EECS 4313: Assignment 3

Wednesday, 16 March 16

Team Members

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

1. Initial Coverage Metrics

Element	Coverage	Covered Inst.	Missed Inst.	Total Inst.
getNValue(String)	79.2 %	42	11	53
calculateRepeatNumber(Calendar, Appointment)	100.0 %	50	0	50
encrypt(String, String)	100.0 %	40	0	40

Table 1: Cover Metrics before White-box testing

2. Test Case Analysis

Method Test #3:

The method under test, `getNValue` exists within `Repeat.java` in the `net.sf.borg.model` package, and 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);
```

The coverage metrics before looking into the code (black-box tests) can be seen in the coverage portion, with coverage percent 79.2%. When expanding the code, 2 additional test cases were added. One to check a `null` input, and another to check for repeated input.

This test case checks for when a `null` string has been passed in. Upon inspection of the code and coverage metrics, it was determined that this branch was not tested for in the black-box testing. When looking into the code, it became clear that a null string should return a 0 value multiplier.

```
@Test
public void testGetNValueNull() {
  String repeat = null;
  assertEquals(Repeat.getNValue(repeat), 0);
}
```

However odd, this test was written after the inspection of the code and coverage metrics. Specifically when looking into the `Repeat` class, the branch checks when more than one comma separated, encoded strings. When this occurs, only the first encoded string and a multiplier are considered.

```
@Test
public void testGetNValueMultiple() {
  String f = Repeat.NDAYS + ",1," + Repeat.NDAYS;
  assertEquals(Repeat.getNValue(f), 1);
}
```

3. Final Coverage Metrics

Element	Coverage	Covered Inst.	Missed Inst.	Total Inst.
getNValue(String)	100.0 %	33	0	53
calculateRepeatNumber(Calendar, Appointment)	100.0 %	50	0	50
encrypt(String, String)	100.0 %	40	0	40

Table 2: Coverage Metrics after White-box testing

4. Control Flow Graph

The control flow graph below is for the `static public getNValue(String f);` method in the `net.sf.borg.model.Repeat` class. The legend provides which steps are executed at each step. For the full method specification, refer to method 3 in Appendix A.

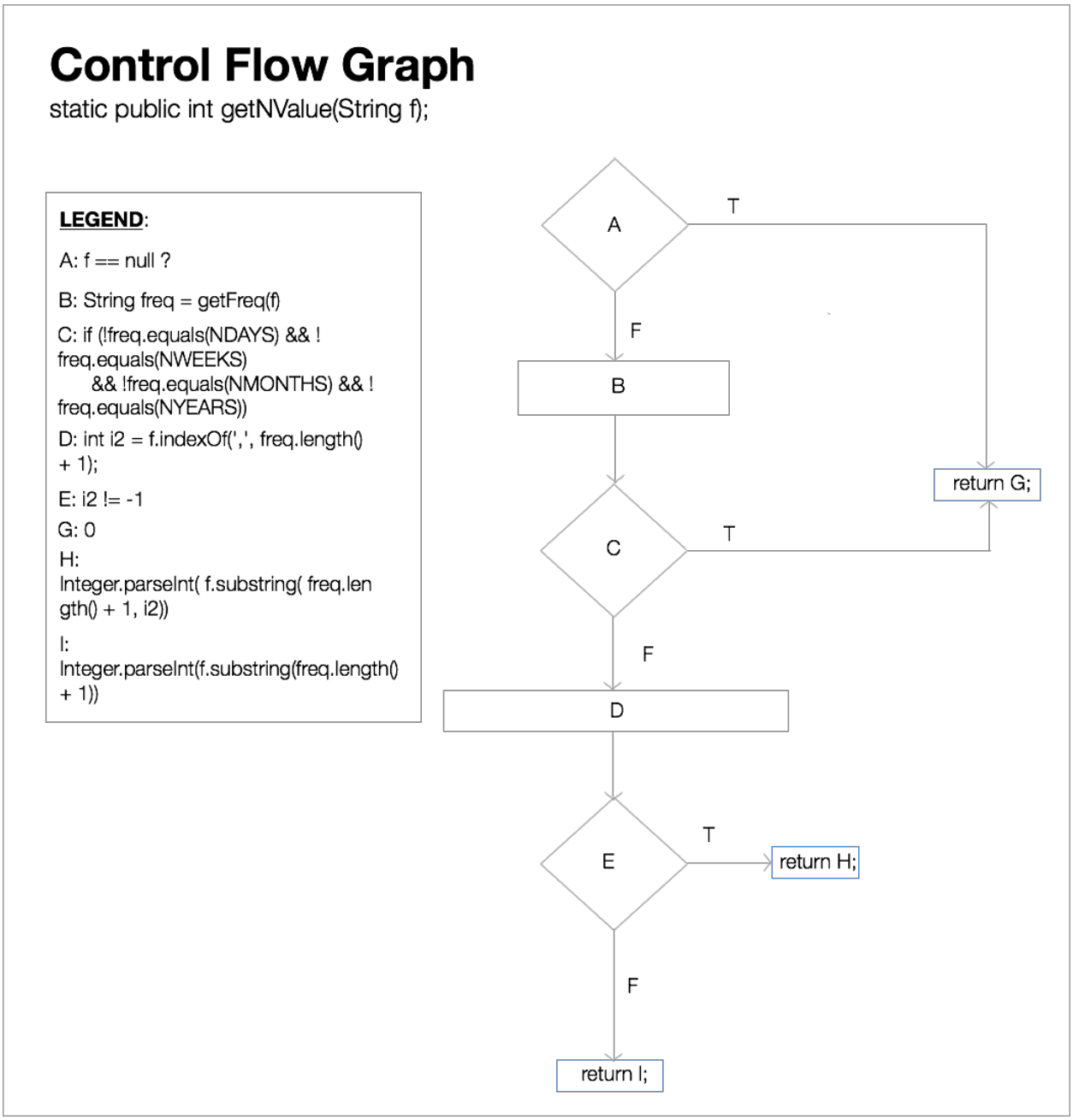

Figure 1: Control Flow Graph

5. Path Coverage Discussion

Paths

The execution paths through the control flow graph presented in section 4 are as follows:

- A(return G);
- ABC(return G);
- ABCDE(return H);
- ABCDE(return I);

Test coverage

As all of these test cases are based on branching, when running the coverage reports from EclEmma we are notified of these as potential test paths in the report. This is important because our test suite checks all possible branches in the `getNValue(Strin f)` method, 100% code coverage . Thereby checking all paths through the control flow graph. Test cases in our suite that do these checks are:

```
/**
 *  Test null input
 */
@Test
public void testGetNValueNull() {
    String repeat = null;
    assertEquals(Repeat.getNValue(repeat), 0);
}

// Test Constant as input
assertEquals(Repeat.getNValue("notavalidconstant" + "," + "5"), 0);

// Test contains only 1 constant and 1 multiplier
assertEquals(Repeat.getNValue(Repeat.NDAYS + "," + "5"), 5);

/**
 *  Test contains more than one constant and/or multiplier
 */
@Test
public void testGetNValueMultiple() {
    String f = Repeat.NDAYS + ",1," + Repeat.NDAYS;
    assertEquals(Repeat.getNValue(f), 1);
}
```

6. Appendix A

This appendix contains each of our method specifications from Assignment 2.

Method 1

```
/**
 * 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) {
  Calendar start = new GregorianCalendar();
  Calendar c = start;
  start.setTime(appt.getDate());
  Repeat r = new Repeat(start, appt.getFrequency());
  for (int i = 1;; i++) {
    if ((c.get(Calendar.YEAR) == current.get(Calendar.YEAR))
        && (c.get(Calendar.DAY_OF_YEAR) == current
            .get(Calendar.DAY_OF_YEAR)))
      return (i);
    if (c.after(current))
      return (0);
    c = r.next();
    if (c == null)
      return (0);
  }

}
```

Method 2

```
    /**
     * 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)
            throws Exception {

        /*
         * get the key and create the Cipher
         */
        Key key = this.keyStore.getKey(keyAlias, this.password.toCharArray());
```

```
        Cipher enc = Cipher.getInstance("AES");
        enc.init(Cipher.ENCRYPT_MODE, key);

        /*
         * encrypt the clear text
         */
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        OutputStream os = new CipherOutputStream(baos, enc);
        os.write(clearText.getBytes());
        os.close();

        /*
         * get the encrypted bytes and encode to a string
         */
        byte[] ba = baos.toByteArray();
        return new String(Base64Coder.encode(ba));

    }
```

Method 3

```
/**
 * 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) {
  if (f == null)
    return 0;

  String freq = Repeat.getFreq(f);

  if (!freq.equals(NDAYS) && !freq.equals(NWEEKS)
      && !freq.equals(NMONTHS) && !freq.equals(NYEARS))
    return (0);

  int i2 = f.indexOf(',', freq.length() + 1);
  if (i2 != -1)
    return (Integer.parseInt(f.substring(freq.length() + 1, i2)));

  return (Integer.parseInt(f.substring(freq.length() + 1)));

}
```