

DESIGN PROJECT REPORT

IntelliJML

16th April 2021

Authors:

Steven Monteiro - s2160501

Erikas Sokolovas - s2171139

Ellen Wittingen - s2095610

Supervisors:

Dr. Tom van Dijk

Prof. Dr. Marieke Huisman

FACULTY OF ELECTRICAL ENGINEERING, MATHEMATICS AND
COMPUTER SCIENCE

TECHNICAL COMPUTER SCIENCE DESIGN PROJECT

UNIVERSITY OF TWENTE.

Table of contents

1	Introduction	4
1.1	Report Outline	4
1.2	Background	4
1.3	Motivation	4
1.4	Goal	5
2	Stakeholders	6
3	Requirements specification	7
3.1	User requirements	7
3.2	System requirements	8
4	Preliminary investigation	10
4.1	Updating OpenJML to support newer versions of Java	10
4.2	JML	10
4.3	Language embedding in the IntelliJ Platform	10
5	Design choices	12
5.1	Choice of IDE, tools and programming language	12
5.2	IntelliJ and Java version support	13
5.3	Supported JML subset	13
5.4	Plugin data flow	14
5.5	Syntax highlighting	16
5.6	Runtime checking	17
6	Main features	19
6.1	Must requirements	19
6.2	Should requirements	23
6.3	Could requirements	24
6.4	JML syntax supported by the plugin	25
7	Testing	29
7.1	Unit tests	29
7.2	Integration tests	31
7.3	Test coverage	31
7.4	System tests	32
7.5	Usability tests	32

8	Organisation, sprints and meetings	34
8.1	Organisational procedures	34
8.2	Original planning	34
8.3	Actual activities	34
8.4	Project meetings	36
9	Reflection	38
9.1	Challenges	38
9.2	Group dynamics	40
9.3	Usability of the project for Software Systems	40
10	Future	42
10.1	Updating the module guide	42
10.2	Use of JML in Software Systems	42
10.3	Maintenance	42
10.4	Future work	43
A	User manual	46
B	JML Reference	51
B.1	Method specifications	51
B.2	Class invariants	52
B.3	In-method specifications	53
B.4	Modifiers	53
B.5	Expressions inside specifications	54
B.6	Comments	56
C	Guide for the maintainer	57
C.1	Tools	57
C.2	Version support	57
C.3	Data flow	58
C.4	Language embedding	59
C.5	Lexing and parsing	61
C.6	Annotation	63
C.7	Code completion	64
C.8	Runtime checking	65
C.9	Testing	66
C.10	Known bugs	67

D	System tests	69
E	Usability testing results	75
E.1	Participant 1	75
E.2	Participant 2	76
E.3	Participant 3	77
E.4	Participant 4	78

1 Introduction

1.1 Report Outline

This report will outline the project process for the JML plugin project, from requirements gathering to the final submission of the project, whilst discussing the rationale and notable occurrences in the project, e.g. major problems, research findings etc. At the end of this report an evaluation of the project will be given, as well as suggestions on the future of the project.

1.2 Background

At the University of Twente, in the module Software Systems, students are introduced to software development practices and are taught the Java programming language. Alongside Java, a specification language called Java Modelling Language (JML) [1] is taught to students. It helps them reason about the behaviour of their Java code, as JML allows for the formal specification of the intended behaviour of Java code.

Up to and including the academic year 2018-2019, JML was included in the curriculum of the Software Systems module. But in the academic year 2019-2020, a decision was made to switch from Java version 8 to Java version 11. This change meant that JML could no longer be taught in the module, as the tooling that supported JML was incompatible with versions of Java higher than Java version 8.

Additionally, the module also provides students with an introduction to integrated development environments (IDE). An IDE is a kind of software that is meant to help programmers develop software by offering facilities such as a text editor, code completion, build tools, a terminal, etc. In the preceding iterations of the Software Systems module, the Eclipse IDE was used in the module. But current planning is that in the next iteration of the Software Systems module, IntelliJ IDEA will replace Eclipse as the IDE used in the module.

1.3 Motivation

The motivation for accepting this project is based on the project group members experience as students in the Software Systems module and the desire

to help provide future students in the Software Systems module with a good first experience with JML.

1.4 Goal

Considering the above mentioned background, the goal of the project is to allow JML to be reintroduced to Software Systems by creating an easy-to-use plugin for IntelliJ IDEA. To achieve this the plugin needs to provide such IDE features as syntax highlighting, code completion, syntax, semantic and type checking. These features should aid students in writing JML specifications in a convenient manner.

The plugin should also support modern versions of Java and ensure, as much as possible, compatibility with future versions of Java. This goal is important to avoid a repeat of the current situation where JML was removed from the module curriculum due to Java version incompatibility.

2 Stakeholders

This section describes the stakeholders involved in the project, which are parties with potential interests in the project. It is important to consider these parties as they can affect the outcome of the project. A short description of relevant stakeholders is given.

Coordinator of Software Systems This is the client of the project. They coordinate the Software Systems module and want to add JML back to the course. They are interested in a plugin that helps students learn JML.

JML teacher of Software Systems They teach JML in the module, and want students to successfully write JML for their Java code. They are interested in a plugin that helps students learn JML.

Student of Software Systems These students are (mostly) completely new to JML when the module starts. They are taught JML by the JML teachers. These students will be the main users of the plugin. They are primarily interested in a plugin that will help them as much as possible with writing JML.

Teaching assistant in Software Systems Teaching assistants (TAs) help students with their questions during tutorials and also grade the final project. Students can ask questions about JML which the TAs then need to answer. The final projects that they grade might also contain JML. They are interested in having a plugin that can help students resolve most if not all JML problems they can encounter.

Student that does not like JML This is a negative stakeholder. For each subject in the course, there exist students that do not like that subject. It is important to include these negative stakeholders as they will still be a part of the user group of the plugin. They are interested in the plugin making their work with JML as easy and as fast as possible.

3 Requirements specification

This section of the document covers the captured project requirements. They are divided into user requirements and system requirements. The user requirements are in user story format. The requirements are based on discussions with the project clients and the authors' own experiences as students in the Software Systems module.

3.1 User requirements

1. As the coordinator of Software Systems, I want to include JML in the curriculum.
2. As the coordinator of Software Systems, I want the plugin to support Java 11 and higher, and also future Java versions.
3. As the coordinator of Software Systems, I want the plugin to support the IntelliJ IDE.
4. As a JML teacher in Software Systems, I want the plugin to support all JML keywords and operators that were taught in Software Systems in 2018.
5. As a JML teacher in Software Systems, I want the plugin to support some more advanced JML keywords and operators than what has been taught, for students who want to learn more.
6. As a JML teacher in Software Systems, I want the plugin to show a warning when there is a space between the start of the JML comment and the @-sign, as then the comment should not be considered as JML.
7. As a JML teacher in Software Systems, I want the plugin to conform to the official JML specification.
8. As a student in Software Systems, I want to get feedback when I use incorrect JML syntax.
9. As a student in Software Systems, I want to get feedback when I use incorrect types in my JML specifications.

10. As a student in Software Systems, I want to get feedback when I put my JML specifications at an incorrect position in the Java file.
11. As a student in Software Systems, I want to get feedback when I make common JML mistakes.
12. As a student in Software Systems, I want syntax highlighting for my JML.
13. As a student in Software Systems, I want intelligent code prediction for my JML.
14. As a student in Software Systems, I want the plugin to be stable, so it does not show me internal errors.
15. As a student in Software Systems, I want the plugin to look and feel integrated into the IDE.
16. As a student, I want to receive feedback on my JML specifications quickly.
17. As a TA in Software Systems, I want JML error and warning messages to be understandable to students, so I do not have to help with every little JML problem.
18. As a TA in Software Systems, I want to check that students wrote their JML correctly according to the JML specification.
19. As a student that does not like JML, I want to write JML as painlessly as possible.

3.2 System requirements

3.2.1 Functional requirements

- Support the syntax of the subset of JML used in the Software Systems module, plus some additional JML keywords and operators for more advanced students.
This supports user stories 4 and 5.
- Ability to embed JML in Java source files in IntelliJ.
This supports user story 3.

- Type check Java expressions in JML.
This supports user story 9.
- Meaningful JML error messages.
This supports user stories 8, 9, 10, 11, and 17.
- Syntax highlighting for JML.
This supports user story 12.
- Code completion for JML.
This partly supports user story 13.
- Provide warnings and errors for common mistakes.
This supports user stories 6, 10, and 11.

One of the main objectives of the plugin is to allow the official JML syntax to be reintroduced into Software Systems. Therefore, all of the above requirements help support user story 1.

Additionally, these items aim to help with the sentiment of user story 19 by making the usage of JML more convenient for them.

3.2.2 Quality requirements

- Ensure as much as possible forward compatibility with future versions of Java.
This supports user story 2.
- Plugin should follow the look and usage style of the IntelliJ IDE platform.
This supports user story 15.
- High stability and robustness.
This supports user story 14.
- Conformity to the official JML specification.
This supports user stories 18 and 7.
- The plugin should give feedback to the user within 1 second after a user action.
This supports user story 16.

4 Preliminary investigation

This section describes the investigations done to implement the project.

4.1 Updating OpenJML to support newer versions of Java

OpenJML [2] was the JML tooling that was previously used in the Software Systems module. At the start of the project, it was examined whether it might be possible to update OpenJML to work with Java versions higher than 8, as this was the primary reason why it could not be used in the module. Unfortunately, it was concluded that this was not realistically achievable in the time-span of the project. This is because OpenJML utilised an internal API in OpenJDK that was deprecated after version 8, and it was found that it was not realistic to adapt the IntelliJ Java API to match the behaviour of the JDK in a reasonable amount of time.

4.2 JML

Since it was decided that it was not feasible to update OpenJML to work with current versions of Java, much of OpenJML's functionality needed to be re-implemented in the new plugin. But it was not necessary to support all the features that JML has, as students are only taught a subset of JML. To determine which JML features should be supported, the JML teaching materials of the last academic year that JML was taught in were examined and cross-referenced against a JML specification reference manual that was linked on the OpenJML website [3]. The clients were also consulted to ensure that the subset that was decided on was appropriate and not missing a feature of JML required for the Software Systems module.

4.3 Language embedding in the IntelliJ Platform

To make JML work as intended, it was necessary to determine how JML could be embedded into Java and have IntelliJ recognise the JML. Additionally, a JML parse tree was needed for semantic functionality, so the custom language API of IntelliJ also needed to be explored. While the IntelliJ platform does provide multiple relatively simple ways to achieve this, the plugin needed to be able to parse Java expressions within JML. This meant that Java needed

to be embedded into JML. However, IntelliJ explicitly forbids embedding another language into an already embedded language.

This is very problematic in the case of JML, as in certain JML specifications, JML and Java expressions can be arbitrarily interwoven and nested within each other. This meant that these parts of JML could not be evaluated properly. The problem was further exacerbated by the fact that many of the more advanced parts of the IntelliJ API that were supposed to allow developers to handle such specific edge cases were completely undocumented.

To solve this issue, an extensive investigation of many official and community made plugins that add support for various programming languages had to be conducted. The hope was that one these plugins had a clear solution to this kind of problem. After multiple weeks of sifting through documentation and other plugin repositories, a solution was engineered by combining multiple solution patterns that were found in the many open-source plugins that were examined. For the specifics of the solutions, see section C.4.

5 Design choices

This section of the report covers the core design choices made during the project and the reasoning behind them.

5.1 Choice of IDE, tools and programming language

The clients indicated in the first meeting that they would like the plugin to support IntelliJ, as the module coordinator was certain that only IntelliJ will be used during the module next year. As such, the supported platform decision was made by the client.

But there were other considerations to make. The first consideration was what tools to use to make the grammar for JML, as there are several tools available for that, such as ANTLR [4] and IntelliJ's built-in Language API [5]. After conducting research (see section 4), it was decided to use IntelliJ's built-in Language API. This choice was made as it is part of the IntelliJ platform, and thus maintained by IntelliJ developers instead of a third party. This would be of benefit to the client as it would be supported for longer. The client indicated that support for future Java versions is a core requirement.

IntelliJ gives developers some choice in what language they want to write their plugins in, such as Kotlin or Java [6]. It was decided to write the plugin in Java as the authors are most familiar with Java and most documentation on the IntelliJ platform provides code examples in Java.

A tool called Grammar-Kit [7] and a tool called PsiViewer [8], were used to write the grammar in BNF form. The tools provide facilities where a developer can inspect the parse tree generated from some sample text in real time and any changes made to the BNF grammar are immediately reflected in the parse tree. This saves a large amount of development time as there is no need to regenerate the parser and lexer each time a change is made to the BNF. The tool Grammar-Kit also allows for automatic generation of a lexer and parser from the BNF grammar.

JetBrains also provides a plugin project template that has a large amount of the infrastructure needed to make a plugin pre-configured [6]. This template was used as the basis of the plugin to speed up development at the start of the project. The template includes run configurations, Gradle configuration that already include most build tasks, code style checks, and a Gradle task that automatically configures a sandbox instance of IntelliJ where the

plugin is automatically loaded at launch. This dramatically simplifies and speeds up the testing process and also allows the use of debugging tools to diagnose issues in the plugin.

At the beginning of the project, the official tutorial on adding custom language support to the IntelliJ platform was followed due to unfamiliarity with the IntelliJ platform [9]. But the tutorial only went over the basics of this process and an explanation on much of the more advanced features was omitted. This necessitated the need for exploration of the source code of other IntelliJ plugins that add support for custom languages to fill in the missing information.

5.2 IntelliJ and Java version support

The plugin was written in Java 8 and attention was given to avoid using deprecated, experimental, or scheduled-for-removal parts of the IntelliJ API. The plugin itself attempts to avoid handling Java as much as possible, and where that is not possible, it hands that off to the IntelliJ API.

The supported IntelliJ platform versions are from version 2020.1 to the latest version, which is 2021.1 at the time of writing. Further backwards compatibility was not practical to achieve due to API changes made in version 2020.1.

The only plugin dependencies are the IntelliJ platform itself and the Grammar-Kit plugin. Both of these dependencies are maintained by JetBrains staff.

So long as IntelliJ developers keep IntelliJ up to date with newer Java versions and do not change the API too heavily, the plugin should keep working. Some minor changes might be necessary to keep the plugin working if a major change in the Java specification or the IntelliJ API occurs in the future, but this is difficult to predict.

5.3 Supported JML subset

The subset of supported JML was decided on after consultation with the clients. The clients indicated that the subset of JML that was taught in the 2018 edition of the Software Systems module had to be supported, but that a somewhat larger subset should also be supported to allow more ambitious students to explore JML. To determine the planned subset, the JML teaching materials used in Software Systems in 2018 were examined and

cross-referenced against the JML manual [3]. On pages 18-24 of the JML reference manual the JML language feature levels are described. Only parts of feature level 0 and one level 1 feature were found to be taught in Software Systems. Based on this finding and with the approval of the clients, the primary focus of the project was on implementing as many languages features from level 0 as possible, along with the `\min`, `\max`, `\sum`, `\product`, and `\num_of` quantifiers and the keyword `pure` from language level 1. The quantifiers were included as they can help avoid writing helper method when iterating over arrays.

5.4 Plugin data flow

To help illustrate how the plugin works, figure 1 contains a basic data flow diagram of the plugin.

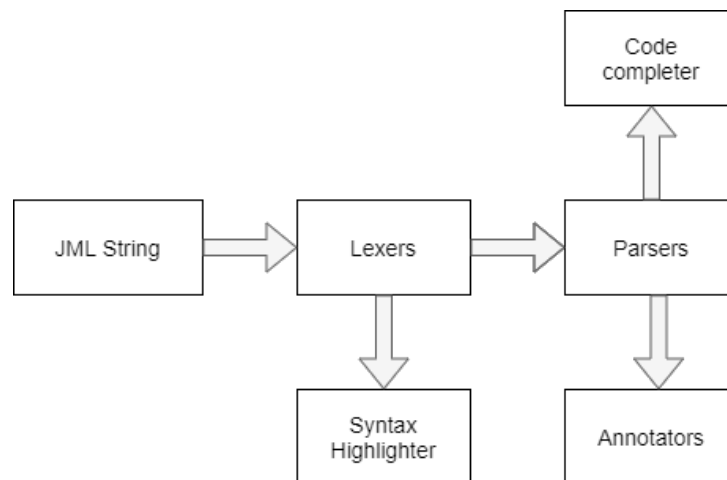


Figure 1: The flow of data through the plugin

Lexing First the plugin receives a string of a JML comment from the IDE. This string is then passed to the JML lexer. This lexer is technically four different lexers in one. This single layered lexer was achieved by having a lower layer lexer pass up its output to a higher level lexer. The different layers tokenize the string into JML and Java tokens, as well doing some token merging. The complexity of the lexer is due to the need to have Java tokens embedded into JML. This was necessary as both code completion and

syntax highlighting are dependent on Java tokens to correctly interpret Java expressions inside JML.

Syntax highlighting As just mentioned, the tokens generated by the lexer are used by the syntax highlighting to decide how each part of the text should be coloured.

Parsing The lexer output is passed up to the JML parser that then generates a JML parse tree. For the JML parser to be able to recognise and correctly parse Java tokens an extension to the JML parser had to be written manually. This was necessary to generate usable parse trees for the sections of JML that can contain both JML and Java. This is particularly needed with quantified expressions.

Error checking The parse tree generated by the parser is used by various annotators. Annotators perform error checking, and if they encounter an error, they generate error and/or warning messages and place them at the appropriate positions in the editor. There are four annotators, each responsible for generating a different type of message. There are four types of messages: JML validity (whether or not the IDE is interpreting this part of the code as JML), syntax checking, semantic checking, and type checking. Type checking additionally checks that code references can be resolved correctly.

The reason for the division is to save resources and avoid confusion for the user. The annotators have a hierarchy and run in sequence. If one annotator finds that an error needs to be displayed, it will stop the annotation there to prevent other error messages from showing. This saves resources as incorrect JML will not be further annotated and multiple completely different error messages will not be displayed to the user, thereby avoiding confusion.

One of the main complaints made by students about the old JML tooling were the indecipherable error messages. An effort was made to avoid this by making the error and warning messages generated by the annotators as descriptive and helpful as possible. For more details on annotation refer to section 6.1.3

Code completion The original intention was to reuse the code completion functionality that the IntelliJ platform provides for Java that is embedded in

JML. However, this was not possible as the Java parse tree that is embedded in the JML parse tree is simply not what the Java completion service is expecting. It fails to provide any meaningful completion as it relies heavily on contextual information to provide relevant completions.

Therefore it was necessary to implement custom code completion for the Java code in JML. It was decided that it would be best that the custom completion service would mimic the behaviour of regular IntelliJ code completion service as closely as possible. This was to ensure that the student would not feel a difference between writing JML and Java in the IDE in terms of how the IDE treated the two languages.

Context-aware JML keyword completion is also included in the plugin. The completion service tries to ensure that only context-appropriate JML keyword completions are offered as suggestions. This is to make sure that if students use the completion service as a crutch to write JML, they will not write incorrect JML. But for the cases that were not accounted for, there are still the syntax and semantic checkers that ensure that the error will be caught and highlighted so that the student can fix it manually.

Certain common JML programming patterns were also included as templates. For example, typing `\forall` and then pressing the enter key will change the `\forall` to `(\forall int ; ;)`. This type of pattern is available for all quantifiers. This should help students write JML boilerplate code faster.

5.5 Syntax highlighting

For the colour highlighting it was decided that JML should not stand out more than the Java code. Otherwise users might believe that it is part of the Java code or be overwhelmed by the combination of bright Java and JML highlighting. Thus, more muted colours were chosen for JML highlighting. For Java tokens within JML, the highlighting was kept the same as the regular Java highlighting provided by IntelliJ. This was done to keep a consistent style between the regular Java and the Java within JML. An example of the colour highlighting can be found in figure 2.

A highlighting colour settings page was also implemented which allows users to change the highlighting colours manually if they do not like the default colours.

```

/*@
  requires (\forall int i; i <= 0 && i < boxes.size(); boxes.get(i).getWeight() >= 0);
  ensures \result == (\sum int i; i<= 0 && i < boxes.size();boxes.get(i).getWeight());
  ensures (\forall int i; i <= 0 && i < boxes.size(); \result >= boxes.get(i).getWeight());
  ensures \result >= 0;
*/
// returns the weight of all the boxes together
public float getTotalWeight() {
  int currentWeight = 0;
  for (Box box : boxes) {
    currentWeight += box.getWeight();
  }
  return currentWeight;
}

```

(a) in the default dark theme (Darcula)

```

/*@
  requires (\forall int i; i <= 0 && i < boxes.size(); boxes.get(i).getWeight() >= 0);
  ensures \result == (\sum int i; i<= 0 && i < boxes.size();boxes.get(i).getWeight());
  ensures (\forall int i; i <= 0 && i < boxes.size(); \result >= boxes.get(i).getWeight());
  ensures \result >= 0;
*/
// returns the weight of all the boxes together
public float getTotalWeight() {
  int currentWeight = 0;
  for (Box box : boxes) {
    currentWeight += box.getWeight();
  }
  return currentWeight;
}

```

(b) in the default light theme

Figure 2: JML syntax highlighting colours

5.6 Runtime checking

For the purpose of implementing runtime checking, it was hoped that IntelliJ had a simple API available with the ability to insert code at compile time. Unfortunately, no direct support existed for this use case, but it was possible to work around this limitation using related methods from the API. For example, extensive use is made of an API ability to walk the parse tree of a Java file. However, this workaround only permitted making insertions visible to the user, which is considered undesirable since the inserted code duplicates what a human reader can already infer from the JML specification itself.

Two possible approaches to this problem were considered. The first involved inserting checks at runtime in the form of Java bytecode by using a modified Java executable or Java agent. The second involved automatically copying the project's source files during compilation and performing insertions into the copied files, leaving the originals untouched. The consideration was made that while the former solution would be more robust, none of the authors had any prior experience with bytecode insertions, and as such the implementation would be prohibitively difficult. Furthermore, making use of a modified Java executable would create a potential for version compatibility problems arising from changes in as of yet unreleased Java versions. For these reasons, the latter approach of copying source files was chosen.

Fully implementing runtime checking was not possible due to time constraints. If fully implemented, this system would have had the ability to transparently insert runtime checks into any Java file annotated with JML that was compiled while the plugin was active. Such functionality is currently partially present in the code, but not accessible to the user.

6 Main features

At the start of the project a feature proposal was presented to the clients, divided into different priorities according the MoSCoW model [10]. This section will describe which proposed features the plugin supports.

6.1 Must requirements

All of the "must" requirements are supported by the plugin. These consisted of IntelliJ support, future Java version support, error checking, correctness, and stability.

6.1.1 IntelliJ support

The plugin supports IntelliJ IDEA Community and IntelliJ IDEA Ultimate from version 2020.1 up to the current version at the time of writing, 2021.1. The plugin is expected to support future versions of IntelliJ as well, as it was built mainly using IntelliJ's own APIs.

6.1.2 Future Java versions

Java 8 up until the newest version (Java 16 at the time of writing) is supported. As much Java (in JML comments) parsing and processing as possible is offloaded to the IntelliJ platform to try to ensure forward compatibility. This means that as long as IntelliJ developers maintain IntelliJ support for future versions of Java, which is to be expected, the plugin will likely support those versions as well.

6.1.3 Error checking

The plugin should provide meaningful error messages as feedback to the users of the plugin, such that they will be able to quickly and easily diagnose issues that arise from malformed JML. To satisfy this requirement, the plugin checks for whitespace before the first at-sign, syntax, semantics, and the return type of JML expressions and whether references in the expressions can be resolved. Checks for common mistakes are also preformed.

Another important aspect is that the error messages should be displayed in a similar fashion as Java syntax errors are displayed in the IntelliJ platform.

This is to make sure that the plugin feels integrated into the IntelliJ workflow, rather than something that was tacked on. To accomplish this, the generated error messages are given to an IntelliJ API which displays the messages as if they are IntelliJ's own error messages.

6.1.4 Checking done by the plugin

Whitespace before the first at-sign First, it is checked that there is no whitespace between the start of the comment and the first at-sign. If that is the case, a weak warning is shown saying that it will not be considered a JML comment, and the comment is not further checked. Also, the entire comment is then coloured the same as a normal comment, to further indicate that is not considered JML.

Syntax checking For syntax checking, error messages are received from the parser, and then some of them are replaced with clearer messages. The situations in which such replacements are done can be found in the list below.

- When a `requires` clause comes after some other method specification clause that is not a `requires` clause (in the same comment without using `also`).
- When a method specification clause comes after a modifier.
- When a `signals` clause does not have parentheses around the exception name.
- When a `signals_only` clause is missing exception names.
- When an assignable clause has a variable with a suffix of the form `[x]` or `[0..x]`, as those are not supported.
- When there are no parentheses around a quantified expression.
- Some general replacements are done to make all default messages a bit clearer, such as replacing a list with all possible modifiers with the word "modifiers".

Another feature in this category is that when there is a syntax error in the JML comment, the entire comment is coloured the same as a default

comment. This was done because when there is a syntax error, the comment is not checked for semantic and type errors. So if the user would not notice the syntax error (which might only show an error on a single character), they would think their entire JML comment is correct, whilst there might be semantic and type errors besides the syntax error. Not highlighting the comment as usual makes sure the user notices that there is a mistake in the syntax.

Semantic checking Some semantic checking is done as well. Most of these checks also work when the JML specifications are spread across multiple comments.

Error messages are shown to the user:

- When the specification is not allowed at its position in the Java file (e.g. a method specification is not allowed above a field).
- When a method specification clause comes after a modifier, checked across comments.
- When a loop invariant is not above a loop.
- When `\result` is not allowed to be used in the specification.
- When `\old()` is not allowed to be used in the specification.
- When a certain modifier is not allowed in the specification.
- When the `helper` modifier is not allowed (requires more checks than the previous check)
- When multiple visibility modifiers of different sorts are used in the same specification (e.g. using both `public` and `private` on an invariant).
- When multiple specification visibility modifiers of different sorts are used in the same specification (e.g. using both `spec_public` and `spec_protected` on a field).
- When the modifiers `static` and `instance` are both used in the same specification.

Warning messages are shown to the user:

- When multiple `signals_only` clauses are used, as that can be confusing.
- When `\not_specified` is redundant as the clause has already been defined.
- When the same modifier is used multiple times.
- When there is a suspicion that a called method inside the JML comment is not pure. (Sometimes IntelliJ cannot infer whether the method is pure, so that is why a warning is given instead of an error).

For the `pure` modifier, it is not checked whether the method this modifier is linked to is actually pure. There was the intention to implement a manual check for this, but due to time constraints it could not be implemented.

Type checking Type checking and checking that all references can be resolved in JML clauses is also done.

These checks are done for:

- Assignable clauses. It is also checked that the references are allowed to be used with `.*` or `[*]`, and that the references do not point to final fields.
- Signals and `signals_only` clauses. It is also checked that the reference can be cast to `java.lang.Exception`, and, if it cannot be casted to a `RuntimeException`, it is also checked that it (or a super or subclass) is mentioned in the `throws`-clause of the method.
- References in an invariant. It is checked whether the specification visibility of the references in an invariant corresponds to the visibility of the invariant. This also includes a check that static invariants do not use references to instances.
- JML expressions that start with a backslash and take a Java expression, such as `\old()`, `\typeof()`, etc. It is checked that the Java expressions inside the parentheses are correct.
- Quantified expressions. It is checked that the declared variables do not already exist, that the range predicate is of type boolean, and that the body returns the correct type.

- JML operators. It is checked that the expressions on both sides are of type `boolean`.
- Most of Java's different kinds of expressions, such as a boolean expressions, type casts, array accesses, references, method calls, etc.
- When assigning variables in a Java expression inside a JML comment, an error is given, as that is not pure.

The only kind of Java expressions that are not checked are lambda expressions. This means that no errors will be shown for those, so the user needs to check themselves whether they wrote their lambda expression correctly. These are not checked as they are rather complicated to check properly.

6.1.5 Correctness and stability

It is important that the error checking of this plugin is implemented correctly. This means that if the written JML is incorrect, the correct error needs to be given, and if it is correct, no error should be given. If this is not the case, students might learn JML incorrectly, which would directly conflict with one of the main goals of this project. To ensure this correctness (adherence to the JML specification), the guidelines laid out in JML reference manual [3] were closely followed and when ambiguities arose one of the supervisors was consulted, who is highly experienced with JML. To check that the plugin actually followed the JML guidelines, extensive testing (see section 7) was performed on the plugin.

The plugin should also be stable and not show internal errors to the user if they occur. It should also interface correctly with IntelliJ and not make the IDE unstable. To ensure this, most internal errors that could be thrown are caught and handled. Also, the code was written in such a way to minimise the number of internal errors that could be thrown, such as always checking whether an object exists before using it. This stability was put to the test by the same testing previously mentioned.

6.2 Should requirements

All "should" requirements were also implemented, which consisted of syntax highlighting, warnings for common mistakes and code completion.

6.2.1 Syntax highlighting

The requirements for syntax highlighting were the following: JML keywords and Java expressions embedded into JML should be highlighted with colours. And, preferably, the embedded Java expression highlighting should look similar to the highlighting done by IntelliJ's own highlighter for Java expressions. Both of these requirements were met. As already mentioned in the design choices (section 2), all JML keywords are highlighted with a custom colour theme, and embedded Java expressions are highlighted with the default colours for Java expressions of the user's currently selected theme.

6.2.2 Common mistake warnings

Sometimes students make some common JML mistakes that are not errors, but are pointless or specify potentially unwanted program behaviour. Therefore, it was proposed to implement some checks for those. These common mistake warnings were implemented together with the error checking, so they were already described in the sub-section about error checking (sub-section 6.1.3).

6.2.3 Code completion

Basic code completion was implemented in the same fashion as IntelliJ provides for Java. When one starts typing a word, based on what has already been typed, the IDE will suggest what could be the fully finished word. This will be helpful to students as it will help them remember what keywords JML has and also explore other JML features.

This kind of code completion is supported by the plugin for most of JML and embedded Java expressions inside JML. The code completion for the embedded Java expressions is not as good as the IntelliJ's Java code completion as that could not be reused and thus had to be manually implemented. However, it should be sufficient for most Java expressions that students could feasibly write.

6.3 Could requirements

There were two "could" requirements, namely support for runtime checking and a private repository for distribution of the plugin. Due to time con-

straints, neither of these were implemented, though it should be mentioned that there is a partial runtime checking implementation in the source code.

6.3.1 Runtime checking

If time had permitted it, some runtime checking would have been implemented. This feature would have inserted assertions at the correct positions in the code to verify that the preconditions and postconditions specified in the JML specifications are met. Then the JML specification would have an actual effect on the code that students write, which would help students see the usefulness of JML. A start was made with implementing this, such as a system to copy files and insert custom assertions in them. However, the generation of the actual assertions corresponding to the JML specifications could not be implemented due to time constraints. Therefore, runtime checking is not available as a feature to the user. However, as mentioned, a basis was put in place in the code, allowing for runtime checking to be implemented in the future.

6.3.2 Private repository

It is possible to host one's own private plugin repository for IntelliJ for distributing plugins. This means that it is possible to avoid publishing the plugin in IntelliJ's marketplace while still letting users receive automatic updates. The installation process is a bit more involved than just downloading from the marketplace, but mostly consists of adding an extra repository source from the user side.

As mentioned before, this could not be implemented due to time constraints. Instead, students can install the plugin by downloading a .jar file and selecting it in IntelliJ's plugin menu. It can be concluded from the usability testing that installing the plugin in this fashion is sufficiently user-friendly. In appendix A, a guide can be found on how to install the plugin in this fashion.

6.4 JML syntax supported by the plugin

In this section the JML syntax that the plugin ended up supporting will be described. These were divided according to the Moscow rules. For explanations about what the JML syntax represents, see appendix B.

6.4.1 Must requirements

The plugin supports all of the "must" requirements of the proposal, which are all the JML syntax that was supported in Software Systems in 2018. These consist of the following:

- requires clauses
- ensures clauses
- class invariants
- modifiers `pure` and `nullable`
- `\result` and `\old()`
- quantified expressions using `\forall` and `\exists`
- operators `==>` and `<==>`

6.4.2 Should requirements

All "should" requirements are supported as well. These consist of level 0 features (and one level 1 feature) which seemed useful to support, namely:

- `\typeof()`, `\elementype()`, and `\type()`
- `\nonnullelements()`
- `\not_specified`
- `assignable` clauses (and its synonyms `modifiable` and `modifies`)
- `signals` and `signals_only` clauses
- loop invariants (and its synonym `maintaining`)
- quantified expressions using `\max`, `\min`, `\sum`, `product`, and `num_of`. These are a level 1 feature.
- modifiers `spec_public` and `spec_protected`
- operators `<==` and `<!=>`

- JML comments of the form `(* comment *)`

A note should be made that for assignable clauses, variable names with the suffix `[*]` and `.*` are supported, but the suffixes `[0..x]` and `[x]` are not supported. These are not supported because `x` can be a Java expression and it is hard to detect where those expressions stop in the parser. However, when a user does try to use the unsupported suffixes, and error is shown that those are unsupported.

6.4.3 Could requirements

Most "could" requirements are supported. These can already be expressed using the "must" and "should" requirements, but these make it syntactically simpler. The supported "could" requirements are:

- `\nothing` and `\everything` in assignable clauses
- the modifiers `helper` and `instance`
- the keyword `also`
- `\TYPE`
- `assume` and `assert` statements

For the `assert` and `assume` statements, only the syntax `assert/assume expression ;` is supported and not `assert/assume expression : expression ;`. The latter is not supported because it is hard for the parser to distinguish between the first and second expression, as `:` could be used inside the expression as well.

6.4.4 Could requirements that are not supported

There are two JML features mentioned in the "could" section of the proposal that are not supported by the plugin.

Data groups Data groups are not supported, because when the JML reference manual was read more precisely at a later point, it became clear that data groups can only be used with model fields. And it had been already decided to not support model fields, as those add a lot of new syntax, so that is why it was decided to not support data groups.

Sub-type operator The operator "<:" is also not supported, because this was one of the last items on the to-do list. And, unfortunately, a lot of time was lost on more crucial features, so there was no time left to still implement support for this operator.

7 Testing

This section will describe what components of the plugin were tested, how they were tested, and the results of this testing.

7.1 Unit tests

Unit tests were written to check that correct parse trees are generated, that all the error checkers return correct error messages, and that the code completion provides correct suggestions. A tutorial in IntelliJ's documentation [11] was followed to write these tests. All the unit tests were found to pass.

7.1.1 Parse tree tests

This test suite checks whether correct parse trees are created. Both correct syntax and wrong syntax are tested. The syntax that is tested includes specification clauses such as requires, ensures, assert, loop invariants, etc.; quantifiers; unfinished expressions; missing parentheses or parentheses in the wrong spot; etc. In total, there are 29 of these tests.

7.1.2 Error checker tests

The error checker tests are categorised the same way as the error checking in section 6.1.3, namely syntax, semantic, and type checkers. The test data consists of Java classes containing JML comments, which contain special tags that describe where a certain error message should be in the file. IntelliJ then runs the file and checks that the error messages that are shown correspond to the expected tags.

Syntax annotator tests This is a test suite that tests that the correct error message is shown at the appropriate position when there is a syntax error. In short, all custom error messages that are mentioned in section 6.1.4 are tested. Also, all possible syntax errors for quantified expressions (as those consist of multiple parts) are tested. The placement of error messages when an expression or semicolon is missing is tested as well. Additionally, the default messages, that are given when the syntax is wrong in general, were checked for inconsistencies. In total there are 11 tests, where each

test contains multiple combinations of correct and wrong syntax of the same category.

Semantics annotator tests This test suite contains tests for each check in section 6.1.4, again validating that correct error messages are shown at the correct positions. For each test, both correct semantics and wrong semantics were tested. The most extensive tests are the ones where specifications or modifiers are in the wrong position in the Java file. All possible positions were tested for those, for example, a method specification above a method, above a class, above a field, inside a method, etc. This was done because the position of the JML comment is mentioned in the error message, so the correct position needs to be displayed. The checks for duplicate modifiers and redundant use of `\not_specified` were also tested. In total there are 14 tests, where each test contains multiple combinations of semantic errors of the same category.

Type annotator tests This test suite contains tests for each check in section 6.1.4, again validating that correct error messages are shown at the correct positions. Quantified expressions are extensively tested, as those consist of multiple parts. Assignable clauses are also tested extensively, as it has multiple checks, such as whether the reference exists, that the reference is not final, and whether using `.*` or `[*]` with that reference is allowed. For embedded Java expressions, extensive testing was done for everything that is type checked by the plugin. This consists of testing whether references can be resolved, creating new instances of classes and arrays, boolean expressions, method calls, use of `instanceof` and casting, literals, parenthesised expressions and unary expressions. In total there are 22 tests, where each test contains multiple combinations of types errors or other Java errors of the same category.

7.1.3 Code completion tests

The code completion test suite tests for common code completion patterns as well as context sensitive patterns. The tests check that expected code completions are present, but most tests do not check whether incorrect completions are also given, with the exception of context-aware code completions (e.g. loop invariants can only be used inside a method), for which the tests do

check whether there are incorrect completion suggestions. Most code completions do not check for the presence of incorrect suggestions due to the sheer amount of suggestions that can be given in certain contexts. Enumerating these is simply not feasible, for example, when a class name reference needs to be completed. In total there are 55 tests for code completion.

7.2 Integration tests

Integration testing is the act of testing multiple components of the system as a group. As mentioned above, there are separate unit test suites for the syntax, semantics, and type checkers. For example, the type checker tests will contain both valid and invalid Java expressions, but no invalid syntax. However, when students use the plugin, they will probably encounter a mix of wrong syntax, wrong semantics, wrong Java expressions, and wrong types. Thus, it is useful to test all of those checkers together. These tests were automated and consisted of two classes annotated with JML that contained combinations of mistakes of different categories. Parsing, syntax highlighting, and code completion are separate entities, so they were not tested with any integration tests, as there is nothing to integrate them with. Those were however tested in the system tests.

7.3 Test coverage

Since the unit and integration tests were automated, coverage data of those tests could be collected. Table 1 lists the results.

Component	Class Coverage	Method Coverage	Line Coverage
Lexer Extension	71%	100%	98%
Parser Extension	100%	95%	90%
Error Checkers	100%	98%	91%
Code Completion	86%	85%	82%
Project Total	82%	72%	80%

Table 1: Test code coverage results

7.4 System tests

The full system test was written as a document that contains individual tests in text form that can be executed one by one, analogous to JUnit tests. It focuses on the components that are not included in the unit or integration tests, namely installation, syntax highlighting under all conditions, and performance. The document can be found in appendix D. All system tests were last performed on the 15th of April 2021 and were confirmed to have passed.

7.5 Usability tests

7.5.1 Setup of the tests

Two Software Systems TAs that had prior experience with JML and two TAs without prior experience participated in usability testing. A few days before the actual testing took place, some of the participants were sent a link to a lecture about JML from 2018 by Luís Ferreira Pires [12] and the slides that belong to that lecture, so they could learn about or get a refresher on JML. The extra JML information was not sent to the TA with the most JML experience as it was thought they did not need a refresher, and was accidentally not sent to one TA without experience. At the start of the meeting, each participant was provided with a document containing a list of all JML syntax supported by the plugin, a preliminary version of appendix B.

During the test, participants were asked to install the plugin and were provided with a copy of a sample project for which they were asked to write JML specifications. Participants were asked to share their screen, so it could be observed how they interacted with the plugin. A lot of small bugs were noticed, written down and fixed before the next usability test where possible. Remarks made by the participants whilst writing the JML were also noted down. Sometimes the participants got stuck and either did not know what JML to write, or had forgotten or did not know the syntax. In those cases, some hints were given. This should not be taken as a sign that the plugin was not helpful, as Software Systems students will have fresh JML knowledge, while three out of the four participants did not. After this, a small interview was conducted with the participants and the following questions were asked:

- How much experience do you have with JML?
- What do you think of the plugin?

- If you could change one thing, what would it be?
- Do you think that the plugin would help the students in Software Systems?
- What do you think of the code completion?
- Any other comments?

7.5.2 Summary of the test results

Here a summary is given of the results of usability testing. For the raw results, see appendix E. The first usability test did not go too well, as this participant had no prior experience with JML, and they had not received information on JML beforehand due to a clerical error. However, the participant could still give some feedback. They were most annoyed with the (then broken) code completion, but found the error messages useful. The second participant was also inexperienced with JML, but had received the information about JML beforehand, which meant the test was more successful. This participant thought the installation of the plugin was simple, and they liked the syntax highlighting. They did make the remark that the auto-indentation inside JML needed to be fixed. Sometimes the code completion still did not work. The third usability test was with the most experienced participant and they gave the most feedback. They remarked that it felt like an IntelliJ plugin and that, besides the code completion, it felt fast. They noticed some minor bugs but did not feel that those bugs were disruptive. They also liked the colour palette of the syntax highlighting. They did suggest add some code snippets for commonly used JML. They also suggested to make some error messages more clear, but did think the error messages were more clear than OpenJML's error messages. The last participant had some prior experience with JML, but not a lot. They indicated that adding quick fixes would be nice and that the code completion pop-ups were not aggressive enough. They did however think that the plugin would be helpful to Software Systems students.

Overall, these usability tests revealed a lot of previously unnoticed bugs. Most of these bugs could be fixed between the usability tests. A lot of usability issues were also pointed out by the participants. Each usability test was an improvement on the previous usability test with regard to the opinion of the participants of the plugin. For that reason, these usability tests can be considered very useful for the development of the plugin.

8 Organisation, sprints and meetings

This section describes the organisation of the project, what was planned for each sprint and what was really done and the meetings with the supervisors.

8.1 Organisational procedures

The project was executed using scrum principles [13]. Most sprints were two weeks long, except the first sprint which was one week long. The original plan was to conduct a short meeting every weekday morning to discuss progress and plan for the day. However, this was changed to being in a conference call the entire day for better collaboration between the project group members. A Trello board was used to keep track of tasks and task assignments. Each Monday morning after a sprint the Trello board was updated to reflect the state of the project. A GitLab repository hosted on the SNT network was used for version control.

8.2 Original planning

The original planning consisted of the following:

- Sprint 1 (1 week): Figuring out different aspects of the project implementation.
- Sprint 2 (2 weeks): Get syntax checking and syntax highlighting working and write requirements specification and test plan.
- Sprint 3 (2 weeks): Get code completion working. Write preliminary version of design report. Write integration and system tests. Do user testing. See if runtime checking is feasible to do in time. Individually, work on reflection take-home exam.
- Sprint 4 (± 2 weeks): Finalise design report and finalise implementation. Make user manual, presentation and poster. Give presentation at the chair of the supervisors.

8.3 Actual activities

Activities were tracked during each sprint using a Trello board with task tickets. The activities list is based on a summary of these tickets.

- Sprint 1 (1 week):
 - Steven: IntelliJ API research, embedding JML into Java.
 - Erikas: IntelliJ API research, plugin infrastructure setup, basic syntax highlighting.
 - Ellen: JML research, writing BNF.
 - <all>: Writing the project proposal.
- Sprint 2 (2 weeks):
 - Steven: Attempting to embed Java into JML, runtime checking research.
 - Erikas: Syntax highlighting for Java expressions, plugin options menu, basic code completion, researching embedding Java into JML, various small cosmetic changes.
 - Ellen: Parser unit testing, error checkers for syntax and semantic checking, some tests for those error checkers, colour theme for syntax highlighting, started on type checking.
 - <all>: Writing requirements specification and test plan.
- Sprint 3 (2 weeks):
 - Steven: Runtime checking, fixing version compatibility, removing green background in syntax highlighting.
 - Erikas: Embedding Java into JML, Java code completion, visibility checking for code completion.
 - Ellen: Custom parts of parser and lexer, type checking and a visitor for Java expressions, more error checking testing, bug-fixing in the error checkers, mock project for usability testing, writing supported JML document.
 - <all>: Usability testing.
- Sprint 4 (± 2 weeks):

- Steven: Runtime checking, system testing.
- Erikas: Code completion unit tests, code completion bug-fixing.
- Ellen: Bug-fixing error checkers and custom part of the parser, finished unit testing, finished integration testing, made project poster.
- <all>: Writing the project report, cleaning up the source code, presentation at the chair of the supervisors.

8.4 Project meetings

Every Thursday morning a meeting was held with the project supervisors, who were also the project clients. The topics discussed in each of the meetings is given below.

2021-02-04 Project requirements, alternatives to previous tooling, IDE selection, future plans of Software Systems module, development priorities.

2021-02-11 Plugin requirements, runtime and static checking, maintainability requirements, project proposal, need for a custom plugin, Java version compatibility.

2021-02-18 Project proposal, JML grammar, messages for unsupported features, code maintainability.

2021-03-04 Project infrastructure, Java embedding issues, private plugin repository.

2021-03-11 Code injection, user testing recruitment, Java embedding issues, maintainers guide.

2021-03-18 Requirements analysis edits, JML specification ambiguity.

2021-03-25 Code submission, method calls in JML expressions.

2021-04-01 Risk analysis, design report source code, design report structure.

2021-04-08 Design report structure, poster feedback.

2021-04-15 Final project delivery details.

9 Reflection

Overall the project went pretty well, though there were some challenges. There was a lot of time spent on researching how to implement certain features. There were also problems with figuring out how to parse and check embedded Java expressions. Later there were problems with the testing framework. It meant that testing and writing the report had to be moved to later weeks, so the project fell behind the planning. Luckily, everything was still finished on time, albeit with some work outside the usual working hours.

9.1 Challenges

9.1.1 Starting from scratch

The first challenge was that there were no already existing frameworks for JML that could be extended to meet the requirements of the clients. There was some exploratory research done to find out whether OpenJML could be extended to support more recent Java versions and IntelliJ, but this turned out not to be feasible. Therefore, JML support had to be built from scratch.

In the last week of the project it was discovered that there exists an open source JML plugin for Eclipse called KeY [14]. While not quite meeting all the project requirement, it could have been helpful to look at the source code of KeY and figure out how certain problems were solved by its developers. That might have saved some time. Additionally, it might have been possible to use some of their CLI tools to provide more functionality to the plugin, like static checking. But this was discovered far too late in the project to be implemented.

9.1.2 Lack of documentation

One of the major issues encountered was that the vast majority of the IntelliJ API was completely undocumented and that the documentation that was available only covered the very basics of plugin development. Additionally, there were very few resources to help with the development of more complex plugins.

This meant that a lot of research was required into how the IntelliJ API functions and how other plugin developers applied parts of it to enable the functionality that their plugin provides. This meant that large amounts of

time were spent going through other plugins' source code to understand how they solved certain problems. This was extremely time consuming and it can be estimated that about 30-50% of time spent on the project went to researching the API and how to apply it to enable certain functionality in the IDE.

9.1.3 Parsing and type checking of Java expressions

Tying into the problem of lack of documentation, there were issues trying to find out how to implement type checking on Java expressions inside JML. This caused some frustrations, so the problem was passed around between the group members. In the end, it took the cooperation of two group members to find a solution. However, this caused a two-week delay in its implementation, which meant that the implementation of the project fell behind on schedule.

9.1.4 Troubles with the testing framework

There were also some issues regarding the setup of the tests. First of all, there was an issue where classes from the Java standard library could not be resolved if used inside the test data. After some time, one of the authors came across a forum post [15] that mentioned that one should clone the GitHub repository of the IDE source code and point towards that in the Gradle build settings. This fixed this issue, but by that time several hours had already been lost. A few days after this, a new issue occurred. Up to that point, all test suites had been run separately from each other. When they were ran together, the tests showed different results based on the order in which the test suites were ran. It is suspected that somehow test suites were not properly terminated before the next one was run. However, it was not possible to determine why this was the case. Therefore, a work-around was needed. Luckily a solution could be found in the Gradle documentation, namely creating a new instance for every test suite. However, again, several hours were lost before this solution was found. The loss of this time was quite impactful because it happened towards the end of the project, where a lot of time was already lost to the Java embedding issue.

9.1.5 Implementing code completion

Having to re-implement code completion for Java embedded in JML was unfortunate. The functionality provided by IntelliJ is substantial and im-

possible to replicate to the same degree in the duration of a design project. Therefore, functionality is limited in some places. For example, loop variables are not recognised by the completion service, not all JML keywords can have their types properly resolved in the completion service.

However, there were a couple of lucky breaks when it comes to using parts of the completion API. Visibility and accessibility checking was sufficiently separate from the rest of the completion logic that it could be reused. The API also provided a way of querying classes existing in the project that allowed for relatively simple static class reference completion.

There are definite places for improvement, especially when it comes to local variable referencing and better support for mixed Java-JML expressions, but overall the code completion functions well enough to be useful in the vast majority of cases.

9.2 Group dynamics

The group dynamics were perceived as enjoyable and relatively productive. As mentioned in section 8.1, all group members were in a conference call all day, which made it easy to ask each other for help, which proved to be beneficial. However, this also meant that sometimes group members would end up having banter about unrelated subjects, usually near the end of the day, which was considered very enjoyable, but not very productive. That might be a point for improvement, though the counterargument can be made that it is normal to not always be productive, and it did keep the group members in good spirits.

9.3 Usability of the project for Software Systems

Overall, the plugin is definitely usable and should prove helpful to students when writing their JML specifications. The code completion is a bit wonky, as it lacks local variable (not field variables) resolution in places and does not provide complete support for mixed JML-Java expressions. However, it covers all the basic completion cases and a fair number of more complicated completion cases so it should prove quite helpful. The code completion does not always give context appropriate completion suggestions but in those instances that it does - error checking should be able to catch the mistake and highlight it to the student. The error messages were made as clear as possible, so those should prove to be helpful to students, especially those that

are just starting out with programming.

10 Future

The intention is that the plugin will be used as a tool in the Software Systems module in the upcoming academic year and potentially the years after that. This goal stretches beyond the reach of the Design Project module and does not necessarily involve the members of the current project group. As such, only a general overview of events that may take place in the future is given here.

10.1 Updating the module guide

Software Systems makes use of an extensive module guide, containing most of the module's practical information for students, as well as exercises for the course material. This module guide will be updated for the next academic year by a team of teaching assistants during the summer holiday. As a result of the existence of this plugin, JML can be taught again. This means that JML exercises will most likely be added to the module guide again. Also, the guide on how to install the plugin included in this report (see appendix A) will probably be included, as well as the explanation of the JML supported by the plugin (see appendix B).

10.2 Use of JML in Software Systems

Following the inclusion in the module guide, JML will be part of the curriculum of Software Systems again. It can be expected that a lecture about the basics of JML will be included in the module. Since JML has not been included in the module in the years 2019-2020 and 2020-2021 according to the module coordinator, it is not unlikely that there will be a knowledge gap for newer teaching assistants. However, the coordinator believes that those TAs can catch up, and that this will not be a significant issue.

10.3 Maintenance

Although the plugin has been designed with forward compatibility in mind, it is likely that a person proficient in Java will need to be appointed as a maintainer to perform minor adjustments for full compatibility with newer versions of Java and IntelliJ, as well as to address potential bugs that were

not discovered or could not be fixed within the project's time limit. To help the maintainer, a guide is included with this document (see appendix C).

10.4 Future work

The plugin contains a sufficient amount of features to be used in Software Systems. However, as explained in the features (see section 6), several features were not included due to time constraints. With more investment, such as another design project or a master's thesis, the plugin could be made to include all JML features defined in the reference manual [3], as well as fully featured runtime checking, static checking, and intelligent code prediction. A more feature complete version of the plugin would be suitable for public release and have uses beyond undergraduate courses in the field of formal verification. One possible extension lead might be the KeY project, it appears to have a compatible static verification CLI back-end that could be used to provide the static analysis functionality for the plugin.

References

1. Leavens, G. T. *The Java Modeling Language (JML)* December 2017. <https://www.cs.ucf.edu/~leavens/JML/index.shtml>.
2. David, C. *OpenJML* 2018. <https://www.openjml.org/>.
3. Leavens, G. T. *et al. JML Reference Manual* DRAFT, Revision 2344. 31st May 2013. <http://www.jmlspecs.org/refman/jmlrefman.pdf>.
4. ANTLR Project. *ANTLR v4 - Plugins: JetBrains* 22nd January 2021. <https://plugins.jetbrains.com/plugin/7358-antlr-v4/>.
5. JetBrains s.r.o. *Custom Language Support* 14th January 2021. <https://plugins.jetbrains.com/docs/intellij/custom-language-support.html>.
6. JetBrains s.r.o. *IntelliJ Platform Plugin Template FAQ* 29th March 2021. <https://github.com/JetBrains/intellij-platform-plugin-template#faq>.
7. JetBrains s.r.o. *Grammar-Kit* 1st January 2021. <https://github.com/JetBrains/Grammar-Kit>.
8. Armstrong, A. *et al. PsiViewer* 9th March 2021. <https://plugins.jetbrains.com/plugin/227-psiviewer>.
9. JetBrains s.r.o. *Custom Language Support Tutorial* 14th January 2021. <https://plugins.jetbrains.com/docs/intellij/custom-language-support-tutorial.html>.
10. Agile Business Consortium. *Chapter 10: MoSCoW Prioritisation* Accessed at 15th April 2021. https://www.agilebusiness.org/page/ProjectFramework_10_MoSCoWPrioritisation.
11. JetBrains s.r.o. *4. Annotator Test* 14th January 2021. <https://plugins.jetbrains.com/docs/intellij/annotator-test.html>.
12. Ferreira Pires, L. *20-11 - Programming - [18-19 K2]* 20th November 2018. <https://vimeo.com/groups/555404/videos/301818114>.
13. Scrum.org. *What is Scrum?* Accessed at 16th April 2021. <https://www.scrum.org/resources/what-is-scrum>.
14. The KeY Project. *Program Verification* Accessed at 14th April 2021. <https://www.key-project.org/applications/program-verification/>.

15. IDEs Support (IntelliJ Platform) - JetBrains. *Cannot resolve symbol with official example code* 4th August 2020. <https://intellij-support.jetbrains.com/hc/en-us/community/posts/360009436599-Cannot-resolve-symbol-with-official-example-code>.
16. Jemerov, D. *IntelliJ SDK: publish javadoc online* 22nd February 2019. <https://youtrack.jetbrains.com/issue/IJSDK-19>.
17. JetBrains s.r.o. *IntelliJ Platform SDK* 15th March 2021. <https://plugins.jetbrains.com/docs/intellij/welcome.html>.
18. Ellis, M. *IntelliJ project migrates to Java 11* 18th September 2020. <https://blog.jetbrains.com/platform/2020/09/intellij-project-migrates-to-java-11/>.

Appendix A User manual

This manual explains how to install the IntelliJML plugin.

Step 1 - Download the plugin Download the plugin and place it somewhere where it can be found again later.

Step 2 - Go to IntelliJ settings Open IntelliJ. In the top left of the window, click on **File**. In the drop-down menu, click on **Settings...**

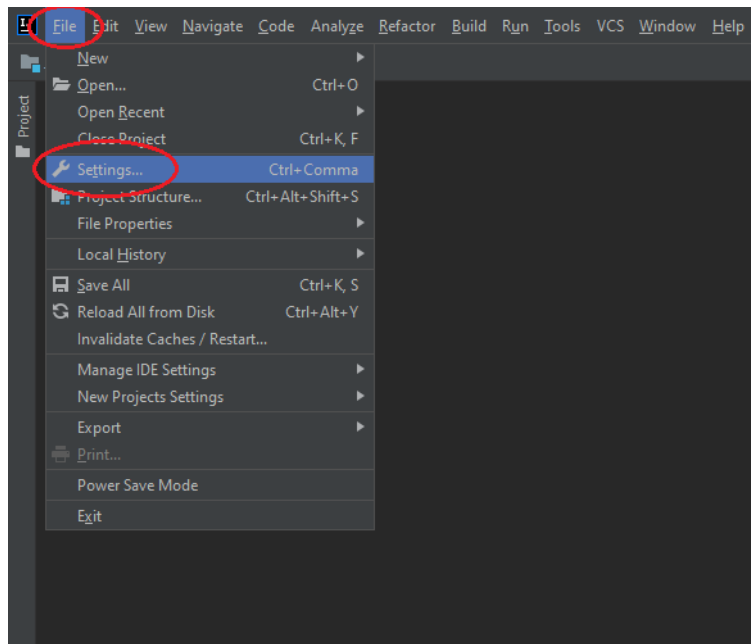


Figure 3: Finding the settings

Step 3 - Go to plugin settings In the left panel in the settings window click on Plugins. When all plugins are visible, click on the cogwheel on the top-right next to Marketplace and Installed (see the red circle in figure 4).

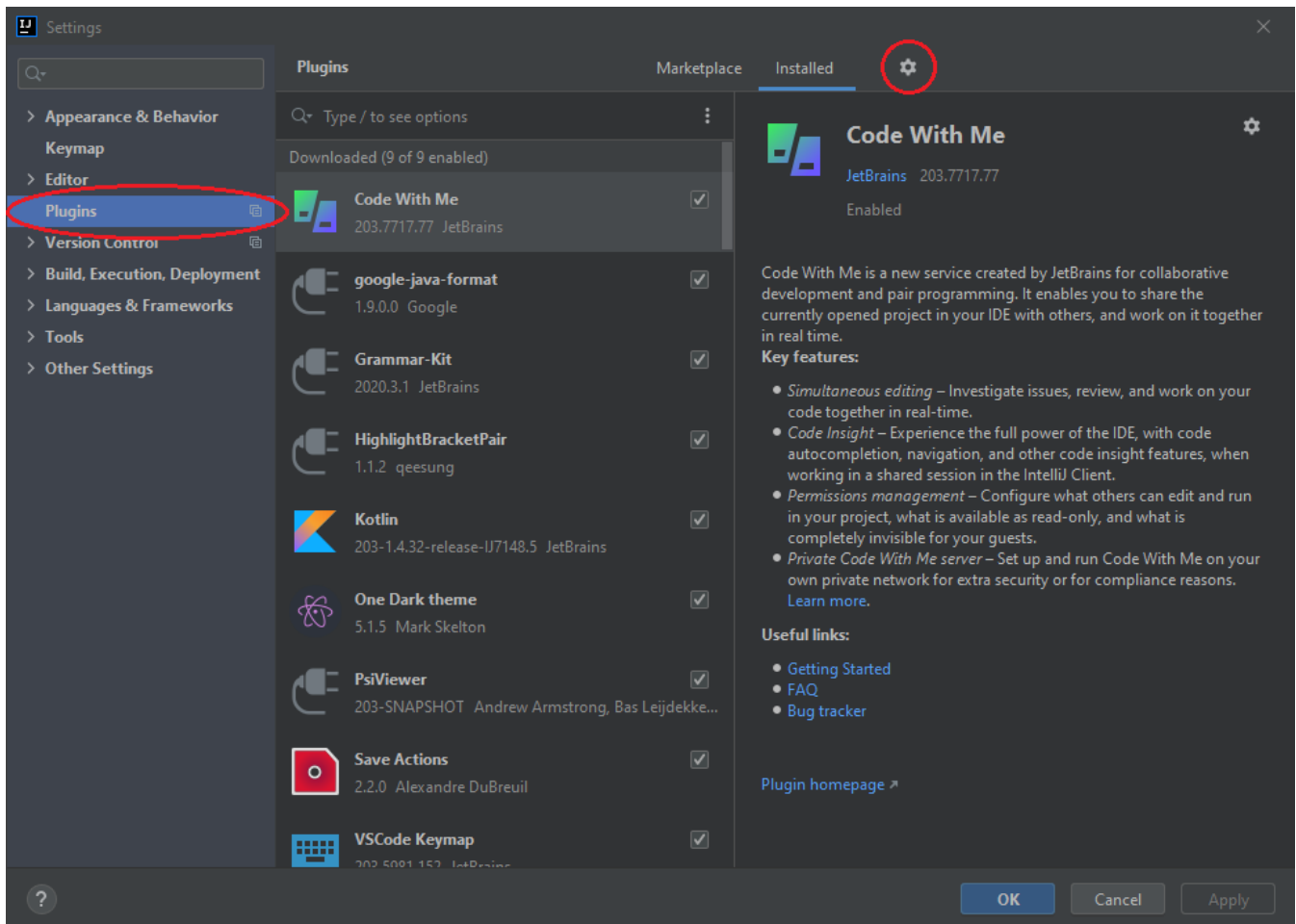


Figure 4: Finding plugin settings

Step 4 - Installing the plugin After clicking on the cogwheel, select **Install plugin from Disk...** This should open a window with a directory tree. Find the plugin file where you saved it beforehand and select it. Then click **OK** to install it.

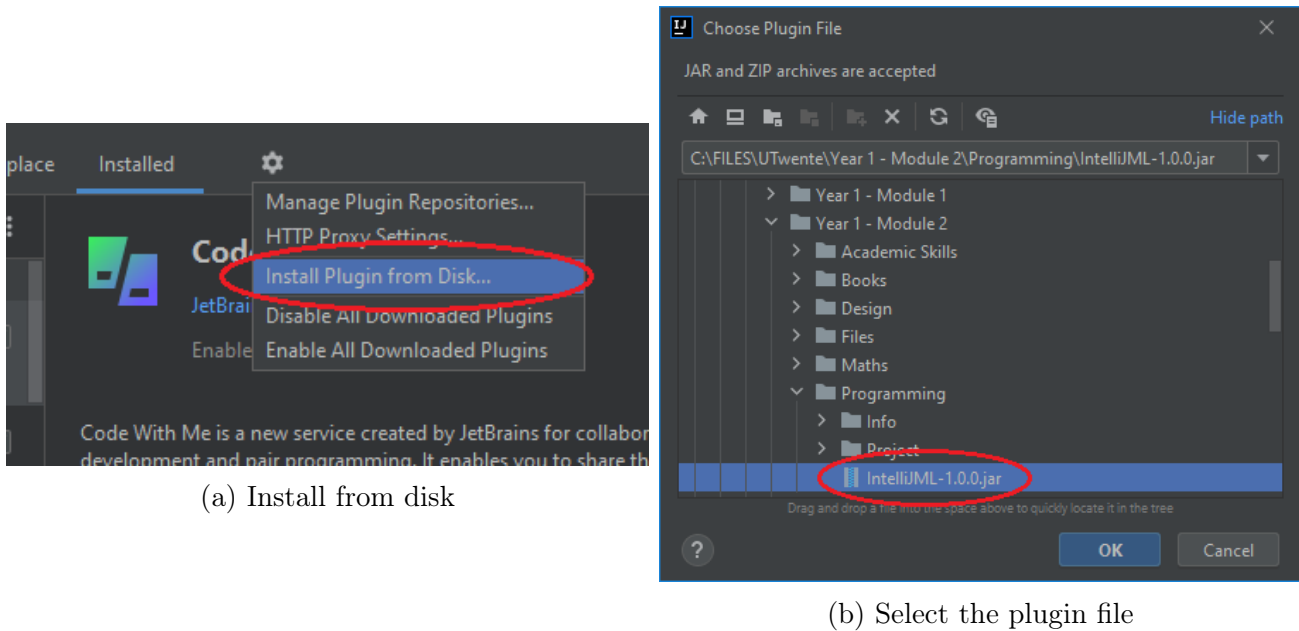


Figure 5: Installing the plugin

Step 5 - Applying changes In the bottom-right of the settings window, click on **Apply** to tell IntelliJ to save the plugin installation. Then click **OK** to close the settings window.

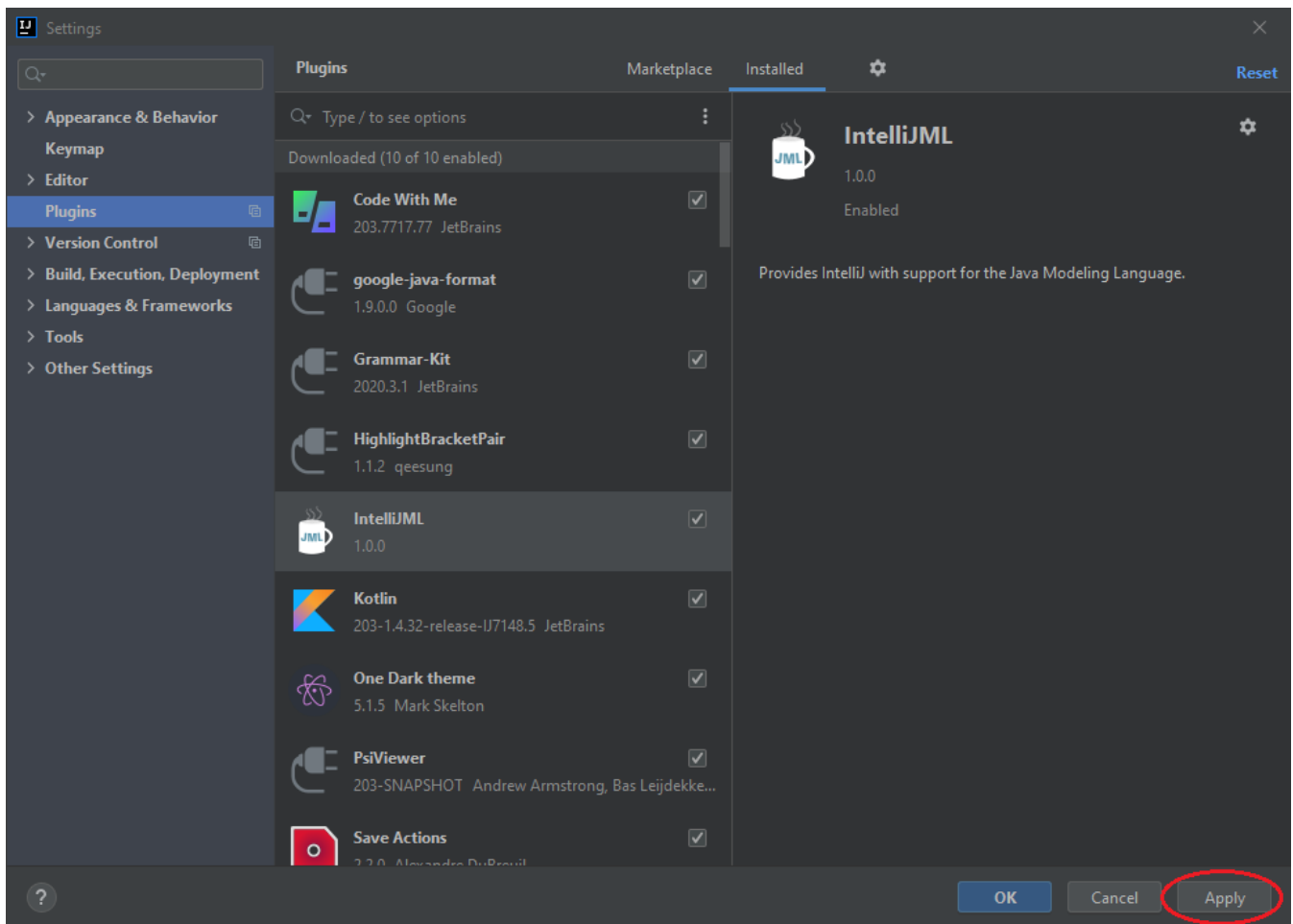
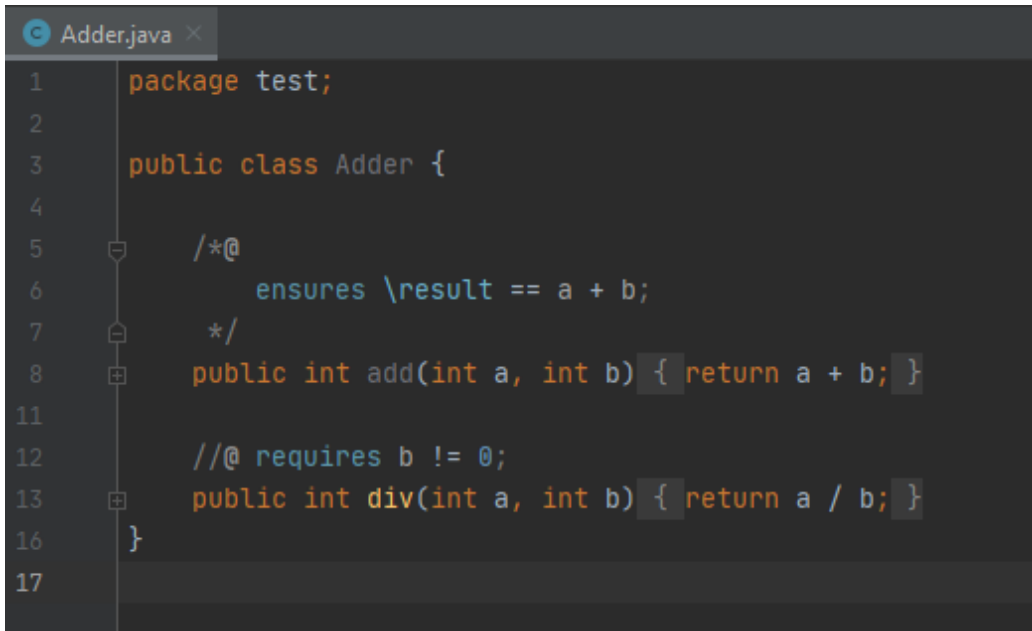


Figure 6: Applying changes

Step 6 - Writing JML Open a Java class and start writing JML by typing `/*@` and pressing the Enter key for a multi-line JML comment, or `//@` for a single-line JML comment. If the syntax of the JML is correct, keywords such as `requires` will be highlighted in blue, as can be seen in figure 7.



```
1 package test;
2
3 public class Adder {
4
5     /*@
6         ensures \result == a + b;
7     */
8     public int add(int a, int b) { return a + b; }
11
12     //@ requires b != 0;
13     public int div(int a, int b) { return a / b; }
16 }
17
```

Figure 7: Writing JML

Appendix B JML Reference

This appendix contains explanations of the JML expressions and keywords that are supported by the IntelliJML plugin.

B.1 Method specifications

The following list contains all supported JML specification clauses used above methods.

- **requires** / **pre** *expression*;
Represents a basic pre-condition, meaning that *expression* must be true at the start of the method body.
- **ensures** / **post** *expression*;
Represents a basic post-condition, meaning that *expression* must be true when the method returns.
- **signals**(*exception-name identifier*) *expression*;
Specifies that *exception-name* can be thrown by the method. The identifier is optional and is used to reference the instance of the exception. The identifier can be used in the (optional) expression, which must be true when the exception is thrown.
- **signals_only** *list-of-exception-names*;
Specifies what exceptions may be thrown by the method. *list-of-exception-names* is a comma-separated list of exception class names, such as `ArithmeticException`.
- **assignable** / **modifiable** / **modifies** *list-of-fields*;
Only the fields listed after one of these keywords may be assigned during the execution of the method. *list-of-fields* is a comma separated list of field names. `arrRef[*]`, `classRef.*`, and `ClassName.*` can be used to add multiple fields to the list at once. `arrRef[*]`, where `arrRef` is a reference to an array, means that all elements of that array can be assigned. `classRef.*` means that all visible static and instance fields of the class referenced by `classRef` can be assigned. `ClassName.*` means that all visible static fields of the class with the name `ClassName` can be assigned.

The keyword `also` can be used between clauses in a method specification. This keyword is usually only for cosmetic purposes.

For `requires`, `ensures`, and `signals` clauses, *expression* must be a Java expression of a boolean type. In these expressions, some JML keywords can also be used. The list below contains the JML keywords that can only be used in *expression* in method specifications. To learn about what other JML keywords and functions can be used in *expression*, see section B.5.

- `\result`: Holds the return value of the method. Can only be used in an `ensures` clause above a method that returns a value.
- `\not_specified`: Can be used as a replacement for *expression*. Mostly useful as a placeholder.
- `\nothing`: Can be used in an `assignable` clause instead of a list of fields. Indicates that no fields from any class can be assigned by the method.
- `\everything`: Can be used in an `assignable` clause instead of a list of fields. Indicates that all fields from all classes can be assigned by the method.

B.2 Class invariants

Class invariants and the modifiers that belong to them (section B.4) are supported. Class invariants are conditions that must be true at the start and end of every method call, with the exception of methods that are marked as `helper`. Class invariants have the following syntax:

- *modifiers* `invariant` *expression*;

For class invariants, it is important that the visibility of the methods and/or fields used inside the expression is the same as the visibility of the class invariant itself. Consider a field with `private` visibility and a class invariant with `public` visibility that uses the field. If one looks at the class with `public` visibility goggles, the class invariant can be seen, but the field cannot, making it unclear what field is being referenced by the invariant. In the opposite case, where the field has `public` visibility and the class invariant has `private` visibility, looking at the class with `public` visibility goggles means that the field can be seen, but the invariant that specifies it cannot.

The visibility of a class invariant can be changed by putting a visibility modifier in front of the keyword `invariant`, as usually done in Java. For example, `//@ public invariant x > 0;` gives the class invariant `public` visibility. This implies that `x` also needs to have `public` visibility.

In some situations, it is undesirable to change the visibility of the invariant or to change the visibility of a field in the code itself. If this is the case, the field can be made visible only to JML specifications by using the keyword `spec_protected` for `protected` visibility and `spec_public` for `public` visibility. `spec_protected` and `spec_public` must be placed inside a JML comment.

B.3 In-method specifications

There exist several JML specification clauses that can be used inside methods.

- `maintaining / loop_invariant expression;`
Must be placed above a loop and indicates that *expression* must be true at the start of each iteration of the loop.
- `assume / assert expression;`
Works the same as Java's `assert` statements. *expression* must be true when the location of the specification is reached.

B.4 Modifiers

Several modifiers are supported, which can be seen in the list below. These modifiers can be used as JML comments by themselves, and can also be used inside certain specifications.

- `spec_public`: Gives a field, method, or class `public` visibility for JML specifications.
- `spec_protected`: Gives a field, method, or class `protected` visibility for JML specifications.
- `pure`: Can be used above methods to indicate that the method does not change the state of the program.
- `instance`: Indicates that a field or class invariant is not static.

- **helper**: Can be used above a method that is either `pure` or `private`, or a `private` constructor. Indicates that class invariants do not hold at the start and/or end of the method.
- **nullable**: Indicates that a field, local variable, or parameter can be null, or that a method can return null.
- **public**, **private**, **protected**: Changes the visibility of a class invariant.
- **static**: Indicates that a class invariant is static.

B.5 Expressions inside specifications

Most specifications require an expression. For the purpose of JML, an expression is a Java expression, with several additions that can only be used in JML specifications. These additions can be found in the next sections.

B.5.1 Quantified expressions

Quantified expressions have the following syntax:

- $(\backslash\textit{keyword variable-declarations ; range-condition ; body-expression})$

The parentheses are mandatory. *variable-declarations* must declare but not initialize the variable(s). That means `int i` is correct, `int i = 0` is not. *range-condition* is optional and indicates the range that the declared variable(s) in *variable-declarations* have. It must return true when the variable is in the range. For quantified expressions with the keywords `\max`, `\min`, `\product` or `\sum`, *body-expression* must return a numeric type. The result of the quantified expression itself will be of the same type. For the other keywords, *body-expression* must return a boolean type.

The list below contains the different kinds of quantified expressions that are supported.

- `\forall`: Tests that a predicate holds for all elements in the range (\forall in mathematics). Returns a `boolean`.
- `\exists`: Tests that a predicate holds for at least one element in the range (\exists in mathematics). Returns a `boolean`.

- `\max`: Returns the maximum value in the range. Returns a number.
- `\min`: Returns the minimum value in the range. Returns a number.
- `\product`: Returns the product of all the items in the range. Returns a number.
- `\sum`: Returns the sum of all the items in the range. Returns a number.
- `\num_of`: Returns the amount of items in the range that satisfy the expression in the body. Returns a long.

To give an example,

```
(\forall int i; 0 <= i && i < array.length; array[i] > 0)
```

means that for all integers `i` that are greater than or equal to zero and smaller than the length of `array`, `array[i]` must be greater than zero. In other words, all elements of the array must be positive.

B.5.2 Type-related expressions

Several expressions regarding the type of sub-expressions are supported.

- `\typeof(expression)`: Returns the most-specific dynamic type of the expression's value. The returned value is of type `java.lang.Class`.
- `\elemtype(expression)`: Returns the most-specific static type shared by all elements of the expression. The expression must be of type `java.lang.Class`. The returned value is also of type `java.lang.Class`.
- `\type(type-name)`: Creates a type from a type name, such as `int`, `String`, etc. The returned value is of type `java.lang.Class`.
- `\TYPE`: Equivalent to `java.lang.Class`.

To give an example, `\elemtype(int[].class) == \type(int)` is true, and so is `\elemtype(int[].class) == \typeof(0)`.

B.5.3 Operators

Several JML operators are supported that can be used inside specifications.

- `==>` Logical implication (if).
- `<==` Reverse logical implication.
- `<==>` Logical equivalence (if and only if).
- `<!=>` Logical inequivalence.

B.5.4 Miscellaneous

- `\old(expression)`: Represents the value of *expression* before the method is called or right before a certain location in the method is reached. Can be used in `ensures`, `signals`, `assert`, `assume`, `maintaining`, and `loop_invariant` statements.
- `\nonnullelements(expression)`: Indicates that an array and its elements are all non-null. The expression must be of an array type.

B.6 Comments

Comments can be placed inside JML specifications by using the following syntax:

- `(* this is a comment *)`

Appendix C Guide for the maintainer

This document is provided as a guide to anyone who needs to work with the source code of the IntelliJML plugin. It is intended to be a technical overview of the plugin's inner working, not an explanation of design choices. It is written in such a way that it is not necessary to read the project's full Design Report to understand this guide.

The reader is warned that the IntelliJ API is highly undocumented and therefore not pleasant to work with. The IntelliJ bug tracker states "Why haven't we done [Javadoc] yet: Because a) it creates the illusion that we have documentation" [16]. The limited documentation that is available can be found at [17].

C.1 Tools

Working with the IntelliJML source code requires the following tools:

- Java 8 or higher (set to language level 8 when higher).
- IntelliJ IDEA 2020.1 or higher. Attempting to work with the source code in any other IDE is highly discouraged and not covered by this document.
- The Gradle plugin for IntelliJ.
- The Grammar-Kit plugin for IntelliJ.
- Optional but highly recommended: The PsiViewer plugin for IntelliJ.

C.2 Version support

IntelliJML is written to be source-compatible with Java 8 and IntelliJ IDEA 2020.1. Unless there is a good reason to increment the minimum IDE version, it is recommended to keep it this way. The version of Java that the plugin is compiled with does not affect the plugin's support for language features, since these are processed through IntelliJ's API. Note that IntelliJ versions older than 2020.3 can be run with Java 8 [18], so incrementing the project's Java version also implies dropping support for these IDE versions. Support for 2019.3 and older is possible, but requires reworking the use of API methods

that were introduced in newer versions, which is unlikely to be worth the effort.

IntelliJ version support can be configured by setting `pluginSinceBuild` and `pluginUntilBuild` in `gradle.properties`. If the IDE version is not within this range, IntelliJ will refuse to load the plugin. Before changing these values, support should be verified using the Gradle task `runPluginVerifier`. This task checks for the use of deprecated and nonexistent classes and methods for all IDE versions listed at `pluginVerifierIdeVersions` in `gradle.properties`. Its output can be viewed at `build/reports/pluginVerifier`.

To package the plugin for distribution, simply run the `jar` Gradle task and copy the output from `build/libs`.

C.3 Data flow

In section 5.4, the data flow going from a string containing JML to each of the plugin's features was explained on a surface level. In Figure 8, the same data flow is shown more in-depth.

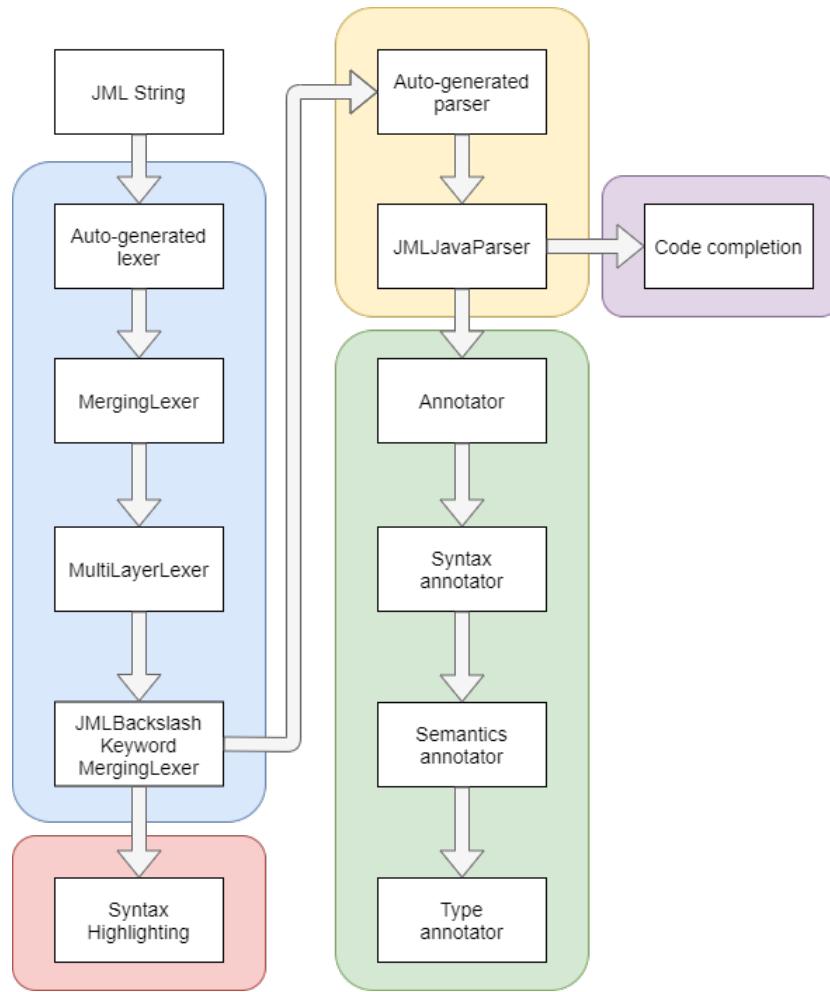


Figure 8: The flow of data through the plugin

C.4 Language embedding

C.4.1 Embedding JML into Java

The topmost level of the language embedding ensures that JML in Java comments is recognized as such by IntelliJ. To do this, `JMLMultiHostInjector` listens for comments in Java files. If one is found, and the first non-whitespace character after the start of the comment is an at-sign, the comment will be marked as JML, which causes IntelliJ to trigger the part of the plugin that processes JML strings.

This form of injection through the IntelliJ API automatically adds a green background to the embedded part, which in the case of this plugin means that all JML specifications have a green background. This is obviously undesirable, but IntelliJ does not provide an option to disable this. To fix this, the green background is manually overridden with one that has the same color as the normal background, effectively making it invisible.

C.4.2 Embedding Java into JML

Java was embedded back into JML (because JML is already being embedded into Java) by means of an extension of the generated JML parser and a multilayered lexer. This complicated approach was necessary due to limitations of the IntelliJ API that explicitly forbids embedding another language into an already embedded language. This essentially meant that a ground-up approach was needed.

First, the generated JML lexer is adapted to work with the IntelliJ API by means of a `FlexAdapter` class that envelopes the generated lexer (the generator names the class `_JMLLexer`) and adapts the interface to work with the API. This all happens in the `JMLMergingLexer` class that also contains the logic for collecting all the individual tokens that constitute a single mixed JML-Java section of JML and returning that entire section as a single token. This was needed because the generated JML lexer does not contain any Java tokenization, basically it returns nonsense in this section. To fix this, the single token is passed to the `JMLMultiLayerLexer`. This lexer dynamically alternates between the `JMLMergingLexer` for lexing the pure JML parts and `JavaLexer` to handle the Java parts (all of which is in the large combined token). This way, the single large token is broken up into a stream of mixed Java and JML tokens. The output of this lexer is then fed into another merging lexer, the `JMLInJavaExprMergingLexer`. This lexer merges/replaces tokens that were previously interpreted as Java into their appropriate JML tokens (JML implication operators and JML keywords with backslashes). Using this method, there is now a correct stream of JML-Java tokens.

Next, this token stream is passed to the JML parser. The generated parser handles the pure JML sections as specified in the BNF grammar, but the mixed JML-Java sections are handled by the custom extension to it. The custom parser extension is the `JMLJavaParser` class (technically this is called a parser util class, but it acts as a parser extension in our instance). This class takes care of building the actual parse tree for the mixed parts.

The parser extension is called by the generated parser when it reaches the `java_expr` node in the generated parser. In the BNF it can be seen that the `<<parseJavaExpression>>` function call is given instead of a regular rule for the `java_expr` node.

This is not an ideal solution, as the parser extension does not provide a proper parse tree for the Java expressions and just dumps the Java tokens under the appropriate JML node. This is, again, because forward compatibility with future Java versions is needed, so implementing custom parsing logic is not possible.

This unfortunately means that both the annotator and code completion services rely on string evaluation of Java expressions quite heavily in certain places, which is inefficient. If a proper Java parse tree could be received, this issue might be fixable.

It might be possible to retrieve a proper Java parse tree for the Java tokens. In the extension section of the parser, it is possible to dynamically switch to the IntelliJ Java parser and dynamically switch back to the calling parser. This was unsuccessfully briefly attempted, but the project was already behind schedule by several weeks because of this embedding issue. More time could not be put into investigating this possible solution, as the current solution allowed the project to move forward.

C.5 Lexing and parsing

C.5.1 Re-generating the lexer and parser

For the lexing and parsing, a grammar was written in BNF form, which can be found in `main/grammar/JML.bnf`. This grammar is used to auto-generate the lexer and the parser. To generate the lexer, right-click the `JML.bnf` file (the grammar-kit plugin in IntelliJ is needed for this), and click **Generate JFlex Lexer**. A window should open that allows choosing where to store it. Store it in `main/gen/nl/utwente/jmlplugin/parser` under the name `_JMLLexer.flex`. Then right-click that file, and click on **Run JFlex Generator**. Then, go back to the `JML.bnf` file, right-click it, and now select **Generate Parser code**. After that, navigate to the `psiToOverwrite` and copy the two files in there, and paste them in `main/gen/nl/utwente/jmlplugin/psi`.

C.5.2 Structure of the lexer

The lexer consists of 4 layers. The first layer is the auto-generated lexer, which converts raw strings to JML tokens. The second layer, the class `MergingLexer`, merges tokens that are part of a Java expression inside JML into one token. After that, `JMLMultiLayerLexer` takes the merged Java expression token and separates it into Java tokens. However, the Java expressions can contain JML keywords such as `\result`, which are not valid Java. For that reason, there is a fourth layer that turns those back into JML tokens (`JMLInJavaExprMergingLexer`). A quick demonstration of the lexers can be seen in Figure 9. The blue sections are JML tokens, the green section is the big Java expression token and the yellow sections are Java tokens.

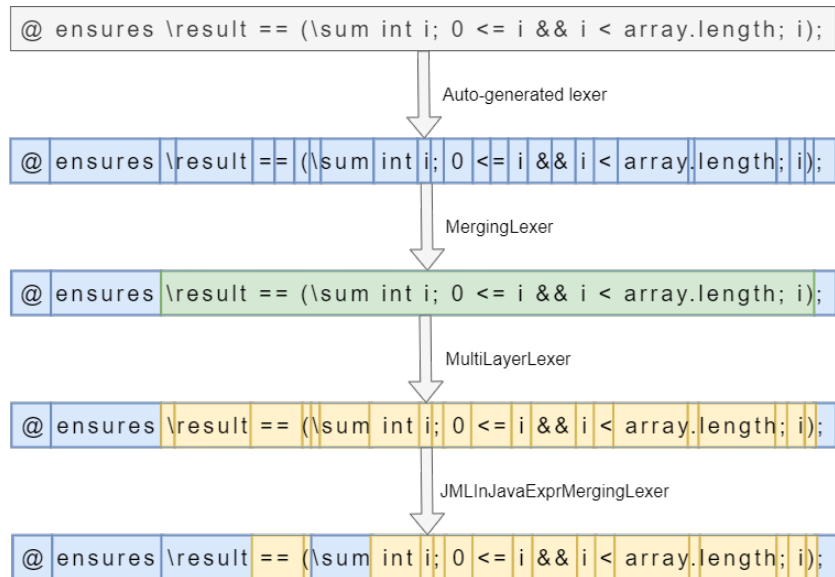


Figure 9: The flow of a JML comment through the lexers

C.5.3 Structure of the parser

There are two parsers. The first parser that is run is the auto-generated one. This parser does all the JML parsing, except for the Java expressions inside JML. For those, the auto-generated parser calls the manual parser, which can be found in `main/java/nl/utwente/jmlplugin/parser/JMLJavaParser`. This parser parses the Java tokens and JML expressions inside the Java expression, and creates a tree for them.

C.6 Annotation

Slightly misleadingly named, annotations refer to IntelliJ's warning and error messages, regardless of the nature of those messages. IntelliJML contains a multi-stage annotator (see `nl.utwente.jmlplugin.annotator`) that checks and provides warnings and errors for syntax, semantics, and typing, in that order. It is important to note is that the parser also provides error annotations, which are intercepted by the custom syntax annotator and adapted to be less confusing and more legible.

The annotator that is first called is the `JMLAnnotator`. In there, the `annotate` method is called for each element in the JML tree. It is checked that comment is an actual JML comment, and if it is, the syntax checker in `JMLSyntaxAnnotator` is called. If no syntax errors have been found, the semantic checker is called (`JMLSemanticsAnnotator`). If no semantic errors are found either, the type checker is called (`JMLTypeAnnotator`). Because this procedure is followed for each tree element, there can be both semantic and type errors in the same JML comment, but only if they are not on the same element in the tree. However, if the JML comment contains a syntax error, no tree elements at all are checked for semantic and type errors.

The type checker makes use of a few classes, of which the most important are mentioned here. First of all, a class called `ExtraVariables` holds a list of variables that should be accessible by the Java expressions inside JML, but cannot be resolved by the Java expression resolver. For example, the variables declared in JML quantified expressions, parameters of methods, initializers of for-loops, etc. are put in `ExtraVariables`. Another class worth mentioning is `RangeManager`. To understand why it is needed, a little more explanation is needed about the `JMLTypeAnnotator`. JML keywords can occur inside Java expressions. Because IntelliJ should create an actual expression from the JML Java expressions, those must be removed, as they are not valid Java. To achieve that, replacements are done on them. However, the replacements can be longer or shorter in amount of characters than the originals, causing the text ranges of errors to be incorrect. This means that errors are shown to the user at the wrong position. The class `RangeManager` is used to keep track of these changes, and to calculate the original position to be used in error messages.

C.7 Code completion

Code completion is provided both for JML keywords and inside Java expressions. The JML keyword implementation is relatively straightforward. For Java expressions, however, it is impossible to obtain a proper parse tree, which is required to be able to use IntelliJ's own Java code completion. This means that the Java code completion inside JML is mostly custom and therefore has subtle differences and possible bugs compared to regular Java code completion.

Some parts of IntelliJ completion were sufficiently generic to be reused. The class finder, responsible for finding all visible classes in the project, is reused from IntelliJ completion, as well as the sorting algorithm that IntelliJ uses for its suggestions.

The main class responsible for completions is the `JMLCompletionContributor` class that houses the `PsiTree` patterns on which completions are called. It also registers all completion providers that provide completion suggestions when a completion is called on the corresponding `PsiTree` pattern.

The `completionProvider` classes are responsible for generating completion suggestions. Each contributor is responsible for providing different suggestions to the result set that stores all collected suggestions from all invoked completion providers. It should be noted that multiple completion providers can be invoked simultaneously, and they contribute to the result set asynchronously. For that reason, it is not possible to have synchronisation between them without risking a deadlock in the completion threads.

Most of the functionality of the completion providers was extracted and put into the `JMLCompletionUtils` class, as a lot of code is shared between the different completion providers. This means that most of the functionality is located there.

Most of the `completionProvider` classes are straightforward and reading the Javadoc should provide a good explanation of what they are doing. However, the `JMLJavaDotCompletionProvider` class is a bit of a hack that handles dot expression completion suggestions, for example (`Integer.getInteger("5")`). The reason for this is that there was not enough time left in the project to write a recursive dot expression evaluator, as accounting for all the import and visibility nuances is complicated. Instead, the complete dot expression string is passed to the Java parser to evaluate the return type up to that point and then provide suggestions based on that. The evaluator does support support some JML-Java expression mixing, but only `\result`.

and `\old()`. keywords are properly supported.

The support for these keywords was achieved by replacing each of these keywords with equivalent valid Java expressions. `\result.` is replaced by `((return_type)0)` or `((return_type)null)` in the string that is passed to the evaluator. `0` is used for primitive types (as any primitive literal is castable to any primitive) and `null` for object return types (as `null` is castable to any object). For `\old()`. Just the `\old` part is removed as the parentheses are valid Java and do not need to be removed.

For accessibility and visibility checking there are two methods that should be kept in mind. First is the `JMLCompletionUtils.isAccessibleHere()` method. This method checks field, method, class and package accessibility at the given location in the project. It accounts for all Java rules regarding to accessibility. However, local variable scoping is not resolved by it, if that is needed - look at `JMLCompletionUtils.localVariableInScope()`. This is written for local variables only, but in theory it is just a much more expensive string evaluation method that also should account for the prior rules as well, it is not used everywhere as it is much more expensive than the `isAccessibleHere()` method.

C.8 Runtime checking

The implementation of runtime checking in the source code is incomplete. It is meant to make use of the following procedure: When a compilation is started, all source files in the project are collected and copied into a temporary directory (the one provided by the operating system). The tree of each copied Java source file is then walked to collect all JML specifications. Each JML specification is converted into an appropriate check in Java code and inserted into the tree of the corresponding copied file. Once all JML specifications have been processed, the copied files are included in the compilation scope, and the originals are excluded. The compilation is then performed by IntelliJ without intervention. After compilation, the temporary directory is cleaned up to ensure that IntelliJ cannot unintentionally show the temporary files to the user through the GUI.

In the current code base, the capability to scan all Java files and transparently insert code is functional. The missing part is the code that converts a given JML specification into an appropriate check. Currently, all code paths related to runtime checking that are present in the plugin are intentionally inaccessible because they are dependent on a setting that cannot be toggled

from the GUI. To make the feature accessible, add a checkbox that enables runtime checking in `nl.utwente.jmlplugin.settings.ConfigurationPanel`. It should be noted that when enabled, the code in its current state does affect the compilation process, but does not do anything useful.

Copying the source files is a workaround best described as a hack. There is however no simple alternative for this: While IntelliJ does support programmatically inserting code at compile time, it forces these changes to be visible to the user, which would be unwanted behavior for this plugin.

If the missing parts of runtime checking are implemented, it is suggested to use if-statements that throw `AssertionErrors` rather than `assert` statements. The latter do not have any effect unless explicitly enabled by a VM argument, which has the potential to confuse students.

C.9 Testing

Unit tests are focused on the parser and annotators, but there are also completion tests. The test framework used is the one used internally by IntelliJ, which is itself based on JUnit 3. Integration tests are also available for the annotators. Unit tests and integration tests can be executed from the IDE as expected, and they can be found in `test/java/nl/utwente/jmlplugin`. Documents describing manual system tests are available in the project's Design Report.

Setup To run the unit and integration tests, the IntelliJ community edition repository needs to be cloned from <https://github.com/JetBrains/intellij-community>. This repository is multiple gigabytes in size, but is unfortunately required: Otherwise, the test runner cannot resolve references to the Java standard library in the test files due to an internal dependency. Make sure that the repository is on the master branch once it has been cloned. After that, navigate to the file `build.gradle.kts` (outside the `src` folder) and look for the `tasks { test {} }` section. Once found, replace `filepath` in `systemProperty("idea.home.path", "filepath")` with the file path of the cloned repository.

Running the tests Now tests can be run. To run all tests after each other open the Gradle tab and in `Tasks` run `verification > test`. Tests can also be run for a single component. For those, run configurations are available

in the IDE. The names of these configurations start with "JML" and end with "Test" and are self-explanatory. For example, the configuration called `JMLSyntaxAnnotatorTest` tests the syntax annotator.

When all tests fail A note should be made that in rare occasions all the tests fail. This is an issue with either Gradle or IntelliJ and not with the plugin. If this occurs, first try to re-run all the tests. If that does not work, look for a folder called `build` and delete it. Then try to run the tests again. If that also does not work, one can try to also delete the `.gradle` folder in the project and in one's user folder, but be warned that it takes Gradle several minutes to rebuild in that case.

Running the tests with coverage The tests can also be run with coverage. Unfortunately, the Gradle task that runs all tests does not run properly with coverage, as it starts a new instance for every test class it runs. This has been done intentionally because there were problems with tests classes not being closed properly, which caused the tests to give different results based on the order they were run. Running them as separate instances fixed this issue. However, this causes only the coverage of the last instance to be shown, and not of all tests together. To solve this, there is a run configuration called `CoverageAllTests`, which runs all test classes after each other. To run with coverage, select `CoverageAllTests` and click `Run with coverage`. During the execution, a popup might appear asking whether coverage data should be displayed for `TestName` results. When this comes up, click on `Add to active suites`. When the task is done running, the coverage should appear on the right side of the screen. It can be verified that the coverage of all classes is being displayed by navigating to `Run > Show Coverage Data...` and then checking all the checkboxes and clicking `Show selected`.

C.10 Known bugs

- An at-sign can be placed after a JML clause without causing a syntax error; it should only be allowed before a clause.
- When an import in a class is only used in JML and not in the Java code, IntelliJ's code cleanup will delete the import because it believes it is unused.

- In some IntelliJ versions, the code that removes the green background overrides the yellow tint that appears on the line where the cursor is.
- Assigning a value to a method parameter within a JML specification gives the incorrect error message "Variable expected". Since assigning variables is not allowed in general, it is correct that there is an error, only the message is wrong.
- Pure checking is insufficient: It does not check if variables are assigned inside the method, and marking a void method as pure is possible.
- No message is given when `modifiable` clauses conflict, such as having `modifiable \everything` and `modifiable \nothing` in the same JML comment.
- Pressing enter inside an incomplete JML comment adds a spurious `*/` under certain conditions.
- Copying and pasting a JML specification does not automatically correct indentation, unlike copying and pasting Java.
- Auto-formatting the code may result in awkward indentation in JML comments under certain conditions.
- Using `Ctrl+/*` inside a JML comment adds `//` to the start of the line, breaking the JML lexer. JML comments of the form `(* *)` should be used instead.
- Generics are not properly type checked. For example, the class name of the generic does not need to exist.
- Code completion does not work on JML backslash expressions related to types.

Appendix D System tests

This document describes a system test for IntelliJML. It centers on the aspects of the plugin that are not tested at the unit test level: The installation of the plugin, the correct functioning of the syntax highlighting feature, and the performance requirement established by the requirements specification. This test assumes a clean installation of IntelliJ IDEA. Note that all features belonging to IntelliJ are assumed to be working as expected.

First, the installation process was tested.

Using the *Install Plugin from Disk...* feature in the *Plugins* window in IntelliJ, select the IntelliJML jar file, confirm the selection, and confirm the changes in the *Plugins* window. Then, restart IntelliJ.

Pass: IntelliJML appears in the *Plugins* window.

Fail: IntelliJML does not appear in the *Plugins* window, has an annotation indicating a problem, or IntelliJ displays an error message during the process.

The items below each describe a test that involves typing certain text into the IDE. After the arrow, it is described what the expected outcome of typing the given text is. The test is considered to have passed if the actual outcome matches the expected outcome.

- `//@ requires;`
-> syntax error, no highlighting
- `//@ requires true`
-> syntax error (missing semicolon), no highlighting
- `//@ requirse true;`
-> syntax error, no highlighting
- `//@ \requires true;`
-> syntax error, no highlighting
- `//@ requires treu;`
-> type error, highlighting except “treu”
- `//@ requires true;`
-> no error, everything highlighted
- `//@ requires true; (* hello *)`
-> no error, everything highlighted

- `//@ requires true; (* hello)`
-> syntax error, no highlighting
- `//@ requires true;`
in class body -> semantic error, no highlighting, only first @ is highlighted
- `//@ (* hello *)`
-> no error, everything highlighted
- `//@ (hello *)`
-> syntax error, no highlighting
- `/*@
requires
true
;
*/`
-> no error, everything highlighted
- `//@@@@@@@@@requires true;;;`
-> no error, everything highlighted
- `/*@
requires true;

ensures true;
*/`
-> no error, everything highlighted
- `/*@
requires treu;

ensures true;
*/`
-> other error, highlighting except “treu”
- `//@ pure`
in a class body -> syntax error, no highlighting, only first @ is highlighted

- `/*@`
`requires true;`
`invariant true;`
`ensures true;`
`*/`
 above method -> syntax error, no highlighting
- `/*@`
`requires true;`
`invariant true;`
`ensures true;`
`*/`
 in class body -> syntax error, no highlighting
- `/*@`
`requires treu;`
`invariant true;`
`ensures true;`
`*/`
 above method -> syntax error, no highlighting
- `/*@`
`requires treu;`
`invariant true;`
`ensures true;`
`*/`
 in class body -> syntax error, no highlighting
- `/*@`
`requires true;`
`invariant true;`
`ensures treu;`
`*/`
 above method -> syntax error, no highlighting
- `/*@`
`requires true;`
`invariant true;`

```
ensures treu;  
*/
```

in class body -> syntax error, no highlighting

- `//@ ensures \old(maxNrOfBoxes)== maxNrOfBoxes;`
-> no error, everything highlighted
- `//@ ensures \odl(maxNrOfBoxes)== maxNrOfBoxes;`
-> syntax error, no highlighting
- `//@ ensures \old == maxNrOfBoxes;`
-> syntax error, no highlighting
- `//@ ensures \old()== maxNrOfBoxes;`
-> syntax error, no highlighting
- `//@ ensures \result == maxNrOfBoxes;`
-> no error, everything highlighted
- `//@ ensures \result(maxNrOfBoxes)== maxNrOfBoxes;`
-> other error, highlighting except after `\result`
- `//@ ensures \result1 == maxNrOfBoxes;`
-> syntax error, no highlighting
- `//@ ensures \result()== maxNrOfBoxes;`
-> other error, highlighting except after `\result`
- `//@requires (\forall int i; true; true);`
-> no error, everything highlighted
- `//@requires (\forall true; true; true);`
-> syntax error, no highlighting
- `//@requires (\forall int i; true; true);`
-> syntax error, no highlighting
- `//@requires (\forall int i; treu; true);`
-> other error, highlighting except “`treu`”
- `//@requires (\forall int i; treu);`
-> syntax error, no highlighting

- `//@requires (\forall int i; treu);`
-> other error, highlighting except “treu”
- `//@requires (\forall int i; true);`
-> no error, everything highlighted
- `//@requires (\forall int i);`
-> syntax error, no highlighting
- `//@requires (\forall int i);`
-> syntax error, no highlighting
- `//@requires \forall int i; true; true;`
-> syntax error, no highlighting
- `//@requires \forall(int i; true; true);`
-> syntax error, no highlighting
- `//@requires (\forall);`
-> syntax error, no highlighting
- `//@requires (\forall);`
-> syntax error, no highlighting
- `//@requires \forall;`
-> syntax error, no highlighting
- `//@signals(Exception e);`
-> no error, everything highlighted
- `//@signals(Exceptino e);`
-> highlighting except “Exceptino”
- `//@signals(Exception e)e;`
-> other error, highlighting except “e”
- `//@signals(Exceptino e)e;`
-> other error, highlighting except “Exceptino” and “e”
- `//@signals(Exception e)e != null;`
-> no error, everything highlighted

- add whitespace in front of @ on single line comment -> highlighting disappears
- remove whitespace in front of @ on single line comment -> highlighting appears
- add whitespace in front of @ on multiline comment -> highlighting disappears
- remove whitespace in front of @ on multiline comment -> highlighting appears
- typing text -> updates within a second
- removing text -> updates within a second
- pasting in a large block of text -> updates within a second
- removing a large block of text by selecting and then pressing backspace -> updates within a second

Appendix E Usability testing results

This appendix contains the full results of usability testing the plugin with four TAs.

E.1 Participant 1

E.1.1 Questions

How much experience do you have with JML? None in particular.

What do you think of the plugin? Having feedback on errors being made is useful.

If you could change one thing, what would it be? Fix code completion.

Do you think that the plugin would help the students in Software Systems? Error might not help you if you do not have docs next to you.

What do you think of the code completion? Was broken.

Any other comments? Not really.

E.1.2 Observation Notes

- Plugin is auto-disabled on tester IDE as the plugin specifies that it only works with the newest IDE version.
- Code snippet for completing multi-line JML comment needed.
- Tester writes "@" on every line in JML comment.
- The need to put parentheses around quantified expressions not immediately clear.
- Code completion auto-prompt is not aggressive enough.
- Indentation levels are not maintained when going to newline in a multi-line JML comment.

- Clearer error messages with regards to quantified expressions needed.

It should be noted that the JML lecture link and slides were not sent to this participant in advance, because it was incorrectly assumed that this participant was a third-year student.

E.2 Participant 2

E.2.1 Questions

How much experience do you have with JML? None, besides the slides that were given.

What do you think of the plugin? Installing is simple, syntax highlighting is nice, auto-indentation needs fixing.

If you could change one thing, what would it be? Fix indentation so the indentation level is kept after going to a new line.

Do you think that the plugin would help the students in Software Systems? It would help because the definitions in JML are much more strictly checked than the loose definitions in Software Systems previously.

What do you think of the code completion? Brackets on method completions are sometimes not there, other than that it's nice.

Any other comments? Provide type error messages, for runtime assertions make a window like for unit tests.

E.2.2 Observation Notes

- Code completion on method completions do put parentheses after the method name.
- In a JML method spec comment there are no code completion suggestions for method parameters.
- Multi-line JML comment completion is needed.

- Indentation levels are not maintained in multi-line JML comments when going to a new line.
- Assignment operator did not throw an error.

E.3 Participant 3

E.3.1 Questions

How much experience do you have with JML? TA'ed Software Systems when it was still using OpenJML. Used it a bit outside Software Systems as well.

What do you think of the plugin?

- Feels like an IntelliJ plugin.
- Completion has performance issues.
- Besides code completion is fast.
- Relatively nice plugin.
- Some minor bugs present, but not that disruptive.
- Inclusion of redundant `\not_specified` is nice.
- Colour palette is nice, not overwhelming.

If you could change one thing, what would it be?

- More code snippets for common JML specifications patterns.
- Entire Java expression is underlined if there is something wrong with it, not just the wrong part.
- More guidance on the syntax.
- More context dependent code completion.

Do you think that the plugin would help the students in Software Systems? Given that the error messages are more descriptive than OpenJMLs error messages it should help out students

What do you think of the code completion? Some issues with performance and code completion should also be more context aware.

Any other comments? Nice work.

E.3.2 Observation Notes

- JML multi-line comment completion only works with the enter key and not tab.
- Plugin does not work on older IntelliJ versions.
- Indentation breaks when copying code.
- Quantified expression error message are not clear.
- Completion ranking is not working properly.
- Java comments in JML do not break when they should.
- Type checking in `\sum` might be broken.
- Add keyword explanations to completion?
- Broken semantics of `\old` in `loop_invariant`?
- Code error in sample project.
- Modifiable does not prompt JML code completion.
- IntelliJ suggestions are broken on JML code injection sites.
- More context aware completions are needed.
- Should allow JML lines to be commented out using `ctrl+\`.
- Weird `\old` highlighting.

E.4 Participant 4

E.4.1 Questions

How much experience do you have with JML? Some experience from taking the Software Systems module.

What do you think of the plugin? Kind of like it.

If you could change one thing, what would it be? Having quick fixes would be nice, message for the "@" space mistake at the start of a JML comment should be a warning instead of an error.

Do you think that the plugin would help the students in Software Systems? Yes it should help out students.

What do you think of the code completion? Code completion popups are not aggressive enough.

Any other comments? Overall likes the plugin.

E.4.2 Observation Notes

- Completion on invariant fields is slow.
- `\nonnullelements` fails to resolve types properly.
- Did not understand the "invariant not allowed above a field" message.
- Code merge fail as some features were missing from the plugin?
- Code completion with quantified expressions does not always work.
- Code templates are shown in inappropriate contexts.
- Non-pure error message is displaying on a pure method.
- JML keyword completion needs fixing.
- Type checking sometimes fails on heavily dot completed expressions.
- Method completion displays two parentheses sets.
- Add parentheses around the `\forall` completion.