EECS 4313: Assignment 2

Monday, 29 February 16

Team Members

```
- Drew Noel (212513784)
- Skyler Layne (212166906)
- Siraj Rauff (212592192)
```

Method Test #1:

The method under investigation, `calculateRepeatNumber` exists within `Repeat.java` in the `net.sf.borg.model` package, and has the following signature including java doc:

```
/**
 * Calculate the number of a repeat given the date and the appointment
 *
 * @param current
 *            the date
 * @param appt
 *            the appointment
 *
 * @return the number of the repeat (starting with 1)
 */
final static public int calculateRepeatNumber(Calendar current, Appointment appt)
```

The method calculates the instance number of the specified repeating event on a particular date. This means that a repeating event will have its instances numerated, and the instance number on the given date will be rturned.

- The first parameter **current**, of type `Calendar`, contains the date for which the instance number is to be determined.
- The second parameter **appt**, of type `Appointment`, contains the event specifics including the start date and repeat frequency.

If a date is given such that it does not exist in the repetitions (*i.e.* a weekend for a event repeating on weekdays, or a date before the start date), the value `0` is returned.

Boundary Value Testing

Having observed the behavior and documentation of the function, it was determined that the distances between the start date of the `Appointment` and the `Calendar` date should be pushed to extreme Integer values. This was chosen after inspection of the code revealed that the repetition was calculated using a loop tracking an integer value, without proper checking for overflow.

Note that the `Appointment` parameter was not Boundary tested as its use in this function was limited to its start date and repeating frequency. The lowest frequency value in BORGCalendar was `DAILY`, therefore we could not test for multiple instances for a given date, and testing for different values would fall within the scope of testing `Calendar`. Testing for extreme values of a start date would not prove to be useful as we would be testing Java's `DATE` class, and not the method at hand.

Given a specific `Appointment`, the following cases were then chosen for the `Calendar`: - The same day - The next day - The previous day - Integer.MAX_VALUE - 1 days in the future - Integer.MAX_VALUE days in the future - Integer.MAX_VALUE + 1 days in the future - Integer.MIN_VALUE + 1 days in the past - Integer.MIN_VALUE days in the past - Integer.MIN_VALUE - 1 days in the past

Special Value Testing

Due to the nature of the `Appointment` parameter, in particular its use for start date and repeating frequency, two special cases were considered: - Appointment without a start date - Appointment with no frequency

These were tested to ensure the method was robust, as this custom class does not ensure that these two fields are set upon creation or modificaiton.

Method Test #2:

The method under test belongs to the `EncryptionHelper` class, located in the package `net.sf.borg.common`. The method has the following signature, including java doc:

```java
/**
* encrypt a String using a key from the key store
*
* @param clearText
*           - the string to encrypt
* @param keyAlias
*           - the encryption key alias
* @return the encrypted string
* @throws Exception
*/
public String encrypt(String clearText, String keyAlias);
```

This method accepts two arguments: a string which should be encrypted, and a string which informs the method which key from the `Keystore` to use. The `Keystore` is created through static methods in the class ahead of time. This method throws an `Exception` if there is any type of exception thrown during its execution.

The technique selected to test this method is the Boundary-Value Test strategy. Through examining the source code, it is clear that `keyAlias` is only manipulated by core Java methods and classes, and so testing values of `keyAlias` was given a lower priority. The argument `clearText` accepts any `String` as input. By treating the length of the string as the component to test, the following value breakdown was created:

- A string with zero length
- A string with length of one
- A string with a nominal length
- A string which is exceptionally large
- A string comprised of binary data

It is not possible to perform a Strong Boundary-Value Test because a String cannot have length that is negative, and String has no practical upper limit. Note that it would be possible to design a test which forcibly exhausts the memory of the JVM by creating an enormous string as input. However this would be a failure point on the JVM, which the application is not expected to be able to handle.

The final test value was created after inspecting the source code. In the method, there is a call to the `String` method `getBytes()`. This method uses the default encoding, which may or may not fundamentally change the data it receives.

By changing the value of `keyAlias` to be either a valid key or an invalid key (similar to a boolean), a second breakdown was created:

- A string which describes a valid key
- A string which does not describe a valid key

The tests were written to attempt every possible combination of these test values, creating a Worst-Case Boundary-Value Test suite. Examination of the source code indicated that the result of an invalid key description will always be a thrown `Exception`. Specifically the type of the exception will be an `InvalidKeyException`. There were several other types of exceptions that this method might throw, but since the signature indicates it may throw any `Exception`, this was not tested.

All the tests described above passed when execute. It is worthwhile to observe that this method, like most methods in the `net.sf.borg.common` package, gives no information about the exception types that it might throw. This significantly increases the amount of code that the client classes will require in order to properly determine why an exception was thrown at all.

Method Test #3:

The method under test has the following signature, including java doc:

```
/**
 * Gets the "N" multiplier value from the encoded appointment string
 *
 * @param f
 *            the encoded appointment string
 *
 * @return the "N" multiplier value
 */
static public int getNValue(String f);
```

Using the testing techniques which we have discussed in class we have devised the following strategy for testing the above stubbed method. Upon inspection, the method under test takes as an argument a `String`. This was troublesome at first however, looking further into this we were able to deduce what, in this case, a valid `String` is. In this context a valid input string is one which contains some word, *Constant*, and a comma separated integer, *Number* such that it has the form `"Constant,Number"`. In the context of testing we can break this up into essentially *2* inputs, the Constant and the Number. This is important to note from the start as we use this context through the entire test.

Boundary Value Testing

As the input is a `String` and the output is an `int`, we can carefully consider the edge cases given that the input is of the form `Constant + "," + Number` (See Appendix A for the JUnit test cases for this strategy):

Constant:
- The empty `String`.
- A very long `String`.
- A desirable `String`.
- Without a comma separator.

Number:
- The empty `String`.
- A non-numeric `String`.
- A very large negative `int` and `float`.
- Zero.
- A very large positive `int` and `float`.

Equivalence Class Testing

Using the following strategy we attempt to cover some of the missing gaps, and remove some of the redundancy from the *Boundary Value Tests*. The following diagrams present this testing strategy, each of the possible Number inputs are on the Y axis, while each of the possible Constant inputs are on the X axis. Each of the filled in boxes presents valid input, while the X represents a corresponding test case (See Appendix A for the JUnit test cases for this strategy).

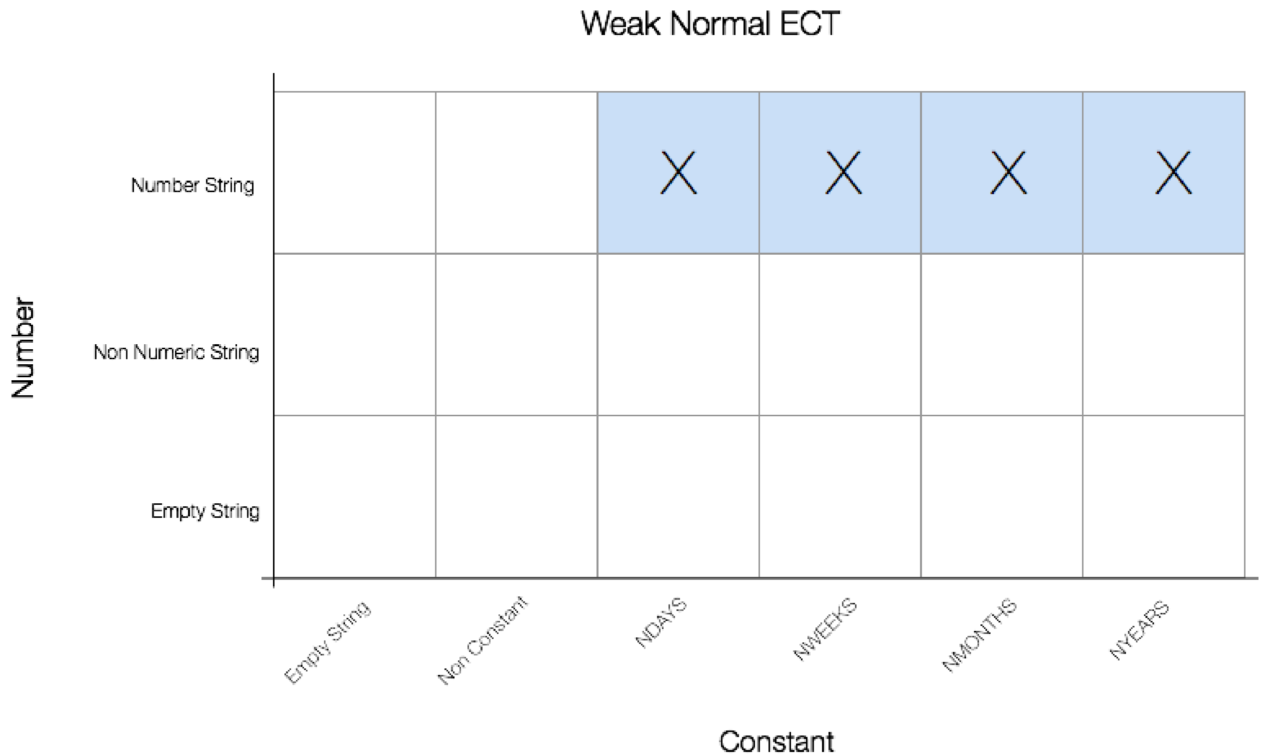

Figure 1: Weak Normal Test Cases

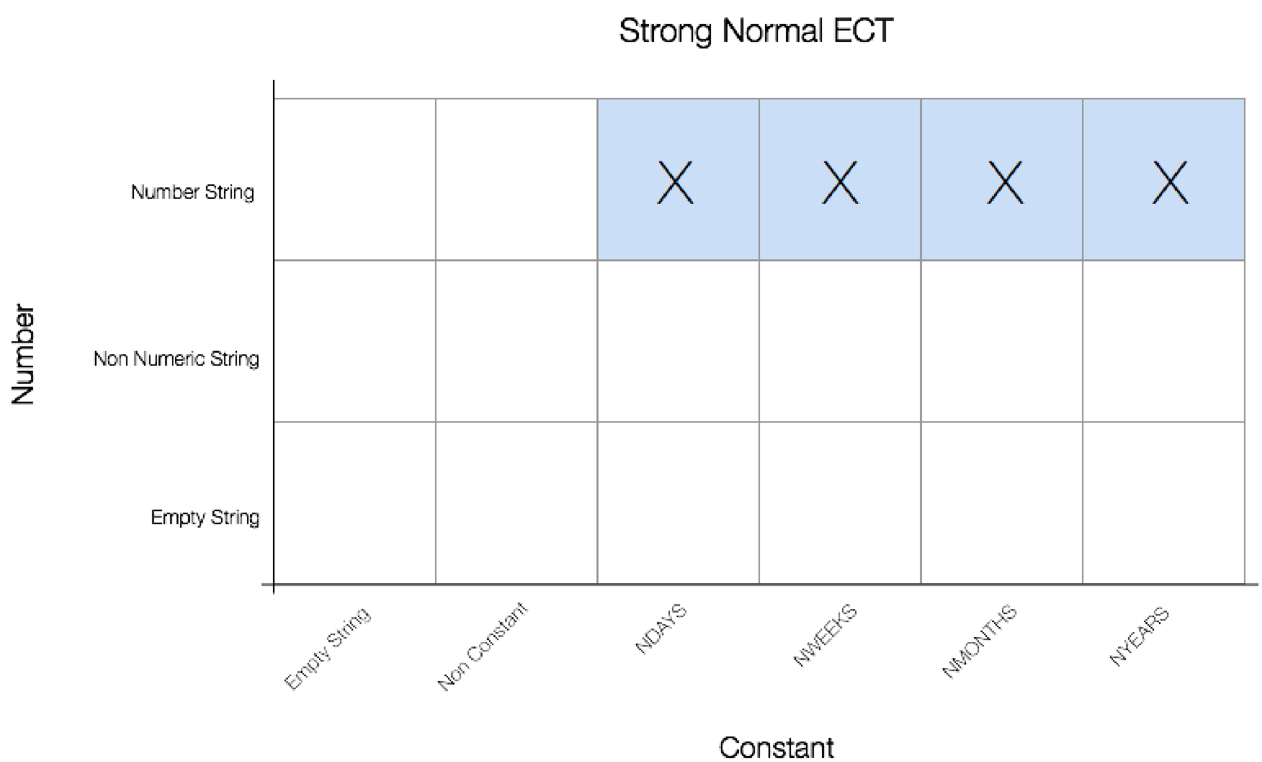

Figure 2: Strong Normal Test Cases

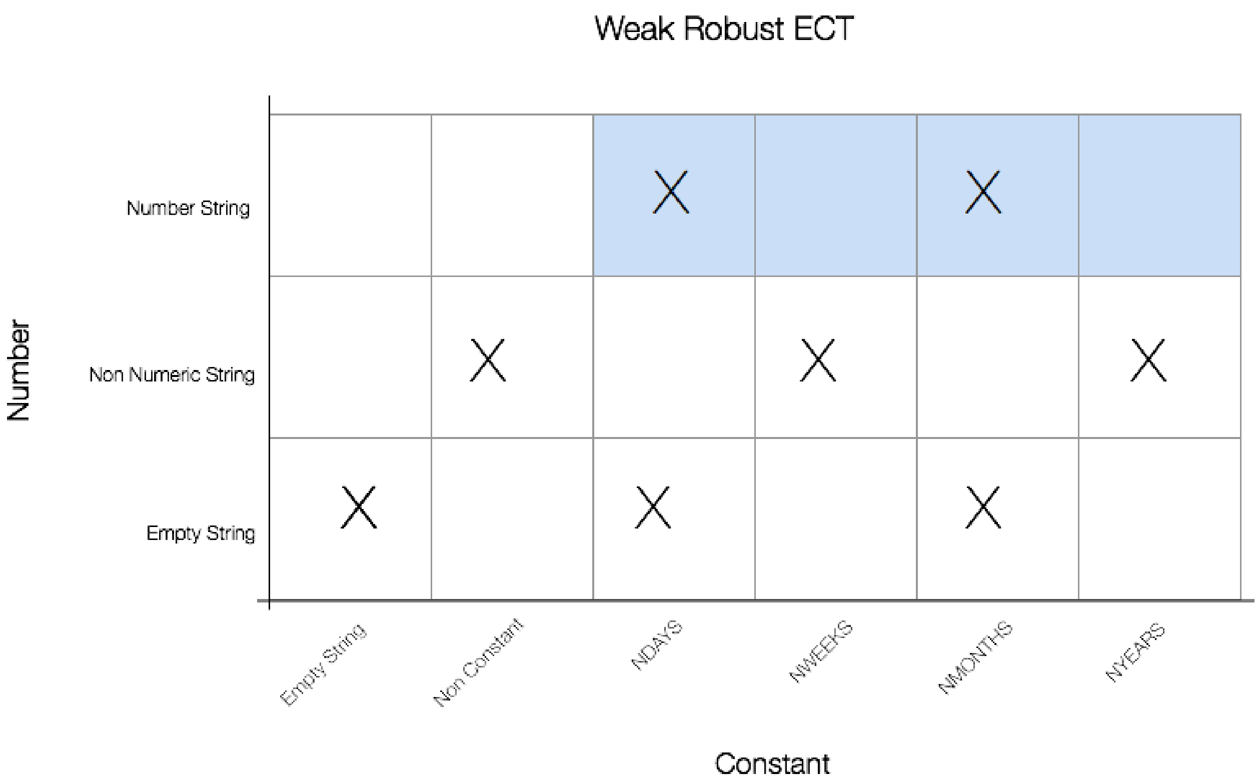

Figure 3: Weak Robust Test Cases

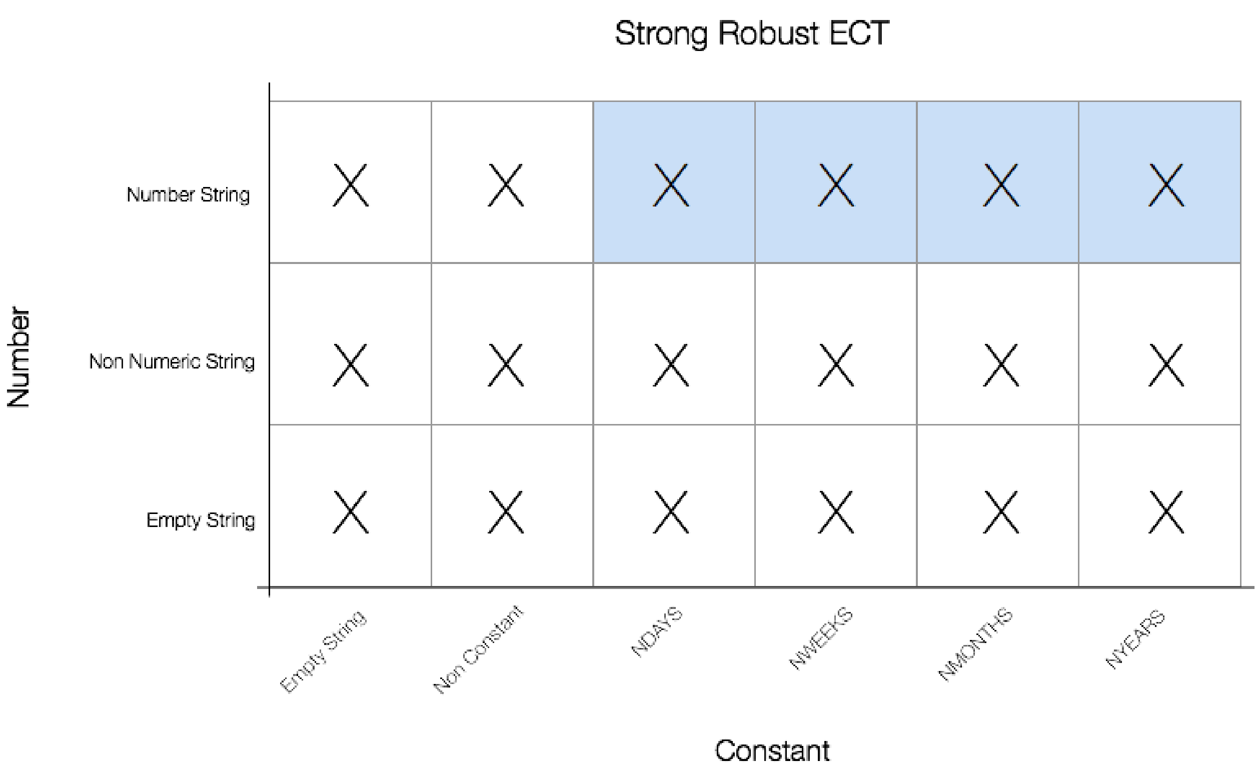

Figure 4: Strong Robust Test Cases

Decision Table Testing

In the following decision table we show the desired behaviour of the system; under the condition what the desired action should be (See Appendix A for the JUnit test cases for this strategy).

Method Under Test: *static public int getNValue(String f)*							
Conditions	Input empty	T	F	F	F	F
	Input does not contain a valid constant or valid number	-	T	F	F	F
	Input contains a valid number but not a valid constant	-	-	T	F	F
	Input contains a constant but no number of occurrences	-	-	-	T	F
	Input contains a valid constant and valid number	-	-	-	-	T
Actions	Return 0	X	X	X	X	
	The extracted number					X

Figure 5: Decision Table

Appendix A

Bug Report #1

Title: Integer Overflow in `calculateRepeatNumber`.
Reported by: Siraj Rauff.
Date Reported: February 28, 2016.
Program name: BORG Calendar.
Configuration: OS X 10.11.3, Java version 1.8.0_71, Runtime build 1.8.0_71-b15.
Report type: Bug.
Reproducibility: Yes – consistently.
Priority: Medium.
Problem Summary:
`calculateRepeatNumber` in `Repeat.java` does not account for integer overflow in its calculation.
Problem Description:
`calculateRepeatNumber` utilizes a for loop, keeping track of the number of occurrences of the repeating event using an integer variable. (`Repeat.java`, [446,0])

```
for (int i = 1;; i++) { ... }
```

This section of the method does not, however, check for integer overflow. This means that any event that occurs more than `Integer.MAX_VALUE` times, the method will simply wrap around to `Integer.MIN_VALUE` and continue counting. Depending on the implementation of infinitely repeating events, a event could quite reasonably have more than `Integer.MAX_VALUE` instances.

This can be reproduced with the following:

```
Calendar testCal = new GregorianCalendar(0,0,1,0,0);
Appointment testAppt = new Appointment();
testAppt.setFrequency("DAILY");
testAppt.setDate(testCal.getTime());

testCal.add(Calendar.DAY_OF_MONTH, Integer.MAX_VALUE);
testCal.add(Calendar.DAY_OF_MONTH, 1);
assertEquals(greaterThan(Integer.MAX_VALUE), Repeat.calculateRepeatNumber(testCal,
    testAppt));
```

New or old bug: old.

Bug Report #2

Title: Unchecked `NullPointerException` in `calculateRepeatNumber`.
Reported by: Siraj Rauff.
Date Reported: February 28, 2016.
Program name: BORG Calendar.
Configuration: OS X 10.11.3, Java version 1.8.0_71, Runtime build 1.8.0_71-b15.
Report type: Bug.
Reproducibility: Yes – consistently.
Priority: Low.
Problem Summary:
`calculateRepeatNumber` in `Repeat.java` does not check if the `Appointment` contains a valid start date before using it.
Problem Description:

`calculateRepeatNumber` utilizes an `Appointment` parameter to get the start date and repeat frequency of an event. It does not, however, check if either of these exist. A `null` frequency is correctly handled in the creation of a repeating event, however a `null` start date will cause a `NullPointerException`.

This can be reproduced with the following:

```
Calendar testCal = new GregorianCalendar(0,0,1,0,0);
Appointment testAppt = new Appointment();

assertEquals(0, Repeat.calculateRepeatNumber(testCal, testAppt));
```

New or old bug: old.

Bug Report #3

Title: Unchecked NumberFormatException.
Reported by: Skyler Layne.
Date Reported: February 28, 2016.
Program name: BORG Calendar.
Configuration: OS X 10.11.3, Java version 1.8.0_60, Runtime 1.8.0_60-b27.
Report type: Bug.
Reproducibility: Yes – consistently.
Priority: Low.
Problem Summary:

Unit testing surfaced an uncaught error in String to Integer conversion.

When setting a repeat event, if the String code passed in takes space, or if it is not an integer, then an uncaught/unhandled error occurs `NumberFormatException` error occurs. Example inputs:

```
Repeat.getNValue("nweeks, ");
Repeat.getNValue("nweeks, 14");
Repeat.getNValue("nweeks, FOO");
Repeat.getNValue("nweeks,FOO");
```

New or old bug: New.

Appendix B

Method #1

Boundary Value Tests

```
/*
 * Testing Boundry Values of Calculate Repeat Number
 */
@Test
public void testCalculateRepeatNumberBVT() {

  /*
   * Boundary Values for Calendar:
   * 1. The same day
   * 2. The next day
   * 3. The previous day
   * 4. Integer.MAX_VALUE - 1 days in the future
   * 5. Integer.MAX_VALUE days in the future
   * 6. Integer.MAX_VALUE + 1 days in the future
   * 7. Integer.MIN_VALUE + 1 days in the past
   * 8. Integer.MIN_VALUE days in the past
   * 9. Integer.MIN_VALUE - 1 days in the past
   */

  Calendar prevDay = new GregorianCalendar(0,0,1,0,0);
  Calendar sameDay = new GregorianCalendar(0,0,2,0,0);
  Calendar nextDay = new GregorianCalendar(0,0,3,0,0);
  Appointment sampleAppt = new Appointment();
  sampleAppt.setFrequency("DAILY");

  // 1. The same day
  sampleAppt.setDate(sameDay.getTime());
  assertEquals(1, Repeat.calculateRepeatNumber(sameDay, sampleAppt));

  // 2. The next day
  assertEquals(2, Repeat.calculateRepeatNumber(nextDay, sampleAppt));

  // 3. The previous day
  assertEquals(0, Repeat.calculateRepeatNumber(prevDay, sampleAppt));

  // 4. Integer.MAX_VALUE - 1 days in the future
  Calendar overMaxDays = sameDay;
  overMaxDays.add(Calendar.DAY_OF_MONTH, Integer.MAX_VALUE - 1);
  assertEquals(Integer.MAX_VALUE, Repeat.calculateRepeatNumber(overMaxDays, sampleAppt));

  // 5. Integer.MAX_VALUE days in the future
  overMaxDays.add(Calendar.DAY_OF_MONTH, 1);
  assertEquals(greaterThan(Integer.MAX_VALUE), Repeat.calculateRepeatNumber(overMaxDays,
      sampleAppt));

  // 6. Integer.MAX_VALUE days in the future
  overMaxDays.add(Calendar.DAY_OF_MONTH, 1);
```

```java
  assertEquals(greaterThan(Integer.MAX_VALUE), Repeat.calculateRepeatNumber(overMaxDays,
      sampleAppt));

  // 7. Integer.MIN_VALUE + 1 days in the past
  Calendar underMaxDays = sameDay;
  underMaxDays.add(Calendar.DAY_OF_MONTH, Integer.MIN_VALUE + 1);
  assertEquals(0, Repeat.calculateRepeatNumber(underMaxDays, sampleAppt));

  // 8. Integer.MIN_VALUE days in the past
  underMaxDays.add(Calendar.DAY_OF_MONTH, -1);
  assertEquals(0, Repeat.calculateRepeatNumber(underMaxDays, sampleAppt));

  // 9. Integer.MIN_VALUE - 1 days in the past
  underMaxDays.add(Calendar.DAY_OF_MONTH, -1);
  assertEquals(0, Repeat.calculateRepeatNumber(underMaxDays, sampleAppt));

}
```

Special Value Tests

```java
/*
 * Special Value cases of Calculate Repeat Number
 */
@Test
public void testCalculateRepeatNumberSpecialValue() {

  /*
   * Special cases for Appointment
   * 1. No Start Date
   * 2. No Repetition Frequency
   */
  Calendar testCal = new GregorianCalendar(0,0,2,0,0);
  Appointment testAppt = new Appointment();

  // 1. No Start Date
  assertEquals(0, Repeat.calculateRepeatNumber(testCal, testAppt));

  // 2. No Repetition Frequency
  testAppt.setDate(testCal.getTime());
  testCal.add(Calendar.DAY_OF_MONTH, 1);
  assertEquals(0, Repeat.calculateRepeatNumber(testCal, testAppt));
}
```

Method #2

Boundary Value Tests

```java
/**
 * Setup class to populate the store with a key
 */
@BeforeClass
```

```java
public static void setup() throws Exception {
  // Create a new store
  EncryptionHelper.createStore(EncryptionHelperTest.STORE_NAME,
      EncryptionHelperTest.STORE_PASS);

  // Add a key, manually
  EncryptionHelper.importKey(EncryptionHelperTest.STORE_NAME, EncryptionHelperTest.KEY,
      EncryptionHelperTest.KEYNAME, EncryptionHelperTest.STORE_PASS);

  // Set up the encryption helper
  InvalidKeystoreWorstCaseBVTest.eh = new
      EncryptionHelper(EncryptionHelperTest.STORE_NAME,
      EncryptionHelperTest.STORE_PASS);

}

/**
 * Method which generates a long string for use in tests
 * @param  len Maximum length of the string
 * @return     String of length `len`
 */
public static String generateLongString(long len) {
  final StringBuilder sb = new StringBuilder();

  for (long i = 0; i < len; i++) {
    sb.append("A");
  }

  return sb.toString();
}

/**
 * Test an invalid key against an empty string
 */
@Test(expected = InvalidKeyException.class)
public void testInvalidEmptyString() throws Exception {
  // Test smallest possible value, and ensure the encrypted form is
  // different than the plain form
  final String enc = InvalidKeystoreWorstCaseBVTest.eh.encrypt(this.EMPTY_STRING,
      "Test");
  Assert.assertNotEquals(enc, this.EMPTY_STRING);
  Assert.assertEquals(this.EMPTY_STRING,
      InvalidKeystoreWorstCaseBVTest.eh.decrypt(enc, EncryptionHelperTest.KEYNAME));
}

/**
 * Test an invalid key against a single-char string
 */
@Test(expected = InvalidKeyException.class)
public void testValidSingleChar() throws Exception {
  // Test a single character encryption
  final String enc = InvalidKeystoreWorstCaseBVTest.eh.encrypt(this.ALMOST_EMPTY_STRING,
      "Test");
  Assert.assertNotEquals(enc, this.ALMOST_EMPTY_STRING);
```

```java
  Assert.assertEquals(this.ALMOST_EMPTY_STRING,
      InvalidKeystoreWorstCaseBVTest.eh.decrypt(enc, EncryptionHelperTest.KEYNAME));
}

/**
 * Test an invalid key against a typical string
 */
@Test(expected = InvalidKeyException.class)
public void testValidNominal() throws Exception {
  // Test medium length encryption
  final String enc = InvalidKeystoreWorstCaseBVTest.eh.encrypt(this.NOMINAL_STRING,
      "Test");
  Assert.assertNotEquals(enc, this.NOMINAL_STRING);
  Assert.assertEquals(this.NOMINAL_STRING,
      InvalidKeystoreWorstCaseBVTest.eh.decrypt(enc, EncryptionHelperTest.KEYNAME));
}

/**
 * Test an invalid key against a very large string
 */
@Test(expected = InvalidKeyException.class)
public void testValidLarge() throws Exception {
  // Test large length encryption
  final String longString = InvalidKeystoreWorstCaseBVTest.generateLongString(4096);
  final String enc = InvalidKeystoreWorstCaseBVTest.eh.encrypt(longString, "Test");
  Assert.assertNotEquals(longString, enc);
  Assert.assertEquals(longString,
      InvalidKeystoreWorstCaseBVTest.eh.decrypt(enc, EncryptionHelperTest.KEYNAME));
}

/**
 * Test a invalid key using a binary string
 */
@Test(expected = InvalidKeyException.class)
public void testValidRawBytes() throws Exception {
  // Test some bytes that are less than 0x80 (String overflows there)
  final byte[] inp = { (byte) 0x00, (byte) 0x0c, (byte) 0x75, (byte) 0x00, (byte) 0x13 };
  final String s = new String(inp);
  final String enc = ValidKeystoreWorstCaseBVTest.eh.encrypt(s, "Test");
  final String dec = ValidKeystoreWorstCaseBVTest.eh.decrypt(enc,
      EncryptionHelperTest.KEYNAME);
  final byte[] out = dec.getBytes();

  boolean bytesMatch = true;
  for (int i = 0; i < inp.length; i++) {
    bytesMatch = bytesMatch & (inp[i] == out[i]);
  }

  Assert.assertTrue(bytesMatch);

}
/**
 * Test a valid key using an empty string
 */
```

```java
@Test
public void testValidEmptyString() throws Exception {
  // Test smallest possible value, and ensure the encrypted
  // form is different than the plain form
  final String enc = ValidKeystoreWorstCaseBVTest.eh.encrypt(this.EMPTY_STRING,
      EncryptionHelperTest.KEYNAME);
  Assert.assertNotEquals(enc, this.EMPTY_STRING);
  Assert.assertEquals(this.EMPTY_STRING,
      ValidKeystoreWorstCaseBVTest.eh.decrypt(enc, EncryptionHelperTest.KEYNAME));
}

/**
 * Test a valid key using a very small string
 */
@Test
public void testValidSingleChar() throws Exception {
  // Test a single character encryption
  final String enc = ValidKeystoreWorstCaseBVTest.eh.encrypt(this.ALMOST_EMPTY_STRING,
      EncryptionHelperTest.KEYNAME);
  Assert.assertNotEquals(enc, this.ALMOST_EMPTY_STRING);
  Assert.assertEquals(this.ALMOST_EMPTY_STRING,
      ValidKeystoreWorstCaseBVTest.eh.decrypt(enc, EncryptionHelperTest.KEYNAME));
}

/**
 * Test a valid key using a standard-length string
 */
@Test
public void testValidNominal() throws Exception {
  // Test medium length encryption
  final String enc = ValidKeystoreWorstCaseBVTest.eh.encrypt(this.NOMINAL_STRING,
      EncryptionHelperTest.KEYNAME);
  Assert.assertNotEquals(enc, this.NOMINAL_STRING);
  Assert.assertEquals(this.NOMINAL_STRING,
      ValidKeystoreWorstCaseBVTest.eh.decrypt(enc, EncryptionHelperTest.KEYNAME));
}

/**
 * Test a valid key against a large 4KB string
 */
@Test
public void testValidLarge() throws Exception {
  // Test large length encryption
  final String longString = ValidKeystoreWorstCaseBVTest.generateLongString(4096);
  final String enc = ValidKeystoreWorstCaseBVTest.eh.encrypt(longString,
      EncryptionHelperTest.KEYNAME);
  Assert.assertNotEquals(longString, enc);
  Assert.assertEquals(longString,
      ValidKeystoreWorstCaseBVTest.eh.decrypt(enc, EncryptionHelperTest.KEYNAME));
}

/**
 * Test a valid key using a binary string
 */
```

```java
@Test
public void testValidRawBytes() throws Exception {
  // Test some bytes that are less than 0x80 (String overflows there)
  final byte[] inp = { (byte) 0x00, (byte) 0x0c, (byte) 0x75, (byte) 0x00, (byte) 0x13 };
  final String s = new String(inp);
  final String enc = ValidKeystoreWorstCaseBVTest.eh.encrypt(s,
      EncryptionHelperTest.KEYNAME);
  final String dec = ValidKeystoreWorstCaseBVTest.eh.decrypt(enc,
      EncryptionHelperTest.KEYNAME);
  final byte[] out = dec.getBytes();

  boolean bytesMatch = true;
  for (int i = 0; i < inp.length; i++) {
    bytesMatch = bytesMatch & (inp[i] == out[i]);
  }

  Assert.assertTrue(bytesMatch);

}
```

Method #3

Using the outlined testing strategies above we have developed the following test cases for the method under test:

Boundary Value Tests

```java
/**
 * Test Case to represent the Boundary Value Tests.
 */
@Test
public void testGetNValueBVT() {

  /*
   * Constant: 1. The empty String. 2. A very long String. 3. A desirable
   * String. 4. Without a comma separator.
   */

  // 1. The empty String.
  assertEquals(Repeat.getNValue(""), 0);

  // 2. A very long String.
  assertEquals(Repeat.getNValue("THISORANGEJUICETASTESLIKECOFFEE"), 0);

  // 3. A desirable String.
  assertEquals(Repeat.getNValue("nweeks"), 0);

  // 4. Without a comma separator.
  assertEquals(Repeat.getNValue("nweeks 5"), 5);

  /*
   * Number: 1. The empty String. 2. A non-numeric String. 3. A very large
```

```java
   * negative int and float. 4. Zero. 5. A very large positive int and
   * float.
   */

  // 1. The empty String.
  assertEquals(Repeat.getNValue("nweeks, "), 0);

  // 2. A non-numeric String.
  assertEquals(Repeat.getNValue("nweeks,SEVEN"), 0);

  // 3. A very large negative int and float.
  assertEquals(Repeat.getNValue("nweeks," + Integer.MIN_VALUE), 0);
  assertEquals(Repeat.getNValue("nweeks," + Float.MIN_VALUE), 0);

  // 4. Zero.
  assertEquals(Repeat.getNValue("nweeks,0"), 0);

  // 5. A very large positive int and float.
  assertEquals(Repeat.getNValue("nweeks," + Integer.MAX_VALUE), 0);
  assertEquals(Repeat.getNValue("nweeks," + Float.MAX_VALUE), 0);
}
```

Equivalence Class Tests

```java
/**
 * Test case to represent the Equivalence Class Tests.
 */
@Test
public void testGetNValueECT() {
  /*
   * 1. Weak Normal Test Cases. 2. Strong Normal Test Cases. 3. Weak
   * Robust Test Cases. 4. Strong Robust Test Cases.
   */

  String number = "";
  String constant = "";

  /*
   * Weak Normal Test Cases A. NDAYS, Numeric String B. NWEEKS, Numeric
   * String C. NMONTHS, Numeric String D. NYEARS, Numeric String
   */
  // A. NDAYS, Numeric String
  number = "5";
  constant = Repeat.NDAYS;
  assertEquals(Repeat.getNValue(constant + "," + number), 5);

  // B. NWEEKS, Numeric String
  number = "5";
  constant = Repeat.NWEEKS;
  assertEquals(Repeat.getNValue(constant + "," + number), 5);

  // C. NMONTHS, Numeric String
  number = "5";
```

```
constant = Repeat.NMONTHS;
assertEquals(Repeat.getNValue(constant + "," + number), 5);

// D. NYEARS, Numeric String
number = "5";
constant = Repeat.NYEARS;
assertEquals(Repeat.getNValue(constant + "," + number), 5);

/*
 * 2. Strong Normal Test Cases :: Same as Weak Normal due to input range.
 */

/*
 * 3. Weak Robust Test Cases
 *
 * A. Empty String, Empty String.
 * B. Non Constant, Non Numeric.
 * C. NDAYS, Numeric String. (COVERED IN 1 and 2)
 * D. NDAYS, Empty String.
 * E. NWEEKS, Non Numeric.
 * F. NMONTHS, Numeric String. (COVERED IN 1 and 2)
 * G. NMONTHS, Empty String.
 * H. NYEARS, Non Numeric String.
 */

 // A. Empty String, Empty String.
  number = "5";
  constant = Repeat.NYEARS;
  assertEquals(Repeat.getNValue(constant + "," + number), 0);

 // B. Non Constant, Non Numeric.
  number = "notanumber";
  constant = "notconstant";
  assertEquals(Repeat.getNValue(constant + "," + number), 0);

 // D. NDAYS, Empty String.
  number = "";
  constant = Repeat.NDAYS;
  assertEquals(Repeat.getNValue(constant + "," + number), 0);

 // E. NWEEKS, Non Numeric.
  number = "notanumber";
  constant = Repeat.NWEEKS;
  assertEquals(Repeat.getNValue(constant + "," + number), 0);


 // G. NMONTHS, Empty String.
  number = "";
  constant = Repeat.NMONTHS;
  assertEquals(Repeat.getNValue(constant + "," + number), 0);

 // H. NYEARS, Non Numeric String.
  number = "notanumber";
  constant = Repeat.NYEARS;
```

```java
  assertEquals(Repeat.getNValue(constant + "," + number), 0);


/*
 * 4. Strong Robust Test Cases
 *
 * A. Constant Empty String, Empty String. (COVERED IN 3[A])
 * B. Constant Empty String, Non Numeric.
 * C. Constant Empty String, Numeric.
 * D. Non Constant, Number Empty String.
 * E. Non Constant, Non Numeric. (COVERED IN 3[B])
 * F. Non Constant, Numeric.
 * G. NDAYS, Empty String. (COVERED IN 3[D])
 * H. NDAYS, Non Numeric String.
 * I. NDAYS, Numeric String. (COVERED IN 1)
 * J. NWEEKS, Empty Number String.
 * K. NWEEKS, Non Numeric. (COVERED IN 3[E])
 * L. NWEEKS, Numeric. (COVERED IN 1[B])
 * M. NMONTHS, Empty Number String. (COVERED IN 3[G])
 * N. NMONTHS, Non Numeric String.
 * O. NMONTHS, Numeric String. (COVERED IN 1[C])
 * P. NYEARS, Empty Number String.
 * Q. NYEARS, Non Numeric String. (COVERED IN 3[H])
 * R. NYEARS, Numeric String (COVERED IN 1[D])
 */


// B. Constant Empty String, Non Numeric.
  number = "notanumber";
  constant = "";
  assertEquals(Repeat.getNValue(constant + "," + number), 0);

// C. Constant Empty String, Numeric.
  number = "5";
  constant = "";
  assertEquals(Repeat.getNValue(constant + "," + number), 0);

// D. Non Constant, Number Empty String.
  number = "";
  constant = "notaconstant";
  assertEquals(Repeat.getNValue(constant + "," + number), 0);

// F. Non Constant, Numeric.
  number = "5";
  constant = "notaconstant";
  assertEquals(Repeat.getNValue(constant + "," + number), 0);

// H. NDAYS, Non Numeric String.
  number = "notanumber";
  constant = Repeat.NDAYS;
  assertEquals(Repeat.getNValue(constant + "," + number), 0);

// J. NWEEKS, Empty Number String.
  number = "";
```

```
    constant = Repeat.NWEEKS;
    assertEquals(Repeat.getNValue(constant + "," + number), 0);

  // N. NMONTHS, Non Numeric String.
    number = "notanumber";
    constant = Repeat.NMONTHS;
    assertEquals(Repeat.getNValue(constant + "," + number), 0);

  // P. NYEARS, Empty Number String.
    number = "";
    constant = Repeat.NYEARS;
    assertEquals(Repeat.getNValue(constant + "," + number), 0);

}
```

Decision Table Tests

```
/**
 * Test case to represent the Decision Table Tests.
 */
@Test
public void testGetNValueDT() {
  fail("Not yet implemented");
  /*
   * Conditions:
   * C1. Input empty.
   * C2. Input does not contain a valid constant or valid number.
   * C3. Input contains a valid number but not a valid constant.
   * C4. Input contains a constant but no number of occurrences.
   * C5. Input contains a valid constant and valid number.
   *
   * Actions:
   * A1. Return 0
   * A2. The extracted number
   *
   * C1 -> A1.
   * C2 -> A1.
   * C3 -> A1.
   * C4 -> A1.
   * C5 -> A2.
   */

   // C1 -> A1.
    assertEquals(Repeat.getNValue(""), 0);

   // C2 -> A1.
    assertEquals(Repeat.getNValue("notavalidconstant" + "," + "notavalidnumber"), 0);

   // C3 -> A1.
    assertEquals(Repeat.getNValue("notavalidconstant" + "," + "5"), 0);

   // C4 -> A1.
    assertEquals(Repeat.getNValue(Repeat.NDAYS + "," + "notavalidnumber"), 0);
```

```
    assertEquals(Repeat.getNValue(Repeat.NDAYS + "," + "-1"), 0);
    assertEquals(Repeat.getNValue(Repeat.NDAYS + "," + "2.5"), 0);
    assertEquals(Repeat.getNValue(Repeat.NDAYS + "," + "0"), 0);

   // C5 -> A2.
    assertEquals(Repeat.getNValue(Repeat.NDAYS + "," + "5"), 5);
    assertEquals(Repeat.getNValue(Repeat.NWEEKS + "," + "5"), 5);
    assertEquals(Repeat.getNValue(Repeat.NMONTHS + "," + "5"), 5);
    assertEquals(Repeat.getNValue(Repeat.NYEARS + "," + "5"), 5);
}
```