Assignment 3: Code complexity, coverage

Annie Kihlert Alexander Widman Alva Sundström
Milad Sarbandi Farahani Tomas Weldetinsae

February 22, 2024

1 Onboarding

1.1 How easily can you build the project? Briefly describe if anything worked as documenter or not

1.1.1 Did you have to install a lot of additional tools to build the software?

Fortunately, the software uses Apache Maven as a build automation tool. Therefore, we did not have to do much in terms of building the software sinc we already had installed Maven on our computers for the previous assignment. Alternatively, Gradle, which is another common build tool, can be used to build the software.

1.1.2 Were those tools well documented?

The project itself is mostly well documented on the corresponding GitHub page. The README included general information about the project as well as details regarding tools such as Maven and JUnit which were used when developing the project. Every other aspect of the project looks to be documented in detail.

1.1.3 Were other components installed automatically by the build script?

Since Maven was used, the command *mvn clean install* automatically installed everything needed, such as dependencies, to be able to build and run the project.

1.1.4 Did the build conclude automatically without errors?

In the project, several modules are included. When building the project, Maven provides info about the results of attempting to build each module. During our intitial tries of building the project, our groupmembers acquired various results. We noticed that some of us were able to build the entire project, including all modules, flawlessly. However, some of us were only able to build some modules while failing on the rest. Since we had decided a specific module to work on

for this assignment (**karate-core**), it was only needed for all of us to be able to successfully build the specified module we had agreed upon. Fortunately, we were all able to do so.

1.1.5 How well do examples and tests run on you system(s)?

Maven has the ability to automatically run all tests included in the project. A status report is also provided which highlights the tests that have failed. All of us were able to run the tests in the specific module we had chosen for this assignment.

2 Part 1: Complexity measurement

To choose functions that were going to be examined for this section of the assignment, we intitially used *Lizard* as a tool for providing the cyclomatic complexity (CC). Since *Lizard* sorts the functions based on the CC, we decided to go through some of the functions with relatively high CC and choose five from that list. All functions chosen belonged to a file called **ScenarioEngine.java**. The five functions were:

- **httpInvokeOnce()** : used to invoke an HTTP request, handle the response and capture relevant information such as performance events for further processing.
- **recurseXmlEmbeddedExpressions()** : used to process and embed expressions within an object of type Variable(part of karate.core). It performs different operations based on the type of Variable received as parameter.
- **set()** : used to set values to variables nested within objects or arrays and accommodating XML and JSON data strcutres. It's main purpose is dynamic data manipulation.
- **match()** : used for performing matching operations between values received as parameters. It matches JSON, XML, JavaScript expressions.
- **evalKarateExpression()** : used to evaluate expressions from the karate framework syntax. These expressions are present in karate tests which can be JSON, XML, JavaScript expressions, function calls and more.

Three members of our group were assigned to calculate the cyclomatic complexity.

2.1 What are your results? Did everyone get the same result? Is there something that is unclear? If you have a tool, is its result the same as yours?

A discussion occurred before actually beginning the count regarding what method to use when calculating the cyclomatic complexity. We did some research and understood that there are multiple ways of calculating the CC by using different formulas. To keep it simple, we decided to use the formula provided in the lecture slides: $M = \pi - s + 2$ in which π is the amount of decisions such as *if,* ||, &&*, try, etc...* and where s is the number of exit points in the code. Following the calculations, we decided to compare our count to the count provided by *Lizard*. These are the results we found:

2.2 Our Count

- **httpInvokeOnce():** 17
- **recurseXmlEmbeddedExpressions():** 29 and 32
- **set():** 17
- **match():** 20
- **evalKarateExpression():** 13

2.3 Lizard's Count

- **httpInvokeOnce():** 17
- **recurseXmlEmbeddedExpressions():** 29
- **set():** 20
- **match():** 21
- **evalKarateExpression():** 22

We noticed that two groupmembers calculated the CC for the method **recurseXmlEmbeddedExpressions** to be 29, while another groupmember calculated it to be 32. We also noticed that the difference between our count and *Lizard's* count for the method **evalKarateExpression** differed significantly in comparison to the other methods.

2.4 Are the functions/methods with high CC also very long in terms of LOC?

There was no clear sign of any connection between high CC and LOC. The function we found to have the highest CC was **recurseXmlEmbeddedExpressions**, but this function came second to **httpInvokeOnce** in terms of

LOC. The function we found to have the lowest CC was **evalKarateExpression**, but this function only had 2 fewer LOC compared to **recurseXmlEmbeddedExpressions**. Therefore, we could not find any significant relationship between CC and LOC.

The order of the methods with the CCs in decreasing order:

- **recurseXmlEmbeddedExpressions()**
- **evalKarateExpression()**
- **match()**
- **set()**
- **httpInvokeOnce()**

The order of the methods with the highest LOC first:

- **httpInvokeOnce():** 89 LOC
- **recurseXmlEmbeddedExpressions():** 82 LOC
- **evalKarateExpression():** 80 LOC
- **set():** 64 LOC
- **match():** 55 LOC

2.5 What is the purpose of these functions? Is it related to the high CC?

We noticed that the methods with higher cyclomatic complexity contain multiple conditional statements such as *if*. Each if-statement adds a new path through the code which increases the cyclomatic complexity. There are also quite many *try-catch* blocks used for exception handling which also add more paths through the code, also increasing the CC. Some of the methods with higher CC also had more looping constructs. The loops, too, introduce new paths thorugh the code leading to higher CC.

In addition to the entry points mentioned above, some of the functions with a high CC also contain a lot of exit points, such as *return* statements. This creates a lot of potential paths through the code, which leads to the high CC.

2.6 If your programming language uses exceptions: Are they taken into account by the tool? If you think of an exception as another possible branch (to the catch block or the end of the function), how is the CC affected?

We used *Lizard* as a tool for calculating CC. In the GitHub page for *Lizard*, there is no documentation that states whether *Lizard* accounts for exceptions in Java

or any other language. However, since it is not stated in the limitations section, we assume that the tool does take exceptions into account. We, ourselves, account for exceptions when calculating by hand. Considering that we are using the formula $M = \pi - s + 2$, an exception would add to the π value, therefore increasing the overall CC of the function.

2.7 Is the documentation of the function clear about the different possible outcomes induced by different branches taken?

Some of the functions are documented well when analysing different paths that can be taken. For example, in the **set** function an *if-statement* is dependent on whether a JSON-file is being worked with, which is documented as *"assume json-path* in the code. However, some paths and outcomes are not directly documented.

3 Part 2. Coverage measurement & improvment

3.1 Task 1: DIY

3.1.1 What is the quality of your own coverage measurement? Does it take into account ternary operators (condition ? yes : no) and exceptions, if available in your language

The coverage measurment we have implemented is based on declaring a boolean array for the five functions with a size that corresponds to the number of branches each function has. Each index in the boolean arrays functions as a flag that is set to true if the branch was executed during a test execution. These flags are set for every test methods present in **ScenarioEngineTest.java** and after the execution of every test is finished, the number of branches executed in the functions is calculated and the result shown in the console. The coverage measurment we have implemented does take into account ternary operators incase they exist in the methods. For instance, in **recurseXmlEmbeddedExpressions()** there is a ternary operator that determines whether a for-loop is executed or not, which has a flag set up for it in the coverage measurement we have implemented.

3.1.2 What are the limitations of your tool? How would the instrumentation change if you modify the program?

One limitation with this approach of implementing coverage measurement is that the measurment are taken at the level of the function we have chosen. Coverage information for other functions that maybe called within the functions we have chosen is not going to be measured at all. If there are modifications

to the program then the coverage measurement maybe affected in several ways. For instance, if some code is removed or modified in a way that makes them unreachable then the coverage of those branches will decrease. Changes to the functions may also affected how they handle errors when they arise. It is also important to extend the coverage measurment to accomdoate for error handling.

3.1.3 If you have an automated tool, are your results consistent with the ones produced by existing tool(s)?

The results from the coverage measurement we have implemented align to a high degree to the ones produced by Jacoco.

3.2 Task 2: Coverage improvement

3.2.1 Identify the requirements that are tested or untested by the given test suite.

3.2.2 Document the requirements (as comments), and use them later as assertions

3.2.3 Create new test cases as needed to improve branch coverage in the given functions. Can you call the function directly? Can you expand on existing tests? Do you have to add additional interfaces to the system (as public methods) to make it possible to set up test data structures?

As the functions chosen for this lab turned out to have a quite high coverage already, new tests where therefore done on additional functions. Some of the functions with low coverege were private, and to be able to create tests for these, public help functions that called on the private functions had to be set up.

Coverage before and after additional tests

Function	Coverage before	Coverage after
getImageOptions()	0%	66%
replacePlaceholderText()	61%	81%
renderHtml()	0%	33%
DriverOptions.selector()	27%	66%
set()	84%	87%
config()	43%	54%

One method that was noticed to have the possibility for increased coverage was **replacePlaceholderText()**. This method was found to have 69 % coverage initially. The choice of writing tests for this method was made spontaneously without any specific consideration as to why. However, there were no tests for this method to be found. This could be because of the difficulties we faced when trying to navigate the code and the immense amount of files.

The **replacePlaceholderText()** method is used to replace placeholders in a

given text with a specified replacement string. The most obvious tests were written to assess this method. One test checks the behaviour of the method when the *text* parameter is null. The method should then return *null*. Another test assesses the behaviour when the *replaceWith* parameter is null. The expected behaviour is then that the method should return the original *text* without any changes. The third test written, analyses the case in which the *token* parameter is null. The method should then return the original *text* without any changes. The final test checks the behaviour of the method when all parameters are valid. The *text* contains the *token* and the *replaceWith* is a valid string. The expected behaviour is then that the method should replace the *token* in the *text* with the *replaceWith* string.

By including these tests, the coverage of the method increased to 81 %.

3.2.4 If you have 100 % branch coverage, you can choose other functions or think about path coverage. You may cover all branches in a function, but what does this mean for the combination of branches? Consider the existing tests by hand and check how they cover the branches (in which combinations).

3.3 Task 3: Refactoring plan

Is the high complexity you identified really necessary? Is it possible to split up the code (in the five complex functions you have identified) into smaller units to reduce complexity? If so, how would you go about this? Document your plan.

httpInvokeOnce()

This method is quite long with 80 LOC. When having a long method, it can get difficult to understand and maintain the code. Therefore, it can be broken into smaller, more manageable methods, each handling different parts. Parts of the **httpInvokeOnce()** method can be extracted into separate methods. For example, the code that sets up the HTTP request could be separated into a method called **setUpHttpRequest()**.

The method also currently has several levels of nested if-else statements and try-catch blocks. This can make the control flow more challenging to follow. Some of this logic can be extracted into separate methods to reduce duplication and improve maintainability. For example, the level of nesting could be reduced by using early returns or to break out of the method when certain conditions are met. In **httpInvokeOnce()**, a separate method could be used called **handleResponse()** which includes the code needed for response handling.

The **httpInvokeOnce()** method currently uses string literals such as *"binary", "json", "xml" and "string"* to represent response types. These could be replaced with constants to avoid potential typos and to make the code overall easier to

understand. For example, instead of writing *"binary"*, a constant called *RESPONSE_TYPE_BINARY* could be used.

evalKarateExpression()

For the **evalKarateExpression()** method, the same principles that were used for refactoring **httpInvokeOnce()** can be applied. Again the method is quite long with 80 LOC. The code which handles the case in which a text is a call syntax can be extracted into a seperate method called **handleCallSyntax()**.

There exists several levels of nested if-else statements in this method as well. One path that can be taken is the case where the text is a JSON path. This part can be separated into a method called **handleJsonPath()**. Also, string literals such as *"callBegin", "get" and "$"* are again used. These can be replaced with constants to make the code more apprehendable. An example would be replacing *"$"* with *DOLLAR*.

recurseXmlExpressions

The techniques mentioned for the previous methods can be applied to **recurseXmlExpressions** as well. Since this method also has several blocks of code that perform similar operations such as evaluating an embedded expression, these could potentially be extracted into separate methods to reduce duplication.

3.3.1 set() and match()

There would be no different approach to refactoring the remaining **set()** and **match()** methods. Every technique and approach mentioned previously can be applied to these methods as well. The **set()** function could for example be refactord through creating two smaller functions, one that handles variables with xmlPath and one that handles Json-paths. This would make the function easier to follow and it's purpouse would be more clear.

3.4 Task 4: Self-assesment

We are currently in the state of "Working Well" as we feel that we have had time to improve and adapt our way-of-working progressively in a way that works well for everyone in the team. We have been able to adapt and tune this way-of-working in accordance to the current assignment, just like before, and we feel that our way-of-working has now become natural for us. The next, and very last step, for us is to share lessons that we have learned from our way-of-working for future use. Since there is only one more assignment to go, that will happen together with taking the step of not using our way-of-working anymore. These are the last steps for us to reach the state of "Retired".

What we have learned most is the importance of teamwork and the ability to

adapt this way-of-working in a way that fits both the team and the task at hand. These abilities have been significantly improved with this course, and we feel that we have excelled in these areas in particular. Where we feel improvement could be possible is the ability to even more naturally apply these practices, and we feel that this would come with even more time with this way-of-working.