Report for assignment 3

Group 1
Authors: Anders Blomqvist, Elliot Darth, Hannes Stig, Yening Liu, ZOU Hetai

Project

Name: Mockito

URL: https://site.mockito.org/
Our fork: https://github.com/DD2480-Group1/mockito
Original repo: https://github.com/mockito/mockito

People who will attempt P+:

- Elliot Darth
- Hannes Stig
- ZOU Hetai
- Yening Liu

Mockito is a mocking framework for Java, which can create substitutes (mockups) of real objects for testing purposes. This means you can create fake objects programmatically, instead having to do set up work for creating the objects you want to test with.

Onboarding experience

The project we choose is relatively easy to build and run. The dependencies are managed by gradle, which allows a mostly pain free build experience. One little obstacle was that the project README states the minimum Java version of 11, but in reality it requires Java 17. Updating the Java version resolves this issue quite easily.

Code coverage tool of Jacoco is also integrated in the gradle project. However, there was no obvious documentation of running the Jacoco and generating the code coverage report. We need to dig a bit deeper into the build script to obtain the relevant task for generating the code coverage report. Some of the team members also use JetBrain IntelliJ IDEA IDE to run the gradle tasks, which streamlined part of the user experiences.

Complexity

Automated tool: Lizard tool(https://github.com/terryyin/lizard)

1. What are your results for five complex functions?

- **(Anders) Result for: matches()**
  Lizard: 21
  JaCoCo: 21
  GitHub Copilot: 11 or 12
  Manual: 12

  Lizard claims its CCN is 21. From our understanding, it counts all if-statements as +2 and then +1 for the last else if no if-statements succeeded. There are 10 if-statements, which means 10 * 2 + 1 = 21.

  GitHub Copilot is not entirely sure. When selecting the function and asking, it simply says 12. Then when asked to explain why, it instead answers 11.

  By manual calculation, the CCN can be given by “McCabe’s complexity metric”: M = p - s + 2, where p = 21 (two decisions per if-statement) and s = 11 (11 returns). This gives us M = 21 - 11 + 2 = 12.

- **(Hetai) Result for: mockClass()**
  Lizard: 34
  GitHub Copilot: 6
  Manual: 35
  JaCoCo: 35

The results are different. 1. Lizard does not count return in the CCN. 2. Lizards count the ternary operator as 1 additional CCN, while manually we count 2. Jacoco uses a different calculation v(G) = B - D + 1, where B is the number of branches and D is the number of decision point.

GitHub copilot result is completely off when analyzing the CCN, possibly due to exceeding the maximum token number in a copilot chat session.

- **(Elliot) Result for: returnValueFor()**
  Lizard: 23
  GitHub Copilot: 24
  Manual: 2 (McCabe formula)
  JaCoCo: 23

The CNN results provided by Lizard and JaCoCo were the same. Using CoPilot, it counted very similarly to the other metrics, however it counted one extra if-statement somehow (an error).

When counting manually using the McCabe formula we get a surprisingly different result. We have 22 if-else statements, with one decision in each statement, which means p = 22. Furthermore, each if statement has an exit point, as well as an exit if no case in the if-else tree is met, which means s = 23. Plugging in these values in the formula, we get:

$$M = p - s + 2 \Rightarrow M = 22 - 23 + 2 = 1$$

This means that the CC counted manually is 1.

- **(Yening) Result for: wrap()**
  Lizard: 27
  GitHub Copilot:3
  Jacoao : 27
  Manual : 26

The CNN results provided by Github Copilot is much different from that of other kinds of method, which may because of there are functions inside the function, and Compilot only count possible paths. Meanwhile, Lizard don't count exit points therefore Manual counts less than it. There are 2 exit points and 26 deciding points in the function.

- **(Hannes) Result for equals()**
  Lizard : 14
  Jacoco: 14
  Manual 4

The cyclomatic complexity results from Jacoco and Lizard were the same, both reporting a complexity of 14. However the results from manually calculating the complexity gave a result of just 4. We believe this is because the automatic methods don't consider how multiple return statements impact the results.

2. Are the functions just complex, or also long?

(Anders) matches()
The function is short and simple. But complex for testing. It's 24 LOC

(Hetai) mockClass()
The function is both long and complex. It contains 141 LOC. It has to be long and complex by construction,since it is one of the method with more complicated logics.

(Elliot) returnValueFor()
The function is pretty short (48 LOC) and quite simple in the sense that each if-statement checks one single case. However, there are a total of 22 if statements which makes the testing repetitive (many single cases to cover). A pretty straight forward yet cumbersome function.

(Hannes) equals()
This method consists of multiple if statements. Some of them nested. The cyclomatic complexity could be reduced by restructuring these if statements. One part of the code that

adds unnecessary complexity is the nested if-statements that are used to guard against null values. These could be written a different way to reduce complexity, probably without reducing performance or code readability as well. Overall the function is not very long, so this is not really a problem even though the length could be reduced through refactoring as well.

(Yening) wrap()
The method is 234 LOC, composed of several sub-functions, which is also very long. Both complexity and length are very large. One part of it has too many decisions to be made.

3. What is the purpose of the functions?

- **Purpose of: matches()**
  The purpose of this function is to check whether your array matches a specific type of array. This is done by directly comparing what type the array it is and see if it matches the given array. Depending on what type of arrays mockito supports, this function will simply grow depending on the number of supported types. Thus, it's directly related to the high CCN.

- **Purpose of: mockClass()**
  mockClass() will creates the mockclass based on MockFeatures. It creates a mock class with the appropriate ClassLoader, give it a name for mock class, and give it the same interface. High CC is related to its function, since the generated Mock class could be different depending on the MockFeatures. The comments are appropriate in critical branches for better understanding the functions of the method.

- **Purpose of: returnValueFor()**
  The function returns an empty object depending on the library used as the data structure. The class type is sent in as a generic variable, which is then compared to multiple different types from different libraries.
  1. Firstly, if a primitive type is found, it just returns the default value for that type.
  2. If not primitive, it checks against multiple library-types and if matched returns an empty collection for said library-type (for instance, `HashSet` ⇒ `return new HashSet<>();` from HashSet library)
     a. If the library type is not found, it just returns a common empty value (Optional<T>).

- **Purpose of: wrap()**
  This Java code snippet is an implementation of the wrap method from the MethodVisitor interface. It checks if the method being visited is a constructor and not the Object class constructor. It then selects the shortest constructor with non-package-private access from the super class, and adds bytecode to the start of the method to handle constructor mocking. If a suitable constructor is found, a new MethodVisitor is returned with the added bytecode. Otherwise, the original MethodVisitor is returned.

- **Purpose of: equals()**

This method fills the same function as most java standard library equal methods. Its purpose is to check if two different SerializableMethod objects are equal to each other. It achieves this by multiple different if-statement checks if all parts of two different SerializableMethod objects are equal.

4. Are exceptions taken into account in the given measurements?

We are using Java together with the lizard v1.17.10. By conducting a little experiment with this Java code:

```java
public class TestExceptionsJava {
    public static void hasNOExceptions() {
        int a = 10 / 0;
    }
    public static void hasExceptions() {
        try {
            int a = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Error");
        }
    }
    public static void hasThrow() throws ArithmeticException {
        int a = 10 / 0;
        throw new ArithmeticException();
    }
}
```

We ran lizard on the code, and it gave us this output:

```
================================================
  NLOC CCN   token  PARAM  length  location
------------------------------------------------
       3      1      12     0      3 TestExceptionsJava::hasNOExceptions@3-5@.\TestExceptionsJava.java
       7      2      31     0      7 TestExceptionsJava::hasExceptions@7-13@.\TestExceptionsJava.java
       4      1      20     0      4 TestExceptionsJava::hasThrow@15-18@.\TestExceptionsJava.java
1 file analyzed.
```

We see that the function which has the try-catch clause, lizard claims its CCN is 2, while the other two are both 1. Therefore we can expect that exceptions are taken into account.

5. Is the documentation clear w.r.t. all the possible outcomes?

matches()
The documentation only states that if there is no match, then false will be returned. Otherwise true. However, it also states that the method should never assert if its argument does not match. This is useful to know when you are overriding the method.

returnValueFor()
The function itself is missing documentation. However, the function is a method in a class with the same name, and said class has documentation for the whole class including some

functionality contained in the function as well. All overarching outcomes are stated, not in detail, but enough to understand the premise of the function. For example, each library return type is not expanded upon, only summarized as “commonly used empty collections”.

equals()
There is no real documentation in form of comments. But the function is pretty self documenting and its easy to understand what is supposed to return for any given input based on the structure of the if statements. For example the if statement if(this == other) should pretty clearly return true since an object is always equal to itself.

Refactoring

Plan for refactoring complex code:

Estimated impact of refactoring (lower CC, but other drawbacks?).

- **(Anders) Refactor plan for matches()**
  This method needs to figure out what type two arrays are so that the correct equals method in the standard Java Arrays library can be called. The current code goes straight into comparing what type `wanted` and `actual` has but also if they are the same. If they are not the same, false will be returned. This means we can refactor by adding an if-statement at the beginning, making sure `wanted` and `actual` are of the same type, regardless of which. This reduces each if statement to only check which type one of either `wanted` or `actual` has. Thus reducing the CCN and the number of branches. The drawbacks are 1 additional if-statement, but that is minor compared to the benefits of 9 simplified if-statements.

- **(Yening P+) Refactor plan for wrap()**
  As we can see that there are two sub-functions inside wrap(), we can first split it by putting the sub-functions out of wrap(). However, one of the functions contributes most to the complexity. The complexity only reduces from 27 to 19 and it’s not enough. We have to check the reasons. We found that one part is composed of a for loop with some ‘if’ and ‘or’. We can split this part into a new function and the complexity is halved.

- **(Hetai optional P+) Refactor plan for mockClass()**
  The reason behind such high CCN is that the method are constructing the builder based on a variety of features and criterions. Therefore, the plan to refactor the code is just split the logic into several smaller independent private methods, and call the methods from mockClass(). The plan for carry out refactor is detailed in this issue. Each smaller private method will carry out a part of the builder logic and allow the overall logic to be easier to be understood by code reader while also reduce CCN.

- **(Elliot) Refactor plan for returnValueFor()**
  The returnValueFor gets a high CCN when looking at it through lizard and JaCoCo because it must compare the input variable to many different collections/data

structures. These branches all have unique exit points as well, which makes it difficult to condense any if-statements. One potential plan would be to separate out some of the if-statements into smaller methods, but this would clutter the code since there would be a lot of methods created (up to 22 if all branches were to be selected) that would only be used by this code.

- **(Elliot P+) Refactor plan for SerializableMethod equals()**
  Instead, another function with better refactoring potential was chosen. This function has many if-statements that can be condensed into single branches since many of the cases lead to the same result. This leaves a lot of potential for the CCN to be reduced. More details regarding the function can be found in this issue.

  For instance, checking if two parameters are not equal, as well as if one is null while the other is not can be condensed into a single comparison. That would reduce the "branch count" from three (with the current implementation) to one.

  The drawback is that initially the method looks less detailed, since not all cases will be as clear as before the refactoring. However, the logic will remain the same, all original cases should be able to pass in the refactored version.

- **(Hannes P+) Refactor plan for equals()**
  The method consists of multiple different if-else statements. This is where the complexity of the method comes from. By combining some of these if statements into one it is possible to reduce the complexity of the function.
  The function takes a generic "Object" parameter as input. The first part of the equals method checks if this object can actually be converted to a SerializableMethod object. The checks to do this can be combined into one.

  The next part of the method checks if all
  the fields of two different SerializableMethod objects are equal. This part of the code uses nested if statements to guard against potential null values. These nested if statements can be refactored using a helper function such that they don't need to be nestled which should reduce CC and branches.

.

Carried out refactoring (optional, P+):

- (Hannes) SerializableMethod equals refactoring
    - **Before:** Lizard and Jacoco report CC of 14
    - **After:** Lizard reports CC of 7 and Jacoco CC of 8
      Cyclomatic complexity reduced by ~50%
- (Elliot) SerializableMethod equals refactoring
    - **Before**: Lizard and JaCoCo report cyclomatic complexity of 14.
    - **After refactor**: CC reduced to around 6 (Lizard) to 7 (JaCoCo), improved by ~50%.
- (Hetai) MockClass refactoring
    - Lizard CCN reduce from 34 (before lizard output) to 20 (after lizard output)
    - Improved by 41%
- (Yening) wrap refactoring
    - Lizard CCN reduce from 27 to 10
    - Improved by 62%

Coverage

Tools: JaCoCo

Document your experience in using a "new"/different coverage tool.

The experience was straight forward since deploying the coverage tool was as simple as just running a command. This is because we used the IntelliJ IDE which streamlines the coverage tool process.

How well was the tool documented? Was it possible/easy/difficult to integrate it with your build environment?

JaCoCo is a free and open-sourced Java code coverage tool that came integrated with the open-source project we picked. It generates a detailed branch coverage report in the format of an html, with easy navigation to check for coverage result for different classes and methods. Specific line of code covered is also highlighted with colors. The coverage tool is well integrated with gradle, the build tool of the project.

Your own coverage tool

Anders:

- matches coverage
- matches coverage result

Hetai:

- mockClass coverage and result

Elliot:

- returnForValue coverage
- returnForValue result

Hannes:

- equals coverage

Yening:

- wrap result and coverage

What kinds of constructs does your tool support, and how accurate is its output?

The tool was tailored to each method where it was adapted to what constructs it had to support. The accuracy was sometimes exactly as JaCoCo but in some cases our tool was off by a few percent (max 5%). Our tool supports all the branching keywords (if, for, while, etc.), as well as all the logical operator (&&, ||, etc.). Tenery operator is also supported.

Evaluation

1. How detailed is your coverage measurement?

The measurement is not detailed at all due to it only prints: “branch X: true/false”. It gives no information about the branch which means it’s very hard for another person to understand which branches are covered.. You have to look into the source code and manually search for the branch IDs..

2. What are the limitations of your own tool?

The limitations are many, but the major one is that our “tool” is made within the original source code. This is so bad it should probably be illegal due to it modifying the original source code and making it *very* unreadable. The amount of additional work when a change is made is enormous because you have to rewrite the coverage “tool” as well. And the tool is not reusable for other files, so for each file you want to test coverage, you have to embed the tool into the original source code.

3. Are the results of your tool consistent with existing coverage tools?

It depends on which function DIY tool was used:

- For the **matches** function, the result is exactly the same as theirs (JaCoCo).

- For the **returnValueFor** method, the result of the DIY tool is exactly the same as the JaCoCo and lizard results.

- For the **equals** function, the result differs by ~7 percentage points. This could be attributed to Jacoco and the DIY tool counting coverage differently, for example I might have identified more branches by checking manually than the automatic tool did.

- For the **mockClass** function, the result differs by 3 percentage. This could be due to Jacoco count different coverage rate, i.e. partial coverage. But there may have been some mistake when implementing the DIY coverage tool.

- For the **wrap** function, the result is exactly the same as theirs (JaCoCo).

Coverage improvement

- **(Anders) Documentation of matches():**
  - **link**
  - **link**

- **(Elliot) Documentation :**
  - **returnValueFor() documentation.**
  - **SerializableMethod equals() documentation.**
  - **InstanceField equals() documentation.**

- **(Hetai) Documentation of mockClass()**
  - **link**

- **(Hetai) Documentation of ByteBuddyMockMaker**
  - **link**

- **(Hannes) Documentation for SerializableMethod.java equals()**
  - **link**

- **(Yening) Documentation for HashCodeAndEqualsMockWrapperTest.Java equals()**
  - **link**

Test cases added

- **(Anders) Test cases for matches():**
  - 2 new test cases
  - DIY result
  - DIY improved result
  - Old DIY coverage: 72.5 %, Old JaCoCo coverage: 72.5 %
  - New DIY coverage: 100 %, New JaCoCo coverage: 100 %

- **(Elliot) Test cases for returnValueFor():**
  - first new test case
  - Old DIY coverage: 95.6%, Old JaCoCo coverage: 97%
  - New DIY coverage: 100%, New JaCoCo coverage: 100%
  - *Now since I have reached 100% branch coverage with only one additional test, I will add my second mandatory test to a different function (seen below)*

- **(Elliot) Test cases for SerializableMethod equals():**
  - second new test case
  - P+ 7 additional test cases
  - Old JaCoCo coverage: 46%
  - New JaCoCo coverage: 50%
  - P+ JaCoCo coverage: 88%

- **(Elliot) Test cases for InstanceField equals():**
  - *Because I accidentally worked on an already taken function, I also opted to create some new unit test for a function with zero branch coverage:*
  - extra unit test cases
  - Old JaCoCo coverage: 0%
  - New JaCoCo coverage: 100%

  All Elliot's JaCoCo results can be viewed here as screenshots.

- **(Hannes) Test cases for equals():**
  - 11 Tests added for equals
  - Old DIY coverage: 39%, Old Jacoco coverage: 46%
  - New DIY coverage: 100%, New Jacoco coverage: 100%

- **(Hetai) Test case for mockClass():**
  - 1 test added
  - Old JaCoCo coverage: 63%
  - New JaCoCo coverage: 66%

- **(Hetai) Test cases for ByteBuddyMockMakerTest Class:**
  - 3 tests added
  - Old JaCoCo class coverage: 45%
  - New JaCoCo class coverage: 72%

- **(Yening) Test cases for HashCodeAndEqualsMockWrapperTest Class**
  - *Since the variables of wrap() are too complex, it's very difficult to write multiple individual test to improve coverage. Through discussion we choose another function and improve coverage to 100%.*
  - 4 tests added
  - Old JaCoCo class coverage: 0%
  - New JaCoCo class coverage: 100%(four branches, each test covers one)

Self-assessment: Way of working

Current state according to the Essence standard:

We are currently in the In Use state, specifically at the point of:

- *Procedures are in place to handle feedback on the team's way of working.*

This is the same point as from the first assignment (DECIDE), and even if looking ahead some points could be argued to be achieved, the proposed solution to handle feedback did not work out / was utilized. So in that sense, we are still stuck on this point.

Was the self-assessment unanimous? Any doubts about certain items?

The self-assessment was unanimous without doubt from any of the team members. Everyone was present in a discord voice call, and the way-of-working table was shared on screen and walked through together, examining each point linearly.

How have you improved so far?

We think the team has gotten more comfortable working with some of the tools needed for this project. In our previous CI-Server project we were all very new to setting up projects in java. Now we worked on a pre-existing project that already had extensive tooling setup around it, and we think our previous experience working with a smaller build system helped us with this assignment.

Where is potential for improvement?

The biggest potential for improvement is how we handle feedback within the group. This can be made through future code reviews but also through our communication channels.

Overall experience

What are your main take-aways from this project? What did you learn?

Writing your own branch coverage tool made you think about how seemingly simple code can suddenly become quite complex and be potential for bugs. Depending on the requirements of the software, a lot of branches can become costly in terms of resources and money. For example, if the software is in a safety critical system, then the code needs to be tested extremely thoroughly. That means (very simplified), having a low amount of branches will reduce cost and risk of failure.

This paper Mythical Unit Test Coverage from 2018 attempts to answer whether it's beneficial to push for higher code coverage or not, to eliminate future defects. Their conclusion was that this research question needs more work but there might not be worth aiming for higher coverage because there was no clear correlation between high coverage and low future defects.