Assignment 2 - Internal Softare Quality

Heiko Joshua Jungen
Software Engineering
Chalmers University of Technology
Sweden, Gothenburg
Email: jungen@student.chalmers.se

David Fogelberg
Software Engineering
Chalmers University of Technology
Sweden, Gothenburg
Email: fodavid@student.chalmers.se

CONTENTS

***Abstract*—In this paper measured and human perceived complexity of source code is discussed. It outlines how complexity is perceived by humans and compares it to measured complexity. Further, ...**

I. INTRODUCTION

In this report five open source Java projects are compared by using their internal qualities. The five projects are: Amazequest, Umlet, Expr, BscSoftware and uHabits. SourceMonitor was used as an measurment tool for measuring the different internal quality factors. The different comparisons between the internal qualities will be discussed below.

II. PERCEIVED AND MEASURED COMPLEXITY

A. Ranking of perceived complexity

In this section five methods are compared and ranked by their perceived complexity, where value one is the highest rank. The perceived complexity for a method is determined by looking at cohesion, coupling, documentation and understandability of method names and variable names. Cohesion is seen as one of the most critical factors since a low cohesion leads to a method that does more than its main task. This makes it harder to understand what a method does since every task, including the main task, needs to be analyzed in order to get a clear view. Coupling is also seen as one of the most critical factors since a tightly coupled method is dependent on other methods. In order to understand what a tightly coupled method does, an analyze of the dependent methods are also needed. Meanwhile documentation and the understandability of method names and variable names are seen as less critical when determining the perceived complexity for a method. The reason for this is because even if a documentation is well written and all the names are clear in a method, the reader can still struggle to understand what is happening when for example there is low cohesion. Methods with low cohesion and tight coupling will therefore be perceived as more complex when doing a pairwise comparison. An example of this is the draw method that gets rank one while the getAngle method gets rank five because of the differences in coupling and cohesion, see table I. Even if the variables names are more vague in the getAngle method then the draw method, it is easier to understand what the method does and therefore the draw method gets an higher complexity.

Rank	Measured complexity	Perceived complexity
1	intersect	draw
2	getAngle	checkKeyword
3	checkKeyword	intersect
4	getParameters	getParameters
5	draw	getAngle

TABLE I: This table shows the ranking of measured and perceived complexity for the selected method. Measured complexity is shown in descending order of the measured complexity value in the second column. The order of perceived complexity is shown in the third column. In both columns rank 1 is the most and rank 5 the least complex.

B. Comparison to measured complexity

This section compares the perceived and measured complexity, and concludes about the quality of the latter one. First of all, the criteria for calculating the complexity are presented and compared to criteria for perceived complexity, which are discussed in section II-A. Finally, the quality of the measured complexity is elaborated.

Measured complexity is calculated by an algorithm according to the following criteria. At the beginning the complexity of a method is 1. This value increases by 1 for every conditional statement (i.e. if, else, for, while) and further increases by 1 for additional logical statements of the conditional statements (i.e '&&', 'll'). Hence, the measured complexity linearly increases with every conditional and logical statement.

Defining a reasonable threshold for the measured complexity value limits the amount of branches in a method. In order to not exceed the threshold developers would have to simplify code, or extract parts of the method in subroutines, which may comforts the ease to understand source code.

Human perceived complexity differs greatly from the before presented measured complexity. The algorithm for measuring complexity determines the number of execution paths and results with a value of measured complexity, hence it takes a single aspect into account to express complexity. Unlike this, human perceived complexity is the result of multiple aspects and additionally a subjective decision with no fixed value. It depends on the technical knowledge of the reader and may vary noticeable. When reading method source code, the reader needs to understand what is happening in order to perceive the method as not complex. Vague method or variable names decrease the ease of understanding, too many subroutines require more reading effort, and multiple actions within a method may overload the readers grasping capabilities. Additionally, readers with greater technical or domain knowledge may understand more difficult source code, whereas readers without said knowledge would not capable to understand.

In direct comparison of the measured and perceived complexity of the selected methods the differences become more clear. For instance according to table I, the draw-method is measured as not very complex, although it is perceived as the most complex. In this method the measured complexity is low, because the method contains just a single conditional statement (see line 16 in listing 1). However, various method calls, the usage of global variables and performing multiple actions subsequently the method appears complex to humans. Another interesting outcome is the comparison of the getAngle-method. Measuring the complexity of this method by counting the execution paths results in the second biggest measured complexity value. Anyhow, in the ranking of human perceived complexity it is considered the least complex method. The source code shows various conditional statements with multiple logical statements within (see lines 9, 12, 15 and 18 in listing 5), hence the measured complexity value increases often. But the method is focused on a single task and does not call many subroutines, which makes it easy to understand its purpose and what is happening.

In conclusion, perceived complexity is more complicated than measured complexity, because it is subjective and takes multiple aspects into account. In contrast to the perceived complexity, the presented measured complexity focuses solely on the total number of execution paths. For that reason, measured complexity is not sufficient enough to represent the actual human perceived complexity. Although, this may not hint that measured complexity is bad, but rather is not sufficient to represent human perception on its own. Thresholds for the complexity value will limit the number of conditional and related logical statements, hence will support the ease of understanding. That being said, additional measures should be taken into account to express the actual complexity, because human perception is not based on one aspect as well. Considering measurements for method coupling and cohesion would support a more accurate statement about actual complexity. Anyhow, aspects like understandable method and variable names as well as technical knowledge are hard to examine for an algorithm. Summarising, applying thresholds for measured complexity may decrease human perceived complexity, though it is most certainly not a guarantor, but rather one aspect of a more difficult subject.

Metric	Threshold
Maximum depth	7
Complexity	10
Method calls	20

TABLE II: This table shows the threshold for the selected metrics

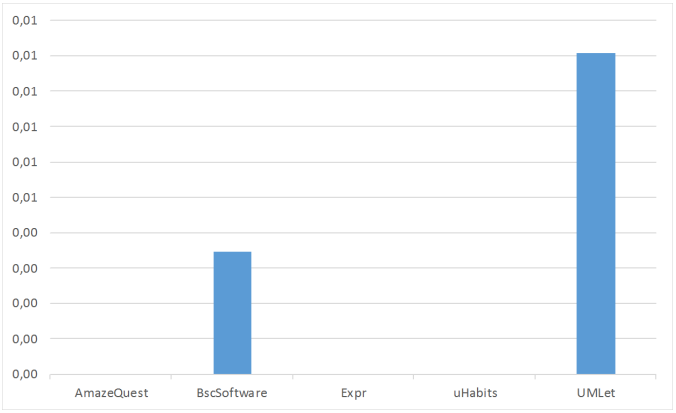

Fig. 1: Relative exceeded maximum depth.

III. Internal Quality

A. Measurements metrics and thresholds

This section introduces the chosen measurements and respective thresholds used to estimate the internal quality of the products. Nested depth, complexity and method calls are the chosen measurements to conclude about the internal quality. The first measurement, namely Nested depth, is the maximum counted number of nested conditional statements within a method. Complexity of a method is expressed by an integer value, that is the higher the more conditional and logical statements the method has. Thereby, the default value is 1 and it increases by 1 for every conditional or logical statements. The last measurement is the number of method calls within the examined method.

Estimation of internal software quality by using said measurements requires a definition of what is good and bad quality. For that reason, every measurement needs a threshold, that when exceeded stand for bad quality. The threshold values for the maximum depth, complexity and method calls are shown in table II and are based upon recommendation from literature or experience.

Project	Depth	Rel. depth	Complexity	Rel. complexity	Calls	Rel. Calls	Number of methods
AmazeQuest	0	0,00	0	0,00	3	0,03	90
BscSoftware	2	0,003	3	0,01	13	0,02	577
Expr	0	0,00	1	0,02	4	0,06	64
uHabits	0	0,00	4	0,00	62	0,03	1888
UMLet	31	0,01	18	0,01	156	0,05	3408

TABLE III: This table shows the absolute and relative number of methods that exceed the threshold for maximum depth, complexity, or methods calls. The relative value is calculating by the number of methods within the respective project.

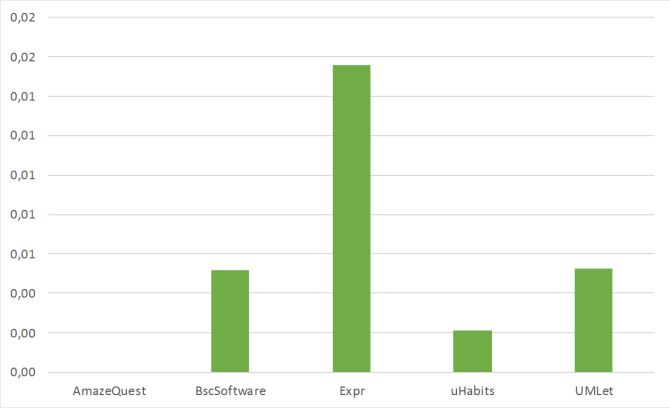

Fig. 2: Relative exceeded complexity.

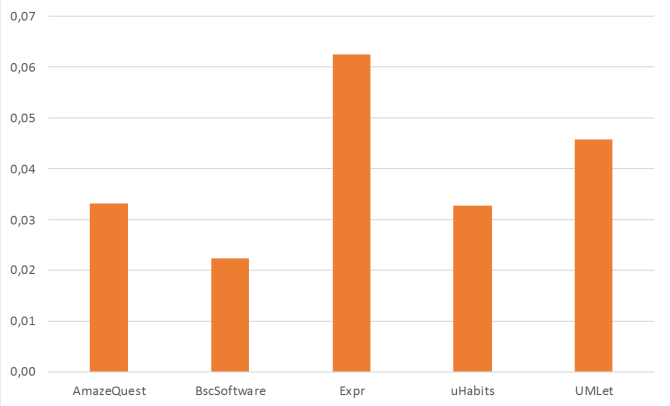

Fig. 3: Relative exceeded method calls.

B. Analysis of measurement metrics

C. Conclusion

IV. Refactoring

Methods that are exceeding defined thresholds of one ore more measurement should be examined carefully. A software engineer analysing a method of insufficient quality has to decide whether refactoring is required or not. In this section the readLinesToTestfile-method, shown in listing 6, is refactored in order to decrease the measured complexity. For this section the defined threshold for the measurement is 20 but the method has a complexity of 32, hence this value has to be decreased by refactoring.

Simply said, the aforementioned method is a for loop that iterates over a huge if-else-cascade, and finally return an object. Each of the if- and else-if-blocks are performing several actions on the object that is return at the end. However, some of these blocks additionally nest conditional statements which are making the method less readable, thus less understandable. A straightforward approach to refactor is to extract the nested conditional statements to separate methods, hence the method's readability increases.

The refactored method is shown in listing 7. The nested statements were extracted in two additional methods, namely the processDescription-method and the processTestlines-method, which increased the readability and understandability. As a result, the complexity value decreased from 32 to 13 and the added methods have a reasonable complexity value of 5 (processDescription) and 9 (processTestlines).

Investigating the algorithm for calculating complexity the presented refactoring results are not surprising. The complexity value of a method increases for every conditional and logical statement, thus extracting these statements into subroutines results in a lower computed value. However, developers could cheat on the complexity value by considering the algorithm. Carrying this to extremes, extracting the whole code of the method to a subroutine would certainly result in a complexity value of 1, which would indicate good quality, but in practice the complexity has solely shifted into another method. For that reason, methods added during refactoring have to examined carefully to make sure that the complexity actually decreased and not merely shifted from one method to another.

V. Acknowledgement

VI. Conclusion

Appendix

Listing 1: The listing shows the source code of the draw-method. Perceived complexity is high, because the method relies on several other methods and global variables. Further, low cohesion and lack of comments additionally increase the human perceived complexity level. The measured complexity for this method is 2.

```
public void draw(DrawHandler drawHandler,
    DrawingInfo drawingInfo) {
  double width = drawingInfo.
    getSymmetricWidth(getFirstLifeline()
    , getLastLifeline(), tick);
  double height = TextSplitter.
    getSplitStringHeight(textLines,
    width - ROUND_PART_WIDTH * 2,
    drawHandler) +
    VERTICAL_BORDER_PADDING * 2;
  double topY = drawingInfo.
    getVerticalStart(tick);
```

```
    topY += (drawingInfo.getTickHeight(tick
        ) - height) / 2;
    double leftX = drawingInfo.
        getHDrawingInfo(getFirstLifeline()).
        getSymmetricHorizontalStart(tick);

    drawHandler.drawArc(leftX, topY,
        ROUND_PART_WIDTH * 2, height, 90,
        180, true);
    width = width - ROUND_PART_WIDTH * 2;
    drawHandler.drawArc(leftX + width, topY
        , ROUND_PART_WIDTH * 2, height, 270,
         180, true);
    drawHandler.drawLine(leftX +
        ROUND_PART_WIDTH, topY, leftX +
        width + ROUND_PART_WIDTH, topY);
    drawHandler.drawLine(leftX +
        ROUND_PART_WIDTH, topY + height,
        leftX + width + ROUND_PART_WIDTH,
        topY + height);
    TextSplitter.drawText(drawHandler,
        textLines, leftX + ROUND_PART_WIDTH,
         topY, width, height,
         AlignHorizontal.CENTER,
            AlignVertical.CENTER);

    for (Lifeline ll : coveredLifelines) {
      drawingInfo.getDrawingInfo(ll).
          addInterruptedArea(new Line1D(topY
          , topY + height));
    }
  }
```

Listing 2: That listing presents the checkKeyword-method source code. The absence of sufficient documentation, vague method and variable names, as well as low cohesion are making the perceived complexity relatively high. The measured complexity value is 6.

```
private boolean checkKeyword(String
    keyword) {
  String libName = null;
  if (keyword.contains(".")) {
    String[] split = keyword.split("\\.")
        ;
    if (split.length != 2) {
      return false;
    }
    libName = split[0];
    keyword = split[1];
  }

  if (libName != null && !
      checkLibraryName(libName)) {
    return false;
  }
  String REGEX_KEYWORD = "[\\w␣\\d␣\\s␣
      \\.␣ß␣äöü␣ÄÖÜ]{1,}";
  if (!Pattern.matches(REGEX_KEYWORD,
      keyword)) {
    return false;
  }
  return true;
}
```

Listing 3: This code listing contains the intersect-method. It is loosely coupled and the variable names are easy to understand. However, the perceived complexity is increased by low cohesion, because the method performs two actions at the same time (calculating a minimum and maximum value). The measured complexity for this method is 11.

```
/**
 * returns the intersection of both
    points [eg: (2,5) intersect (1,4) =
    (2,4)]
 * @param nanPriority if true then NaN
    has priority over other values,
    otherwise other values have priority
 */
public XValues intersect(XValues other,
    boolean nanPriority) {
  Double maxLeft = left;
  Double minRight = right;
  if (nanPriority) {
    if (other.left.equals(Double.NaN) ||
        other.left > left) {
      maxLeft = other.left;
    }
    if (other.right.equals(Double.NaN) ||
         other.right < right) {
      minRight = other.right;
    }
  }
  else {
    if (left.equals(Double.NaN) || other.
        left > left) {
      maxLeft = other.left;
    }
    if (right.equals(Double.NaN) || other
        .right < right) {
      minRight = other.right;
    }
  }
  return new XValues(maxLeft, minRight);
}
```

Listing 4: Here the source code of the getParameters-Method is shown. The method lacks understandable variable names and proper documentation, but due to high cohesion and loose coupling the perceived complexity is relatively low. The measured complexity for this method is 4.

```
/**
```

```
  * Splits up comma-seperated parameters
     into single parameters.
  *
  * @param parameterLine
  * @return
  * @throws TestfileException
  */
 private Object[] getParameters(String
     parameterLine) throws
     TestfileException {
   if (parameterLine.length() == 0) {
     return new Object[0];
   }
   String[] parStrArray = parameterLine.
      split(",");
   Object[] res = new Object[parStrArray.
      length];
   for (int i = 0; i < parStrArray.length;
       i++) {
     String strPar = parStrArray[i].
        toString().trim();

     Object value = null;
     try {
       value = interpretValue(strPar);
     } catch (KeywordException |
        AssertionError e) {
       throw TestfileExceptionHandler.
          InvalidParameter(strPar);
     }
     res[i] = value;
   }
   return res;
 }
```

Listing 5: This listing contains the source code of the getAngle-method. Although it lacks sufficient documentation and meaningful variable names, the high cohesion and low coupling keeps the perceived complexity very low. The measured complexity value is 9.

```
/**
 * Calculates and returns the angle of
    the line defined by the coordinates
 */
public static double getAngle(double x1,
    double y1, double x2, double y2) {
  double res;
  double x = x2 - x1;
  double y = y2 - y1;
  res = Math.atan(y / x);
  if (x >= 0.0 && y >= 0.0) {
    res += 0.0;
  }
  else if (x < 0.0 && y >= 0.0) {
    res += Math.PI;
  }
  else if (x < 0.0 && y < 0.0) {
    res += Math.PI;
  }
  else if (x >= 0.0 && y < 0.0) {
    res += 2.0 * Math.PI;
  }
  return res;
}
```

Listing 6: Not refactored

```
/**
 * Create a {@link Testfile}-Object from
    the specified lines
 *
 * @param lines
 * @return
 * @throws TestfileException
 */
private Testfile readLinesToTestfile(
    String[] lines) throws
    TestfileException {
  Testfile testfile = new Testfile();
  for (int i = 0; i < lines.length; i++)
     {
    String line = lines[i].trim();

    if (line.length() == 0 || line.
       startsWith(Testfile.TAG_COMMENT))
       { // comment
      // ignore
    } else if (line.startsWith(Testfile.
      TAG_AUTHOR)) { // author
      String author = getLineContent(
         Testfile.TAG_AUTHOR, line);
      testfile.setAuthor(author);
    } else if (line.startsWith(Testfile.
      TAG_TESTNAME)) { // testname
      String testname = getLineContent(
         Testfile.TAG_TESTNAME, line);
      testfile.setTestname(testname);
    } else if (line.startsWith(Testfile.
      TAG_DESCRIPTION)) { // description
      String description = getLineContent
         (Testfile.TAG_DESCRIPTION, line)
         ;
      for (; i < lines.length - 1; i++) {
        line = lines[i + 1].trim();
        if (!line.startsWith(Testfile.
           TAG_FIRST_CHAR)) {
          if (line.length() > 0) {
            description += "␣" + line;
          }
        } else {
          break;
        }
      }
```

```
            testfile.setDescription(description
                );
        } else if (line.startsWith(Testfile.
            TAG_LIBRARY_FILE)) { // library
            String libPath = getLineContent(
                Testfile.TAG_LIBRARY_FILE, line)
                ;
            testfile.addLibraryFilePath(libPath
                );
        } else if (line.startsWith(Testfile.
            TAG_VARIABLE_FILE)) {
            String varPath = getLineContent(
                Testfile.TAG_VARIABLE_FILE, line
                );
            testfile.addVariableFilePath(
                varPath);
        } else if (line.startsWith(Testfile.
            TAG_REPEAT)) { // repeat
            String repeat = getLineContent(
                Testfile.TAG_REPEAT, line);
            testfile.setRepeat(repeat);
        } else if (line.startsWith(Testfile.
            TAG_SETUP)) { // setup
            for (; i < lines.length - 1; i++) {
                line = lines[i + 1].trim();
                if (line.length() == 0 || line.
                    startsWith(Testfile.
                    TAG_COMMENT)) {
                    continue;
                } else if (line.startsWith(
                    Testfile.TAG_FIRST_CHAR)) {
                    break;
                } else {
                    testfile.addSetupLine(line, i +
                        2);
                }
            }
        } else if (line.startsWith(Testfile.
            TAG_TEST)) { // test
            for (; i < lines.length - 1; i++) {
                line = lines[i + 1].trim();
                if (line.length() == 0 || line.
                    startsWith(Testfile.
                    TAG_COMMENT)) {
                    continue;
                } else if (line.startsWith(
                    Testfile.TAG_FIRST_CHAR)) {
                    break;
                } else {
                    testfile.addTestLine(line, i +
                        2);
                }
            }
        } else if (line.startsWith(Testfile.
            TAG_TEARDOWN)) { // teardown
            for (; i < lines.length - 1; i++) {
                line = lines[i + 1].trim();
                if (line.length() == 0 || line.
                    startsWith(Testfile.
                    TAG_COMMENT)) {
                    continue;
                } else if (line.startsWith(
                    Testfile.TAG_FIRST_CHAR)) {
                    break;
                } else {
                    testfile.addTeardownLine(line,
                        i + 2);
                }
            }
        }
    }

    checkTestfileContent(testfile);
    return testfile;
}
```

Listing 7: Refactored

```
/**
 * Create a {@link Testfile}-Object from
    the specified lines
 *
 * @param lines
 * @return
 * @throws TestfileException
 */
private Testfile readLinesToTestfile(
    String[] lines) throws
    TestfileException {
    Testfile testfile = new Testfile();
    for (int i = 0; i < lines.length; i++)
        {
        String line = lines[i].trim();
        if (line.length() == 0 || line.
            startsWith(Testfile.TAG_COMMENT))
            {
            // comment ignore
        } else if (line.startsWith(Testfile.
            TAG_AUTHOR)) { // author
            String author = getLineContent(
                Testfile.TAG_AUTHOR, line);
            testfile.setAuthor(author);
        } else if (line.startsWith(Testfile.
            TAG_TESTNAME)) { // testname
            String testname = getLineContent(
                Testfile.TAG_TESTNAME, line);
            testfile.setTestname(testname);
        } else if (line.startsWith(Testfile.
            TAG_DESCRIPTION)) { // description
            processDescription(testfile, lines,
                i);
        } else if (line.startsWith(Testfile.
```

```
        TAG_LIBRARY_FILE)) { // library
      String libPath = getLineContent(
          Testfile.TAG_LIBRARY_FILE, line)
          ;
      testfile.addLibraryFilePath(libPath
          );
    } else if (line.startsWith(Testfile.
        TAG_VARIABLE_FILE)) {
      String varPath = getLineContent(
          Testfile.TAG_VARIABLE_FILE, line
          );
      testfile.addVariableFilePath(
          varPath);
    } else if (line.startsWith(Testfile.
        TAG_REPEAT)) { // repeat
      String repeat = getLineContent(
          Testfile.TAG_REPEAT, line);
      testfile.setRepeat(repeat);
    } else if (line.startsWith(Testfile.
        TAG_SETUP)) { // setup
      processTestlines(testfile, Testfile
          .TAG_SETUP, lines, i);
    } else if (line.startsWith(Testfile.
        TAG_TEST)) { // test
      processTestlines(testfile, Testfile
          .TAG_TEST, lines, i);
    } else if (line.startsWith(Testfile.
        TAG_TEARDOWN)) { // teardown
      processTestlines(testfile, Testfile
          .TAG_TEARDOWN, lines, i);
    }
  }
  checkTestfileContent(testfile);
  return testfile;
}

private void processDescription(Testfile
    testfile, String[] lines, int
    currentIndex){
  String description = getLineContent(
      Testfile.TAG_DESCRIPTION, lines[
      currentIndex]);
  for (; i < lines.length - 1; i++) {
    line = lines[i + 1].trim();
    if (!line.startsWith(Testfile.
        TAG_FIRST_CHAR)) {
      if (line.length() > 0) {
        description += "␣" + line;
      }
    } else {
      break;
    }
  }
  testfile.setDescription(description);
}
```

```
private void processTestlines(Testfile
    testfile, String tag, String[] lines,
    int currentIndex){
  for (; currentIndex < lines.length - 1;
      currentIndex++) {
    line = lines[currentIndex + 1].trim()
        ;
    if (line.length() == 0 || line.
        startsWith(Testfile.TAG_COMMENT))
        {
      continue;
    } else if (line.startsWith(Testfile.
        TAG_FIRST_CHAR)) {
      break;
    } else {
      if (tag.equals(Testfile.TAG_SETUP))
          {
        testfile.addSetupLine(line,
            currentIndex + 2);
      } else if (tag.equals(Testfile.
          TAG_TEST)){
        testfile.addTestLine(line,
            currentIndex + 2);
      } else if (tag.equals(Testfile.
          TAG_TEARDOWN)){
        testfile.addTeardownLine(line,
            currentIndex + 2);
      }
    }
  }
}
```