Handout: Refactoring and Unit Testing

This handout shows examples of refactoring and unit testing. In the refactoring example, a working block of Java code is refactored by a team during a code review. In the unit testing example, several JUnit tests are constructed to verify that refactored code.

Refactoring Example

In this example, a programming team in an investment bank is reviewing a block of code for a feature that calculates fees and bonuses for brokers who sell a certain kind of investment account to their corporate clients. The programmer who wrote the code uses refactoring to clarify problems that were identified during the code review.

The inspection team performed a code review on this block of Java code, which included the class `Account` and a function `calculateFee` from a different class:

```
1   class Account {
2       float principal;
3       float rate;
4       int daysActive;
5       int accountType;
6
7       public static final int  STANDARD = 0;
8       public static final int  BUDGET = 1;
9       public static final int  PREMIUM = 2;
10      public static final int  PREMIUM_PLUS = 3;
11  }
12
13  float calculateFee(Account accounts[]) {
14      float totalFee = 0;
15      Account account;
16      for (int i = 0; i < accounts.length; i++) {
17          account = accounts[i];
18          if ( account.accountType == Account.PREMIUM ||
19              account.accountType == Account.PREMIUM_PLUS ) {
20              totalFee += .0125 * ( account.principal
21                          * Math.exp( account.rate * (account.daysActive/365.25) )
22                          - account.principal );
23          }
24      }
25      return totalFee;
26  }
```

At first, the code seemed reasonably well-designed. But as the inspection team discussed it, a few problems emerged. One of the inspectors was not clear about the purpose of the calculation that was being performed on lines 20 to 22. The programmer explained that this was a compound interest calculation to figure out how much interest was earned on the account, and suggested that they use the *Extract Method* refactoring to clarify it. They performed the refactoring right there during the code review. Since this calculation only used data that was available in the `Account` class, they moved it into that class, adding a new method called `interestEarned` (in lines 12 to 15 below):

```
1   class Account {
2       float principal;
3       float rate;
4       int daysActive;
5       int accountType;
6
7       public static final int  STANDARD = 0;
8       public static final int  BUDGET = 1;
9       public static final int  PREMIUM = 2;
10      public static final int  PREMIUM_PLUS = 3;
11
12      float interestEarned() {
13          return ( principal * (float) Math.exp( rate * (daysActive / 365.25 ) ) )
14                  - principal;
15      }
16  }
17
18  float calculateFee(Account accounts[]) {
19      float totalFee = 0;
20      Account account;
21      for (int i = 0; i < accounts.length; i++) {
22          account = accounts[i];
23          if ( account.accountType == Account.PREMIUM ||
24               account.accountType == Account.PREMIUM_PLUS )
25              totalFee += .0125 * account.interestEarned();
26      }
27      return totalFee;
28  }
```

An inspector then asked what the number `.0125` in line 25 was, and if it could ever change in the future. It turned out that each broker earned a commission fee that was equal to 1.25% of the interest earned on the account. They used the *Replace Magic Number with Symbolic Constant* refactoring, replacing it with the constant `BROKER_FEE_PERCENT` and defining that constant later in line 31 (and adding a leading zero to help people read the code quickly):

```
1   class Account {
2       float principal;
3       float rate;
4       int daysActive;
5       int accountType;
6
7       public static final int  STANDARD = 0;
8       public static final int  BUDGET = 1;
9       public static final int  PREMIUM = 2;
10      public static final int  PREMIUM_PLUS = 3;
11
12      float interestEarned() {
13          return ( principal * (float) Math.exp( rate * (daysActive / 365.25 ) ) )
14                  - principal;
15      }
16  }
17
18  float calculateFee(Account accounts[]) {
19      float totalFee = 0;
20      Account account;
21      for (int i = 0; i < accounts.length; i++) {
22          account = accounts[i];
23          if ( account.accountType == Account.PREMIUM ||
24               account.accountType == Account.PREMIUM_PLUS ) {
25              totalFee += BROKER_FEE_PERCENT * account.interestEarned();
26          }
27      }
28      return totalFee;
29  }
30
31  static final double BROKER_FEE_PERCENT = 0.0125;
```

The next issue that was raised in the code review was confusion about why the `accountType` variable was being checked in lines 23 and 24. There were several account types, and it wasn't clear why the account was being checked for just these two types. The programmer explained that the brokers only earn a fee for premium accounts, which could either be of the type PREMIUM or PREMIUM_PLUS.

By using the *Decompose Conditional* refactoring, they were able to clarify the purpose of this code. Adding the isPremium function to the Account class (lines 17 to 22) made it more obvious that this was a check to verify whether the account was a premium account:

```
1   class Account {
2       float principal;
3       float rate;
4       int daysActive;
5       int accountType;
6
7       public static final int  STANDARD = 0;
8       public static final int  BUDGET = 1;
9       public static final int  PREMIUM = 2;
10      public static final int  PREMIUM_PLUS = 3;
11
12      float interestEarned() {
13          return ( principal * (float) Math.exp( rate * (daysActive / 365.25 ) ) )
14                  - principal;
15      }
16
17      public boolean isPremium() {
18          if (accountType == Account.PREMIUM || accountType == Account.PREMIUM_PLUS)
19              return true;
20          else
21              return false;
22      }
23  }
24
25  float calculateFee(Account accounts[]) {
26      float totalFee = 0;
27      Account account;
28      for (int i = 0; i < accounts.length; i++) {
29          account = accounts[i];
30          if ( account.isPremium() )
31              totalFee += BROKER_FEE_PERCENT * account.interestEarned();
32      }
33      return totalFee;
34  }
35
36  static final double BROKER_FEE_PERCENT = 0.0125;
```

The last problem found during the inspection involved the interestEarned() method which they had extracted. It was a confusing calculation, with several intermediate steps crammed into a single line. When that behavior was buried inside the larger function, the problem wasn't as glaring, but now that it had its own discrete function, they could get a clearer look at it.

The first problem was that it wasn't exactly clear why there was a division by 365.25 in line 13. The programmer explained that in the Account class daysActive represented the number of days that the account was active, but the rate was an annual interest rate, so they had to divide daysActive by 365.25 to convert it to years. Another programmer asked why principal was being subtracted at the end of the interest calculation. The explanation was that this was done because the fee calculation was based only on the interest earned, regardless of the principal that was initially put into the account.

The refactoring *Introduce Explaining Variable* was used to introduce two intermediate variables, years on line 13 and compoundInterest on line 14, to clarify the code:

```
1   class Account {
2       float principal;
3       float rate;
```

```
    int daysActive;
    int accountType;

    public static final int  STANDARD = 0;
    public static final int  BUDGET = 1;
    public static final int  PREMIUM = 2;
    public static final int  PREMIUM_PLUS = 3;

    float interestEarned() {
        float years = daysActive / (float) 365.25;
        float compoundInterest = principal * (float) Math.exp( rate * years );
        return ( compoundInterest – principal );
    }

    public boolean isPremium() {
        if (accountType == Account.PREMIUM || accountType == Account.PREMIUM_PLUS)
            return true;
        else
            return false;
    }
}

float calculateFee(Account accounts[]) {
    float totalFee = 0;
    Account account;
    for (int i = 0; i < accounts.length; i++) {
        account = accounts[i];
        if ( account.isPremium() ) {
            totalFee += BROKER_FEE_PERCENT * account.interestEarned();
        }
    }
    return totalFee;
}

static final double BROKER_FEE_PERCENT = 0.0125;
```

After these four refactorings, the inspection team agreed that the new version of this code was much easier to understand, even though it was almost 50% longer.

Unit Testing Example

The examples in this section are the individual test methods from a test case object called `testFeeCalculation`. There are many tests that would exercise the fee calculation function shown in the Refactoring section above. This example shows six of them. All of them require an instance of the *FeeCalculation* class, which is set up using this `setUp()` function:

```
public FeeCalculation feeCalculation;
public void setUp() {
    feeCalculation = new FeeCalculation();
}
```

The first test simply verifies that the function has performed its calculation and has generated the right result by comparing the output to a known value, which was calculated by hand using a calculator.

```
public void testTypicalResults() {
    Account accounts[] = new Account[3];

    accounts[0] = new Account();
    accounts[0].principal = 35;
    accounts[0].rate = (float) .04;
    accounts[0].daysActive = 365;
    accounts[0].accountType = Account.PREMIUM;
    accounts[1] = new Account();
    accounts[1].principal = 100;
    accounts[1].rate = (float) .035;
    accounts[1].daysActive = 100;
    accounts[1].accountType = Account.BUDGET;

    accounts[2] = new Account();
    accounts[2].principal = 50;
    accounts[2].rate = (float) .04;
    accounts[2].daysActive = 600;
    accounts[2].accountType = Account.PREMIUM_PLUS;

    float result = feeCalculation.calculateFee(accounts);
    assertEquals(result, (float) 0.060289, (float) 0.00001);
}
```

This test passes. The call to `feeCalculation()` with those three accounts returns a value of *0.060289383*, which matches the value passed to `assertEquals()` within the specified tolerance of *.000001*. The assertion does not cause a failure, and the test case completes.

It's important to test unexpected input. The programmer may not have expected `feeCalculation()` to receive a set of accounts that contained no premium accounts. So the second test checks for a set of non-premium accounts:

```
public void testNonPremiumAccounts() {
    Account accounts[] = new Account[2];

    accounts[0] = new Account();
    accounts[0].principal = 12;
    accounts[0].rate = (float) .025;
    accounts[0].daysActive = 100;
    accounts[0].accountType = Account.BUDGET;

    accounts[1] = new Account();
    accounts[1].principal = 50;
    accounts[1].rate = (float) .0265;
    accounts[1].daysActive = 150;
    accounts[1].accountType = Account.STANDARD;

    float result = feeCalculation.calculateFee(accounts);
    assertEquals(result, 0, 0.0001);
}
```

The expected result for this test is *0*, and it passes.

It's not enough to just test for expected results. A good unit test suite will include tests for *boundary conditions*, or inputs at the edge of the range of acceptable values. There are many kinds of boundary conditions, including:

- Zero values, null values or other kinds of empty or missing values
- Very large or very small numbers that don't conform to expectations (like a rate of 10000%, or an account that has been active for a million years)
- Arrays and lists that contain duplicates or are sorted in unexpected ways
- Events that happen out of order, like accessing a database before it's opened
- Badly formatted data (like an invalid XML file)

A few tests will verify that these boundary conditions are handled as expected. This unit test verifies that `calculateFee()` can handle an account with a zero interest rate:

```
public void testZeroRate() {
    Account accounts[] = new Account[1];

    accounts[0] = new Account();
    accounts[0].principal = 1000;
    accounts[0].rate = (float) 0;
    accounts[0].daysActive = 100;
    accounts[0].accountType = Account.PREMIUM;

    float result = feeCalculation.calculateFee(accounts);
    assertEquals(result, 0, 0.00001);
}
```

This test passes in an account with a negative principal (a calculator was used to come up with the expected result by hand):

```
public void testNegativePrincipal() {
    Account accounts[] = new Account[1];

    accounts[0] = new Account();
    accounts[0].principal = -10000;
    accounts[0].rate = (float) 0.263;
    accounts[0].daysActive = 100;
    accounts[0].accountType = Account.PREMIUM;

    float result = feeCalculation.calculateFee(accounts);
    assertEquals(result, -9.33265, 0.0001);
}
```

In this case, the programmer expects the correct mathematical result to be returned, even though it may not make business sense in this context. Another programmer maintaining the code can see this expectation simply by reading through this unit test.

The next test verifies that the software can handle a duplicate reference. `feeCalculation()` takes an array of objects. Even if one of those objects is a duplicate reference of another one in the array, the result should still match the one calculated by hand:

```
public void testDuplicateReference() {
    Account accounts[] = new Account[3];

    accounts[0] = new Account();
    accounts[0].principal = 35;
    accounts[0].rate = (float) .04;
    accounts[0].daysActive = 365;
    accounts[0].accountType = Account.PREMIUM;

    accounts[1] = accounts[0];

    accounts[2] = new Account();
    accounts[2].principal = 50;
    accounts[2].rate = (float) .04;
    accounts[2].daysActive = 600;
    accounts[2].accountType = Account.PREMIUM_PLUS;

    float result = feeCalculation.calculateFee(accounts);
    assertEquals(result, 0.0781316, 0.000001);
}
```

It's also possible to create tests that are expected to fail. The programmer expects `calculateFee()` to choke on one particular boundary condition—being passed *null* instead of an array:

```
public void testNullInput() {
    Account accounts[] = null;
    float result = feeCalculation.calculateFee(accounts);
    assertTrue(true);
}
```

The assertion `assertTrue(true)` will never fail. It's included for the benefit of any programmer reading this unit test. It shows that the test is expected to get to this line. Unfortunately, `calculateFee` throws a *NullPointerException* error.

In this case, that's exactly the behavior that the programmer expects. The unit test can be altered to show that it expects the call to `calculateFee()` to fail:

```
public void testNullInput() {
    Account accounts[] = null;
    try {
         float result = feeCalculation.calculateFee(accounts);
         fail();
    } catch (NullPointerException e) {
         assertTrue(true);
    }
}
```

The `fail()` assertion is placed after `calculateFee()` to verify that it throws an exception and never executes the next statement. The `assertTrue(true)` assertion is then used to show that the call is expected to throw a specific error, and the test expects to catch it.

These test methods by no means represent an exhaustive test case for the `FeeCalculation` class. But even this limited set of tests is enough, for instance, to ensure that a refactoring has not broken the behavior of the class. It would not take a programmer much longer to come up with a more exhaustive test case for this example.