Assignment#3 - Group 16

Henrik Åkesson, Robin , Anton Sederlin, Robert Skoglund, Kelvin osv

February 2024

1 Onboarding

We chose the the following project for this assignment: javaparser. This java parser project is fully written in java, in accordance with the Apache Java Project, and contains $\sim 200k$ NLOC.

1.1 Documentation

As the project uses the Apache Maven Project to manage to code base its quite straight forward how to build, test and run the project. This process is well documented in the README. After cloning the project locally you can simple run: **./mvnw clean install**. mvnw is a bash script wrapping the maven project, which essentially mean that the correct version of maven will be installed and used for you if it don't already exist on your system. **./mvnw clean install** will compile, build and run tests automatically for you. It will also install all the dependencies and tools needed as specified and documented in the pom.xml files.

1.2 Experience

Even though the onboarding process is very simple, the project build result is extremely unreliable. The project is built successfully, almost every time the first time you run **./mvnw clean install**. However, every time you want to rebuild the project, problems and errors occur, even though nothing in the code base is changed. There is no pattern for when the project builds successfully or not across all five group members systems, the unsuccessfully builds appear to be arbitrary. Fixes that work sometimes include, deleting and re-cloning the repository, restarting VScode, or running multiple **./mvnw clean** followed by **./mvnw test**. The extreme extent of the unreliable builds made it very hard to compile and test our own contributions to the project.

2 Complexity Measurement

2.1 Metrics

Complexity measurement metrics was acquired automatically by running a complexity measurement tool named lizard on the code base. After successfully installing lizard lizard the flowing command was run in the root folder of the code base:

```
$ lizard -C 12
```

The above command returns a list with complexity metrics on all functions with $CCN > 12$.

Definitions of metrics in Table 1:

- **Functions:**
- **NLOC:** Non-commented Lines Of Code
- **manCCN:** The cyclomatic complexity count acquired from Lizard
- **manCC:** The cyclomatic complexity count acquired from a manual count $CCN = PredicateNodes) + 1$, where $PredicateNodes$ equals the total number of conditions (Decisions)
- **TOKENS:** Total number of written characters
- **PARAMS:** Total Number of input parameters
- **LENGTH:** Total number of lines of code

Table 1: Metrics Table

	METRICS					
Function	**NLOC**	**CCN**	**manCC**	**TOKENS**	**PARAMS**	**LENGTH**
cleanLines	30	14	14	342	1	33
calculatedSyntaxModelForNode	131	44	44	1296	4	141
isApplicable	86	27	a	729	4	125
compareConsideringTypeParameters	44	21	21	408	1	47
isAssignableMatchTypeParametersMatchingQName	51	19	19	451	3	65

2.2 Function 1: calculatedSyntaxModelForNode - Henrik Åkesson

2.2.1 Purpose

The purpose of the function, given that it's part of the **LexicalDifferenceCalculator** class, seems to be to create a syntax model in a way that it can be used for comparison, identifying any lexical differences between different versions of the parsed code.

2.2.2 Parameters

- **CsmElement csm** (Concrete Syntax Model): Defines the expected syntax of the code.
- **Node node**: The node is likely a node in the AST (Abstract Syntax Tree), representing the structure of the code where a node is some part of the syntax.
- **List<CsmElement> elements list**: List where the calculated syntax models are added.
- **Change object**: Keeps track of any changes to the node.

2.2.3 Documentation

The function has no documentation, except for a few comments. The comments are simple and don't give an explanation to the actual functionality. An example is the comment *// nothing to do*, where if the csm element is a comment node, the function does nothing. While this does in some way explain one of the possible outcomes induced by a branch, it's still one of the few outcomes with an explanation. Another comment is *// Same edge-case as in DefaultPrettyPrinterVisitor.visit(LambdaExpr, Void)*, which is not clear what it means in relation to the outcome unless the reader is already familiar with the mentioned edge-case. The function also contains a comment about a fix that was made; *// fix #2382: // This method calculates the syntax model..*, explaining an issue which was fixed in a later version.

2.2.4 CC & LOC

- The function NLOC is 131, a rather large number of non-comment lines of code for one function, correlating with a large CC.
- The high CC seems to be related to the purpose of the function in the way that it handles the many different syntax elements, leading to the numerous conditions and the high CCN.
- The function uses exceptions, which seems to be taken into account by the lizard tool when counting CC, as this matches the manual count where the exceptions were also counted to the total CCN. Therefore, taking into account exceptions did not affect the CC.

2.2.5 Manual CC count

```
1 Condition 1: if (csm instanceof CsmSequence)
2 Condition 2: else if (csm instanceof CsmComment)
3 Condition 3: else if (csm instanceof CsmSingleReference)
4 Condition 4 & 5: if (change instanceof PropertyChange && ((
    PropertyChange) change).getProperty() == csmSingleReference.
    getProperty())
```

```
Condition 6 & 7: if (node instanceof LambdaExpr && child instanceof
     ExpressionStmt)
Condition 8: (child != null)
Condition 9: else if (csm instanceof CsmNone)
Condition 10: else if (csm instanceof CsmToken)
Condition 11: else if (csm instanceof CsmOrphanCommentsEnding)
Condition 12: else if (csm instanceof CsmList)
Condition 13: if (csmList.getProperty().isAboutNodes())
Condition 14: if (rawValue instanceof Optional)
Condition 15: if (optional.isPresent())
Condition 16: if (!(optional.get() instanceof NodeList))
Condition 17: throw new IllegalStateException("Expected NodeList,
    found " + optional.get().getClass().getCanonicalName())
Condition 18: if (!(rawValue instanceof NodeList))
Condition 19: if (!nodeList.isEmpty())
Condition 20: if (i != 0)
Condition 21: if (i != (nodeList.size() - 1))
Condition 22: if (!collection.isEmpty())
Condition 23: if (!first)
Condition 24: if (value instanceof Modifier)
Condition 25: if (it.hasNext())
Condition 26: else if (csm instanceof CsmConditional)
Condition 27: if (satisfied)
Condition 28: else if (csm instanceof CsmIndent)
Condition 29: else if (csm instanceof CsmUnindent)
Condition 30: else if (csm instanceof CsmAttribute)
Condition 31: if (value instanceof Stringable)
Condition 32 & 33: else if ((csm instanceof CsmString) && (node
    instanceof StringLiteralExpr))
Condition 34: if (change instanceof PropertyChange)
Condition 35 & 36: else if ((csm instanceof CsmString) && (node
    instanceof TextBlockLiteralExpr))
Condition 37: if (change instanceof PropertyChange)
Condition 38 & 39: else if ((csm instanceof CsmChar) && (node
    instanceof CharLiteralExpr))
Condition 40: if (change instanceof PropertyChange)
Condition 41: else if (csm instanceof CsmMix)
Condition 42: else if (csm instanceof CsmChild)
Condition 43: throw new UnsupportedOperationException("Not
    supported element type: " + csm.getClass().getSimpleName() + "
    " + csm)

CCN = #Conditions (Decisions/predicate nodes) + 1 = 44
```

Listing 1: Manual CC count

2.3 cleanLines - Anton Sederlin

2.3.1 Purpose

The **cleanLines()** function is a part if the **JavadocParser** class which is responsible for parsing the content of Javadoc comments and producing Javadoc documents as specified in The Javadoc specification. The propose of the function itself is to identify, normalise , and remove leading and trailing whitspaces from code comments. It is also responsible for removing empty lines, enuring

that only well formatted comment strings are returned.

2.3.2 Documentation

The function is documented to some extent, the top-level (class) javadoc comment describes it purpose briefly. Furthermore, there is some documetnetion regarding the outcome of different branches:

- //lines containing only white-space are normalised to empty lines
- //if the first starts with a space, remove it
- //drop empty lines at the beginning and at the end.

2.3.3 CC & LOC

- The NLOC of the function is 44, this is not a huge number but to make it even more readable and less complex, it should be refactored in such way that none of the sub functions NLOCs exceed $NLOC = 20$ and $CC < 12$. This rather high value of NLOC corresponds to the rather high $CCN = 14$
- The porpuse of the function is related to the high CC, as there are mane different edge cases that needs to be checked and handled differently, hence alot of decisions needs to be made.
- This function does not throw any exceptions, so it cant be said of lizard handles try/catch blocks in this specific case.

2.3.4 Manual CC count

```
1  JavadocParser::cleanLines@91-123@./javaparser-core/src/main/java/
       com/github/javaparser/JavadocParser.java
2
3  Condition 1: if (lines.length == 0)
4  Condition 2: if (asteriskIndex == -1)
5  Condition 3: if (l.length() > (asteriskIndex + 1))
6  Condition 4 & 5: if (c == ' ' || c == '\t')
7  Condition 6: l.trim().isEmpty() ? "" : l
8  Condition 7 & 8 & 9: !cleanedLines.get(0).isEmpty() && (
       cleanedLines.get(0).charAt(0) == ' ' || cleanedLines.get(0).
       charAt(0) == '\t'))
9  Condition 10 & 11: while (cleanedLines.size() > 0 && cleanedLines.
       get(0).trim().isEmpty())
10 Condition 12 & 13: while (cleanedLines.size() > 0 && cleanedLines.
       get(cleanedLines.size() - 1).trim().isEmpty())
11
12 CCN = #Conditions (Decisions/predicate nodes) + 1 = 14
```

Listing 2: Manual CC count

2.4 isApplicable - Robert Skoglund

2.4.1 Purpose

The purpose of the isApplicable function is to determine whether a given method usage is applicable based on certain conditions involving method names, parameter counts, and parameter types. The function performs a series of checks and comparisons between the provided methodUsage and the expected criteria, and it returns a boolean value indicating whether the method is applicable

2.4.2 Documentation

The function is documented briefly, the comments tells us the it returns true: if the given MethodUsage matches the given name/types (normally obtained from a ResolvedMethodDeclaration). It also provides some information on the outcome of a taken branch.

- //If the counts do not match and the method is not variadic, this is not a match.
- // If the counts do not match and we have provided too few arguments, this is not a match. Note that variadic parameters allow you to omit the vararg, which would allow a difference of one, but a difference in count of 2 or more is not a match.
- // If the given argument still isn't applicable even after considering type arguments/generics, this is not a match.

2.4.3 CC & LOC

The function NLOC is 86. The CCN count using lizard is 27. The CCN of 27 in this function is mainly due to intricate nested conditions and loops, involving complex type comparisons type checking. The numerous decision points, especially within the nested loops and conditional statements, contribute to a high CCN, indicating potential code complexity and increased difficulty in understanding and maintaining the code.

2.4.4 Manual CC count

```
1 Condition 1: if (!methodUsage.getName().equals(needleName))
2 Condition 2 & 3: if (!methodIsDeclaredWithVariadicParameter && !(
      needleParameterCount == countOfMethodUsageArgumentsPassed))
3 Condition 4 & 5: if (!(needleParameterCount ==
      countOfMethodUsageArgumentsPassed) && needleParameterCount <
      lastMethodUsageArgumentIndex)
4 Condition 6: for (int i = 0; i < needleParameterCount; i++)
5 Condition 7: boolean reachedVariadicParam =
      methodIsDeclaredWithVariadicParameter && i >=
      lastMethodUsageArgumentIndex;
6 Condition 8: if (!reachedVariadicParam)
```

```
Condition 9 & 10: boolean argumentIsArray = (needleParameterCount
    == countOfMethodUsageArgumentsPassed) && expectedArgumentType.
    isAssignableBy(actualArgumentType)
Condition 11: if (!argumentIsArray)
Condition 12: for (int j = 0; j < countOfMethodUsageArgumentsPassed
    ; j++)
Condition 13: if (parameter.isVariadic())
Condition 14: if (needleParameterCount == j)
Condition 15: if (tp.getBounds().isEmpty())
Condition 16: else if (tp.getBounds().size() == 1)
Condition 17: if (bound.isExtends())
Condition 18: if (tp.getBounds().isEmpty())
Condition 19: else if (tp.getBounds().size() == 1)
Condition 20: if (bound.isExtends())
Condition 21 & 22 & 23 & 24: if (!expectedArgumentType.
    isAssignableBy(actualArgumentType)
&& !expectedTypeWithSubstitutions.isAssignableBy(actualArgumentType
    )
&& !expectedTypeWithInference.isAssignableBy(actualArgumentType)
&& !expectedTypeWithoutSubstitutions.isAssignableBy(
    actualArgumentType))


CCN = #Conditions (Decisions/predicate nodes) + 1 = 24
```

Listing 3: Manual CC count

2.5 compareConsideringTypeParameters: Robin Eggestig

2.5.1 Purpose

This function is part of the ResolvedReferenceType. The compareConsidering-TypeParameters function compares two instances of a ResolvedReferenceType class, accounting for type parameters, raw types, and wildcard types. It ensures nuanced equivalence checks, crucial for understanding the compatibility and similarity between instances with respect to their generic type information.

2.5.2 Documentation

The compareConsideringTypeParameters function is a protected method designed for comparing instances of a ResolvedReferenceType class, considering various scenarios involving type parameters. The method begins by checking object equality and qualified names before delving into more nuanced comparisons. It handles raw types, wildcard types, and type variables, ensuring that type parameters are appropriately matched. The method exhibits a comprehensive understanding of Java generics, providing a detailed and context-aware equivalence check for reference types. While the provided code lacks explicit Javadoc comments, a well-documented version would likely emphasize its role in type parameter comparison, guide users on proper usage, and highlight potential exceptions. Users should consult external documentation or codebase comments for a more detailed understanding of the function's behavior and intended usage.

2.5.3 CC & LOC

The function NLOC is 44. The CCN count using lizard is 21. The CCN of 21 in this function is mainly due to intricate nested conditions and loops, involving complex type comparisons and wildcard handling. The numerous decision points, especially within the nested loops and conditional statements, contribute to a high CCN, indicating potential code complexity and increased difficulty in understanding and maintaining the code.

2.5.4 Manual CC count

```
Condition 1: if (other.equals(this))
Condition 2: if (this.getQualifiedName().equals(other.
    getQualifiedName()))
Condition 3 & 4: if (this.isRawType() || other.isRawType())
Condition 5: if (typeParametersValues.size() != other.
    typeParametersValues().size())
Condition 6: Loop condition - for (int i = 0; i <
    typeParametersValues.size(); i++)
Condition 7: if (!thisParam.equals(otherParam))
Condition 8: if (thisParam instanceof ResolvedWildcard)
Condition 9 & 10: if (thisParamAsWildcard.isSuper() && otherParam.
    isAssignableBy(thisParamAsWildcard.getBoundedType()))
Condition 11 & 12: else if (thisParamAsWildcard.isExtends() &&
    thisParamAsWildcard.getBoundedType().isAssignableBy(otherParam)
    )
Condition 13: else if (!thisParamAsWildcard.isBounded())
Condition 14 & 15: if (thisParam instanceof ResolvedTypeVariable &&
     otherParam instanceof ResolvedTypeVariable)
Condition 16: return thisBounds.size() == otherBounds.size() &&
    otherBounds.containsAll(thisBounds);
Condition 17 & 18: if (!(thisParam instanceof ResolvedTypeVariable)
     && otherParam instanceof ResolvedTypeVariable)
Condition 19 & 20: if (thisParam instanceof ResolvedTypeVariable &&
     !(otherParam instanceof ResolvedTypeVariable))

CCN = #Conditions (Decisions/predicate nodes) + 1 = 21
```

Listing 4: Manual CC count

2.6 isAssignableMatchTypeParametersMatchingQName: Tsz Ho Wat

2.6.1 Purpose

- The provided function, isAssignableMatchTypeParametersMatchingQName, compares two ResolvedReferenceType instances for assignability while considering type parameters and nested parameterizations. It checks if the qualified names of the expected and actual types are equal and then iterates over their type parameters. It handles cases such as nested parameterizations, array types, type variables, reference types, and wildcards.

- In the case of nested parameterizations, it peels off one layer and recursively verifies assignability. For array types, it recursively checks the component types. It handles type variables by matching their names or invoking another function (matchTypeVariable). The function also addresses wildcard types, specifically handling unbounded wildcards and extends-bounded wildcards.
- The method returns true if the types are assignable, considering all the specified conditions. If an unsupported scenario is encountered, an UnsupportedOperationException is thrown.

2.6.2 Documentation

- The function is a private function, it have few comment represent the meaning of code.
- The function iterates through each type parameter, addressing nested parameterizations, arrays, type variables, reference types, and wildcards.
- The comments provide insights into the detailed comparisons made, such as peeling off layers for nested types or verifying component types for arrays.
- The function intelligently handles various scenarios, including type variable matching and wildcard comparisons.
- The documentation effectively outlines the function's purpose and the considerations for determining type assignability, making it a valuable reference for developers.

2.6.3 CC & LOC

- The NLOC is 51.
- The CCN is 19 counted by Lizard.
- The high CCN in the provided function arises due to its intricate logic for comparing and validating assignability of complex type structures, including nested parameterizations, arrays, type variables, and wildcards. Multiple conditional branches and recursive calls contribute to the elevated CCN, reflecting the code's intricate control flow.

2.6.4 Manual CC count

```
1 Condition 1: if (!expected.getQualifiedName().equals(actual.
      getQualifiedName()))
2 Condition 2: if (expected.typeParametersValues().size() != actual.
      typeParametersValues().size())
3 Condition 3: for (int i = 0; i < expected.typeParametersValues().
      size(); i++)
```

```
Condition 4,5: if (expectedParam.isReferenceType() && actualParam.
    isReferenceType())
Condition 6,7: if (expectedParam.isArray() && actualParam.isArray()
    )
Condition 8: if (expectedParam.isTypeVariable())
Condition 9,10: if (!actualParam.isTypeVariable() || !actualParam.
    asTypeParameter().getName().equals(expectedParamName))
Condition 11: if (expectedParam.isReferenceType())
Condition 12: if (actualParam.isTypeVariable())
Condition 13: if (!expectedParam.equals(actualParam))
Condition 14: if (expectedParam.isWildcard())
Condition 15: if (expectedParam.asWildcard().isExtends())
Condition 16,17: if (actualParam.isWildcard() && !actualParam.
    asWildcard().isBounded())
Condition 18: if (actualParam.isTypeVariable())

CCN = #Conditions (Decisions/predicate nodes) + 1 = 19
Same as the counted in lizard
```

Listing 5: Manual CC count

3 Coverage measurement & improvement

Our code coverage tool is very simple, it can determine if elementary branches such as if statements and loops are taken. More complex branches such as different exit points, errors and ternary operators are not supported. This instrumentation does not take into account conditions containing multiple and/or. The tool will only count these as 1 branch, however if you refractory the code to only have atomic conditions, it will be counted as multiple branches.

3.1 calculatedSyntaxModelForNode

LexicalDifferenceCalculator

Source file "com/github/javaparser/printer/lexicalpreservation/LexicalDifferenceCalculator.java" was not found during generation of report.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
calculatedSyntaxModelForNode(CsmElement, Node, List, Change)		74%		80%	12	44	33	146	0	1
loadFromFile()		0%		0%	3	3	14	14	1	1
toToken(Modifier)		36%		38%	8	13	8	14	0	1
calculatedSyntaxModelAfterListAddition(Node, ObservableProperty, int, Node)		61%		50%	1	2	1	6	0	1
calculatedSyntaxModelAfterListRemoval(Node, ObservableProperty, int)		60%		50%	1	2	1	6	0	1
calculatePropertyChange(NodeText, Node, ObservableProperty, Object, Object)		89%		50%	1	2	1	9	0	1
writeToFile(int[])		92%		100%	0	2	2	11	0	1
calculateListAdditionDifference(ObservableProperty, NodeList, int, Node)		100%		n/a	0	1	0	8	0	1
replaceEolTokens(List, LineSeparator)		100%		100%	0	2	0	5	0	1
calculateListReplacementDifference(ObservableProperty, NodeList, int, Node)		100%		n/a	0	1	0	5	0	1
calculateListRemovalDifference(ObservableProperty, NodeList, int)		100%		n/a	0	1	0	5	0	1
calculatedSyntaxModelAfterListAddition(CsmElement, ObservableProperty, NodeList, int, Node)		100%		n/a	0	1	0	4	0	1
calculatedSyntaxModelAfterListReplacement(CsmElement, ObservableProperty, NodeList, int, Node)		100%		n/a	0	1	0	4	0	1
calculatedSyntaxModelAfterListRemoval(CsmElement, ObservableProperty, NodeList, int)		100%		n/a	0	1	0	4	0	1
calculatedSyntaxModelAfterPropertyChange(CsmElement, Node, ObservableProperty, Object, Object)		100%		n/a	0	1	0	3	0	1
calculatedSyntaxModelForNode(CsmElement, Node)		100%		n/a	0	1	0	3	0	1
calculatedSyntaxModelAfterPropertyChange(Node, ObservableProperty, Object, Object)		100%		n/a	0	1	0	1	0	1
calculatedSyntaxModelForNode(Node)		100%		n/a	0	1	0	1	0	1
lambda$calculatedSyntaxModelForNode$1(Node, List, Change, CsmElement)		100%		n/a	0	1	0	1	0	1
lambda$calculatedSyntaxModelForNode$0(Node, List, Change, CsmElement)		100%		n/a	0	1	0	1	0	1
static {...}		100%		n/a	0	1	0	1	0	1
LexicalDifferenceCalculator()		100%		n/a	0	1	0	1	0	1
getNewLineToken(LineSeparator)		100%		n/a	0	1	0	1	0	1
Total	301 of 1,256	76%	32 of 113	71%	26	85	60	252	1	23

3.2 cleanLines

The clean line function had a branch coverage of 75% with the original test suite.

JavadocParser

Source file "com/github/javaparser/JavadocParser.java" was not found during generation of report.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
parse(String)	75	0%	2	0%	2	2	12	12	1	1
trimRight(String)	22	0%	4	0%	3	3	3	3	1	1
cleanLines(String)	21 77	78%	4 4	12 75%	4	9	3	13	0	1
parseBlockTag(String)	20	0%		n/a	1	1	4	4	1	1
lambda$3(String)	9	0%		n/a	1	1	1	1	1	1
lambda$2(String)	7	0%	2	0%	2	2	1	1	1	1
lambda$5(String)	6 32	84%	2 6	75%	2	5	1	8	0	1
isABlockLine(String)	5	0%		n/a	1	1	1	1	1	1
lambda$4(Javadoc, String)	5	0%		n/a	1	1	1	1	1	1
parse(JavadocComment)	4	0%		n/a	1	1	1	1	1	1
JavadocParser()	3	0%		n/a	1	1	1	1	1	1
startsWithAsterisk(String)	34	100%	1 9	90%	1	6	0	8	0	1
static {...}	13	100%		n/a	0	1	0	2	0	1
lambda$6(String)	8	100%	2	100%	0	2	0	1	0	1
Total	177 of 341	48%	15 of 44	65%	20	36	26	53	9	14

Created with JaCoCo 0.8.11.202310140853

After adding 9 tests to the test suite the branch cover was improved up to 93%

JavadocParser

Source file "com/github/javaparser/JavadocParser.java" was not found during generation of report.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
parse(JavadocComment)	4	0%		n/a	1	1	1	1	1	1
lambda$cleanLines$3(String)	3 35	92%	3 5	62%	2	5	0	8	0	1
JavadocParser()	3	0%		n/a	1	1	1	1	1	1
cleanLines(String)	98	100%	1 15	93%	1	9	0	13	0	1
parse(String)	75	100%	2	100%	0	2	0	11	0	1
startsWithAsterisk(String)	34	100%	10	100%	0	6	0	8	0	1
trimRight(String)	22	100%	4	100%	0	3	0	3	0	1
parseBlockTag(String)	20	100%		n/a	0	1	0	4	0	1
static {...}	14	100%		n/a	0	1	0	2	0	1
lambda$parse$1(String)	9	100%		n/a	0	1	0	1	0	1
lambda$cleanLines$4(String)	8	100%	2	100%	0	2	0	1	0	1
lambda$parse$0(String)	7	100%	2	100%	0	2	0	1	0	1
lambda$parse$2(Javadoc, String)	6	100%		n/a	0	1	0	1	0	1
isABlockLine(String)	5	100%		n/a	0	1	0	1	0	1
Total	10 of 343	97%	4 of 44	90%	5	36	2	52	2	14

Created with JaCoCo 0.8.11.202310140853

3.3 isApplicable

MethodResolutionLogic

Source file "com/github/javaparser/resolution/logic/MethodResolutionLogic.java" was not found during generation of report.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
isApplicable(MethodUsage, String, List, TypeSolver)		0%		0%	29	29	66	66	1	1
isApplicable(ResolvedMethodDeclaration, String, List, TypeSolver, boolean)		0%		0%	39	39	61	61	1	1
isMoreSpecific(ResolvedMethodDeclaration, ResolvedMethodDeclaration, List)		0%		0%	44	44	49	49	1	1
findMostApplicable(List, String, List, TypeSolver, boolean)		0%		0%	23	23	45	45	1	1
isAssignableMatchTypeParametersMatchingQName(ResolvedReferenceType, ResolvedReferenceType, Map)		0%		0%	19	19	35	35	1	1
inferTypes(ResolvedType, ResolvedType, Map)		0%		0%	25	25	35	35	1	1
replaceTypeParam(ResolvedType, ResolvedTypeParameterDeclaration, TypeSolver)		0%		0%	9	9	18	18	1	1
findMostApplicableUsage(List, String, List, TypeSolver)		0%		0%	8	8	15	15	1	1
isAssignableMatchTypeParameters(ResolvedType, ResolvedType, Map)		0%		0%	8	8	12	12	1	1
groupTrailingArgumentsIntoArray(ResolvedMethodDeclaration, List, ResolvedType)		0%		0%	7	7	14	14	1	1
isMoreSpecific(MethodUsage, MethodUsage)		0%		0%	8	8	13	13	1	1
matchTypeVariable(ResolvedTypeVariable, ResolvedType, Map)		0%		0%	4	4	11	11	1	1
areOverride(MethodUsage, MethodUsage)		0%		0%	5	5	8	8	1	1
isAssignableMatchTypeParameters(ResolvedReferenceType, ResolvedReferenceType, Map)		0%		0%	4	4	8	8	1	1
groupVariadicParamValues(List, int, ResolvedType)		0%		0%	2	2	7	7	1	1
getMethodsExplicitAndVariadicParameterType(ResolvedMethodDeclaration, int)		0%		0%	3	3	6	6	1	1
isExactMatch(ResolvedMethodLikeDeclaration, List)		0%		0%	3	3	4	4	1	1
findMostApplicable(List, String, List, TypeSolver)		0%		0%	2	2	4	4	1	1
isArrayOfObject(ResolvedType)		0%		0%	4	4	3	3	1	1
solveMethodInType(ResolvedTypeDeclaration, String, List, boolean)		0%		0%	2	2	3	3	1	1
isJavaLangObject(ResolvedType)		0%		0%	3	3	1	1	1	1
findCommonType(List)		0%		0%	2	2	3	3	1	1
convertToVariadicParameter(ResolvedType)		0%		0%	2	2	1	1	1	1
isApplicable(ResolvedMethodDeclaration, String, List, TypeSolver)		0%		n/a	1	1	1	1	1	1
lambda$findMostApplicable$3(String, List, TypeSolver, boolean, ResolvedMethodDeclaration)		0%		n/a	1	1	1	1	1	1
getLastParameterIndex(int)		0%		n/a	1	1	1	1	1	1
distinctByKey(Function)		0%		n/a	1	1	2	2	1	1
solveMethodInType(ResolvedTypeDeclaration, String, List)		0%		n/a	1	1	1	1	1	1
lambda$findMostApplicableUsage$4(String, List, TypeSolver, MethodUsage)		0%		n/a	1	1	1	1	1	1
lambda$distinctByKey$1(Set, Function, Object)		0%		n/a	1	1	1	1	1	1
lambda$findMostApplicable$2(String, ResolvedMethodDeclaration)		0%		n/a	1	1	1	1	1	1
lambda$replaceTypeParam$0(ResolvedTypeParameterDeclaration, TypeSolver, ResolvedType)		0%		n/a	1	1	1	1	1	1
static {...}		0%		n/a	1	1	1	1	1	1
MethodResolutionLogic()		0%		n/a	1	1	1	1	1	1
Total	2,016 of 2,016	0%	464 of 464	0%	266	266	429	429	34	34

3.4 compareConsideringTypeParameters

There is no coverage reporting for compareConsideringTypeParameters. This is because for some reason the jacoco isn't reporting the coverage for the javaparser-symbol-solver-testing module. However since there is at least one test that's called with the function, the coverage is either the same or more. There are at least one test which use a function that depends on compareConsidering-TypeParameters - ReferenceTypeImpl::isAssignableBy(). However we cannot see coverage from this.

3.5 isAssignableMatchTypeParametersMatchingQName

In isAssignableMatchTypeParametersMatchingQName function, only 12/18 branch is covered. There are still some branch is not covered.

4 Refactoring plan

4.1 calculatedSyntaxModelForNode

4.2 cleanLines

To reduce the high complexity in cleanLines the function can be split up into smaller units. The simplest way to achieve this is by splitting up the function on each decision. One alternative is to create a new cleanLines method that calls multiple helper methods containing a subspace of the original decisions each. The new methods and their metrics can be wived bellow:

```
List<String> cleanLines(String content) {}
List<String> processLines(String[] lines) {}
String getProcessedSubstring(String line, int asteriskIndex){}
List<String> normalizeWhitespace(List<String> lines){}
boolean startsWithWhitespace(String line){}
void removeLeadingWhitespace(List<String> lines){}
void removeEmptyLines(List<String> lines){}
  NLOC    CCN   token  param  length
    13      4      88      1      18 cleanLines@92-109@
    12      3      73      1      15 processLines@111-125@
    9       4      70      2      10 getProcessedSubstring@127-136@
    3       2      45      1       3 normalizeWhitespace@138-140@
    3       3      35      1       3 startsWithWhitespace@142-144@
    3       1      29      1       3 removeLeadingWhitespace@146-148@
    8       5     100      1       9 removeEmptyLines@150-158@
```

Listing 6: Manual CC count

4.3 isApplicable

To reduce the high complexity in isApplicable, we split the function to make five extra functions. These first four extra functions retrieves the conditions to be check for in the final extra function. We did opt to fetch certain constant variables multiple times for each function, this might not be the best option for performance depending on how heavy these variables can take to fetch. An example of these constant variables would be the typeParameters variable which is use in multiple of the new extra functions. This was done to reduce the amount of parameters used in the extra functions, some would otherwise need 8+ parameters instead of max 4 for each (except last which has 5).

```
public static boolean isApplicable(MethodUsage methodUsage, String
    needleName, List<ResolvedType> needleParameterTypes, TypeSolver
     typeSolver)
private static ResolvedType getExpectedArgumentType(MethodUsage
    methodUsage, List<ResolvedType> needleParameterTypes, int idx,
    ResolvedType actualArgumentType)
private static ResolvedType
    inferParameterTypesAndReplaceTypeVariables(MethodUsage
    methodUsage, ResolvedType expectedArgumentType, List<
    ResolvedType> needleParameterTypes)
```

```
private static ResolvedType handleTypeVariableReplacementCases(
    MethodUsage methodUsage, ResolvedType expectedArgumentTypeInput
    , TypeSolver typeSolver)
private static ResolvedType handleTypeVariableBoundsCases(
    MethodUsage methodUsage, ResolvedType
    expectedTypeWithoutSubstitutions, TypeSolver typeSolver)
private static boolean isAssignable(ResolvedType
    expectedArgumentType, ResolvedType
    expectedTypeWithSubstitutions, ResolvedType
    expectedTypeWithInference, ResolvedType
    expectedTypeWithoutSubstitutions, ResolvedType
    actualArgumentType)
  NLOC    CCN   token  param  length
    30      8    255      4      56 isApplicable@435-490
    16      5    127      4      27 getExpectedArgumentType@493-519
    22      5    186      3      29
    inferParameterTypesAndReplaceTypeVariables@521-549
    19      5    180      3      25
    handleTypeVariableReplacementCases@551-575
    19      5    170      3      21
    handleTypeVariableBoundsCases@577-597
    10      4     48      5      10 isAssignable@599-608
```

Listing 7: Manual CC count

4.4 compareConsideringTypeParameters

To reduce the cyclomatic complexity, we split the function into 3 more private functions. Two to handle the Wildcard and normal instance checks, and a last one for checking each parameter separately.

```
protected boolean compareConsideringTypeParameters(
    ResolvedReferenceType other)
private boolean compareConsideringTypeParameter(ResolvedType
    thisParam, ResolvedType otherParam)
private boolean compareConsideringVariableTypeWildcardParameter(
    ResolvedWildcard thisParam, ResolvedType otherParam)
private boolean compareConsideringVariableTypeInstanceParameter(
    ResolvedType thisParam, ResolvedType otherParam)
NLOC   CCN   token  param  length
20       8    147      1      21
    compareConsideringTypeParameters@442-462
15       5     73      2      15 compareConsideringTypeParameter@467
    -481
13       6     83      2      13
    compareConsideringVariableTypeWildcardParameter@482-494
14       8    181      2      14
    compareConsideringVariableTypeInstanceParameter@495-508
 9       4    100      2      12
    compareConsideringVariableTypeParameters@509-520
```

Listing 8: Manual CC count

4.5 isAssignableMatchTypeParametersMatchingQName

To reduce the cyclomatic complexity, split the function is a solution.Create multiple function to reduce the number of decision. The new method is below:

```
1  private static boolean isAssignableMatchTypeParametersMatchingQName
       (ResolvedReferenceType expected, ResolvedReferenceType actual,
       Map<String, ResolvedType> matchedParameters) {}
2  private static boolean checkQualifiedNames(ResolvedReferenceType
       expected, ResolvedReferenceType actual) {}
3  private static boolean checkTypeParameterSizes(
       ResolvedReferenceType expected,
4      ResolvedReferenceType actual){}
5  NLOC    CCN   token  param  lenght
6  19      7     151      3      24 isAssignableMatchTypeParameters-
                                      MatchingQName@305-328@
7  3       1      26      2       3 checkQualifiedNames@330-332@
8  3       1      31      2       3 checkTypeParameterSizes@334-336@
9  9       3      65      3       9 isAssignableReferenceType@338-346@
10 9       3      73      3       9 isAssignableArrayType@348-356@
11 11      4      88      3      11 isAssignableTypeVariable@358-368@
```

Listing 9: Manual CC count

5 Self-assesment

The group is in the Use state. The agreed upon practises are being used to do real work. GitHub is being used to track issues to be completed and for assigning work. Discord is being used as the main communication channel for discussing general issues and problems with completing tasks, as well as for updating on current progress in the project. The work is going well, but some improvement in communication is still needed. At this state, everyone has become well accustomed to working with GitHub; branching, creating issues, merging and reviewing pull requests. Coding in Java is working well, with only some minor mistakes and inconsistencies. We should have put more effort on selecting a suitable project. The project we chose turned out to be badly suited for this assignment, even tough it followed all the requirements, but this was discovered when too much work already was made, and there was not enough time to redo the assignment on a different project.