of test information is listed so that the information will be printed by the testResult() method. The description is a String version of the method call that generates the actual result for the test. Since the Paycheck.calculateRegularPay() method returns a int, the result is concatenated to the empty String to generate a String output for the actual results.

/**
  * Test the Paycheck.calculateRegularPay() method.
  */
public static void testCalculateRegularPay() {
    // Only testing valid values for the moment.

    // Less than 40 hours
    // Regular Level 1 36 hours
    String id = "Regular Level 1 36 hours";
    String desc = "Paycheck.calculateRegularPay(Paycheck.LEVEL_1_PAY_RATE," + " 36)";
    String exp = "68400";
    String act = ""
    + Paycheck.calculateRegularPay(Paycheck.LEVEL_1_PAY_RATE, 36);
    testResult(id, desc, exp, act);
}

Figure 6: Requirement test for Paycheck.calculateRegularPay() in PaycheckTest.testCalculateRegularPay().

3.3. Test Equivalence Classes
The strategy of testing representative values from equivalence classes still applies to white box testing. We break up the possible inputs for each parameter into equivalence classes and test representative values for each parameter. There are two parameters for
Paycheck.calculateRegularPay(): the pay rate and the hours worked. For the hours worked, the equivalence classes are hours less than 0, hours between 0 and 40, and hours greater than 40. Since we are focusing on representative values, we can choose values that are away from 40. Good representative values would be 36 and 46, shown in Figure 7.

```java
/**
 * Test the Paycheck.calculateRegularPay() method.
 */
public static void testCalculateRegularPay() {
    // Only testing valid values for the moment.

    // Less than 40 hours
    // Regular Level 1 36 hours
    String id = "Regular Level 1 36 hours";
    String desc = "Paycheck.calculateRegularPay(Paycheck.LEVEL_1_PAY_RATE," + " 36)";
    String exp = "68400";
    String act = "" + Paycheck.calculateRegularPay(Paycheck.LEVEL_1_PAY_RATE, 36);
    testResult(id, desc, exp, act);

    // Over 40 hours
    // Regular Level 1 46 hours
    id = "Regular Level 1 46 hours";
    desc = "Paycheck.calculateRegularPay(Paycheck.LEVEL_1_PAY_RATE, 46)";
    exp = "76000";
    act = "" + Paycheck.calculateRegularPay(Paycheck.LEVEL_1_PAY_RATE, 46);
    testResult(id, desc, exp, act);
}
```

Figure 7: Equivalence class tests for Paycheck.calculateRegularPay() in PaycheckTest.testCalculateRegularPay().

### 3.4. Test Boundary Values

Once representative values of a method are tested, boundary values between the equivalence classes (if there are boundary values) should be tested. For Paycheck.calculateRegularPay(), there are two boundaries: 1) at 0 hours and 2) at 40 hours. We'll focus on the boundary at 40. There are three tests to consider: 39 hours, 40 hours, and 41 hours. These correspond to the standard boundary value tests of at the boundary, one less than the boundary, and one more than the boundary. The boundary value tests are shown in Figure 8. Similar tests should be done for the lower boundaries.
/**
 * Test the Paycheck.calculateRegularPay() method.
 */
public static void testCalculateRegularPay() {
    // Testing boundary
    // Less than 40 hours
    // Regular Level 1 39 hours
    String id = "Regular Level 1 39 hours";
    String desc = "Paycheck.calculateRegularPay(Paycheck.LEVEL_1_PAY_RATE, 39)";
    String exp = "74100";
    String act = "" + Paycheck.calculateRegularPay(Paycheck.LEVEL_1_PAY_RATE, 39);
    testResult(id, desc, exp, act);

    // Regular Level 1 40 hours
    id = "Regular Level 1 40 hours";
    desc = "Paycheck.calculateRegularPay(Paycheck.LEVEL_1_PAY_RATE, 40)";
    exp = "76000";
    act = "" + Paycheck.calculateRegularPay(Paycheck.LEVEL_1_PAY_RATE, 40);
    testResult(id, desc, exp, act);

    // Over 40 hours
    // Regular Level 1 41 hours
    id = "Regular Level 1 41 hours";
    desc = "Paycheck.calculateRegularPay(Paycheck.LEVEL_1_PAY_RATE, 41)";
    exp = "76000";
    act = "" + Paycheck.calculateRegularPay(Paycheck.LEVEL_1_PAY_RATE, 41);
    testResult(id, desc, exp, act);
}

Figure 8: Boundary value tests for Paycheck.calculateRegularPay() in PaycheckTest.testCalculateRegularPay().

3.5. Test All Paths

Since the code under test is known, the code may be used to guide writing unit or integration tests for a method. 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 white box testing should identify most of the possible paths through a method. Supplementing equivalence class testing with information about the possible paths ensures that all statements in the method are executed at least once and increases confidence that the method works correctly.

When performing white-box testing we are interested in the control flow of the program. The control flow is a graph that contains decision points and executed statements. Each decision (conditional test in an if statement) in the method is shown as a diamond. The statements are in rectangles.
There are standard templates for each of the control structures that make decisions in our code. These templates are provided in Figure 9, a-g.

![Control Flow Diagram Templates](image)

Figure 9: Control flow diagram templates for standard control structures.

If the if statement contains compound conditional tests (i.e., the conditional tests are separated by `&&` or `||`), then each conditional test within the compound statement is shown as a separate diamond. Figure 10a-b shows standard templates for conditional statements with compound predicates. Figure 10a shows two predicates that are and-ed together. Both
statements have to be true for the statement on the right to execute. If either predicate is false, then the inner portion of the conditional test will never execute. Figure 10b shows two predicates that are or-ed together. Either statement can be true for the body of the conditional (represented by the lower statement) to execute. If both statements are false, then the body of the conditional test never executes.

A measure called *cyclomatic complexity* provides a guide for the number of possible paths through the code. When creating white box tests, we want to create a test case for each possible path through the code. There are several calculations for cyclomatic complexity, but the easiest is to add one to the number of decision nodes (diamonds) in the control flow graph. Cyclomatic complexity provides us with the upper bound of the number of tests that we should write to guarantee full execution of the method if the tests are chosen appropriately such that they cover the paths. That is the minimum set of test cases that we should write just for execution of all conditionals on their true and false paths. However, one or more of the paths may be invalid. If a program requires that the same conditional predicate is used in two sequential if statements, that predicate will always evaluate the same as long as there is no change to the value. The path where one predicate would first evaluate true and the second predicate would evaluate to false can never occur. That's why cyclomatic complexity is an estimate of the number of tests that you need for a method.

Once we have possible paths, we can create input values that will test each of the paths. Creating tests to consider all paths of statements is straightforward. Creating tests to consider all paths of loops is more complex. We could create many more paths that would execute the loop more than once, leading to a potentially infinite number of test cases. There is typically not enough time to run all possible test cases, so only focus on the paths through the code where a loop is run once through its body.
Pressman [3] provides the following guidance for testing a simple loop (i.e., no nesting), where the loop is expected to iterate \( n \) times.

- Fail the conditional test for entering the loop, so that the loop never executes;
- Execute the body of the loop only once;
- Execute the body of the loop twice;
- Execute the body of the loop \( m \) times, where \( m < n \);
- Execute the body of the loop \( n - 1 \) times;
- Execute the body of the loop \( n \) times; and
- Execute the body of the loop \( n + 1 \) times.

A loop’s execution ranges from the lower boundary to \( n \). The first set of three test cases test the loop’s lower boundary value. The 4\textsuperscript{th} test case is a representative equivalence class test of the loop’s input range. The last 3 test cases test loop’s upper boundary value. Some of the tests may lead to redundancies if the loop’s bounds are dependent on the input, so create as many distinct tests as possible, ensuring at a minimum coverage of all paths through the loop.

Nested loops introduce additional complexity when testing. Pressman [3] gives the following guidance for testing nested loops:

- Keeping all outer loops to minimal values that reduce the number of iterations, test the innermost loop using the techniques listed above.
- Move up the level of nested loops, and test the loop using the techniques listed above. The outer loops should kept to minimal iterations and any inner loops should be iterated a “typical” number of times.
- Continue moving up the level of nested loops until all loops are tested.

We will now use the test all paths strategy to test `Paycheck.calculateRegularPay()`. The control flow diagram for `Paycheck.calculateRegularPay()` is shown in Figure 11.

![Flow Chart](image)

**Figure 11:** Control flow graph for `Paycheck.calculateRegularPay()` method.
The cyclomatic complexity of Paycheck.calculateRegularPay() is $1 + 1 = 2$. This implies that there are as many as 2 valid paths through the method. The possible paths are:

- 2–3
- 2–4

The tests shown in Figure 6 and Figure 7 cover the two valid paths for the Paycheck.calculateRegularPay() method. Identifying the basis set of tests for a method can help identify requirements and equivalence class tests for a method. However, *be careful about only considering valid paths in your code! If your code is missing functionality, you won’t write tests for that. You should always think about requirements and equivalence classes when writing unit tests.*

### 3.5.1 Testing Exceptions

Some paths through code will contain conditions that result in the throwing of an exception. Tests should be written to test that the exceptions are thrown.

For testing exceptions, we will examine testing a factorial method (Figure 12) that is contained within a Factorial class.

```java
/**
 * Calculates the factorial of n
 * @param n number to calculate factorial of
 * @return n factorial (n!)
 * @throws IllegalArgumentException if n < 0
 */
public static int factorial(int n) {
    if (n < 0) {
        throw new IllegalArgumentException("negative n");
    }
    int product = 1;
    for (int i = 2; i <= n; i++) {
        product *= i;
    }
    return product;
}
```

**Figure 12:** factorial method that contains throwing an exception

To get the actual result, we will use a try/catch structure to examine whether the correct exception is thrown. Code for the white-box test is given in Figure 13.

```java
String id = "Invalid n: -4";
```
String desc = "Factorial.factorial(-4)";
String expected = "class java.lang.IllegalArgumentException";
String actual = "";
try {
    actual = "" + Factorial.factorial(-4);
} catch (IllegalArgumentException e) {
    actual = "" + e.getClass();
}
testResult(id, desc, expected, actual);

Figure 13: Code testing exception

4. Summary

Testing is an important part of the software development process and helps increase our confidence that our software meets the requirements set out by the customer (or your instructor). Systematic application of the strategies presented in this document will help you create a suite of test cases that will efficiently test your code. Black box testing is used to test the program as a whole by specifying program inputs and checking the generated outputs with the expected values. Black box testing should be completed along with white box testing. White box testing uses the source code to guide the development of test cases. Writing test cases for each method is a specific type of white box testing called unit testing. White box test cases can be written in another Java program, which allows for the automation of your testing efforts. Overall, testing is a technique that identifies underlying program faults by exercising the program you have developed.

5. References


6. Paycheck Appendix

This Appendix contains the full black box test plan, Paycheck source code, and full white-box test suite for reference. This example tests through Control Structures/Conditionals of the course textbook (Reges & Stepp, Chapter 4).

6.1. Full Black Box Test Plan

http://courses.ncsu.edu/csc116/common/Schmidt/Testing/version_2016_02/Paycheck/BlackBoxTestPlan_ControlStructures.pdf

6.2. Paycheck Program

http://courses.ncsu.edu/csc116/common/Schmidt/Testing/version_2016_02/Paycheck/Paycheck.java
6.3. Full White-Box Tests
http://courses.ncsu.edu/csc116/common/Schmidt/Testing/version_2016_02/Paycheck/PaycheckTest.java

7. RobustPaycheck Appendix
This Appendix contains the full black box test plan, RobustPaycheck source code, and full white-box test suite for reference. The example is robust because it ensures that the user enters valid values for the input. The example also allows the user to enter information for multiple employees. It tests through Indefinite Loops of the course textbook (Reges & Stepp, Chapter 5).

7.1. Full Black Box Test Plan
https://courses.ncsu.edu/csc116/common/Schmidt/Testing/version_2016_02/RobustPaycheck/BlackBoxTestPlan_IndefiniteLoops.pdf

7.2. RobustPaycheck Program
https://courses.ncsu.edu/csc116/common/Schmidt/Testing/version_2016_02/RobustPaycheck/RobustPaycheck.java

7.3. Full White-Box Tests
https://courses.ncsu.edu/csc116/common/Schmidt/Testing/version_2016_02/RobustPaycheck/RobustPaycheckTest.java

8. Paychecks Appendix
This Appendix contains the full black box test plan, Paychecks source code, and full white-box test suite for reference. The example inputs employee information through a file and outputs results to a file and error messages to the console. This example tests through File Input/Output of the course textbook (Reges & Stepp, Chapter 6).

8.1. Full Black Box Test Plan
https://courses.ncsu.edu/csc116/common/Schmidt/Testing/version_2016_02/Paychecks/BlackBoxTestPlan_FileIO.pdf

8.2. Paychecks Program
https://courses.ncsu.edu/csc116/common/Schmidt/Testing/version_2016_02/Paychecks/Paychecks.java

8.3. Full White-Box Tests
https://courses.ncsu.edu/csc116/common/Schmidt/Testing/version_2016_02/Paychecks/PaychecksTest.java