

BA - FS2019 - Documentation

Safe C++ Guidelines for Ceevelop (AUTOSAR)

University of Applied Sciences Rapperswil

February - June 2019

Author:	Viktor Puselja, Gabriel Vlasek
Supervisor:	Peter Sommerlad
Technical Adviser:	Toni Suter
Expert:	Martin Botzler

1. Abstract

CodeAnalysator is an Eclipse plug-in for improving C++ code. Its static analysis uses AST visitors for determining rule violations and notifies the users about them. Often the plug-in even proposes quick fixes to solve a problem. The plug-in features rules from many different C++ programming guidelines such as from AUTOSAR or MISRA. The plug-in was initially developed by another team. After we added a large set of rules in the previous term project, this thesis was all about improvement. By applying refactorings and rewriting code, the complexity of the project was greatly reduced and circular dependencies were removed. By introducing new concepts and improving the naming, we increased understandability and readability of the project. Using performance measuring we also improved the execution speed and responsiveness significantly, reducing the run time on our test projects from almost an hour to less than three minutes. Among many other smaller things, we also incorporated a set of new rules from the AUTOSAR guideline.

2. Management Summary

This chapter consists of a short introduction of this thesis, a description of our approach and the results and findings of the project.

2.1. Introduction

The integrated development environment Cevalop is used to write C++ code. To provide as much help as possible to the developer, Cevalop contains a large number of plug-ins that make programs more safe and reliable. One of these plug-ins is the CodeAnalysator. The plug-in features rules from many different C++ programming guidelines such as from AUTOSAR or MISRA. These guidelines define rules that improve the safety and robustness of C++ code and also reduce ambiguity. The CodeAnalysator plug-in implements checkers for these rules that will notify the developer by underlining the code where it violates such a rule. While the foundation of the project was laid in an earlier project and some rules were already implemented in another thesis, this project was all about improving the existing plug-in.



Figure 2.1.: Cevalop and AUTOSAR.

2.2. Approach

The main goal of this project was to make the CodeAnalysator plug-in viable for real world usage, because the performance was unsatisfying initially. To achieve this, we removed as many bugs and mistakes as we could. We did this by performing functional tests on real C++ projects. Another part of the project was to improve the overall structure and architecture of the plug-in. We analyzed the tool by hand and using tools and identified many things that we could improve. But functionality is not the only thing that needed improvement. The performance was lackluster and was one of the main things that held the plug-in back. Using the same test projects as in the functional tests, we identified performance bottlenecks. Additionally we also increased the amount of supported rules. Such a rule works by having a checker analyze the code and recognize patterns that can lead to an unsafe program. Sometimes the problem can be solved by the plug-in directly. In these cases, we offer a so called quick fix which is a short program to fix a given issue.

<pre>#include <stdint> A typedef int32_t type;</pre> <div><div>A</div><div>A7-1-6: The typedef specifier shall not be used</div><div>3 quick fixes available:</div><div><div>➤</div><div>A7-1-6: Replace typedef with using</div></div><div><div>➤</div><div>Suppress all guidelines for this statement</div></div><div><div>➤</div><div>Suppress autosar A7-1-6 for this statement</div></div></div> <div>Press 'F2' for focus</div>	<pre>#include <stdint> using type = int32_t;</pre>
---	--

Table 2.1.: Example of a checker and a quick fix

2.3. Results

Looking at the plug-in before and after the project, one can notice many differences. First of all, we increased the number of supported checkers and quick fixes. In total we added eight new checkers and two quick fixes. In another part of the project, we improved the readability of the source code by using more appropriate names for the different classes so that a new developer can understand the plug-in better. A few redundant parts of the program were removed and some complicated parts were simplified. The performance tests led to great results as well. We managed to reduce the execution time of the plug-in on our test projects from almost an hour to less than three minutes. In the end, the CodeAnalysator plug-in is in a state where new checkers and quick fixes can be added easily and where C++ developers can benefit from it.

2.4. Outlook

There are still a great amount of rules that can be implemented from the already supported AUTOSAR, MISRA and C++ Core Guidelines as well as from new guidelines. Nonetheless the CodeAnalysator plug-in is in a good state.

Contents

1. Abstract	ii
2. Management Summary	iii
2.1. Introduction	iii
2.2. Approach	iv
2.3. Results	v
2.4. Outlook	v
3. Introduction	1
3.1. Safe C++	1
3.2. Workings of the CodeAnalysator	2
3.3. Previous Work	4
3.4. Project Goals	4
4. Code Improvement	5
4.1. Overview	5
4.2. Guideline Mapping and Priority Resolution	6
4.3. Rule Suppression	12
4.4. CodeAnalysatorVisitor Type Parameter	17
4.5. CodeAnalysator Visitors	18
4.6. Performance Measurement	18
4.7. Rule Reporting	18
4.8. Rule Enablement Checking	19
4.9. CodeAnalysator Checkers	19
4.10. BaseQuickFix	19
4.11. AttributeSuppressQuickfix	21
4.11.1. Remove Bug Workaround	21
4.11.2. Relation to BaseQuickFix	22
4.12. Package Structure	22
4.13. Checker and Quick Fix Tests	24
4.14. Refactor Old Tests	27
4.15. Ordering of Rules	28

4.16. A6-4-1: A switch statement shall have at least two case-clauses, distinct from the default label.	29
4.16.1. Switch statement with non-compound statement as direct child	30
4.16.2. Surrounding compound statement	30
4.16.3. Further errors	31
4.17. A8-5-3: A variable of type auto shall not be initialized using {} or = braced-initialization.	32
4.18. Conclusion	32
4.18.1. Reflection	33
4.18.2. Proposed Future Work	33
5. Functional and performance testing	35
5.1. Finding Suitable Test Projects	35
5.1.1. Possible Projects	36
5.1.2. Decision	36
5.2. Functional Tests	38
5.2.1. Sample Test Cases: jucipp	39
5.2.2. Functional Reports Tests: jucipp	48
5.2.3. Functional Sample Test Cases: LevelDB	52
5.2.4. Functional Reports Test: Leveldb	59
5.2.5. Changes Based on Functional Tests	61
5.2.6. Conclusion of Functional Tests	69
5.3. Performance Tests	71
5.3.1. Test Results	71
5.3.2. Changes Based on Performance Tests	72
5.3.3. Test Results after Changes	80
5.4. Conclusion	80
6. Quick Assists	81
6.1. Getting Started	81
6.2. Overrider	84
6.3. Struct/Class Switcher	85
7. Per-Project Guideline Configuration	88
8. Implementation of checkers and quick fixes for further rules	92
8.1. Analysis	92
8.1.1. Possible Rules to implement	93
8.2. New rules	96
8.2.1. A02-13-05: Hexadecimal constants should be upper case. . .	96

8.2.2.	A5-0-3: The declaration of objects shall contain no more than two levels of pointer indirection.	98
8.2.3.	A05-02-01: <code>Dynamic_cast</code> should not be used	99
8.2.4.	A05-02-06: The operands of a logical <code>&&</code> or <code> </code> shall be parenthesized if the operands contain binary operators. . . .	100
8.2.5.	A5-10-1: A pointer to member virtual function shall only be tested for equality with <code>null-pointer-constant</code>	103
8.2.6.	A7-5-1: A function shall not return a reference or a pointer to a parameter that is passed by reference to <code>const</code>	104
8.2.7.	A13-05-02: All user-defined conversion operators shall be defined <code>explicit</code>	105
8.2.8.	A15-03-05: A class type exception shall be caught by reference or <code>const</code> reference	107
8.3.	Existing rules	108
8.3.1.	A5-0-2: Condition of <code>if/while/for</code> shall be <code>bool</code>	108
8.3.2.	A5-1-1: No magic numbers	109
8.3.3.	A5-1-2: Do not implicitly capture variables in lambda expressions	110
8.3.4.	A5-1-3: Parameter list shall be included in lambda	110
8.3.5.	A5-1-4: Lambda expressions shall not outlive reference captured objects	111
8.3.6.	A5-1-6: Explicit lambda return type	111
8.3.7.	A5-1-8: Do not nest lambda expressions	112
8.3.8.	A5-16-1: Do not use the ternary conditional operator as sub-expression	112
8.3.9.	A6-4-1: Switch shall have at least two cases	112
8.3.10.	A7-1-4: The register keyword shall not be used	113
8.3.11.	The <code>auto</code> specifier shall not be used	113
8.3.12.	A7-1-6: The <code>typedef</code> specifier shall not be used	114
8.3.13.	A7-2-2: Enumeration underlying base type shall be explicitly defined	114
8.3.14.	A7-2-3: Declare enumerations as <code>scoped enum</code> classes	115
8.3.15.	A7-2-4: Initialize none, the first or all enumerators	115
8.3.16.	A7-4-1: The <code>asm</code> declaration shall not be used	115
8.3.17.	A8-5-2: Braced-initialization <code>{}</code> , without equals sign, shall be used for variable initialization	116
8.3.18.	A8-5-3: A variable of type <code>auto</code> shall not be initialized using <code>{}</code> or <code>={} braced-initialization</code>	116
8.3.19.	A9-5-1: Unions shall not be used	117

8.3.20. A10-1-1: Class shall not be derived from more than one base class which is not an interface class.	117
8.3.21. A10-2-1: Do not redefine non-virtual member functions . . .	117
8.3.22. A10-3-1: Virtual function declarations shall have exactly one of the specifiers virtual, override or final	118
8.3.23. A10-3-2: Each overriding virtual function shall be declared with the override or final specifier.	119
8.3.24. A10-3-3: Do not introduce virtual member functions in a final class	119
8.3.25. A10-3-5: User defined assignment operators shall not be virtual	120
8.4. Conclusion	120
9. Quality Measures	121
10. Conclusion	122
10.1. Retrospective	122
10.2. Considerations for future projects	123
Glossary	129
A. Developer Guide	131
A.1. Checker	132
A.1.1. Visitor	132
A.1.2. Testing	136
A.2. Quick Fix	138
A.2.1. Quick Fix	138
A.2.2. Testing	139

List of Listings

1.	A compiling code snippet	2
2.	Member variable declarations of the class Rule	8
3.	Example of a suppression attribute for the rule "A7-1-6: The typedef specifier shall not be used"	12
4.	CodeAnalysatorVisitor declaration with type parameter	18
5.	Switch with neither a compound, case nor default statement as child	30
6.	A lamdba returning a nullptr	62
7.	A lamdba returning a nullptr, fixed incorrectly	62
8.	A lamdba returning a nullptr, fixed correctly	62
9.	Example for a binary expression with a non-expression second operand	63
10.	Example of a wrongly inserted semicolon for control declarations . .	63
11.	Example of a ranged for loop	64
12.	The visitor marked the constructor on line 9 wrongly	64
13.	Resolving this marker failed before our fix	65
14.	The A05-01-01 marker did not recognize this exception	65
15.	B.doSomething() is virtual even though there is neither a virtual keyword or an override specifier. Because of this, C.doSomething() does not shadow but override.	66
16.	Example of a method that can be made final	67
17.	Example made final without violating A10-03-01	67
18.	Example of a method that can be made final with virtual keyword .	67
19.	Example made final but violating A10-03-01	67
20.	A ternary conditional operator used in an assignement	68
21.	The AvoidLossyConversionVisitor before our refactoring.	75
22.	The AvoidLossyConversionVisitor after our refactoring.	76
23.	The Quick Fix for A07-01-06 is not always appliccable.	77
24.	Checker generating and passing a contextFlagsString	78
25.	Quick Fix computing isApplicable by contextFlagsString	79
26.	ContextFlagsHelper containing contextFlagsString indices and flags	79
27.	A02-13-05 checker example	97

28.	An example of the quick fix for A02-13-05	98
29.	Examples of declarations violating rule A5-0-3	99
30.	Usage of dynamic_cast. Taken from [AUTOSAR 2018]	100
31.	Example of a binary expression	101
32.	102
33.	The complicated example resolved	102
34.	Example of comparison violating rule A5-10-1	104
35.	Example of comparison violating rule A7-5-1	105
36.	The difference between an explicit and a non explicit conversion operator	106
37.	Examples of catching a reference by value and by reference	107
38.	AutosarIdHelper	132
39.	AutosarIdHelper	133
40.	fragment.xml	134
41.	fragment.xml	135
42.	AutosarChecker.java	136
43.	Adding the test class to the test suite	136
44.	Adding the test class to the test suite	137
45.	An example rts file	137
46.	An example quick fix	138
47.	Registering your quick fix	139
48.	A new quick fix test class	140
49.	Adding the test class to the test suite	140
50.	An example quick fix rts file	141

List of Figures

2.1. Cevelop and AUTOSAR.	iii
3.1. Our checker marks the rule. By hovering over it, the developer gets a list of possible solutions.	3
3.2. The code after the quick fix was applied	3
4.1. Guideline before refactoring	10
4.2. Guideline after refactoring	11
4.3. The AttributeSuppressionStrategy	13
4.4. Suppression attributes and quick fixes before refactoring	14
4.5. Suppression attributes and quick fixes after refactoring	15
4.6. Marker resolution generation before refactoring	16
4.7. Marker resolution generation after refactoring	17
4.8. BaseQuickFix and derived RuleQuickFix before refactoring	20
4.9. BaseQuickFix and derived RuleQuickFix after refactoring	21
4.10. The code to the left is rewritten with the code to the right	22
4.11. The package structure before the refactoring	23
4.12. The package structure after refactoring	24
4.13. Checker tests before refactoring	25
4.14. Quick fix tests before refactoring	25
4.15. Checker tests after refactoring	26
4.16. Quick fix tests after refactoring	27
4.17. The C++ Code Analysis preferences page before renaming the rules	28
4.18. The C++ Code Analysis preferences page after renaming the rules .	29
5.1. The results of the performance test	71
5.2. The results of the performance test while scrolling	72
5.3. The results of the final performance tests	80
6.1. The general structure of a quick assist	82
6.2. Diagram of the completed Override quick assist and refactoring . .	83
7.1. The existing guideline preferences page for the workspace	89
7.2. The guideline preferences page now for projects	91

8.1.	A part of the AST for the example in Listing 31	101
8.2.	Example for A5-0-2 [AUTOSAR 2018]	108
8.3.	Unclear example [AUTOSAR 2018]	109
8.4.	Different ways of initializing an auto variable[<i>Initialization - cppreference.com</i> 2018]	116
8.5.	B.F() shadows A.F()	118

List of Tables

2.1. Example of a checker and a quick fix	iv
4.1. Example of the quick fix adding a surrounding compound statement	31
5.1. Test protocol for A08-05-02	39
5.2. Test protocol for A05-01-01	40
5.3. Test protocol 2 for A08-05-02	40
5.4. Test protocol for A08-05-00	41
5.5. Test protocol for A07-02-02	41
5.6. Test protocol for A07-02-03	42
5.7. Test protocol for A05-01-02	43
5.8. Test protocol for A07-01-06	44
5.9. Test protocol for A05-01-03	45
5.10. Test protocol for A07-02-02	46
5.11. Test protocol for A10-01-01	46
5.12. Test protocol for A05-01-08	47
5.13. Test protocol for jucipp checked rule violation reports	48
5.13. Test protocol for jucipp checked rule violation reports	49
5.13. Test protocol for jucipp checked rule violation reports	50
5.13. Test protocol for jucipp checked rule violation reports	51
5.14. Test protocol for A05-01-01 and A08-05-02 test	52
5.15. Test protocol for A05-00-02 test	52
5.16. Test protocol for A05-01-03	53
5.17. Test protocol for A05-16-01	53
5.18. Test protocol for A07-01-05	53
5.19. Test protocol for A07-01-06	54
5.20. Test protocol for A07-02-02	54
5.21. Test protocol for A07-02-03	54
5.22. Test protocol for A08-05-00	55
5.23. Test protocol for A07-02-03	56
5.24. Test protocol for A10-03-03	57
5.25. Test protocol for A12-00-01 test	58
5.26. Test protocol for leveldb checked rule violation reports	59
5.26. Test protocol for leveldb checked rule violation reports	60

5.27. How the tests for AvoidLossyConversions where changed	74
6.1. Examples of the Override	84
6.2. Struct to class switch cases	85
6.3. Class to struct switch cases	85
6.4. Examples of switching from struct to class	86
6.5. Examples of switching from class to struct	87
8.1. Estimated hours per task	92
8.2. List of possible rules	96
8.3. Assignment of clauses to if-, else- and following-parts	113

3. Introduction

The C++ programming language is used in many different situations in a lot of different branches of the industry. Good code quality is always important, but in some situations lives might even depend on it. One of the branches where a lot of such situations occur is the automobile industry. It is of uttermost importance that a car does not fail because of a preventable programming error. Unfortunately, there are many mistakes one can make when working with C++ in safety critical systems. This is where the CodeAnalysator plugin comes in and provides support for writing safe C++ code by marking code that might be problematic and proposing solutions to these problems.

3.1. Safe C++

The foreword of [*MISRA C++ : 2008 Guidelines for the use of the C++ language in critical systems* 2018] begins with the following citation of Bjarne Stroustrup: "C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off". This sums up what working with C++ is like. It is a very powerful language but it is also more difficult to get right than some other languages. The main danger in C++ programs is undefined behaviour. As the term suggests, undefined behaviour can not be reliably predicted. Effects of this can lead to program crashes, faulty program results or even software or hardware corruption. This is obviously unwanted in critical systems (or any other system). Luckily C++ is a very mature and tried language and a lot of common situations where programming errors lead to undefined behaviour are known. Many different guidelines suggest coding standards that, among other things, prevent such behaviour. For this project the two guidelines AUTOSAR [AUTOSAR 2018] and MISRA [*MISRA C++ : 2008 Guidelines for the use of the C++ language in critical systems* 2018] from the automobile industry are relevant. These two address not only undefined behaviour, but also topics like readability or clarity. Of course, no guideline can guarantee completely fault-free programs, but they can at least help mitigate common mistakes.

3.2. Workings of the CodeAnalysator

The guidelines mentioned in Section 3.1 are rather long documents. They define a wide range of rules on how to write safe C++ code. It takes a lot of time and effort to learn all these rules. This is why the CodeAnalysator plugin is needed. It uses checkers to statically analyze the code and recognize when rules are violated. It will then underline the corresponding code to signal this to the developer. This direct feedback is what makes the CodeAnalysator special. In many cases it is possible to automatically transform the code into a compliant state and fix the issue. These solutions are called quick fixes. When the user hovers over a marked piece of code, all available quick fixes are proposed. If he/she wants, the developer can activate one by clicking on it. This will resolve the problem without changing the semantics of the code.

```
1  #include <cstdint>
2  void Fn() noexcept
3  {
4      auto lambda = []() {
5          std::uint8_t ret = 0U;
6          return ret;
7      };
8  }
```

Listing 1: A compiling code snippet

For example the code snippet in Listing 1 violates AUTOSAR rule A5-1-6 [AUTOSAR 2018] but will compile and is valid code. Rule A5-1-6 states that a lambda return type shall be explicitly specified as opposed to implicitly as it is done in the snippet above. The CodeAnalysator plugin will underline the corresponding part of the code. In this case it is the entire lambda expression. This will not go unnoticed. See Figure 3.1.

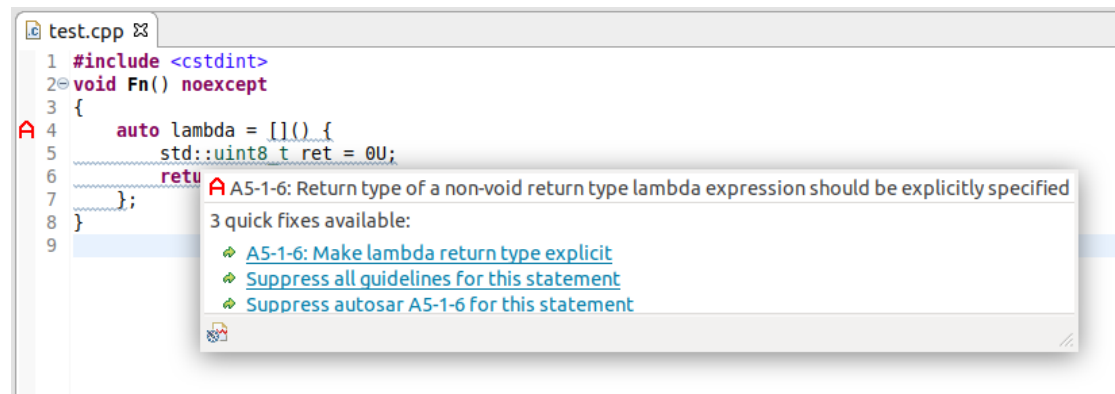


Figure 3.1.: Our checker marks the rule. By hovering over it, the developer gets a list of possible solutions.

If the developer chooses the quick fix "A5-1-6: Make lambda return type explicit", the abstract syntax tree is analyzed and the lambda rewritten. This time with the deduced type explicitly specified as the trailing return type. See Figure 3.2 for what the corrected code looks like. Note that most IDEs will mark this lambda even after applying the quick fix because the variable lambda is unused.

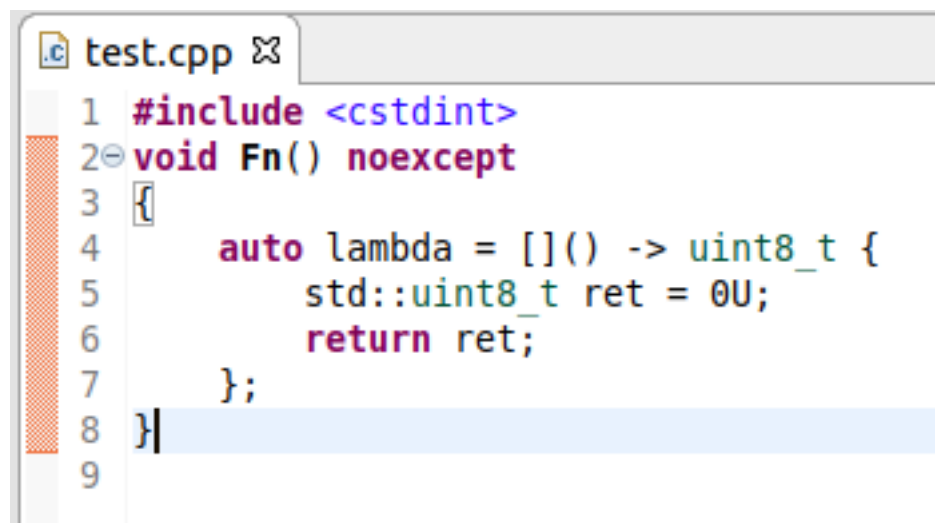


Figure 3.2.: The code after the quick fix was applied

In summary, this means that the CodeAnalysator plugin helps C++ developers apply the guideline and as a result improves the quality of the written code.

3.3. Previous Work

This project builds on the already existing CodeAnalysator plugin. This is an Eclipse plugin for the C++ development environment Codeloop. It was developed over the course of two term projects by two different teams. In the first project [Deicha and Bertschi 2018] the plugin was initially built. This means the main focus was on the overall architecture and design of the plugin. There were also a few checkers and quick fixes implemented, but in a proof-of-concept way. In the second project [Puselja and Vlasek 2019] a large number of checkers and quick fixes were implemented for the rules of the AUTOSAR guideline [AUTOSAR 2018].

3.4. Project Goals

The team of this project is the same as it was during the second term project mentioned in 3.3. Since we worked closely with the CodeAnalysator plugin during this project, we learned a lot about it. We noticed that there are some things that could be improved. Because of this we will analyze the architecture of the plugin and find out how we can improve it. We will also test the performance and functionality of the checkers and quick fixes by running them on a real project. Hopefully, we will gain a lot of insight and find many possibilities for improvements. We will also add some new features and new checkers and quick fixes. In summary, this means we will focus more on architectural topics as opposed to just implementing rules like in the previous project. This also means that the amount of analysis required will increase.

4. Code Improvement

As the first major task of our bachelor thesis we refactored the CodeAnalysator plug-in. In the preceding term project we have implemented many checker and quick fixes for the plug-in. [Puselja and Vlasek 2019] While doing so, we have become very familiar with the CodeAnalysator plug-in, its architecture and its shortcomings. Some examples were the complex and seemingly redundant registrations in the `IGuidelineMapper` implementations, the many responsibilities of the `GuidelineConflictResolver` or the large number of sub-classes related to rule suppression containing only little configuration. While the issues were out of scope for our term project, we took notes of our findings in order to address them over the course of our bachelor thesis. During the refactorings some errors surfaced, in which case we addressed them within the scope of this code improvement.

This chapter begins by giving a broad overview over the refactorings we performed. Afterwards, we elaborate on each self-contained refactoring, giving an analysis of the problem as well as our solution. Finally, we end the chapter with our conclusion, where we reflect on what has been done and propose a list of possible future work, which could not fit into our thesis.

4.1. Overview

We performed two major refactorings within this thesis. Firstly, the largest and most important, the separation of the `GuidelineConflictResolver` into his distinct responsibilities. This led to the creation of the various `GuidelinePreferences`, `GuidelinePriorityResolver`, etc. classes, of which the introduction of the `Rule` class is perhaps the most pervasive. It also led to a simplification of most of the previous uses of the `GuidelineConflictResolver`.

Secondly, the comprehensive refactoring of rule suppression, which led to a stark reduction in the number of classes and in complexity. On one hand this was done by moving configuration into constructor or method arguments, on the other hand by replacing the registry-based approach on suppression with a factory-based one.

Besides these two major refactorings we performed a number of smaller refactorings. These are mostly surrounding the visitor and quick fix base classes, the package structure and tests. The refactorings surrounding the base classes were

mostly concerned with simplifying, removing unnecessary code and duplication. We resolved the circular dependencies and inconsistencies in the package structure and cleaned up the tests. In general, we cleaned up small things when we came across them.

Lastly, we go into the refactorings of certain checker and quick fixes. We resolved some errors and simplified the code for A6-4-1 and removed duplication for A8-5-3. Further, we generally performed small refactorings and cleaned up some code not worth explicitly mentioning.

4.2. Guideline Mapping and Priority Resolution

The class `GuidelineConflictResolver` had many responsibilities that, despite its name, did not include resolving conflicts between guidelines. It was merely a participant of this process. The various responsibilities were:

- Collecting all guidelines and mapping guideline IDs to guidelines
- Mapping problem ids to visitor ids
- Mapping problem ids to guidelines
- Mapping visitor ids to a list of guideline, problem id tuples, sorted by guideline priority

Problem ids are reported to the Codan framework and identify the specific problem to report. Visitor ids were private to the `CodeAnalysator` plug-in. Their purpose was to be able to map problem ids of different guidelines to shared visitor ids in order to avoid reporting the same problem multiple times for different guidelines. Problem ids would map to the same visitor id if they were concerned with the same problem. Instead of reporting the same problem multiple times, the `CodeAnalysator` plug-in would only report it once, for the guideline with the highest priority. This is the actual guideline conflict resolution (or as we later refer to it, the guideline priority resolution), though the `GuidelineConflictResolver` only provides the presorted priority list of problems for a visitor id and the bulk of the resolution is done by the `CodeAnalysatorChecker`.

Further issues with the `GuidelineConflictResolver` were:

- Singleton-like static instance
- Unused instance change listener
- Initialization spread among three classes
- Full initialization on every preference change

The static instance of the `GuidelineConflictResolver` is initialized on start-up and on every preference change. It is a serial singleton in that there is a single

global instance but that the instance might get replaced in time. There exists an accompanying change listener, though it is not used by any part of the code. We would generally like to avoid the use of singletons, but especially such a more complicated serial singleton.

The initialization is spread among the classes `GuidelineLoader`, `GuidelineConflictResolverFactory` and `GuidelineConflictResolver` itself, which would better be contained in one place for each responsibility. Lastly, there were also issues in general regarding the concept of visitor ids and the mapping of problem ids to visitor ids. The issues were:

- Only a single rule reported by a visitor can be suppressed
- A rule reported by multiple visitors can only be suppressed for a single visitor
- Conflict resolution fails, when one of multiple visitors reporting a rule is shared

The visitor base class `CodeAnalysatorVisitor` explicitly supported reporting multiple problems, while the derived base class for shared visitors `SharedVisitor` limited that to one. The `GuidelineConflictResolver` only supports visitors which report a single problem and while conflict (or priority) resolution is only needed for shared rules, mapping problem ids and visitor ids was needed for rule suppression. As a result, only a single rule can be suppressed per visitor. This issue was mitigated though, by there only being a single visitor, `BoolExpressionOperandsVisitor`, reporting more than one rule. The second rule was also only an info problem. It was an extension of the first rule, with the same rule number as the first one. It might be argued whether it needs suppression, though it probably should be suppressed together with the first rule.

The second issue is caused similarly like the first. The `GuidelineConflictResolver` only supports problems reported by a single visitor. If multiple visitors report the same problem, only the problems reported by a single visitor can be suppressed. Registering suppression attributes for the same problem for more than one visitor results in broken rule suppression. As there were no problems reported by two visitors, this did not lead to any error.

In the third case even conflict resolution fails, in addition to rule suppression. If, after mapping a shared visitor to a problem, another visitor is mapped to the problem, then the final problem id to visitor id mapping will map to the second visitor. When the shared visitor then reports a problem and conflict resolution is performed, it will be wrongly performed for the second visitor, instead of the actual visitor reporting the problem. Thus the problem might be reported even if there is a higher priority active rule. Same as in the second issue, there were no

problems reported by two visitors, and therefore the error did not express. Though, from the documentation of the term project of the preceding group this use case is clearly meant to be supported. [Deicha and Bertschi 2018, p.25]

One of our main goals was to get rid of the mappings in the `GuidelineConflictResolver` and the `IGuidelineMapper` implementations of the guidelines or at least to refactor them into their own class. We began by introducing a new class called `Rule`. The class holds various information regarding a rule. Among them a reference to the guideline it belongs to, the quick fixes for the rule and a shared problem id. The shared problem id replaces the earlier visitor id. This made three of the mappings superfluous, but required a new one to map problem ids to rules.

```
1 public class Rule {
2
3     private String          ruleNr;
4     private IGuideline      guideline;
5     private String          problemId;
6     private IMarkerResolution[] markerResolutions;
7     private boolean         isShared      = false;
8     private String          sharedProblemId = null;
9
10    ...
11 }
```

Listing 2: Member variable declarations of the class `Rule`

As shown in Listing 2, a rule has two ids. One is the problem id, which as the name suggests, refers to the corresponding problem. The other one is the shared problem id, which is shared between all rules that are equivalent, i.e. use the same visitor. It is used by the priority resolution to decide which marker is going to be set. For example: the rules `A4-7-1AvoidLossyConversion` and `M5-0-6AvoidLossyConversion` do not have the same `problemId` but share the same `sharedProblemId`. If a rule has no `sharedProblemId` it means, that it is only relevant for one guideline.

We then set out to divide the `GuidelineConflictResolver` and its helper classes into their responsibilities. We refactored the code retrieving and parsing guideline preferences out of the classes `GuidelineLoader` and `GuidelineConflictResolverFactory` into the class `GuidelinePreferences`. It provides methods to check whether a guideline is enabled and which priority it has. Loading the guideline extensions and storing the guidelines was moved into the class `GuidelineRegistry`. We further moved the mapping of problem ids to rules into the class `RuleRegistry`.

With the conflict (or priority) resolution being the only thing left, we refactored the priority resolution from the classes `GuidelineConflictResolver` and `CodeAnalysatorChecker` into the new class `GuidelinePriorityResolver` and removed the `GuidelineConflictResolver` and its various helper classes.

Now, with the singleton-like `GuidelineConflictResolver` gone, we needed some way to access instances of the various replacement classes. While in Eclipse in general dependency injection is used, it is not supported in CDT. The colloquial solution in CDT is to have a central static instance of a class which then provides access to the various services required. The instance is set when the plug-in is started and disposed again when it is stopped. We therefore added our various services to the `Activator` class of the `CodeAnalysator` plug-in and renamed it to `CodeAnalysatorRuntime`.

Figure 4.1 shows the `GuidelineConflictResolver` and its surrounding classes in the `Guideline` package. It helps to visualize with what we mean by the `GuidelineConflictResolver` being a central class with no real defined purpose.

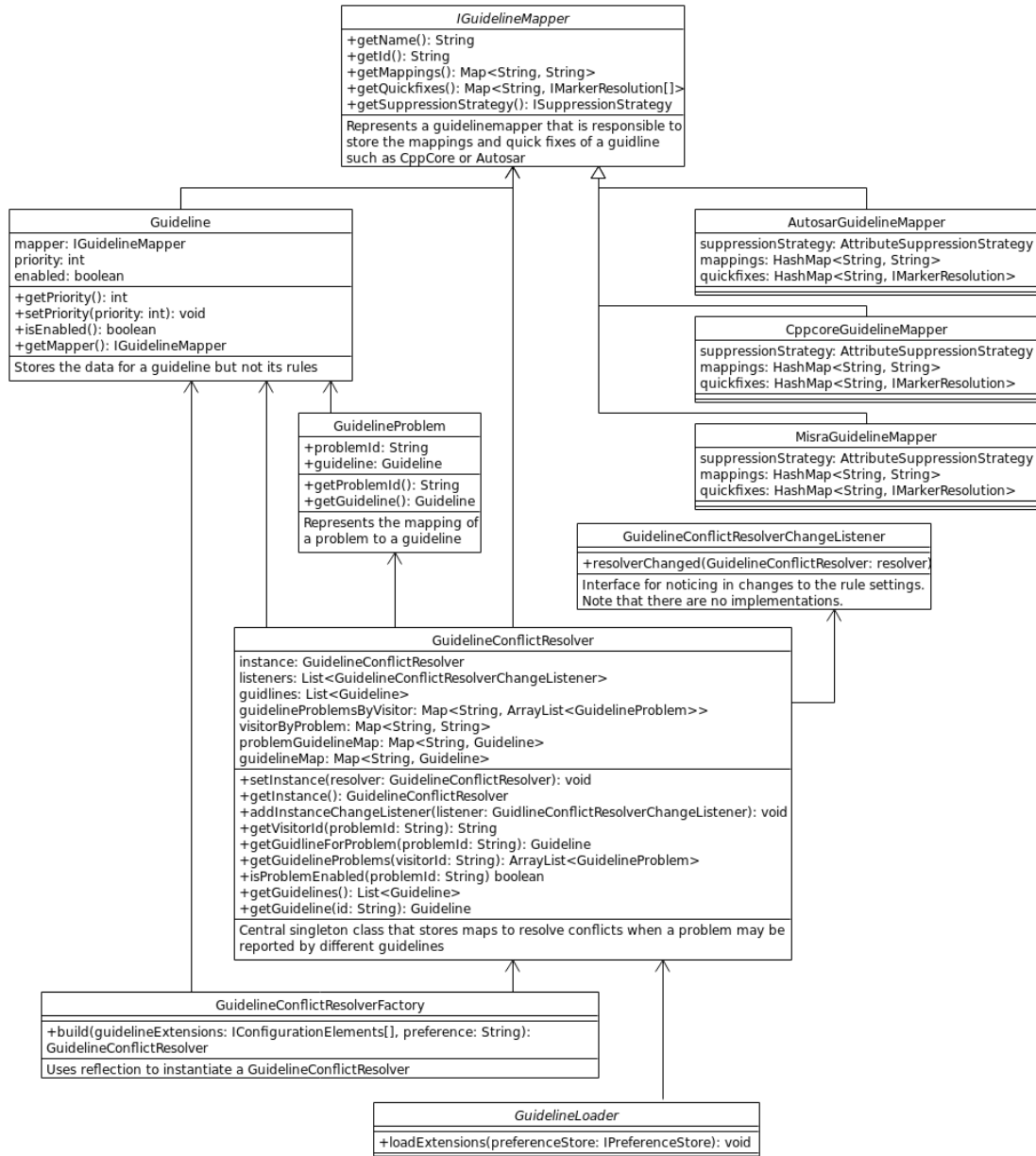


Figure 4.1.: Guideline before refactoring

Figure 4.2 shows the guideline package after removing the **GuidelineConflictResolver** and reassigning the responsibilities. On first glance it might look like we made it more complicated because there are more new classes than we removed. But readability has been greatly improved because it is more clear what each class

does. We also reduced the amount of "empty" classes that only held a little data but did not have any functionality or reason to exist.

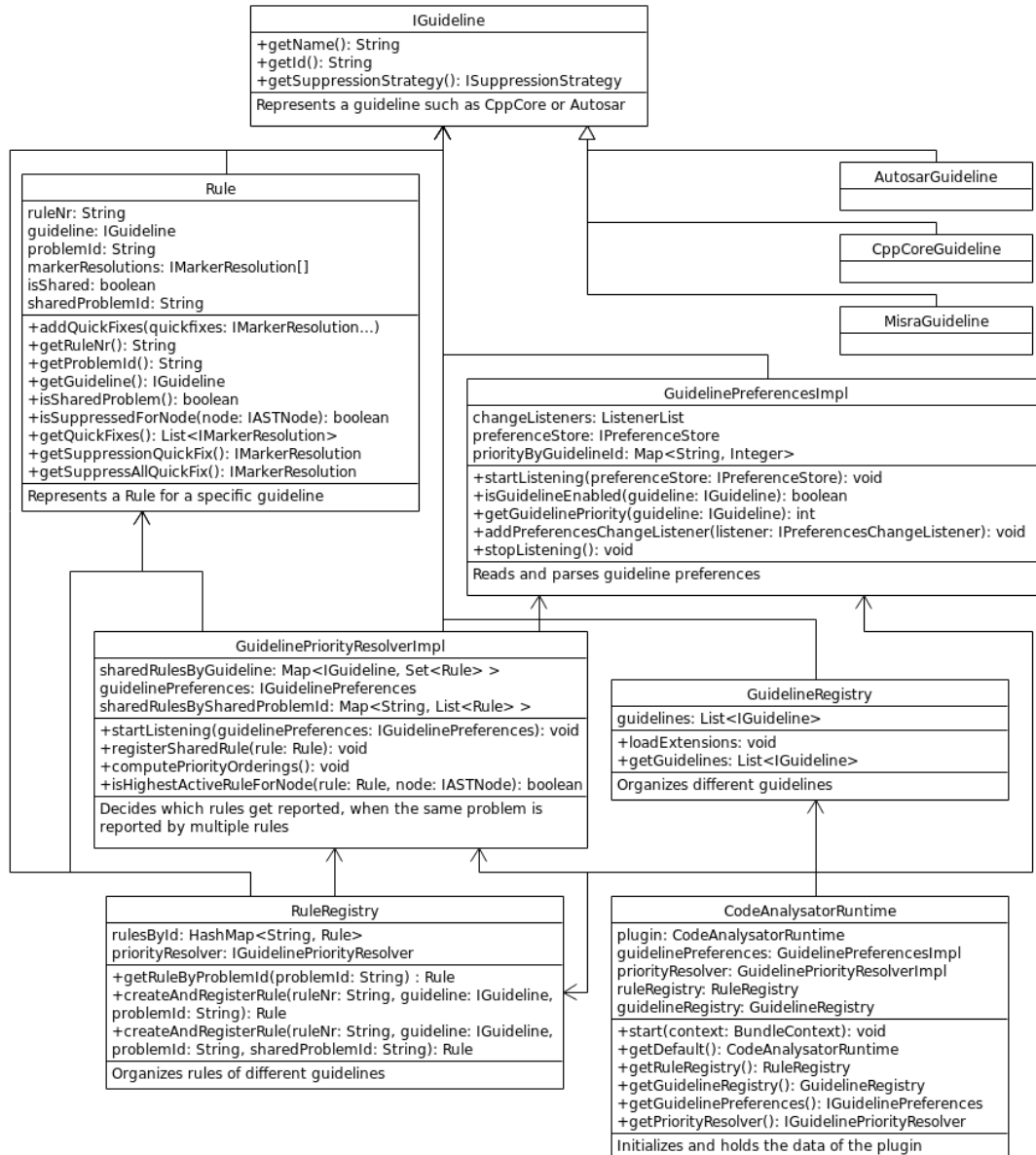


Figure 4.2.: Guideline after refactoring

4.3. Rule Suppression

Many rules in guidelines are not absolute and sometimes, there are good reasons to disregard them. Further, some rules like “A5-1-4: Lambda expressions shall not outlive reference captured objects” require complex analysis, which, due to practical reasons, may return false positives. Due to both reasons, it is desired to be able to suppress rule violations. Otherwise, one runs the risk of a significant rule violation going unnoticed between disregarded ones. Or, given enough insignificant reports, one may abandon the rule checking as a whole.

Suppression of rules in the CodeAnalysator plug-in is implemented using C++ attributes. The advantage of using attributes instead of comments is that there is a syntactic relationship to the element on which the rule is suppressed. This makes it always clear to what the suppression applies and does not potentially break when the source code is reformatted [*attribute specifier sequence(since C++11) - cppreference.com* 2019]. An example of a suppression attribute can be seen in Listing 3.

```
1 [[autosar::suppress("A7-1-6")]] typedef int int_t;
```

Listing 3: Example of a suppression attribute for the rule ”A7-1-6: The typedef specifier shall not be used”

The central class in the suppression mechanism is the `AttributeSuppressionStrategy`, as shown in Figure 4.4. It acts as a registry for `SuppressionAttributes`, mapping visitor ids to suppression attributes. There is one instance per guideline mapper. For every rule a guideline specific suppression attribute needs to be created and registered with the suppression strategy by visitor id. It is not clear, here, why the visitor id is used instead of the problem id, as the suppression strategy instances are unique to the guidelines and there is always a one-to-one mapping between problem id and visitor id.

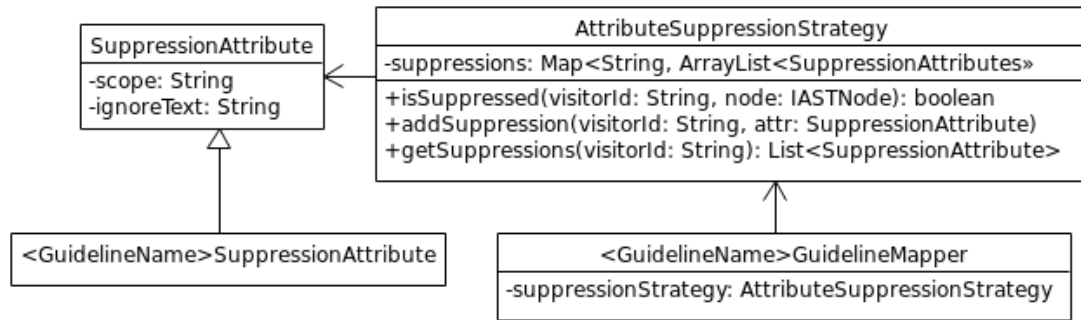


Figure 4.3.: The **AttributeSuppressionStrategy**

The purpose of the guideline specific suppression attribute is to supply the **scope**, which consists of the attribute namespace and identifier for the suppression attribute. In case of AUTOSAR this would be "autosar::suppress". The **ignoreText** specifies the argument for the suppression attribute and is initialized with the rule number in the guideline mapper.

Now to the issues with the current implementation. The suppression mechanism requires a guideline to subclass many classes in order to use it. These subclasses contain just a bit of configuration and hardly any behaviour. This results in many nearly empty classes which do not do much. The suppression mechanism in general is spread over many classes and packages, instead of being contained in one place.

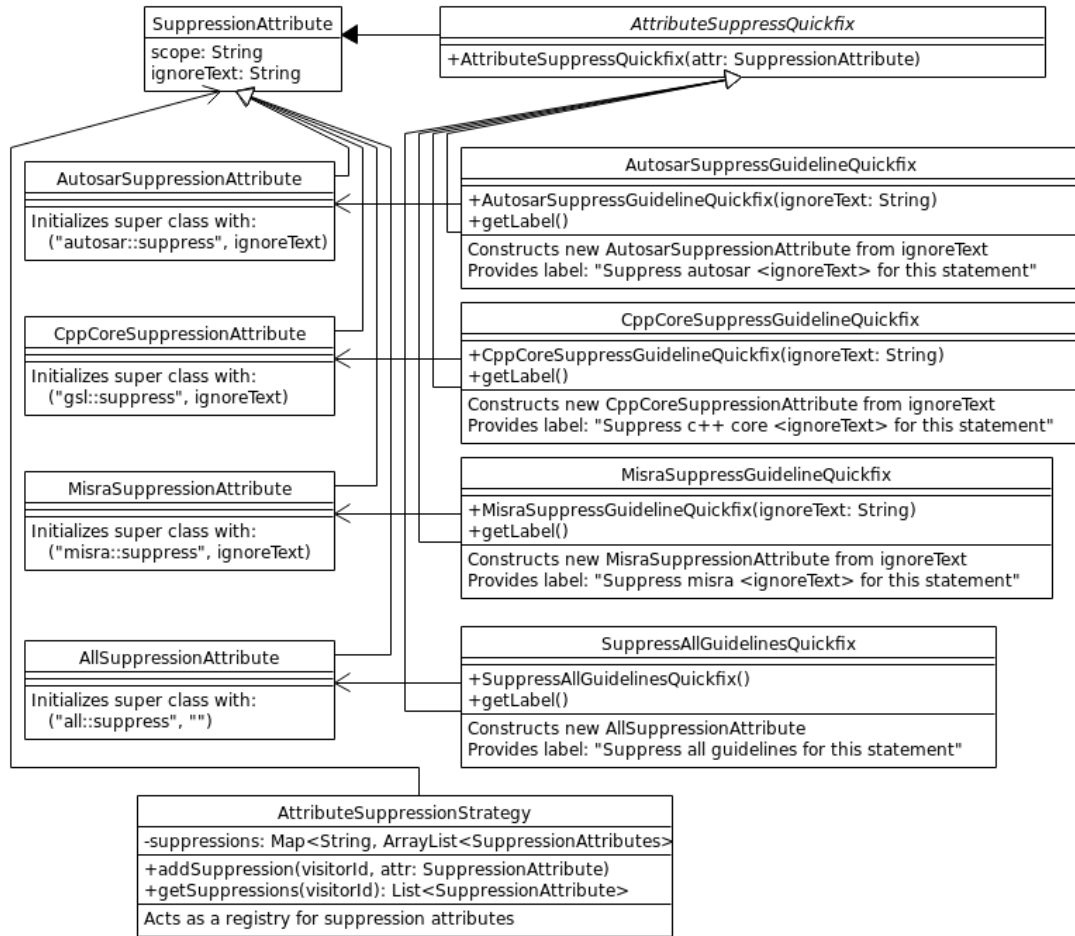


Figure 4.4.: Suppression attributes and quick fixes before refactoring

Figure 4.4 shows the classes related to attribute suppression, the only implemented suppression mechanism, before the refactoring. As you can see, for every guideline the classes `SuppressionAttribute` and `AttributeSuppressQuickfix` are derived, supplying guideline specific configurations but otherwise having no behaviour. The `AttributeSuppressionStrategy` acts as a registry for suppression attributes which are registered by visitor id.

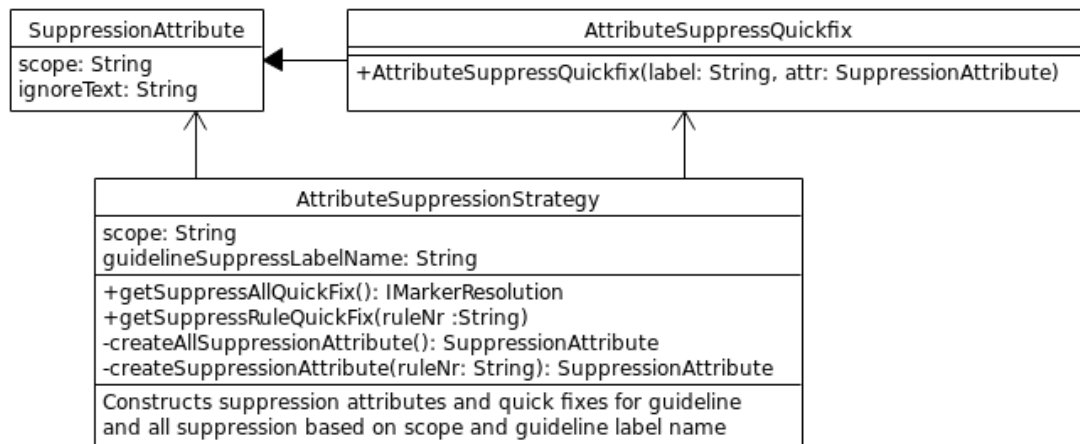


Figure 4.5.: Suppression attributes and quick fixes after refactoring

In the refactoring, **AttributeSuppressionStrategy** was changed to generate suppression attributes instead of storing them. To achieve this a scope argument is passed to the strategy on construction, containing the guideline specific scope previously contained in derivations of **SuppressionAttribute**. With the scope for suppression of all guidelines being hard coded ("all"), this made derivations of **SuppressionAttribute** superfluous. Further, this removes the need to create and register suppression attributes for every visitor in guideline mappers.

Something similar was done for suppress quick fixes. The guideline name to use in the label of the suppression quick fix is passed to the constructor of **AttributeSuppressionStrategy**, which then uses the name to generate suppress quick fixes. The label for the quick fix suppressing all guidelines is hard coded again ("all guidelines"). This change again makes derivations of **AttributeSuppressQuickfix** obsolete.

The result, as you can see in Figure 4.5, is a change in responsibility for the **AttributeSuppressionStrategy** as well as a reduction in the number of classes.

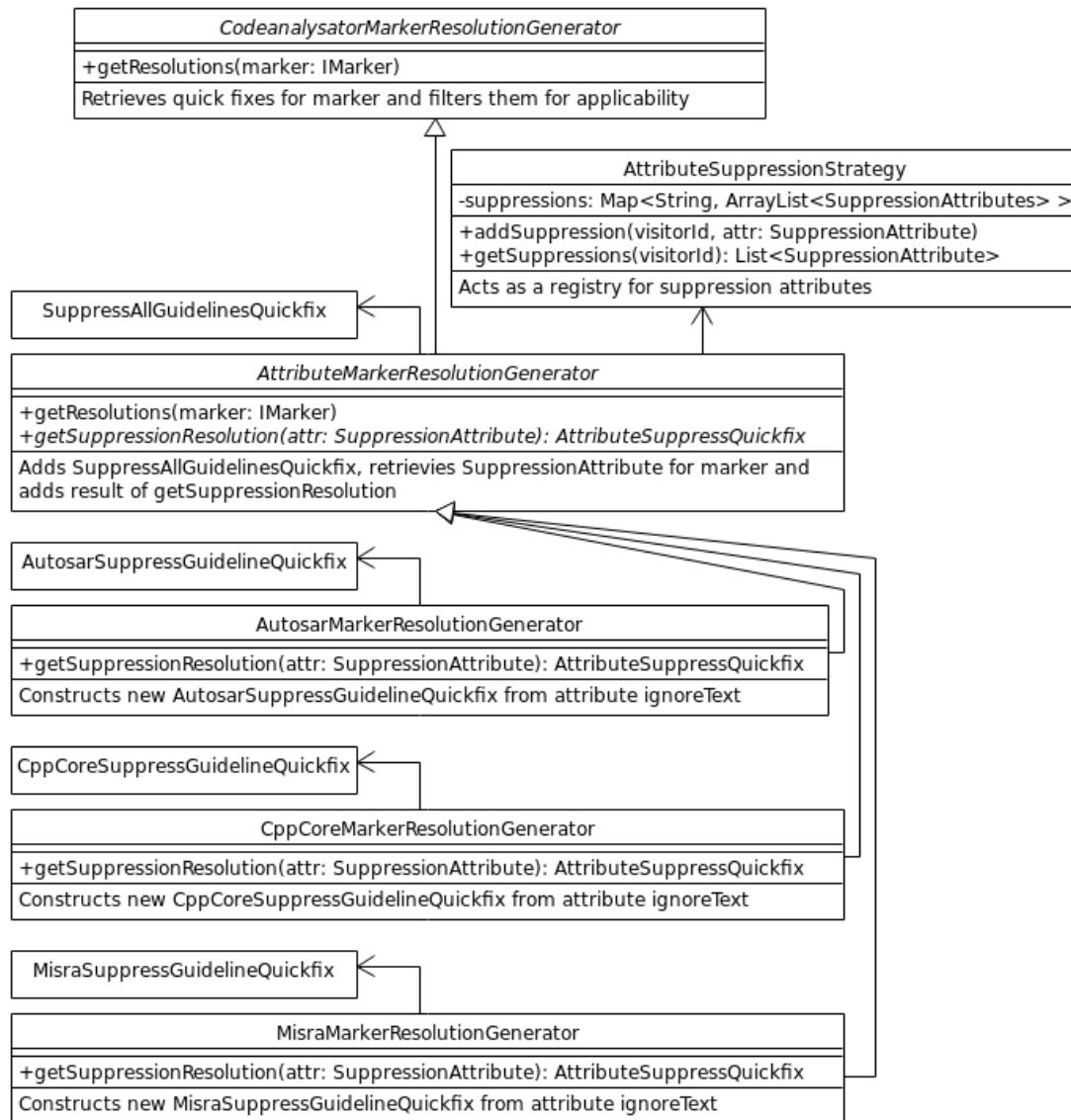


Figure 4.6.: Marker resolution generation before refactoring

In Figure 4.6 you can see the classes related to implementing marker resolution generation using attribute suppression. Every guideline derives from **AttributeMarkerResolutionGenerator**, instantiating the respective suppression quick fix.

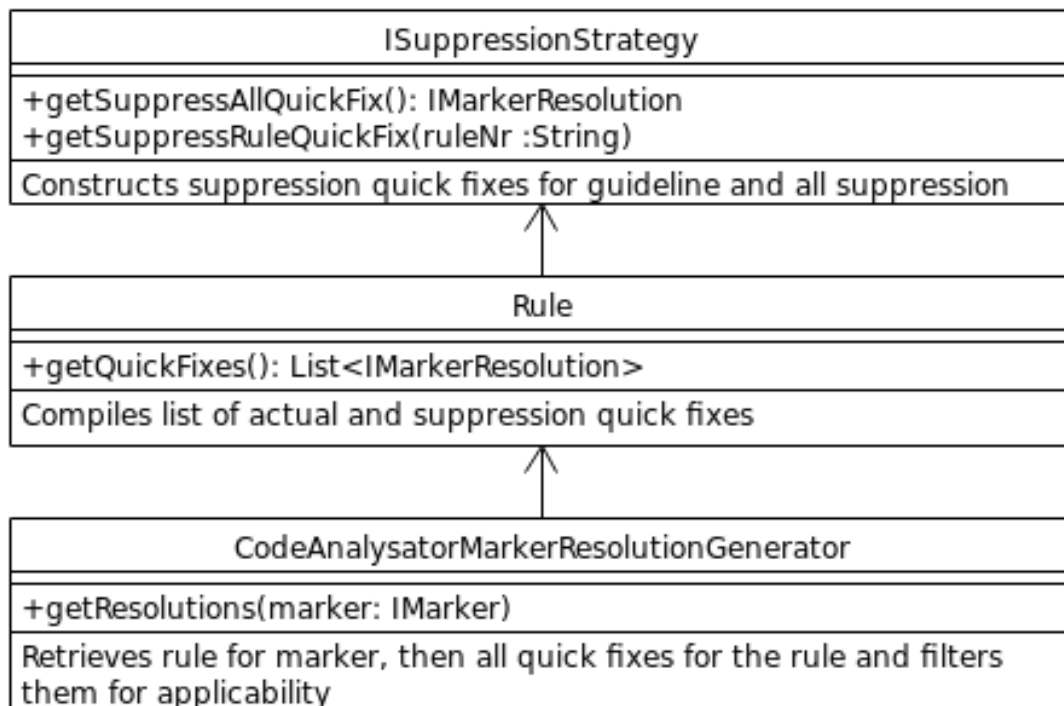


Figure 4.7.: Marker resolution generation after refactoring

Marker resolution generators specific to a guideline are not needed anymore because the suppression quick fixes are now being generated by `ISuppressionStrategy`. The compilation of quick fixes, consisting of actual and suppression quick fixes was moved to the `Rule` class. This class was also newly created in a different refactoring, with the `CodeAnalysatorMarkerResolutionGenerator` retrieving the quick fixes and filtering them based on applicability.

As shown in Figure 4.7, this results in fewer classes while at the same time being more general, as the marker resolution generation code is now independent of the suppression method used, with all suppression specific code belonging to the concrete suppression strategy.

4.4. CodeAnalysatorVisitor Type Parameter

The `CodeAnalysatorVisitor` was parameterized by a type parameter specifying the type of the checker, see Listing 4. A visitor needed a reference to a checker in order to report problems or to check whether a rule is suppressed. The type

parameter was likely meant to allow a visitor private to a guideline to use additional functionality from that guidelines checker. But no such use existed in the CodeAnalysator plug-in and so we replaced uses of the type parameter with CodeAnalysatorCompositeChecker and removed the type parameter.

```
1 public abstract class CodeAnalysatorVisitor
2     <TChecker extends CodeAnalysatorChecker>
```

Listing 4: CodeAnalysatorVisitor declaration with type parameter

4.5. CodeAnalysator Visitors

The CodeAnalysator plug-in contained two visitor base classes, CodeAnalysatorVisitor and SharedVisitor. The only distinction or purpose of the SharedVisitor was being a base class for visitors which report a single problem id. As there was only a single visitor which reported more than one problem id, we decided to split that visitor and have visitors always only report a single problem id. This made the distinction between the two visitor base classes superfluous and we therefore merged SharedVisitor into CodeAnalysatorVisitor.

4.6. Performance Measurement

The CodeAnalysator plug-in contained some code for performing performance measurements. The measurement was not very accurate, did not make it easy to measure the execution times under realistic conditions and we intended to use a profiler for performance measuring. We therefore removed the performance measuring code from CodeAnalysatorChecker and the class CheckerMeasurement.

4.7. Rule Reporting

Visitors initially contained a reference to CodeAnalysatorChecker in order to report problems and check whether a rule is suppressed. This led to a circular dependency between checkers and visitors. We refactored the code for reporting rules into a RuleReporter class and replaced the reference with one of its base class AbstractIndexAstChecker to break up the circular dependency.

4.8. Rule Enablement Checking

The check for which rules are enabled was not working i.e. Codan returned true even for disabled problems (rules), which resulted in visitors being run unnecessarily for disabled rules. The error was checking `isEnabled` on the problems retrieved directly from the checker which always has the default value for a problem. Instead we had to query the respective problem profile and check `isEnabled` on the problems returned from the problem profile. A problem profile represents the problem settings for a specific resource and therefore contains the actual values of `isEnabled` from workspace and project settings.

4.9. CodeAnalysator Checkers

The classes `CodeAnalysatorChecker` and `CodeAnalysatorCompositeChecker` are the base classes for the various checkers in the CodeAnalysator plug-in. With the preceding refactorings completed there was no behaviour left in the `CodeAnalysatorChecker`. The only class deriving from `CodeAnalysatorChecker` was `CodeAnalysatorCompositeChecker`, from which every guideline specific checker derived. This is why we replaced references to `CodeAnalysatorChecker` with `AbstractIndexAstChecker` and renamed the `CodeAnalysatorCompositeChecker` into `CodeAnalysatorChecker`. The purpose of the `CodeAnalysatorChecker` class now is to filter the visitors created by its subclasses based on whether the rule they report is enabled and to run those who are.

4.10. BaseQuickFix

The class `BaseQuickFix` is the abstract base class for implementing quick fixes in the CodeAnalysator plug-in. It has been taken over from the GSLator plug-in. This class was not designed to support resolving bindings in the `isApplicable` method, requiring derived classes that need to do so, to acquire an index lock themselves. Further, it contained some unused members. Also, the protected node factory member variable was defined with the concrete type, requiring derived classes to suppress restriction warnings in order to use it. Lastly, every quick fix defined a label instance variable, initialized with a constructor parameter.

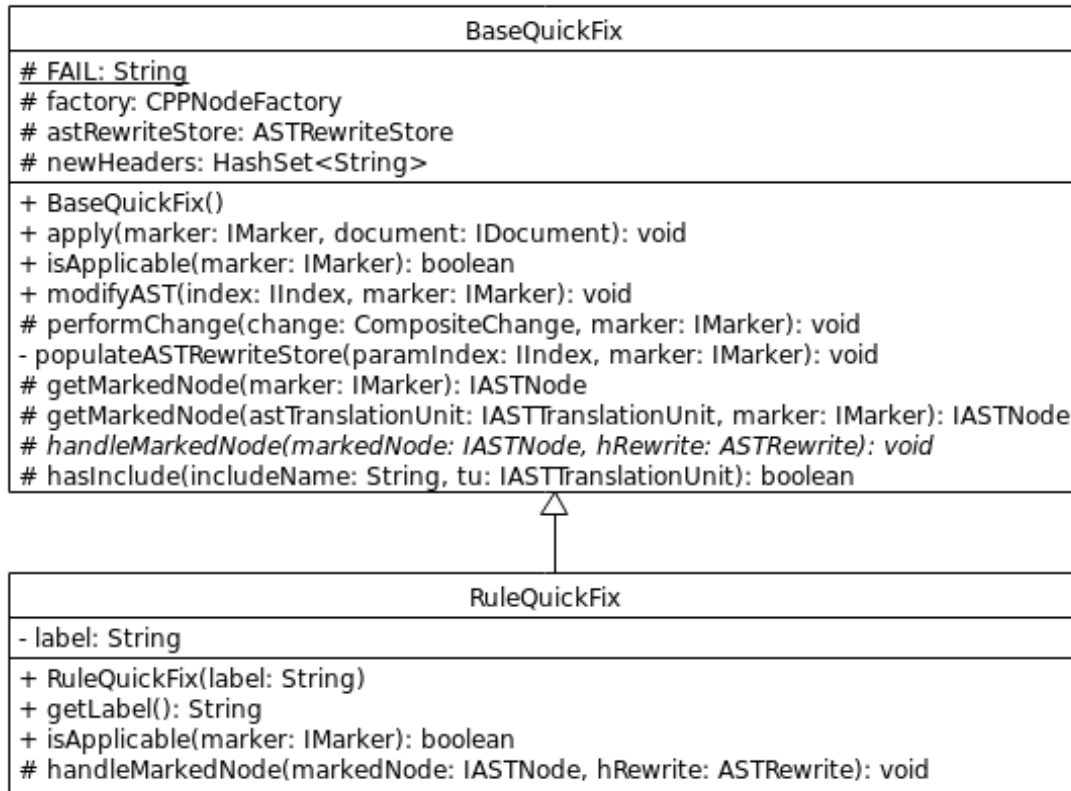


Figure 4.8.: BaseQuickFix and derived RuleQuickFix before refactoring

First, we introduced the abstract method `canHandleMarkedNode` for derived classes to implement. It replaces `isApplicable` and is passed a node retrieved with locked index, allowing bindings to be resolved. The index locking and retrieval is performed in `isApplicable` in the `BaseQuickFix` which then calls `canHandleMarkedNode`. Further, we removed unused or obsolete members (`newHeaders`, `hasInclude`, `getMarkedNode` overload) and refactored instance variables away (`populateASTRewriteStore`, `performChange` and `astRewriteStore`). We also changed the type of the node factory instance variable to `IBetterFactory`, made it static and refactored the quick fixes to use it. Lastly, we changed the `BaseQuickFix` constructor to take a `String` label and implemented the `getLabel` method in the base class, removing it from derived classes.

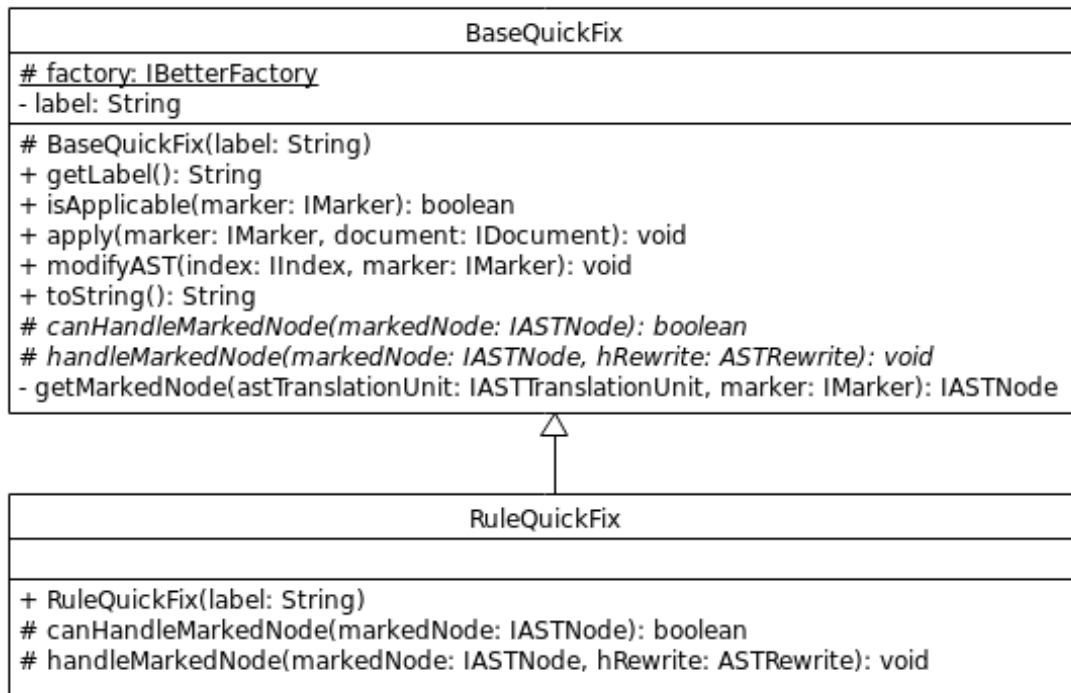


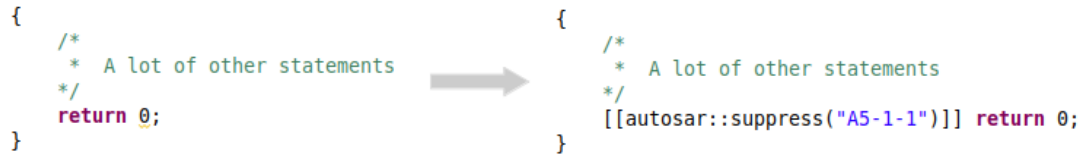
Figure 4.9.: BaseQuickFix and derived RuleQuickFix after refactoring

4.11. AttributeSuppressQuickfix

The AttributeSuppressQuickFix is the quick fix used for suppression of a rule using C++ attributes. See Section 4.3 for more information on rule suppression.

4.11.1. Remove Bug Workaround

As documented in [Deicha and Bertschi 2018, p.28], there was a bug in CDT which prevented the rewrite from working correctly. Because of this bug it was not possible to rewrite the node itself but instead the parent node had to be rewritten. As can be seen in Figure 4.10, the whole code block around the statement that should be suppressed has to be rewritten. While this yields the correct results, it can be inefficient. Often, the parent of a statement is a compound statement with many other children. They all have to be copied and then rewritten.



```

{
    /*
     * A lot of other statements
     */
    return 0;
}

→

{
    /*
     * A lot of other statements
     */
    [[autosar::suppress("A5-1-1")]] return 0;
}

```

Figure 4.10.: The code to the left is rewritten with the code to the right

We tested whether it would still work if we removed the workaround and it did. So we removed the unnecessary code. To ensure the correctness of the refactoring, we tested it manually but also wrote automated tests for suppressing all rules and for suppressing a single rule.

4.11.2. Relation to BaseQuickFix

The class `AttributeSuppressQuickFix` contained copy-pasted sections from `BaseQuickFix`, presumably due to the workaround. A lot of functionality is similar to other quick fixes. There is a class called `SetAttributeQuickFix` in the `GSLator` which has a similar functionality to the `AttributeSuppressQuickFix`. In the `CodeAnalysator` plug-in the class extended directly `AbstractAstRewriteQuickFix` while the version from `GSLator` extended `BaseQuickFix`. Since `AttributeSuppressQuickFix` shared a lot of functionality with `BaseQuickFix`, we assumed, that it could be possible to have it extend `BaseQuickFix` instead. In summary, this means the goal of this refactoring was to find out whether it was possible to have `AttributeSuppressQuickFix` extend `BaseQuickFix` and, if possible, refactor it. As it turned out it was possible. We made the class `AttributeSuppressQuickFix` extend `BaseQuickFix` instead of `AbstractAstRewriteQuickFix` and refactored the rewrite logic from `modifyAst()` to `handleMarkedNode()`. Lastly, we removed the parts copied from `BaseQuickFix`, which had now become obsolete.

4.12. Package Structure

The `CodeAnalysator` plug-in contained many interdependent `util` packages, leading to two-way dependencies between packages. These needed to be reorganized. The package structure also contained inconsistencies, including package names `quickfix` and `quickfixes` and `util` and `helper`. To analyze the package structure we used `STAN` [*STAN - Structure Analysis for Java* 2019].

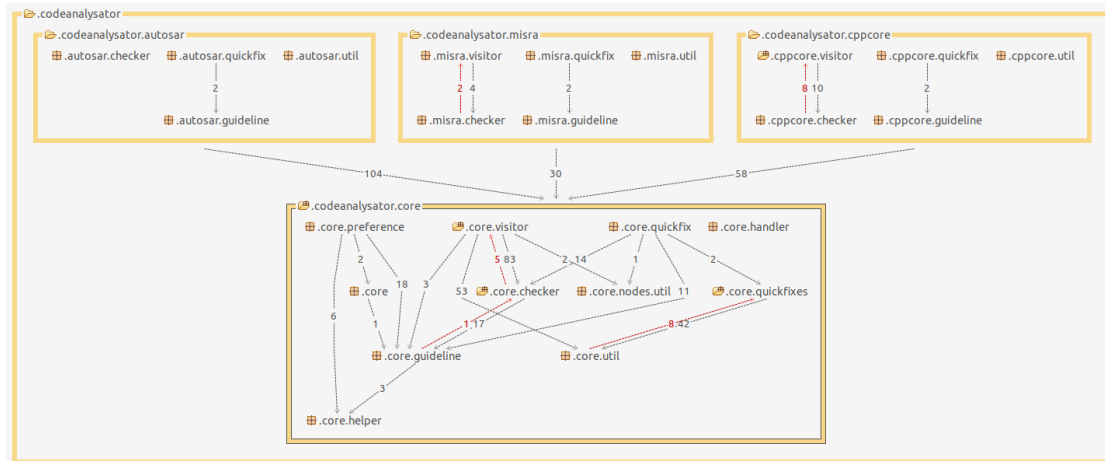


Figure 4.11.: The package structure before the refactoring

The issues with cyclic dependencies are, among others, that they are more difficult to understand, make the code less reusable and indicate that a certain concept has not been understood or discovered yet. The following list shows all the problems we approached in this task. Most of them can be derived from looking at Figure 4.11. Some others are not visible there because there is not enough space to include a fully expanded image.

- Two-way dependencies between `misra.visitor` and `misra.checker`
- Two-way dependencies between `cppcore.visitor` and `cppcore.checker`
- Two-way dependencies between `core.visitor` and `core.checker`
- Two-way dependencies between `core.checker` and `core.guideline`
- Two-way dependencies between `core.quickfixes` and `core.util`
- Inconsistency: Many different `util` packages with many different purposes
- Confusing naming: In `codeanalysator.core` there is a package `quickfix` and one named `quickfixes`

In order to solve the aforementioned issues we performed the following refactorings. To decouple the packages `visitor` and `checker` we introduced a `RuleReporter` class, as described in Section 4.7.

We also moved the `CodeAnalysatorMarkerResolutionGenerator` class into its own `markerresolution` package, as it was the only class in the `quickfix` package depending on the packages `core` and `guideline` and did not have any dependencies in common with the other classes in the package.

We further introduced a new package called **suppression** and moved the suppression logic there, which was previously spread between the packages `core.checkers.util` and `core.visitor.util`.

Then we moved the remaining classes in the various sub-util packages to the root util packages of the bundles, as they were mostly interdependent.

Lastly, the `core` package contained visitors and quick fixes specific to AUTOSAR. We had developed them within the preceding term project and had placed them in the `core` for simplicity. Now, we moved them to their appropriate place, the `autosar` package.

The resulting package structure can be seen in Figure 4.12. All cyclic dependencies have been removed and the package structure has become simpler to understand.

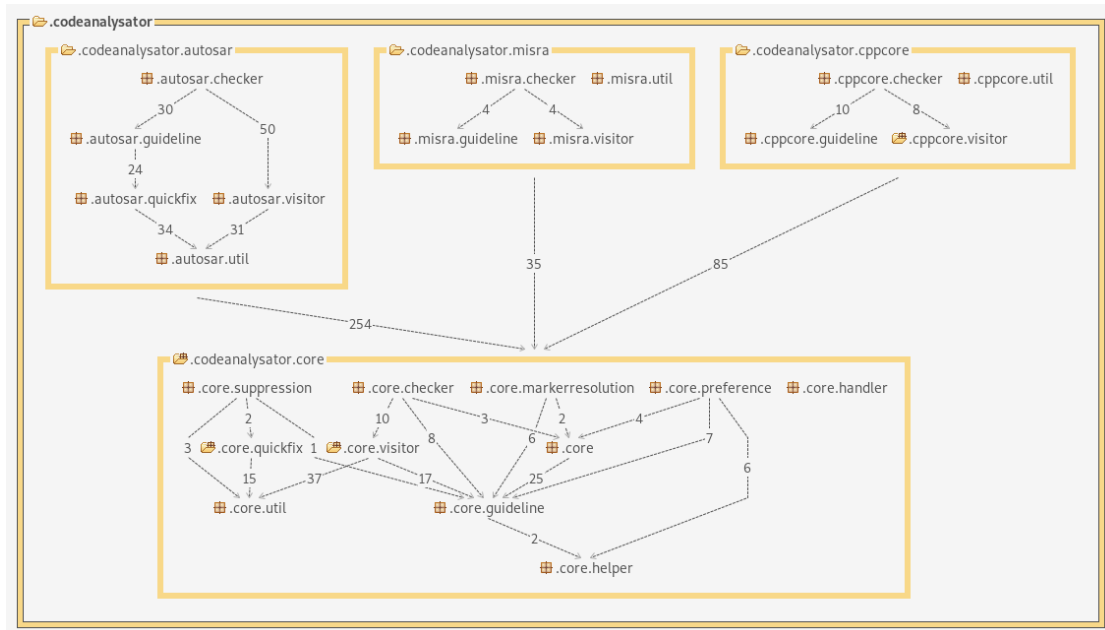


Figure 4.12.: The package structure after refactoring

4.13. Checker and Quick Fix Tests

Checker and quick fix tests derive from guideline specific `AbstractCheckerTest`, respectively `AbstractQuickFixTest` classes. The guideline specific classes are in both cases duplicates of each other, except for the `getPreferenceConstants` method which returns a different `IdHelper` class for each guideline. Figure 4.13 and Figure 4.14 show the class hierarchies for the checker and quick fix tests respectively. Note the similarities between the base classes.

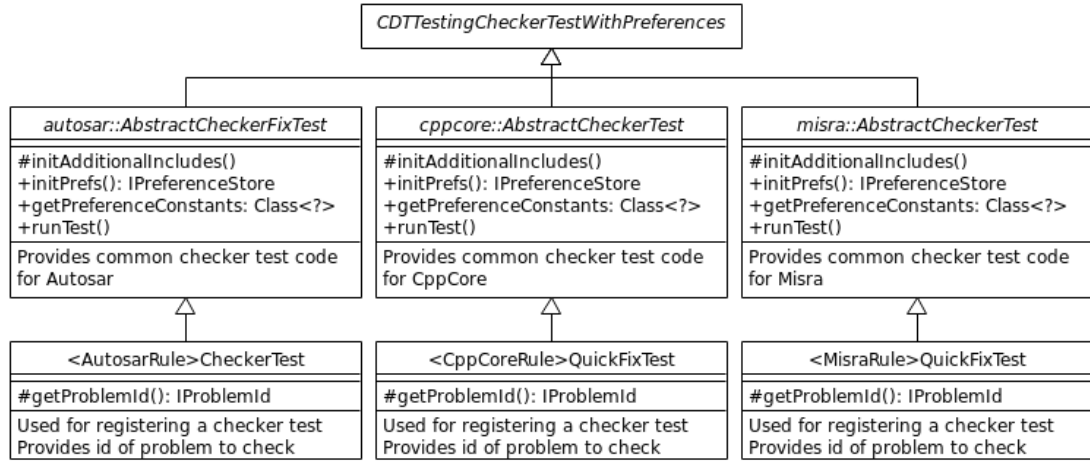


Figure 4.13.: Checker tests before refactoring

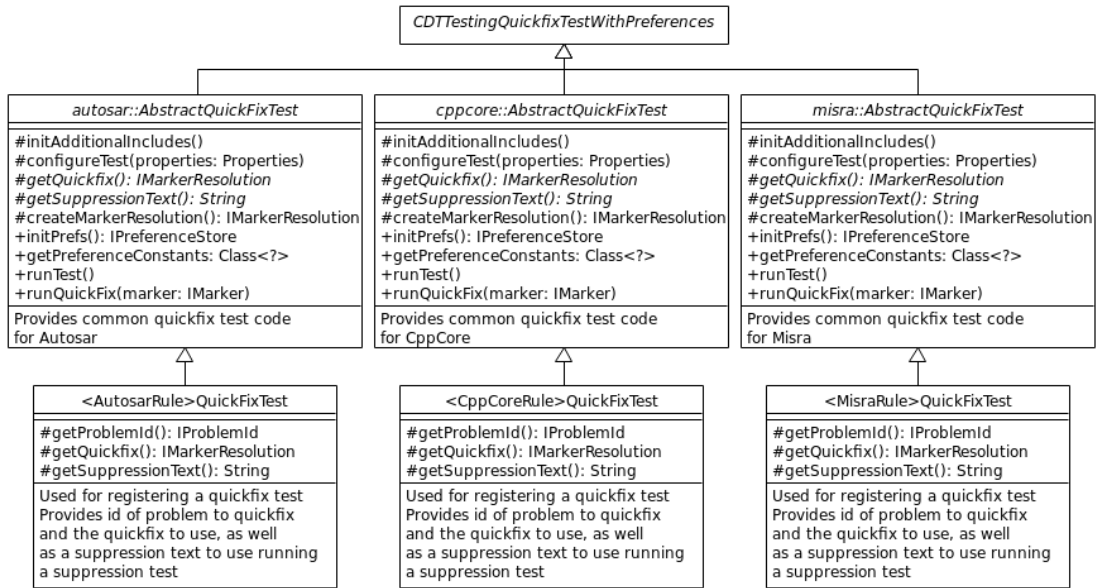


Figure 4.14.: Quick fix tests before refactoring

In the refactoring we extracted the common code into the `CodeAnalysatorCheckerTestBase` and `CodeAnalysatorQuickFixTestBase` classes, with the guideline specific base classes only retaining the `getPreferenceConstants` method. We further

refactored checker and quick fix tests to use the newly-created `Rule` class, simplifying them and reducing duplication. The result can be seen in Figure 4.15 and Figure 4.16.

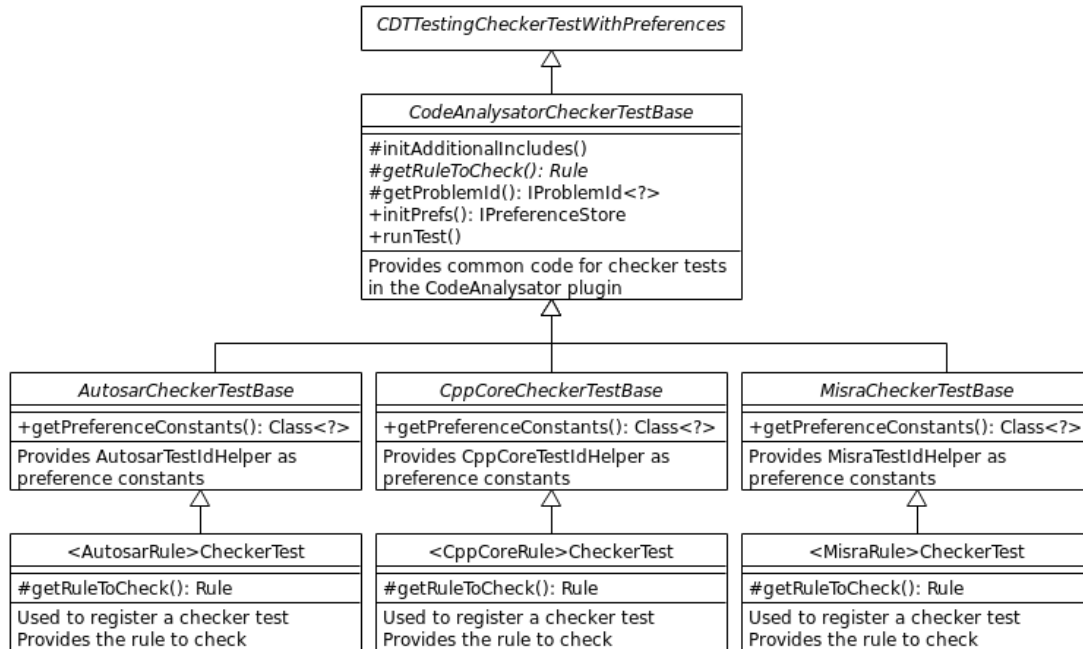


Figure 4.15.: Checker tests after refactoring

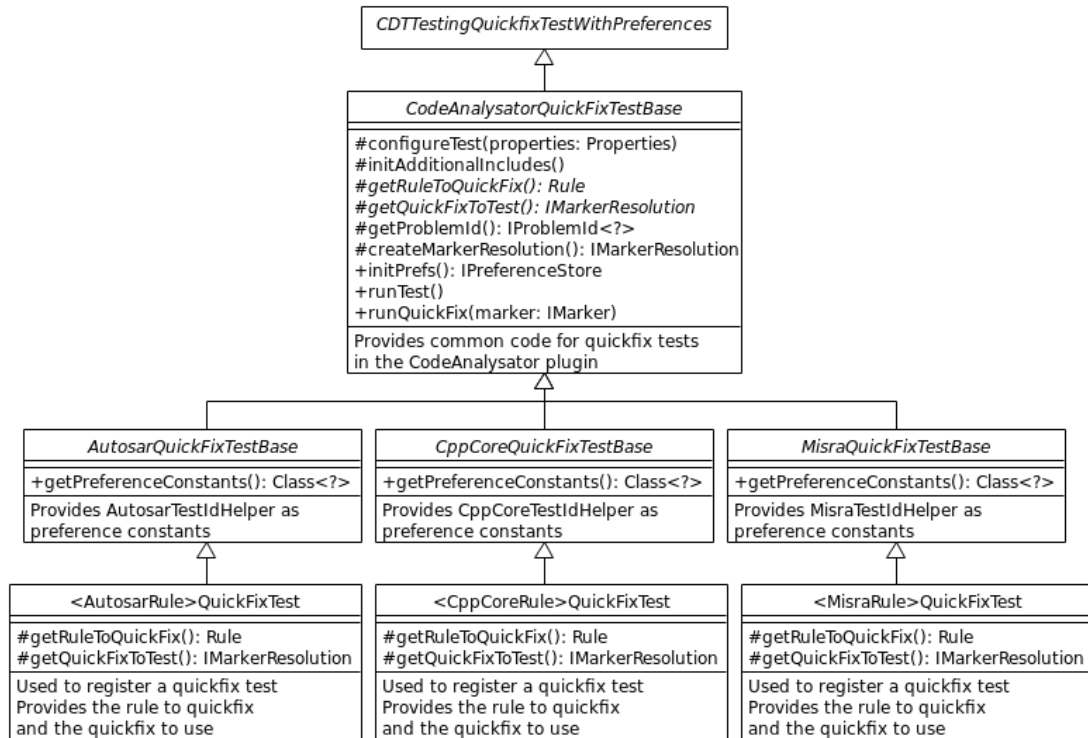


Figure 4.16.: Quick fix tests after refactoring

4.14. Refactor Old Tests

Problem The old tests that were copied from the GSLator plug-in have the problem of being badly named and written. The first problem is, that the names are not written as sentences but as long camel cased words, which is marked by Eclipse. This has a large usability impact, due to the files having many tests and these markers for some reason being computationally expensive, resulting in lags. The second problem is, that a lot of test cases are combined into single tests. This makes it hard to read the tests and determine where a problem lies.

Solution We split camel cased test names into sentences. We also split too large tests into smaller tests and named them with fitting descriptions. This improves the performance, usability and in particular the readability.

4.15. Ordering of Rules

With the new guideline system introduced in Section 4.2, we noticed that the rules are ordered differently in different files such as the checker, the guideline or the `fragment.xml`. Similarly, rules were not displayed in the correct order in the overview of the C++ Code Analysis page. Refer to Figure 4.17 to see that for example the rules A10-XX appear before A5-XX.

Name	Severity
▼ <input checked="" type="checkbox"/> AUTOSAR Guidelines	
<input checked="" type="checkbox"/> A10-1-1: Class shall not be derived fr	⚠ Warning
<input checked="" type="checkbox"/> A10-2-1: Do not hide non-virtual me	⚠ Warning
<input checked="" type="checkbox"/> A10-3-1: Virtual functions shall have	⚠ Warning
<input checked="" type="checkbox"/> A10-3-2: Each overriding virtual func	⚠ Warning
<input checked="" type="checkbox"/> A10-3-3: Do not introduce virtual fur	⚠ Warning
<input checked="" type="checkbox"/> A10-3-5: A user-defined assignment	⚠ Warning
<input checked="" type="checkbox"/> A12-0-1: Avoid redundant default op	⚠ Warning
<input checked="" type="checkbox"/> A12-0-1: Missing special member fu	⚠ Warning
<input checked="" type="checkbox"/> A13-5-3: Avoid conversion operators	⚠ Warning
<input checked="" type="checkbox"/> A4-7-1: Avoid lossy (narrowing, trun	⚠ Warning
<input checked="" type="checkbox"/> A5-0-2: The condition of an if or iter	⚠ Warning
<input checked="" type="checkbox"/> A5-1-1: Literal values shall not be us	⚠ Warning

Figure 4.17.: The C++ Code Analysis preferences page before renaming the rules

To bring some order and system into these files we ordered all listings of classes alphabetically. This has also the advantage of not introducing a merge conflict every time two people add a visitor or quick fix. We also changed the labels in the

preferences. We added padding zeros into the names to ensure correct ordering. For example the rule A7-1-6 would now be renamed to A07-01-06. Figure 4.18 shows that the names are now ordered correctly in the preferences page.

Name	Severity
▼ AUTOSAR Guidelines	
<input checked="" type="checkbox"/> A04-07-01: Avoid lossy (narrowing, truncation) conversions	Warning
<input checked="" type="checkbox"/> A05-00-02: The condition of an if or if-else statement shall not be a constant expression	Warning
<input checked="" type="checkbox"/> A05-01-01: Literal values shall not be used in constant expressions	Warning
<input checked="" type="checkbox"/> A05-01-02: Variables shall not be implicitly converted to bool	Warning
<input checked="" type="checkbox"/> A05-01-03: Parameter list (possibly empty) shall not be a constant expression	Warning
<input checked="" type="checkbox"/> A05-01-04: A lambda expression object shall not be a constant expression	Warning
<input checked="" type="checkbox"/> A05-01-06: Return type of a non-void function shall not be a constant expression	Info
<input checked="" type="checkbox"/> A05-01-08: Lambda expressions shall not be used in constant expressions	Info
<input checked="" type="checkbox"/> A05-16-01: The ternary conditional operator shall not be used in constant expressions	Warning
<input checked="" type="checkbox"/> A06-04-01: A switch statement shall have at least two case-clauses, distinct from the default label	Warning

Figure 4.18.: The C++ Code Analysis preferences page after renaming the rules

4.16. A6-4-1: A switch statement shall have at least two case-clauses, distinct from the default label.

We performed significant refactorings in order to improve the readability and maintainability of the switch analysis code and the quick fix. During the refactoring, in addition to improving the code, we also found and fixed some errors. These were mostly due to not fully understanding the non-strictness of C++ Switch statements. There was also a feature left-over from the term project, i.e. omitting to insert a compound statement surrounding the replacement code when not needed.

4.16.1. Switch statement with non-compound statement as direct child

Problem The switch analysis does not correctly handle the case, when a switch statement contains a direct child statement which is neither a compound statement, case statement nor default statement, See Listing 5. As a result of this, the quick fix deletes the whole switch statement without a replacement, instead of declaring itself not applicable.

```
1 switch (0)
2     break;
```

Listing 5: Switch with neither a compound, case nor default statement as child

Analysis CDT automatically inserts a compound statement into the AST for switch statements containing a case statement or default statement as a direct child. This allows us to treat all switch statements as containing a compound statement. Except, if the direct child is not a case statement or default statement, CDT adds the child as the direct child of the switch statement. CDT automatically inserting a compound statement initially led us to believe we can always expect a compound statement as the body of a switch statement when it is not empty. Until we found out this is only the case for case or default statements. [*switch statement* - *cppreference.com* 2018]

Solution Extended switch handling for switch statements without a compound statement body. If a switch statement does not contain a compound statement in the AST, the single direct child is treated as if it were contained in a compound statement for the purpose of that analysis.

4.16.2. Surrounding compound statement

Problem The quick fix for this rule always adds a surrounding compound statement around the replacement code for the switch, because of possible identifier conflicts with identifiers introduced in switch scope. This is undesirable, because the vast majority of time there is no such conflict (hiding a variable of an outer scope is often even considered bad style) and the unnecessary compound statement is intrusive. Table 4.1 shows an example of an unnecessary compound statement being added.

Before Quick Fix		After Quick Fix	
1	<code>switch (variable) {</code>	1	<code>{</code>
2	<code>case 0:</code>	2	<code>if (variable == 0) {</code>
3	<code>return 0;</code>	3	<code>return 0;</code>
4	<code>default:</code>	4	<code>} else {</code>
5	<code>return 1;</code>	5	<code>return 1;</code>
6	<code>}</code>	6	<code>}</code>
		7	<code>}</code>

Table 4.1.: Example of the quick fix adding a surrounding compound statement

Analysis In order to be able to omit the surrounding compound statement when it is not needed, we need to know whether there would be an identifier conflict in the surrounding scope without it. An identifier conflict would occur if a top-level declaration in the replacement code would re-declare an identifier already existing in the surrounding scope, thereby shadowing the existing identifier if it is defined in a higher scope or lead to a compiler error if is defined in the surrounding scope. The only allowed type of declaration in a block is a simple declaration. As a switch statement can only occur inside a block, we only need to check for simple declarations. A simple declaration can contain multiple declarators, each declaring an identifier. [*Declarations* - [cppreference.com](#) 2019] [*switch statement* - [cppreference.com](#) 2018] [*Statements* - [cppreference.com](#) 2019]

Solution The quick fix has been extended with a check whether the replacement code re-declares identifiers in the top-level scope. A visitor is run on the replacement code, visiting all declarators and collecting their identifiers (names), skipping child compound statements, which introduce a child block scope. Then each collected identifier is tried to be resolved in the surrounding scope, to check whether it exists. If none of the identifiers exist in the surrounding scope, the surrounding compound statement is omitted and the replacement code is inserted directly in place of the switch statement. Otherwise, the switch statement is replaced by a compound statement containing the replacement code.

4.16.3. Further errors

While fixing the issue above and implementing the new feature the following further errors were found and fixed:

- Checker reporting switches with nested case-clauses: Extended to check nested statements
- Quick Fix failing on switches with nested case-clauses: Made inapplicable
- Quick Fix failing on switches with nested break-statements: made inapplicable
- Quick Fix omitting last return-statement in clauses: Fixed clause parsing

4.17. A8-5-3: A variable of type auto shall not be initialized using `{}` or `=` braced-initialization.

Problem The implementation of the quick fix for this rule has a large overlap in functionality with the `DeclarationGeneratorImpl`. This redundancy results in unnecessary maintenance effort and duplicated code.

Analysis Large parts of the quick fix overlap with the `DeclarationGeneratorImpl`, but are not identical, as they are interleaved with code specific to the quick fix. The implementation of `DeclarationGeneratorImpl` supports more cases of types and has a more complete declaration generation, but the implementation of the quick fix contains better support for template instances with non-type template arguments. Replacing the implementation of declaration generation in the quick fix with `DeclarationGeneratorImpl` would result in the tests depending on this improved template support failing.

Solution As a temporary solution a local copy of the `DeclarationGenerator` has been created, called `TemplateExtendedDeclarationGeneratorImpl`, with the template support improvements added. This makes it possible to move the improvements upstream in the future. The quick fix has been refactored to use this new local version for declaration generation.

4.18. Conclusion

Now, that we have completed our code improvement, we will come to our conclusion. We have performed various refactorings on the CodeAnalysator plug-in and thereby simplified it and made it easier to understand. But there are still some issues left. In the first part of this section we will reflect on what we have done and the results. In the second part we will propose a list of possible future work.

4.18.1. Reflection

During our preceding term project on the same plug-in we became aware of certain issues and other things that could be improved. In this part of our bachelor thesis we intended to address these issues and we believe we have been largely successful at that. There are still issues or areas to improve on left, but those were simply out of the scope of our code improvement for this thesis. Besides many smaller refactorings there were two major refactorings, also the most significant. We broke up the `GuidelineConflictResolver` and its related classes according to their responsibilities. And we reorganized the suppression logic.

The `GuidelineConflictResolver` and its related classes divided the initialization logic between them without a clear principle and the `GuidelineConflictResolver` contained the majority of the core logic of the CodeAnalysator plug-in. We redivided these classes along responsibilities, for example extracting a class `GuidelinePreferences` for reading and parsing the preferences or `RuleRegistry` for mapping problem ids to rules. While this increased the number of classes it also increased the cohesion and made it clear what belongs to what.

The second major refactoring was concerned with the suppression logic. We removed the need for many sub classes whose only purpose was to configure their base classes and were successful in general in simplifying rule suppression. Among other things, we removed the need to register `SuppressionAttributes` for each rule with the `AttributeSuppressionStrategy`, instead generating everything when needed based on the rule.

We believe that with these two major refactorings and the many smaller refactorings we improved the architecture of the CodeAnalysator plug-in and contributed to the understandability and therefore maintainability of the code.

4.18.2. Proposed Future Work

While we performed many refactorings and changes in the scope of our bachelor thesis, there are things leftover we would have wanted to do or believe should be done. In the following we will give a list of proposed future work. It contains areas of possible improvement as well as some unresolved errors.

Mono Checker Currently, there is one checker per guideline. Each checker runs all the visitors for the rules of its guideline. In case a rule is common to multiple guidelines and the visitor is shared, the `GuidelinePriorityResolver` is used to determine which rule should be reported. Regardless of this, the visitor is run for each checker.

An alternative would be a single checker running all visitors for all guidelines and part of the `core` of the CodeAnalysator plug-in. The visitors would report abstract

identifiers. Guidelines would specify mappings from these abstract identifiers to concrete rules. A class similar to the `GuidelinePriorityResolver` would then determine the highest active mapping and report that problem corresponding to that rule.

The advantages would be a simpler priority resolution and a conceptually simpler problem reporting. Mapping found issues to the highest active rules instead of mapping concrete rules to abstract issues and then checking whether it is the highest active one. There would also be a performance improvement as visitors are only run once and only a single checker is run. On the downside, a new extension-point would have to be defined to keep the problem definitions in the `fragment.xmls`, as the Codan extension-point does not allow checkers to be extended. The `CodeAnalysator` plug-in would then have to dynamically register those categories and problems to the mono checker.

Double Suppress All The Suppress All quick fix is offered as a resolution to all rules. It suppresses all rules of all guidelines for that node. Now, when two rules are reported on the same node (or overlapping range) the Suppress All quick fix is offered twice. This is a result of marker resolution working per marker (and therefore rule) and then the resolutions of all markers as well as other quick assists being merged and offered to the user together. This is part of CDT and it is not clear how it should be amended. While it is annoying it has no other effects and does not break the code.

Pushing Upstream The `CodeAnalysator` plug-in contains a local copy of the `DeclarationGeneratorImpl` with our added modifications, which is undesirable. (Initially named `TemplateExtendedDeclarationGeneratorImpl` though in testing further extended and renamed `CodeAnalysatorDeclarationGeneratorImpl`). Our modifications and fixes would most likely be useful in CDT as well and would ideally be merged upstream to avoid having to maintain a local version.

Also, for the rule A6-4-1 we had to implement logically parsing a switch into its case and default clauses. It works for trivial switches (not containing statements between clauses or nested case clauses for example) and otherwise marks the switch as non-trivial. There is a comment in the switch statement implementation in CDT suggesting to perform logical grouping and our implementation in the `SwitchHelper` might be considered as a starting point.

5. Functional and performance testing

The second major part of this project consists of performing functional and performance testing. The goal is to find problems in the plugin from a functional as well as a performance perspective. In particular we have the following goals:

- **Functional:** We want to find out whether our checkers and quick fixes work correctly. False positives and false negatives are very likely in multiple checkers and quick fixes. There could also be some unexpected exceptions which might make our checkers and quick fixes crash. With the functional tests, we want to identify such problems and fix as many of them as possible. Additionally, we want to find out whether there are tests that are not useful in a real scenario, because they are annoying for example.
- **Performance:** By letting all of our checkers and quick fixes run on a large project, we want to find out how the performance of the CodeAnalysator plug-in is. Especially interesting are differences between execution times of multiple checkers. In the case that such differences exist, we want to identify the corresponding problems and improve the performance to a level where it can be reasonably used by a developer.

Summarized we can say, that we want to make the CodeAnalysator plug-in more viable to use in a real-life scenario.

5.1. Finding Suitable Test Projects

Before we can begin testing, we need at least one suitable test project to run the tests on. There are countless C++ projects of varying quality on GitHub. Suitable for us means on one hand that it is large enough to be non-trivial while making sufficient use of the C++ language. On the other hand it should contain many different applications of C++ constructs, while also being small enough for us to review the reported violations of rules and parts of the code base. And lastly, it must not be written by ourselves, because we might be biased by our understanding of the rules and their edge cases. This will make for more realistic examples than our test cases that were specifically designed for this purpose. We hope to find

cases that we did not think of during the implementation of the checkers and quick fixes.

5.1.1. Possible Projects

This section covers the projects that we considered more closely to use for performance and functional tests and our decision on which to use.

- **Real Time C++** is a companion project for the book "Real Time C++" by Christopher Kormanyos [*GitHub - ckormanyos/real-time-cpp: Real-Time C++ Companion Code* 2019], [Kormanyos 2018]. It consists of code samples and a reference app. In total there are about 30'000 lines of code.
- **Doom 3** and the other Doom games are often used as test and reference projects. It has around 500'000 lines of code. [*id-Software/DOOM-3: Doom 3 GPL source release* 2019]
- **LevelDB** is a key/value store by Google. It has a medium size and triggers many different checkers and also has tests to verify that our quick fixes did not break anything. The code base consists of about 21'000 lines of code [*GitHub - google/leveldb: LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values.* 2019]
- **Jucipp** is an integrated C++ development environment. Its code triggers many of our checkers and can be tested automatically. It amounts to about 38'000 lines of code. [*cppit / jucipp · GitLab* 2019]

5.1.2. Decision

To be able to get the most value out of these tests, we decided to use both LevelDB and Jucipp. We decided against Real Time C++ because the code is too well written. Even though it covers many aspects of C++, only a few checkers are triggered. This results in a lack of test cases for our checkers and quick fixes, which is not what we need. Doom, on the other hand, has the problem, that it was released before C++ 11 and therefore uses a very different kind of C++ than is used today. It also is an extremely large project. LevelDB and Jucipp are moderately modern and are large enough to trigger many checkers. A big plus is also the presence of tests to verify that our quick fixes did not break anything. The exact commits and fork dates are listed below for reproducibility reasons.

- **Jucipp:** We forked commit d56eb33385945b1ea757c23486700985239412c5 on April 12th, 2019.

- **LevelDB:** We forked commit 7b1174519044339f07a023dc445b0d36425bd6db on April 13th, 2019

5.2. Functional Tests

This section contains the results of the functional tests. As can be seen in Section 5.1.2, we decided to use the two projects LevelDB and Jucipp. There are two kinds of tests for both projects.

- **Sample Test Cases:** This category of tests is primarily used to find false negatives. The approach is to disable the checkers and then look for code snippets that we think should trigger our checkers. To apply the tests, the checkers are re-enabled. The result of a test is whether our predictions were correct.
- **Reports Test Cases:** This category of tests is primarily used to find false positives. To run these tests, we enable all checkers and work through the reported markers. We verify that the corresponding piece of code is marked correctly. To reproduce the results, set AUTOSAR as the first, CPPCore as the second and MISRA as the last priority.

5.2.1. Sample Test Cases: jucipp

This section contains the sample test cases for Jucipp. An explanation, why Jucipp was chosen can be found in Section 5.1.2. An explanation of how these tests work and why they are performed can be found at the beginning of Section 5.2.

Rule A08-05-02

Braced-initialization {}, without equals sign, shall be used for variable initialization.

jucipp/src/source.cc:37

```

1  Glib::RefPtr<Gsv::Language> Source::guess_language(
2      const boost::filesystem::path &file_path) {
3      auto language_manager = LanguageManager::get_default();
4      bool result_uncertain = false; // <--- [A08-05-02]
5      auto content_type = Gio::content_type_guess(file_path.string(), nullptr,
6          0, result_uncertain);
7      if(result_uncertain)
8
9  ==
10
11  Glib::RefPtr<Gsv::Language> Source::guess_language(
12      const boost::filesystem::path &file_path) {
13      auto language_manager = LanguageManager::get_default();
14      bool result_uncertain { false }; // <--- quick fixed
15      auto content_type = Gio::content_type_guess(file_path.string(), nullptr,
16          0, result_uncertain);
17      if(result_uncertain)

```

Result: Pass

Table 5.1.: Test protocol for A08-05-02

Rule A05-01-01

Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead.

jucipp/src/source.cc:45

```

1      auto filename = file_path.filename().string();
2      if(filename == "CMakeLists.txt") // <--- [A05-01-01]
3          language = language_manager->get_language("cmake");
4          // ^--- [A05-01-01]
5      else if(filename == "meson.build") // <--- [A05-01-01]
```

Result: Pass

Table 5.2.: Test protocol for A05-01-01

Rule A08-05-02

Braced-initialization {}, without equals sign, shall be used for variable initialization.

jucipp/src/source.cc:88

```

1  std::string Source::FixIt::string(
2      const Glib::RefPtr<Gtk::TextBuffer> &buffer) {
3      auto iter = buffer->get_iter_at_line_index(offsets.first.line,
4          offsets.first.index);
5      unsigned first_line_offset = iter.get_line_offset() + 1;
6      // ^--- [A08-05-02]
7      iter = buffer->get_iter_at_line_index(offsets.second.line,
8          offsets.second.index);
9      unsigned second_line_offset = iter.get_line_offset() + 1;
10     // ^--- [A08-05-02]
```

Result: Pass

Table 5.3.: Test protocol 2 for A08-05-02

Rule A08-05-00	
All memory shall be initialized before it is read.	
jucipp/src/source.cc:94	
1	<code>std::string text; // <--- [A08-05-00]</code>
2	<code>if(type == Type::INSERT) {</code>
Result: Pass	

Table 5.4.: Test protocol for A08-05-00

Rule A07-02-02	
Enumeration underlying base type shall be explicitly defined.	
lib/libclangmm/src/token.h:14	
1	<code>public:</code>
2	<code>enum Kind {</code>
3	<code> Punctuation,</code>
4	<code> Keyword,</code>
5	<code> Identifier,</code>
6	<code> Literal,</code>
7	<code> Comment</code>
8	<code>}; // <--- [A07-02-02]</code>
Result: Pass	
Note: Double 'Suppress all guidelines for this statement' entry	

Table 5.5.: Test protocol for A07-02-02

Rule A07-02-03

Enumerations shall be declared as scoped enum classes.

lib/libclangmm/src/token.h:

```
1  public:
2      enum Kind {
3          Punctuation,
4          Keyword,
5          Identifier,
6          Literal,
7          Comment
8      }; // <--- [A07-02-03]
```

Result: Pass

Note: Double 'Suppress all guidelines for this statement' entry

Table 5.6.: Test protocol for A07-02-03

Rule A05-01-02

Variables shall not be implicitly captured in a lambda expression.

tests/lldb_test.cc:83

```
1  Debug::LLDB::get().on_exit.emplace_back(
2      [&](int exit_status_) { // <--- [A05-01-02]
3          exit_status = exit_status_;
4          exited = true;
5      });
6  Debug::LLDB::get().on_event.emplace_back(
7      [&](const lldb::SBEvent &event) { { <-- [A05-01-02]
8          std::unique_lock<std::mutex> lock(Debug::LLDB::get().mutex);
9          auto process = lldb::SBProcess::GetProcessFromEvent(event);
10         auto state = lldb::SBProcess::GetStateFromEvent(event);
11         if(state == lldb::StateType::eStateStopped) {
12             auto line_entry = process.GetSelectedThread()
13                 .GetSelectedFrame().GetLineEntry();
14             if(line_entry.IsValid()) {
15                 lldb::SBStream stream;
16                 line_entry.GetFileSpec().GetDescription(stream);
17                 line_nr = line_entry.GetLine();
18             }
19         }
20     });
21
22 ==
23
24  Debug::LLDB::get().on_exit.emplace_back(
25      [&exit_status, &exited](int exit_status_) { // <--- [A05-01-02]
26          exit_status = exit_status_;
27          exited = true;
28      });
29  Debug::LLDB::get().on_event.emplace_back(
30      [&line_nr](const lldb::SBEvent &event) { { // <--- [A05-01-02]
31          std::unique_lock<std::mutex> lock(Debug::LLDB::get().mutex);
32          auto process = lldb::SBProcess::GetProcessFromEvent(event);
33          auto state = lldb::SBProcess::GetStateFromEvent(event);
34          if(state == lldb::StateType::eStateStopped) {
35              auto line_entry = process.GetSelectedThread()
36                  .GetSelectedFrame().GetLineEntry();
37              if(line_entry.IsValid()) {
38                  lldb::SBStream stream;
39                  line_entry.GetFileSpec().GetDescription(stream);
40                  line_nr = line_entry.GetLine();
41              }
42          }
43      });
```

Result: Pass

Table 5.7.: Test protocol for A05-01-02

Rule A07-01-06

The typedef specifier shall not be used.

lib/tiny-procss-library/process.hpp:27

```
1  #else
2      typedef pid_t id_type; // <--- [A07-01-06]
3      typedef int fd_type; // <--- [A07-01-06]
4      typedef std::string string_type; // <--- [A07-01-06]
5  #endif
6      typedef std::unordered_map<string_type, string_type> environment_type;
7      // ^--- [A07-01-06]
8
9  ===
10
11 #else
12     using id_type = pid_t; // <--- quick fixed
13     using fd_type = int; // <--- quick fixed
14     using string_type = std::string; // <--- quick fixed
15 #endif
16     using environment_type = std::unordered_map<string_type, string_type>;
17     // ^--- quick fixed
```

Result: Pass

Table 5.8.: Test protocol for A07-01-06

Rule A05-01-03

Parameter list (possibly empty) shall be included in every lambda expression.

src/autocomplete.h:4

```
1 Autocomplete::Autocomplete(Gtk::TextView *view,
2   bool &interactive_completion, guint &last_keyval,
3   bool pass_buffer_and_strip_word)
4   : view(view), interactive_completion(interactive_completion),
5   pass_buffer_and_strip_word(pass_buffer_and_strip_word) {
6   view->get_buffer()->signal_changed().connect(
7     [this, &last_keyval] { // <--- [A05-01-03]
8     if(CompletionDialog::get() && CompletionDialog::get()->is_visible()) {
9       cancel_reparse();
10      return;
11    }
12  ==
13
14  Autocomplete::Autocomplete(Gtk::TextView *view,
15   bool &interactive_completion, guint &last_keyval,
16   bool pass_buffer_and_strip_word)
17   : view(view), interactive_completion(interactive_completion),
18   pass_buffer_and_strip_word(pass_buffer_and_strip_word) {
19   view->get_buffer()->signal_changed().connect(
20     [this, &last_keyval]() { // <--- quick fixed
21     if(CompletionDialog::get() && CompletionDialog::get()->is_visible()) {
22       cancel_reparse();
23     }
24     return;
25   }
```

Result: Pass

Table 5.9.: Test protocol for A05-01-03

Rule A07-02-02

Enumeration underlying base type shall be explicitly defined.

src/autocomplete.h:16

```

1 public:
2     enum class State { IDLE, STARTING, RESTARTING, CANCELED };
3     // ^--- [A07-02-02]
```

Result: Pass

Table 5.10.: Test protocol for A07-02-02

Rule A10-01-01

Class shall not be derived from more than one base class which is not an interface class.

src/source_clang.h:107

```

1 class ClangView : public ClangViewAutocomplete, public ClangViewRefactor
2 { // <--- [A10-01-01]
3 public:
4     ClangView(const boost::filesystem::path &file_path,
5               const Glib::RefPtr<Gsv::Language> &language);
6
7     void full_reparse() override;
8     void async_delete();
```

Result: Pass

Table 5.11.: Test protocol for A10-01-01

Rule A05-01-08

Lambda expressions should not be defined inside another lambda expression.

src/directories.cc:781

```
1  if(it != directories.end() && it->second.repository) {
2      auto repository = it->second.repository;
3      std::thread git_status_thread(
4          [this, dir_path, repository, include_parent_paths] {
5              Git::Repository::Status status;
6              try {
7                  status = repository->get_status();
8              }
9              catch(const std::exception &e) {
10                 Terminal::get().async_print(std::string("Error (git): ") + e.what()
11                     + '\n', true);
12             }
13
14             dispatcher.post([this, dir_path, include_parent_paths,
15                 status = std::move(status)] { // <--- [A05-01-08]
16                 auto it = directories.find(dir_path->string());
17                 if(it == directories.end())
18                     return;
```

Result: Pass

Table 5.12.: Test protocol for A05-01-08

5.2.2. Functional Reports Tests: jucipp

This section contains the reports test cases for Jucipp. An explanation, why Jucipp was chosen can be found in Section 5.1.2. An explanation of how these tests work and why they are performed can be found at the beginning of Section 5.2.

Table 5.13.: Test protocol for jucipp checked rule violation reports

RuleNr	Location	Result	Comment
A05-00-02	src/autocomplete.cc:67	Notes	marked, when index not fully build
A05-00-02	src/compile_commands.cc:16	Fail	std::string equality operator yields bool, index error?
A05-00-02	src/compile_commands.cc:49	Fail	size_t smaller than operator yields bool, excessive marking
A05-00-02	src/debug_llvm.cc:205	Research	relational operator used with enum StopReason, error?
A05-00-02	src/filesystem.cc:23	Fail	std::ifstream implements explicit cast to bool
A05-01-01	src/autocomplete.cc:28	Notes	true literal, return value, exception?
A05-01-01	src/autocomplete.cc:31	Notes	nullptr literal, return value, exception?
A05-01-01	src/cmake.cc:21	Notes	true literal, while condition, exception?
A05-01-01	src/cmake.cc:22	Pass	file name literal
A05-01-01	src/cmake.cc:132	Notes	zero index literal, exception?
A05-01-03	src/autocomplete.cc:45	Pass	
A05-01-03	src/dispatcher.cc:5	Notes	excessive marking
A05-01-03	src/selection_dialog.cc:122	Pass	
A05-01-03	src/terminal.cc:301	Notes	excessive marking
A05-01-03	src/source_language_protocol.cc:409	Notes	excessive marking
A05-01-06	src/autocomplete.cc:29	Pass	
A05-01-06	src/autocomplete.h:33	Fail	syntax error: return type missing
A05-01-06	src/autocomplete.h:52	Notes	quick fix works, but verbose fully specified string declaration
A05-01-06	lib/libclangmm/src/completion_string.cc:77	Pass	
A05-01-06	src/directories.cc:17	Pass	
A05-16-01	lib/tiny-process-library/examples.cpp:17	Notes	usefull case?
A05-16-01	src/source_language_protocol.cc:121	Notes	weird use case
A05-16-01	src/source_language_protocol.cc:595	Pass	difficult expression

Table 5.13.: Test protocol for jucipp checked rule violation reports

RuleNr	Location	Result	Comment
A05-16-01	src/source_language_protocol.cc:1032	Notes	usefull case?
A05-16-01	src/terminal.cc:187	Pass	difficult, magic expression
A07-01-05	src/cmake.cc:10	Fail	lambda expression is of non-fundamental type , quick fix just removes auto, syntax error
A07-01-05	src/cmake.cc:11	Fail	auto used as decl specifier in for-each, quick fix inserts semicolon, syntax error
A07-01-05	src/cmake.cc:20	Pass	
A07-01-05	src/cmake.cc:134	Pass	
A07-01-05	src/completion_string.cc:78	Notes	auto assigned static_cast
A07-01-06	lib/tiny-process-library/process.hpp:28	Pass	
A07-01-06	lib/tiny-process-library/process.hpp:29	Pass	
A07-01-06	lib/tiny-process-library/process.hpp:30	Pass	
A07-01-06	lib/tiny-process-library/process.hpp:32	Pass	
A07-01-06	src/source_base.cc:74	Notes	weird case, preprocessor makros used
A07-02-02	src/autocomplete.h:17	Pass	
A07-02-02	src/git.h:57	Pass	
A07-02-02	src/source.h:41	Pass	
A07-02-02	src/source_diff.h:14	Pass	
A07-02-02	src/token.h:14	Pass	
A07-02-03	lib/libclangmm/src/completion_string.h:9	Pass	
A07-02-03	lib/libclangmm/src/token.h:14	Pass	
A08-05-00	src/autocomplete.cc:63	Pass	
A08-05-00	src/autocomplete.cc:105	Fail	for-each declarator
A08-05-00	src/cmake.cc:13	Pass	
A08-05-00	src/config.cc:29	Pass	
A08-05-00	lib/libclangmm/src/cursor.cc:127	Pass	
A08-05-02	src/autocomplete.h:24	Pass	
A08-05-02	src/autocomplete.h:28	Pass	
A08-05-02	src/cmake.cc:12	Pass	
A08-05-02	src/cmake.cc:20	Pass	
A08-05-02	src/compile_commands.cc:89	Pass	

Table 5.13.: Test protocol for jucipp checked rule violation reports

RuleNr	Location	Result	Comment
A09-05-01	src/source_base.cc:74	Notes	error in makro expansion, wrong error code A09-05-001
A09-05-01	src/source_base.cc:75	Notes	error in makro expansion, wrong error code A09-05-001
A10-01-01	src/project.h:155	Pass	
A10-01-01	src/source.h:52	Pass	
A10-01-01	src/source_clang.h	Pass	
A10-02-01	src/directories.h:27	Fail	ctor of type with same name as base class
A10-02-01	src/directories.h:50	Fail	static class function not a member function
A10-02-01	src/entry_box.h:48	Notes	info on which method is hidden would be nice
A10-02-01	src/source.h:61	Fail	member function overrides base function, no hiding?, diamond problem?
A10-02-01	src/source_generic.h:18	Notes	info on which method is hidden would be nice, class::method
A10-03-01	src/dialogs.cc:29	Research	Should report when no virtual specifier? (redundant with A10-03-02)
A10-03-02	src/dialogs.cc:29	Research	override in header declaration, marked in implementation?
A10-03-02	src/source_generic.h:11	Pass	dtor of type
A12-00-01	src/config.h:10	Fail	rule of zero followed
A12-00-01	src/cursor.h:11	Pass	only copy ctor implemented
A12-00-01	src/project.h:40	Pass	move ctor and dtor only
A12-00-01	src/terminal.h:11	Fail	rule of zero followed, maybe due to base class?
A12-00-01	src/translation_unit.h:15	Pass	dtor only
A13-05-03	src/ctags.h:17	Pass	
A13-05-03	src/git.h:21	Pass	
A13-05-03	src/source.h:31	Pass	
ES.026	src/cmake.cc:281	Pass	
ES.026	src/menu.cc:518	Pass	
ES.026	src/notebook.cc:215	Pass	
M03-04-01	lib/libclangmm/src/completion_string.cc:85	Notes	loop variable used afterwords
M03-04-01	src/directories.cc:82	Notes	loop variable used before
M03-04-01	src/filesystem.cc:133	Pass	

Table 5.13.: Test protocol for jucipp checked rule violation reports

RuleNr	Location	Result	Comment
M04-05-01	src/cmake.cc:205	Fail	assignment operator
M04-05-01	src/config.cc:82	Notes	belongs to assignment or logical operators?

5.2.3. Functional Sample Test Cases: LevelDB

This section contains the sample test cases for LevelDB. An explanation, why LevelDB was chosen can be found in Section 5.1.2. An explanation of how these tests work and why they are performed can be found at the beginning of Section 5.2.

Rule A05-01-01 (required, implementation, partially automated) Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead.	
Rule A08-05-02 (required, implementation, automated) Braced-initialization , without equals sign, shall be used for variable initialization.	
db/autocompact_test.cc:52ff	
1	<code>static const int kValueSize = 200 * 1024;</code>
2	<code>// ^--- 2x [A05-01-01], [A08-05-02]</code>
Result: Pass	

Table 5.14.: Test protocol for A05-01-01 and A08-05-02 test

Rule A05-00-02 (required, implementation, automated) The condition of an if-statement and the condition of an iteration statement shall have type bool.	
db/db_impl.cc:91	
1	<code>if (static_cast<V>(*ptr) > maxvalue) ... // <--- [A05-00-02]</code>
Result: Pass	

Table 5.15.: Test protocol for A05-00-02 test

Rule A05-01-03 (required, implementation, automated) Parameter list (possibly empty) shall be included in every lambda expression.

env_windows.cc:689

```

1      bgsignal_.wait(lk, [this] { return !queue_.empty(); });
2                                  // ^--- [A05-01-03]

==

1      bgsignal_.wait(lk, [this](){ return !queue_.empty(); });
2                                  //^--- quick fixed

```

Result: Pass

Table 5.16.: Test protocol for A05-01-03

Rule A05-16-01 (required, implementation, automated) The ternary conditional operator shall not be used as a sub-expression.

test_c.cc:39ff

```

1      fprintf(stderr, "%s: expected '%s', got '%s'\n",
2              phase,
3              (expected ? expected : "(null)"), // <--- [A05-16-01]

```

Result: Pass

Table 5.17.: Test protocol for A05-16-01

Rule A07-01-05 (required, implementation, automated) The auto specifier shall not be used apart from following cases: (1) to declare that a variable has the same type as return type of a function call, (2) to declare that a variable has the same type as initializer of non-fundamental type, (3) to declare parameters of a generic lambda expression, (4) to declare a function template using trailing return type syntax.

envposix.cc:795

```

1      auto background_work_function =
2      background_work_queue_.front().function; // <--- [A07-01-05]

```

Result: Pass

Table 5.18.: Test protocol for A07-01-05

Rule A07-01-06 (required, implementation, automated) The typedef specifier shall not be used.

db/db_test:2033

```
1     typedef std::map<std::string, std::string> KVMMap; // <--- [A07-01-06]
==
1     using KVMMap = std::map<std::string, std::string>; // <--- quick fixed
```

Result: Pass

Table 5.19.: Test protocol for A07-01-06

Rule A07-02-02 (required, implementation, automated) Enumeration underlying base type shall be explicitly defined.

include/leveldb/c.h:192ff

```
1     enum { // <---[A07-02-02]
2         leveldb_no_compression = 0,
3         leveldb_snappy_compression = 1
4     };
```

Result: Pass

Note: A07-02-03 is also reported

Table 5.20.: Test protocol for A07-02-02

Rule A07-02-03 (required, implementation, automated) Enumerations shall be declared as scoped enum classes.

db/db_iter.cc:203

```
1     enum Direction { // <--- [A07-02-03]
2         kForward,
3         kReverse
4     };
```

Result: Pass

Note: A07-02-02 is also reported

Table 5.21.: Test protocol for A07-02-03

	Rule A08-05-00 (required, implementation, automated) All memory shall be initialized before it is read.
1	db/c.cc:203 <pre>std::string tmp; // <--- [A08-05-00]</pre>
	Result: Pass

Table 5.22.: Test protocol for A08-05-00

Rule A10-03-02 (required, implementation, automated) Each overriding virtual function shall be declared with the override or final specifier.

include/leveldb/iterator.h 24ff

```

1      class LEVELDB_EXPORT Iterator {
2          ...
3      public:
4          ...
5          virtual void SeekToFirst() = 0;
6          virtual void SeekToLast() = 0;
7          virtual void Seek(const Slice& target) = 0;
8          virtual void Next() = 0;
9          virtual void Prev() = 0;
10         ...
11     };

```

db/db_iter.cc 82ff

```

1      class DBIter: public Iterator {
2          ...
3      public:
4          ...
5          virtual void Next(); // <--- [A10-03-02]
6          virtual void Prev(); // <--- [A10-03-02]
7          virtual void Seek(const Slice& target); // <--- [A10-03-02]
8          virtual void SeekToFirst(); // <--- [A10-03-02]
9          virtual void SeekToLast(); // <--- [A10-03-02]
10         ...
11     };

```

Result: Pass

Table 5.23.: Test protocol for A07-02-03

Rule A10-03-03 (required, implementation, automated) Virtual functions shall not be introduced in a final class

util/env_posix.cc:104ff

```

1      class PosixSequentialFile final : public SequentialFile {
2          ...
3          Status Read(size_t n, Slice* result, char* scratch)
4              override {...} // <--- [ A10-03-03]
5          ...
6      }
==
1      class PosixSequentialFile final : public SequentialFile {
2          ...
3          Status Read(size_t n, Slice* result, char* scratch) final {...}
4          ... // quick fixed ---^
5      }

```

Result: FAIL. Rule is reported but quick fix fails.

Table 5.24.: Test protocol for A10-03-03

Rule A12-00-01 (required, implementation, automated) If a class declares a copy or move operation, or a destructor, either via “=default”, “=delete”, or via a user-provided declaration, then all others of these five special member functions shall be declared as well.

util/arena.h:16ff

```

1      class Arena { // <---[A12-00-01]
2      public:
3          Arena();
4          ~Arena();
5
6          char* Allocate(size_t bytes);
7
8          char* AllocateAligned(size_t bytes);
9
10         size_t MemoryUsage() const {
11             return memory_usage_.load(std::memory_order_relaxed);
12         }
13
14     private:
15         char* AllocateFallback(size_t bytes);
16         char* AllocateNewBlock(size_t block_bytes);
17
18         char* alloc_ptr_;
19         size_t alloc_bytes_remaining_;
20
21         std::vector<char*> blocks_;
22
23         std::atomic<size_t> memory_usage_;
24
25         Arena(const Arena&);
26         void operator=(const Arena&);
27     };

```

Result: Pass

Table 5.25.: Test protocol for A12-00-01 test

5.2.4. Functional Reports Test: Leveldb

This section contains the reports test cases for LevelDB. An explanation, why LevelDB was chosen can be found in Section 5.1.2. An explanation of how these tests work and why they are performed can be found at the beginning of Section 5.2.

Table 5.26.: Test protocol for leveldb checked rule violation reports

RuleNr	Location	Result	Comment
A04-07-01	db/log_writer.cc 17	Pass	
A05-00-02	util/coding.cc 10	Fail	is static bool, should not be reported
A05-00-02	util/coding.cc 21	Fail	is static bool, should not be reported
A05-00-02	util/coding.cc 145	Pass	bitwise and leads to implicit conversion
A05-00-02	util/crc32c.cc 328	Fail	macro
A05-01-01	util/arena.cc 12	Pass	But does it really make sense to mark nullptr?
A05-01-01	util/arena.cc 32	Pass	
A05-01-01	util/arena.cc 41	Pass	
A05-01-01	table/block_builder.cc 44	Pass	
A05-01-01	include/leveldb/c.h 193	Pass	But does it really make sense to mark in enum declaration?
A05-01-01	util/crc32c.cc 65	Pass	But does it make sense to mark array indices?
A05-01-01	db/db_bench.cc 62	Fail	Initialization
A05-01-01	util/env_posix.cc 89	Pass	Method call
A05-16-01	util/arena_test.cc 29	Pass	
A05-16-01	util/bloom_test.cc 64	Pass	Does it make sense to mark this though?
A05-16-01	db/c.cc 273	Pass	
A05-16-01	db/db_impl.cc 100	Fail	Used for initialization, technically sub-expression but not very usefull
A07-01-05	util/env_posix.cc 797	Pass	
A07-01-06	include/leveldb/c.h 54	Pass	Quick fix fails, typedef struct
A07-01-06	helpers/menenv/memenv.cc 399	Pass	
A07-01-06	db/skiplist_test.cc 20	Pass	
A07-01-06	table/two_level_iterator.cc 16	Pass	
A07-02-02	include/leveldb/c.h 192	Pass	
A07-02-02	db/version_edit.cc 14	Pass	
A07-02-02	db/fault_injection_test.cc 369	Pass	
A07-02-03	include/leveldb/c.h 192	Pass	
A07-02-03	db/version_edit.cc 14	Pass	
A07-02-03	db/fault_injection_test.cc 369	Pass	

Table 5.26.: Test protocol for leveldb checked rule violation reports

RuleNr	Location	Result	Comment
A07-02-03	doc/bench/db_bench_tree_db.cc 282	Pass	
A08-05-00	util/arena.cc 46	Pass	
A08-05-00	util/arena_test.cc 20	Pass	
A08-05-00	table/block.cc 173	Pass	
A08-05-02	util/arena.cc 9	Pass	
A08-05-02	util/arena.cc 17	Pass	
A08-05-02	util/arena.cc 43	Pass	
A10-02-01	db/db_impl.h 186	Pass	
A10-02-01	db/db_iter.cc 127	Pass	
A10-02-01	db/fault_injection_test.cc 138	Fail	Overridden method is virtual. Overrides, but no shadowing
A10-03-01	db/c.cc 514	Pass	
A10-03-01	db/db_test.cc 154	Pass	
A10-03-02	db/db_test.cc 2045	Pass	
A10-03-02	table/block.cc 139	Pass	
A10-03-03	util/env_posix.cc 129	Pass	
A10-03-03	util/env_posix.cc 301	Pass	
A10-03-03	util/windows_logger.cc 34	Pass	
A12-00-01	util/cache.cc 361	Pass	
A12-00-01	include/leveldb/db.h 30	Pass	
A12-00-01	db/db_test.cc 2044	Pass	
C.020	include/db/db.h 38	Pass	
C.020	db/db_test.cc 2044	Pass	
ES.074	db/db_impl.cc 1300	Pass	
ES.074	db/fault_injection_test.cc 352	Pass	
ES.074	db/log_test.cc 230	Pass	
M03-04-01	util/logging.cc 63	Pass	
M05-00-06	db/log_writer.cc 17	Pass	
M05-00-06	util/arena_test.cc 27	Pass	
M05-00-06	db/db_bench.cc 146	Pass	

5.2.5. Changes Based on Functional Tests

This section covers all the changes that we performed on the code based on the functional tests.

Binary Expression Handling with Non-Expression Second Operand

While conducting the functional tests we noticed a `NullPointerException` being thrown in the `BoolExpressionOperandsWarningVisitor`. This was caused by the `IASTBinaryExpression` returning null for `getOperand2()`. This is a bit counterintuitive, since one would expect a binary expression to always have two operands. The reason for this is that with an assignment operator the second operand may be a list-initializer, which is an initializer clause but not an expression (`getOperand2()` returns an expression). In every case, as every expression is an initializer clause, the second operand can be accessed using `getInitOperand2()`. After fixing the `BoolExpressionOperandsWarningVisitor` to check for null, we checked all uses of `getOperand2()` in the `CodeAnalysator` for a possible error. `BoolExpressionOperandsInfoVisitor` and `ConditionMustBeBoolVisitor` did use the method but were not affected as the operator in their case is never the assignment operator and therefore `getOperand2()` never returns null. In contrast, we were able to induce similar errors for `DontUseVariableForTwoUnrelatedPurposesVisitor` and `DeclareLoopVariableInTheInitializerQuickFix`, which we fixed by replacing the call with `getInitOperand2()`.

Assignment Operator with Bool Operand Reported

An assignment operator with a bool operand was reported as violating rule M4-5-1. This rule forbids the use of a boolean as an operand to built in operators except for a few cases. One of these cases is the use of bool operands in assignment operators [*MISRA C++ : 2008 Guidelines for the use of the C++ language in critical systems* 2018]. `BoolExpressionOperandsVisitor` was fixed by adding a check for the assignment operator and not marking it if it is one.

Implicit Lambda Expression Returning a nullptr

Our quick fix for the rule A5-1-6, which advises against implicit lambda return types, failed to correctly fix a lambda expression returning a `nullptr`. It inserted just the arrow of the trailing return type without specifying any actual type leading to a syntax error. See Figure 6 for the initial situation and Figure 7 for the faulty fix.

```
1 auto lambda = []() {  
2     return nullptr;  
3 };
```

Listing 6: A lambda returning a nullptr

```
1 auto lambda = []() -> {  
2     return nullptr;  
3 };
```

Listing 7: A lambda returning a nullptr, fixed incorrectly

Upon investigation we discovered the error in the `DeclarationGeneratorImpl` of the CDT. The issue was that while the type of a `nullptr` is represented using a basic type, it cannot be declared using a simple declaration specifier. It requires a named type declaration specifier instead. Since we cannot modify the CDT code directly and we had already created our own modified version of the declaration generator with improved ability to generate template declarations, we decided to implement our fix there. We renamed the `TemplateExtendedDeclarationGeneratorImpl` to `CodeAnalysatorDeclarationGeneratorImpl`, fixed the generator to construct a named type specifier in the case of a basic type of the kind `nullptr` and changed the quick fix to use this generator instead. See Figure 8 for the correctly fixed lambda.

```
1 auto lambda = []() -> std::nullptr_t {  
2     return nullptr;  
3 };
```

Listing 8: A lambda returning a nullptr, fixed correctly

Ideally, these changes will be included into the CDT in the future, therefore we have kept the changes to the class to a minimum. Now the quick fix correctly inserts `std::nullptr_t` as the return type.

Auto Declared Variable Initialized with Lambda Expression

Rule A07-01-05 prohibits the use of `auto` except for a few certain cases. Refer to the AUTOSAR document for a detailed explanation [AUTOSAR 2018]. Declarations that are initialized using lambda expressions and are using the `auto` specifier, were

reported by the visitor for this rule. But one of the exceptions is that the auto specifier is allowed, if a non-fundamental type is used. A lambda expression is considered to be of a non-fundamental type and therefore should not be reported by this rule. We fixed the `UseAutoSparinglyVisitor` by adding a check for lambda expressions.

```
1 int i {};  
2 i = { 1 }; // idExpression(i), operator(assignment),  
3           // initializerClause({ 1 })
```

Listing 9: Example for a binary expression with a non-expression second operand

A07-01-05 Quick Fix Inserts Semicolon after Declaration in Ranged-for

One application of the Quick Fix for A07-01-05 is to replace auto in a declaration of a range based for. It replaces the auto specifier with the deduced type, but inserts a semicolon after the declaration and before the double colon, leading to a syntax error. See Listing 10. Upon debugging the error was found to be due to the `ASTWriterVisitor` calling the `DeclarationWriter.writeDeclaration` overload which unconditionally inserts a semicolon when replacing an `IASTDeclaration`. Regardless of whether the for loop is ranged or not. We further tested and verified that the same error occurs also for replacing declarations in if-conditions, loop-conditions and the controller declaration in the switch statement. The bug might be fixed by adding a check, either in the `visit` method of the `ASTWriterVisitor` or in the `DeclarationWriter`. It would need to test whether the declaration is a control declaration, i.e. whether the parent is a if-, loop- or switch-statement, and in that case omit the semicolon. As a workaround we have changed the quick fix to be not applicable in the case of a control declaration.

```
1 for (int &i;: list) {}
```

Listing 10: Example of a wrongly inserted semicolon for control declarations

A08-05-00: Range-declaration Reported for Not Initializing Object

The `AlwaysInitializeAnObjectVisitor` reported range-declarations in ranged-for statements as violating A08-05-00: Always initialize an object. That is because a range-declaration does not have an initializer, but is initialized by the ranged-for loop on every iteration. We added an additional check, whether the parent is a

ranged-for statement, next to the check whether the parent is a catch handler and do not report the declaration in that case.

```
1 for(int i: list){
2     //^--- No initialization here, but in every iteration
3 }
```

Listing 11: Example of a ranged for loop

A10-2-1: Constructor of Class with Equal Name As Base Class Reported

The `DoNotHideMemberFunctionsVisitor` wrongly reported constructors of classes, with a name equal to one of its base classes, as hiding a member function. We added a check whether the method is a constructor and, in that case, skip testing for member function hiding. See Figure 12 for an example.

```
1 namespace a_namespace {
2 struct someStruct {
3     someStruct() {
4     }
5 };
6 }
7 namespace another_namespace {
8 struct someStruct: a_namespace::someStruct {
9     someStruct() {
10    }
11 };
12 }
```

Listing 12: The visitor marked the constructor on line 9 wrongly

A10-03-03 Quick Fix Is Not Working with NamedDeclSpecifiers

A10-03-03 is the rule that every overriding function is supposed to be written with either the `override` or `final` keyword. This did not work if the `DeclSpecifier` was a class or struct as opposed to a primitive. We fixed this and added tests accordingly. See Figure 13 for an example of such a case.

```
1 struct X{};
2 struct A {
3     virtual X doSomething(){}
4 };
5
6 struct B : A{
7     X doSomething(){} <--- A10-03-03 marker
8 };
```

Listing 13: Resolving this marker failed before our fix

Note: The Overrider quick assist can also be used for this scenario. This quick assist was also developed during this project. More information about the Overrider and quick assists in general can be found in Chapter 6.

A05-01-01 Checker Is Not Working Properly with Exception

According to rule A05-01-01, literal values should be avoided. An exception thereof is a literal value in an initialization. This did not work properly before, since the checker did not recognize when the literal value occurred in an initializer list. See Figure 14 for an example. This has now been changed, which also slightly reduced the amount of times the rule was reported.

```
1 int main(){
2     int i {5};
3 }
```

Listing 14: The A05-01-01 marker did not recognize this exception

A10-02-01 Checker Does Not Check for Virtuality Because of Overriding

A10-02-01 defines that non-virtual functions shall not be overridden because it results in shadowing [AUTOSAR 2018]. This means that the checker must find out, whether a method has any method in a base class, which it overrides but which is not virtual. Up until now, the checker used the method `isVirtual()` of the class `ICPPMethod`. Unfortunately this function does not take into consideration, that a method might be virtual because it overrides another method. This resulted in false positives.

```
1 struct A {
2     virtual void doSomething() {
3     }
4 };
5
6 struct B: A {
7     void doSomething() {
8     }
9 };
10
11 struct C: B {
12     void doSomething() {
13     }
14 };
```

Listing 15: B.doSomething() is virtual even though there is neither a virtual keyword or an override specifier. Because of this, C.doSomething() does not shadow but override.

To solve this problem, we introduced a check, that also accounts for this type of virtual. A final method is theoretically also virtual. But overriding a final method poses other threats, which is not in scope of this rule.

Note: While the example in Figure 15 compiles and is correct code, it still violates another AUTOSAR rule. Rule A10-03-02 defines that every overriding method should be marked with either the override or final keyword [AUTOSAR 2018].

A10-03-02 Quick Fix Resulted in Code That Violates A10-03-01

A10-03-02 wants the developer to mark every overriding method with either the override or final keyword. Currently this rule offers two quick fixes. One to make it final and one to make it overriding. This is simply a matter of adding one of the two keywords. The problem is now, that while this approach is correct, the resulting code sometimes violates A10-03-01 which wants only one virtual keyword per method. For example the following method in Listing 16 would turn into Listing 17, which is good.

```
1 S doSomething();
```

Listing 16: Example of a method that can be made final

```
1 S doSomething() final;
```

Listing 17: Example made final without violating A10-03-01

But if the original code looked like in Listing 18, the resulting code would look like the example in Listing 19. This would violate Rule A10-03-01.

```
1 virtual S doSomething();
```

Listing 18: Example of a method that can be made final with virtual keyword

```
1 virtual S doSomething() final;
```

Listing 19: Example made final but violating A10-03-01

We fixed this by extracting some code from the A10-03-01 quick fix and used it in the A10-03-02 quick fix. That way the code complies with both rules.

A05-01-01 Marks This Pointer

There was a small bug in the checker for no magic numbers A05-01-01. The **this** pointer in C++ is considered a literal expression which means, every occurrence of it was marked by the A05-01-01 checker. This is obviously not intended. We added an exception for the **this** pointer to be excluded from the rule.

Adding Exceptions of A05-01-01

According to the AUTOSAR rule for A05-01-01, every literal value that is not in either an initialization or a log statement shall be reported by the checker. But this leads to a huge amount of markers. Even though correct, there is no way it would be practicable to use the checker as it was. So we added a few exceptions to the checker. These exceptions are the literal values for **true** and **false**, **nullptr** and the numbers 1 and 0. Right now, these exceptions are hard coded. This could be made configurable in a future project.

Change Missing Special Member Functions Checking to Rule of Five

The `MissingSpecialMemberFuntionsVisitor` is shared between AUTOSAR rule A12-00-01 and C++ Core rule C.21, but while A12-00-01 advocates the rule of zero and rule of five, C.21 advocates the rule of zero and rule of six, i.e. counts the default constructor as well. After a discussion in a meeting it was decided to chose the rule of five for both cases as more rational. We therefore changed the visitor to ignore the default constructor.

A05-16-01: Ternary Conditional Operator Reported in Assignment Expression

Rule A05-16-01 specifies that the ternary conditional operator shall not be used as a sub-expression. The operator is therefore also reported when used in an assignment expression. We discussed this special case in a meeting and decided to make an exception. We changed the visitor to allow use of the operator in assignment expressions. See Listing 20 on line 3 for an example of this exception.

```
1 int foo(bool condition){  
2     int i = 10;  
3     i = condition ? 1 : 0;  
4 }
```

Listing 20: A ternary conditional operator used in an assignement

A07-01-06: Make not Applicable for Typedef Struct

The use of typedef struct can be useful in C and the quick fix to the rule that forbids typedefs can not correctly fix this. The CodeAnalysator is intended to be used with modern C++. This is why we decided to simply make the quick fix not applicable for typedef structs. In a future project, the visitor could check whether the code is in a `extern C` environment.

A05-00-02: Bool or Contextually-Convertible to Bool Expressions Wrongly Reported as Not Boolean

The `ConditionMustBeBoolVisitor` reported boolean expressions or expressions contextually-convertible to bool as not being boolean, which is obviously wrong. This was due to multiple errors. The simplest were caused by semantic problems, which we changed to be ignored. Another error was caused by a bug due to which only the first explicit member of a class was being tested for a conversion to bool

operator. The last type of error was caused by a non-general mechanism for testing whether a member is a conversion to bool operator, which for example did not work for operators of template specializations. As there does not exist a general, semantic mechanism for obtaining that information from a binding we resorted to textually comparing the name of the members to `"operator bool"`. Lastly, during the checking it occurred to us that too much of the code was being marked, i.e. the whole statement containing the condition instead of the offending condition expression itself and we adapted the visitor accordingly.

5.2.6. Conclusion of Functional Tests

Most of our checkers and quick fixes work correctly, though we managed to find some that could be improved and some that needed fixing. Most of these were some edge cases that were not considered when initially writing them. Some others were technically correct but proved to be unusable in a real live scenario. Of course we do not claim to have found every problem, but we are confident to have improved the quality of most checkers and quick fixes by a great amount.

There were both false positives and false negatives. It is easy to find false positives. We looked at the reported markers and checked, whether they were rightfully reported. If we found one that was not supposed to be reported, we knew that it was wrong and could start fixing it. To find false negatives on the other hand was a bit more difficult. We approached it by looking at the code without markers and choosing places where a marker should appear in our opinion. We then ran the analysis and verified our expectations. We found false negatives as well as false positives, but we eliminated the false positives to a greater extent than the false negatives.

There are some checkers that do not do well in a real life scenario. As we expected, the rule no magic numbers can be annoying because it marks every single occurrence. An exception for that rule is, that it is OK to use literal values in log statements. But with our checker, a user would have to suppress the rule every time he/she logs something with a literal value because the checker can not differentiate the cases. Another rule that can be annoying especially in older projects is Always initialize an object. Some time ago it was common practice to declare every variable at the beginning of a function but not necessarily initialize it. This makes the checker trigger countless times in large old objects. There is not really something that can be done about this from a program side. A possible solution would be to have these checkers disabled by default.

In conclusion, we can say that it was a good idea to run the functional tests. We were able to implement many fixes and improvements to our plug-in. The overall robustness was improved and the amount of bugs decreased, which leaves the project in a better state than before.

5.3. Performance Tests

We have measured the run time for each checker and quick fix as well as the aggregated run time of the CodeAnalysator plug-in. The goal was to find relevant execution duration differences and bottlenecks. All our results were collected using the software JProfiler [*Java Profiler - JProfiler 2019*].

5.3.1. Test Results

We ran Code Analysis on the complete code of Jucipp and LevelDB. Since the results were almost equal, there is only a screenshot for Jucipp. We performed two kinds of tests. First we ran all the checkers to measure their performance. We also noticed, that scrolling in large files was slow and froze often. Because of this, we also measured the performance for simply scrolling through the files.

Running the checkers

As can be seen in Figure 5.1, there is indeed a performance bottleneck. Apparently the method `getAST` uses up a majority of computation time. The measurement below was stopped after about five and a half minutes. Further tests showed, that it would run for about an hour, which is of course unacceptable performance. Refer to Section 5.3.2 for our actions taken to solve this problem and to Section 5.3.3 for another test run after we applied the changes.

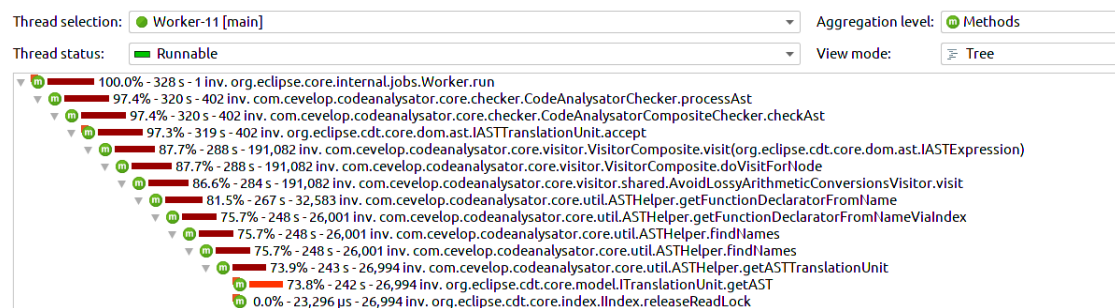


Figure 5.1.: The results of the performance test

Scrolling

Measuring the performance while scrolling led to the results shown in Figure 5.2. The performance problem is apparent on first glance. Same as before, the problem lies in the method `getAST()`. This time it is called from the `isApplicable()` part

of our quick fixes. Refer to Section 5.3.2 for our actions taken to solve this problem and to Section 5.3.3 for another test run after we applied the changes.

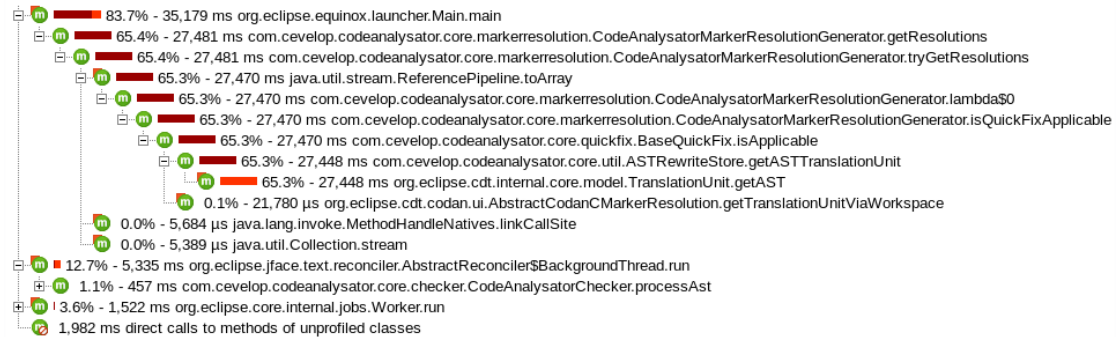


Figure 5.2.: The results of the performance test while scrolling

5.3.2. Changes Based on Performance Tests

This section covers all the changes we performed to solve the performance problems found in Section 5.3.1.

Updating the ASTHelper

As can be read in Section 5.3.1, the biggest performance bottleneck was the method `getASTTranslationUnit()` in the `ASTHelper`. Unfortunately we did not write this file ourselves. It is a very large helper class that contains a variety of support methods for AST traversal. This class was initially written by the team that created the `GSLator`. The first project team that worked on the `CodeAnalysator` plug-in simply copied it, because they needed it for a few visitors. The problem was, that the `AvoidLossyConversionsVisitor`, among other things, checks every method call for a lossy conversion. To find the original method declaration, the `ASTTranslationUnit` must be resolved, which happens in the `getAST()` method mentioned above. As a first step we updated the `CodeAnalysator` version with the newer version from the `GSLator`, which was subject to a few updates in the meantime. By debugging, we found out, that some sections of the code are never accessed and thus were basically dead code. We removed this dead code. This and the update improved the performance very slightly, but not by a significant amount.

Updating the `AvoidLossyConversionsVisitor`

To further improve the performance, we also updated the `AvoidLossyConversionsVisitor` with its newer version from the `GSLator`. `AvoidLossyConversionsVisitor` is the class that calls the bottleneck function the most. We managed to reduce the number of calls to it slightly.

Do not run a checker if its rule will not be reported

As we noticed, shared visitors are run for each guideline that has a corresponding rule. This means that sometimes the same visitor is run three times on the same code. To prevent this, we introduced a check, that only runs a visitor when its rule has the highest priority. Because otherwise it would not be reported anyway, so it does not make sense to have the visitor run. This improved the performance significantly, because the problematic visitor `AvoidLossyConversionsVisitor` has a corresponding rule in AUTOSAR [AUTOSAR 2018], MISRA [*MISRA C++ : 2008 Guidelines for the use of the C++ language in critical systems* 2018] and C++ CORE Guideline [*C++ Core Guidelines* 2019].

Updating the `AvoidLossyConversion` tests

There are a lot of tests for the problematic visitor `AvoidLossyConversionsVisitor`. Over the course of the refactoring of this visitor, we noticed, that the visitor apparently did not work correctly before. Many tests had no marker lines even though they should. And now that we fixed the visitor, all these tests obviously failed. So we added the marker lines. Another issue with these tests, was that there usually were a lot of markers for one single tests. This made it hard to find the problem, when a test failed. Because of this we split the tests into singular test cases, which took a lot of effort, but made the tests and test results in particular way better readable. Refer to Table 5.27 for an example of how the tests look like before and after our changes.

Before	After
<pre> 1 <i>//!FloatToIntConversionInitializer</i> 2 <i>//@.config</i> 3 setPreferencesEval= 4 (GUIDELINE_SETTING_ID 5 AUTOSAR_GUIDELINE_ID) 6 <i>//@main.h</i> 7 int main() { 8 long double ld = 345.5; 9 double d = 123.6; 10 float f = 7.9; 11 long long ll = -12345; 12 unsigned long long ull = 12345; 13 long l = -213; 14 unsigned long ul = 213; 15 int i = -42; 16 unsigned u = 42; 17 short s = -4; 18 unsigned short us = 4; 19 20 long long ll_1 = ld; 21 unsigned long long ull_1 = ld; 22 long l_1 = ld; 23 unsigned long ul_1 = ld; 24 int i_1 = ld; 25 unsigned u_1 = ld; 26 short s_1 = ld; 27 <i>// Many more test cases</i> 28 }</pre>	<pre> 1 <i>//!ld to ll initializer</i> 2 <i>//@.config</i> 3 setPreferencesEval= 4 (GUIDELINE_SETTING_ID 5 AUTOSAR_GUIDELINE_ID) 6 markerLines=3 7 <i>//@main.h</i> 8 int main() { 9 long double ld = 345.5; 10 long long ll_1 = ld; 11 } 12 13 <i>//!ld to ull initializer</i> 14 <i>//@.config</i> 15 setPreferencesEval= 16 (GUIDELINE_SETTING_ID 17 AUTOSAR_GUIDELINE_ID) 18 markerLines=3 19 <i>//@main.h</i> 20 int main() { 21 long double ld = 345.5; 22 unsigned long long ull_1 = ld; 23 } 24 25 <i>// Many more test cases</i></pre>

Table 5.27.: How the tests for AvoidLossyConversions where changed

Stop using the ASTHelper

After many failed attempts to improve the `ASTHelper` we decided to try and stop using it in the cases where possible. Unfortunately it took us very long to figure this out, because we were confused by the `ASTHelper`. A lot of code is executed when

its methods are called from the `AvoidLossyConversionsVisitor`. We wasted a lot of time trying to fix the `ASTHelper`, when we could have just changed the visitor itself. It boiled down to only a few lines of code, to improve the performance to the same level as the other visitors. Refer to Section 5.3.3 to see a report of how the performance improved. Listing 21 shows the corresponding section of the `AvoidLossyConversionsVisitor` before our changes and Listing 22 shows what it looks like afterwards.

```
1 IASTFunctionDeclarator functionDeclarator =
2   ASTHelper.getFunctionDeclaratorFromName(name, astCache);
3 if (functionDeclarator != null) {
4   IASTInitializerClause[] params = functionCall.getArguments();
5   Map<Integer, IType> paramsspec =
6     ASTHelper.getFunctionArguments(functionDeclarator, true);
7   int i = 0;
8   for (IASTInitializerClause iastInitializerClause : params) {
9     List<String> intermediateTypes = new ArrayList<String>();
10    String from = getTypeStringFromExpressionElement(iastInitializerClause,
11      intermediateTypes);
12    String to = "";
13    if (paramsspec.get(i) != null) to = paramsspec.get(i).toString();
14    analyseLossy(from, to, iastInitializerClause, true, intermediateTypes);
15    i++;
16  }
17 }
```

Listing 21: The `AvoidLossyConversionVisitor` before our refactoring.

```

1 IBinding binding = name.resolveBinding();
2 if (binding instanceof IFunction) {
3     IFunction function = (IFunction) binding;
4     IType[] parameterTypes = function.getType().getParameterTypes();
5     IASTInitializerClause[] arguments = functionCall.getArguments();
6     for (int i = 0; i < parameterTypes.length && i < arguments.length; i++) {
7         String from = getTypeStringFromExpressionElement(arguments[i]);
8         String to = parameterTypes[i].toString();
9         analyseLossy(from, to, expression);
10    }
11 }

```

Listing 22: The `AvoidLossyConversionVisitor` after our refactoring.

Quick Fix Applicability Checking Optimization

The quick fixes in the `CodeAnalysator` plug-in implement applicability checking to avoid showing quick fixes to the user when they are not applicable. The applicability checking also serves the purpose to verify the reported rule violation, in case the quick fix got out-of-sync with the checker that reported it. Now, the issue is that in order to perform these checks a traversal of the AST is required. Also, the `CodeAnalysatorMarkerResolutionGenerator` is only passed a marker from the Eclipse environment, which requires the file to be parsed every time resolutions for markers are retrieved. These parsing operations are relatively computationally expensive. Eclipse requests those resolutions not only when a menu is invoked, but as soon as the respective marker comes in to view. This results in a large number of file parses while scrolling a file with many markers in it, leading to noticeable lags of up to multiple seconds while scrolling.

As a temporary measure we removed the defensive applicability checking, running the defensive checks only when applying the quick fix. Also performing applicability checks only for partially applicable quick fixes. This greatly improved UI responsiveness in files with many `CodeAnalysator` markers, but is still an issue for markers with partially applicable quick fixes and still introduces lag when such a marker is encountered. Partially applicable means, that the quick fix is only applicable in certain cases. The code in Listing 23 shows such an example of a marker that can be applicable or not. The rule A07-01-06 forbids the use of typedef, but the quick fix cannot be used for typed struct. This means that the quick fix should only be proposed in the first case as shown below.

```
1  /* A07-01-06 Quick Fix applicable */
2  typedef int someType;
3
4  /*A07-01-06 Quick Fix not applicable*/
5  typedef struct someOtherType someOtherType;
```

Listing 23: The Quick Fix for A07-01-06 is not always applicable.

Any solution must remove the need for AST traversal in the applicability checking. The AST traversal and hence the check, must be moved to somewhere where the AST is already available, therefore the checker. Then, that information must somehow be passed to the quick fix. We came to two possible solutions, yet both with drawbacks.

1. The simplest solution would be to define separate problems for the applicable and non-applicable cases, but this would result in redundant entries for the rule, likely resulting in user confusion and error.
2. The second solution would be to pass the information over the marker using attributes, but, in the current implementation of the Codan base classes, the marker created when reporting a problem is not directly accessible to the checker.
3. While researching other possible solutions we came across a discussion on a bug report about the exact same issue: how to avoid parsing source code in the `isApplicable` check in quick fixes. [*Bug 309760 - [checker]/[quick fix] Provide checker for name resolution problems found by indexer 2019*] Among the members of the discussion were people who developed the initial Codan API or extended it. The solution chosen at the end was to perform the necessary checks in the checker and to pass the information to the quick fix by string problem arguments. The problem arguments are used as argument for placeholders in problem messages but superfluous ones are ignored by the formatter. The quick fix then implements the `isApplicable` check by retrieving those arguments from the marker.

While this last solution is not fully semantic and seems a bit like a hack, it avoids the above issues and is not too complex. We have therefore chosen the third solution.

```

1  @Override
2  public int visit(IASTDeclarator decl) {
3      if (violatesRule(decl)) {
4          reportRuleForNode(decl, createContextFlagsString(decl));
5      }
6      return super.visit(decl);
7  }
8  // ...
9  private String createContextFlagsString(IASTDeclarator decl) {
10     StringBuffer contextFlagsBuffer = new StringBuffer();
11     Optional<ICPPMethod> declaredMethod = getDeclaredMethod(decl);
12     declaredMethod.ifPresent(method -> {
13         if (method.isPureVirtual()) {
14             contextFlagsBuffer.append(ContextFlagsHelper.Virtual
15 FunctionShallHaveExactlyOneSpecifierContextFlagPureVirtual);
16         }
17         if (!VirtualHelper.overridesVirtualMethod(method)
18             && method.isFinal()) {
19             contextFlagsBuffer.append(ContextFlagsHelper.Virtual
20 FunctionShallHaveExactlyOneSpecifierContextFlagIntroducingFinal);
21         }
22     });
23     return contextFlagsBuffer.toString();
24 }

```

Listing 24: Checker generating and passing a contextFlagsString

Most of the solution is rather straightforward. When we determine that we need to report a node for a rule violation which has partially applicable quick fixes we generate a string called `contextFlagsString` and pass it as an args to the `reportRuleForNode` function, see Listing 24. The quick fix then retrieves the `contextFlagsString` from the marker arguments and computes whether it is applicable, see Listing 25. The index of the `contextFlagsString` and the various context flags are defined as constants in the new class `ContextFlagsHelper`, see Listing 26.

```

1  @Override
2  public boolean isApplicable(IMarker marker) {
3      if (!super.isApplicable(marker)) return false;
4
5      String contextFlagsString = getProblemArgument(marker,
6          ContextFlagsHelper.VirtualFunctionShallHaveExactlyOne
7  SpecifierContextFlagsStringIndex);
8      return !contextFlagsString.contains(ContextFlagsHelper.Virtual
9  FunctionShallHaveExactlyOneSpecifierContextFlagPureVirtual)
10         && !contextFlagsString.contains(ContextFlagsHelper.
11  VirtualFunctionShallHaveExactlyOneSpecifierContextFlagIntroducing
12  Final);
13  }

```

Listing 25: Quick Fix computing isApplicable by contextFlagsString

```

1  public class ContextFlagsHelper {
2
3      public static final int    DoNotIntroduceVirtualFunctionInFinal
4  ClassContextFlagsStringIndex    = 0;
5      public static final String DoNotIntroduceVirtualFunctionInFinal
6  ClassContextFlagPureVirtual      = ":purevirtual";
7      public static final String DoNotIntroduceVirtualFunctionInFinal
8  ClassContextFlagIntroducingVirtual = ":introducingvirtual";
9      public static final int    VirtualFunctionShallHaveExactlyOne
10  SpecifierContextFlagsStringIndex    = 0;
11      public static final String VirtualFunctionShallHaveExactlyOne
12  SpecifierContextFlagPureVirtual      = ":purevirtual";
13      public static final String VirtualFunctionShallHaveExactlyOne
14  SpecifierContextFlagIntroducingFinal = ":introducingfinal";
15      public static final int    SwitchMustHaveAtLeastTwoCasesContext
16  FlagsStringIndex                  = 0;
17      public static final String SwitchMustHaveAtLeastTwoCasesContext
18  FlagTrivial                        = ":trivial";
19  }

```

Listing 26: ContextFlagsHelper containing contextFlagsString indices and flags

5.3.3. Test Results after Changes

As can be read above in Section 5.3.1, the CodeAnalysator plug-in had two big problems with performance. The first problem was that, some of the checkers used methods that were very slow. The other problem was, that the `isApplicable` checking was performed unnecessarily often. Please refer to the Section mentioned above for details. As can be seen in Figure 5.3 the test results of the performance measuring look a lot better than before we implemented the changes documented in Section 5.3.2.

Conclusion of performance tests

It was definitely a good idea to run performance tests. We found out, that most of our checkers run very fast. We were also able to fix those that did not. Look at the screenshot in Figure 5.3 for the results of the performance tests after applying changes. The execution time was reduced to less than one minute.

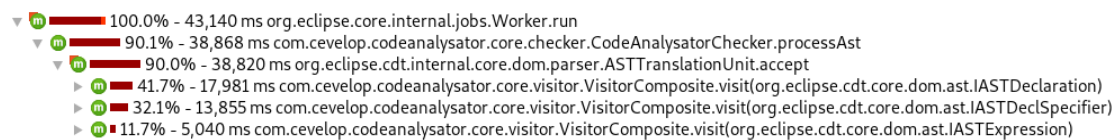


Figure 5.3.: The results of the final performance tests

In conclusion we can say that the performance of the CodeAnalysator plug-in was measurably improved. The execution time of the code analysis was reduced by a great amount and the scrolling no longer stutters and freezes. We consider these tests and the resulting changes made to the plug-in a success.

5.4. Conclusion

Both the functional and performance tests led to satisfying results. Both tests yielded a lot of opportunities to improve the plug-in. We do not claim to have found every problem and there are probably still some mistakes and edge-cases, that we did not consider. But our goal of making the plug-in more viable in a real-life project was definitely achieved. Together with the other parts of this thesis, such as the code improvement, the overall quality has been improved.

6. Quick Assists

During discussions of guideline rules there came up multiple ideas for quick assists in order to quickly refactor between different versions of code fragments, based on coding preferences. A quick assist, as opposed to the in this project more commonly used quick fix, does not resolve a checker. Instead it can be applied without being related to a rule. This major task consists of becoming acquainted with Eclipse quick assists and their respective API and designing and implementing those quick assists. The two planned quick assists for this projects are an Overrider and a Struct/Class-Switcher. The former is supposed to be able to switch between the virtual qualifiers (virtual, override, final). The other one is to transform a class into a struct in one click and vice versa.

6.1. Getting Started

Since the CDT documentation is a bit lacking in this regard, we tried to find out about quick assist from existing examples. Unfortunately though, there are not that many existing quick assists we could draw inspiration from. In the plugins project, there are only two such occurrences. The first is the `MacroExpansionQuickAssist` in the `Macronator`. The other is the `NamespactorQuickAssist` in the `Namespactor` project. Even though they both work a bit different from each other, we found out the base structure that such a quick assist has.

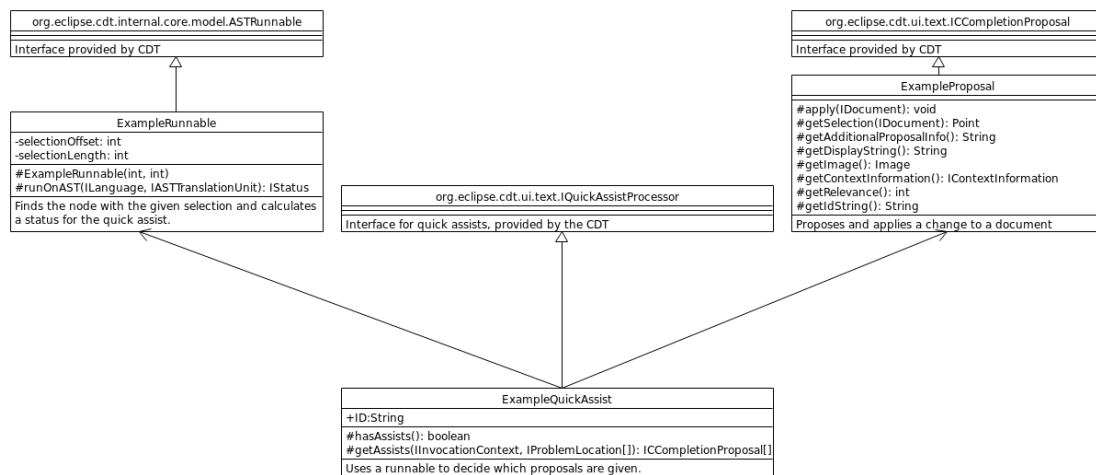


Figure 6.1.: The general structure of a quick assist

Figure 6.1 shows a general layout of a quick assist. The central class is the quick assist itself. Its ID field is the same that gets registered in the plugin.xml to identify the endpoint. The method `getAssists()` is called when the quick assists are loaded, for example when a user uses `ctrl + 1` somewhere in a code. In this method, a new runnable is generated. This runnable then finds the node matching the selection and checks which proposals are applicable for the found node. The quick assist then returns a list of proposals, which then get displayed to the user. If the user decides to apply one of the assists, the corresponding apply function is called. Unfortunately, this function only takes a `IDocument` parameter as opposed to an `ASTRewrite`, **which means we can only replace text instead of whole nodes.**

Solving the problem with IDocument As concluded in the previous paragraph, with the quick assist as is, we can only replace a section in the code with a String. This means we would have to build a String of the code we want to insert. For the Overrider this would be doable but even there can be things like additional spacing that could mess up the result. And for the Struct/Class-Switcher, the problem would, of course, be even worse. After discussing it in a meeting we decided to use refactorings. With a refactoring, we can manipulate the AST in the same way as with a quick assist. Figure 6.2 displays the class structure of the OverriderQuickAssist with its refactoring. The class structure for the struct/class switcher looks the same, just with different names. Note that the refactoring can now be called from both the quick assist menu (`Ctrl + 1`) but also via context menu and refactoring menu.

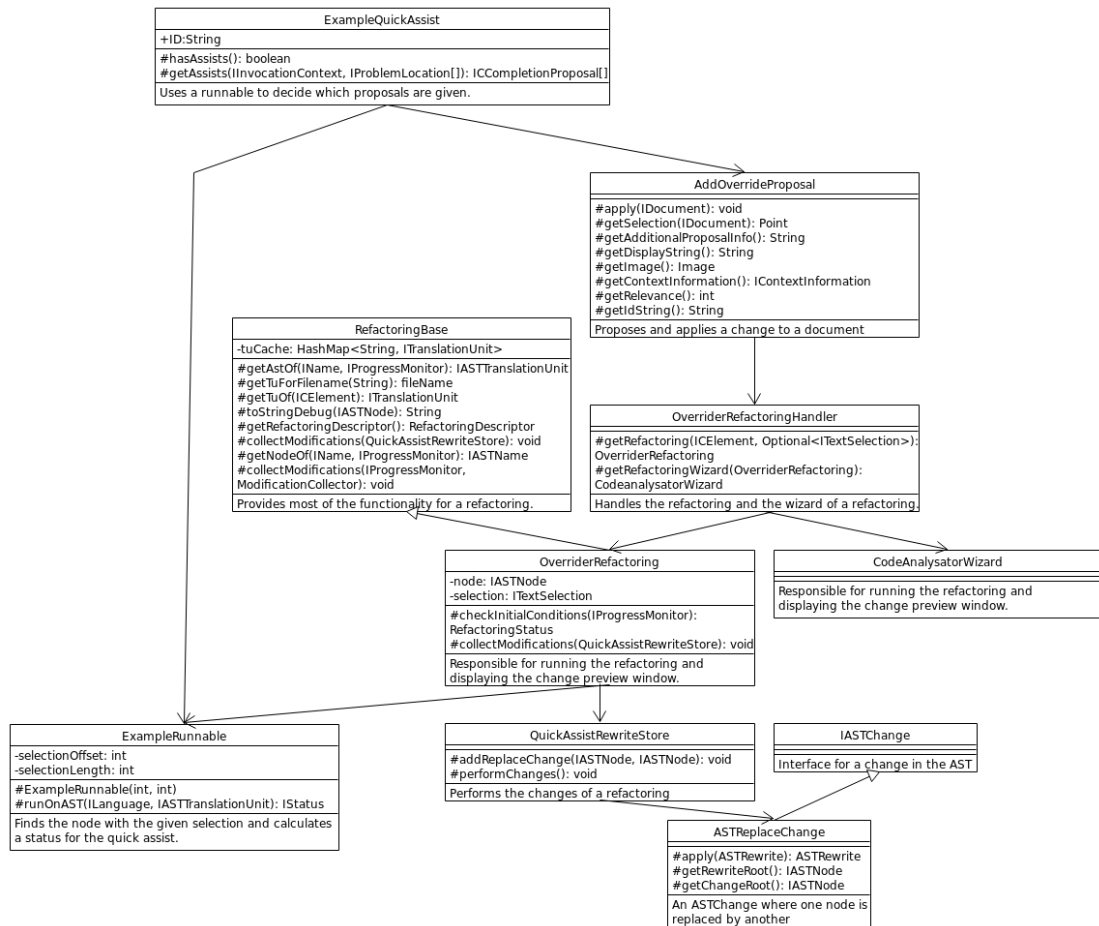


Figure 6.2.: Diagram of the completed Override quick assist and refactoring

For simplicity reasons, the diagram in Figure 6.2 omits a few helper classes and interfaces, that do not contribute directly to the concept of the refactoring or quick assist. More specifically, the classes `OverrideQuickAssist`, `AddOverrideProposal`, `OverrideRefactoringHandler`, `RefactoringBase` and `CodeAnalysatorWizard` implement an interface from the CDT.

6.2. Overrider

The goal of the Overrider is to fix the virtual specifiers of a function definition. The rules for what fixed means, is based on a discussion with Prof. Sommerlad. In general the idea is to have only one keyword for overriding functions which should in most cases be `override`. This means, if a method has more than one virtual specifier, all of them except `override` are removed and `override` is added, if it was not there. It can also add the `override` specifier to a function definition, if there was only the `virtual` keyword or no keyword at all. Of course this is not supposed to work on non-overriding functions. The Overrider first checks, whether a function overrides another or not. Please refer to Table 6.1 for a few examples.

Before	After
<pre> 1 <i>// The base class</i> 2 class A { 3 public: 4 virtual void F() {} 5 virtual void G() {} 6 virtual void H() {} 7 virtual void I() {} 8 }; 9 10 class B: public A { 11 public: 12 <i>// No virtual keyword</i> 13 void F{} 14 <i>// Final keyword</i> 15 void G final{} 16 <i>// Virtual keyword</i> 17 virtual void H{} 18 <i>// Virtual and Final</i> 19 virtual void final I(){} 20 }; </pre>	<pre> 1 <i>// The base class</i> 2 class A { 3 public: 4 virtual void F() {} 5 virtual void G() {} 6 virtual void H() {} 7 virtual void I() {} 8 }; 9 10 class B: public A { 11 public: 12 <i>// No virtual keyword</i> 13 void F override{} 14 <i>// Final keyword</i> 15 void G override{} 16 <i>// Virtual keyword</i> 17 void H override{} 18 <i>// Virtual and Final</i> 19 void I override (){} 20 }; </pre>

Table 6.1.: Examples of the Overrider

6.3. Struct/Class Switcher

The Struct/Class-Switcher enables one to switch from struct to class and the reverse, depending on which is preferred. The only difference between a struct and a class is in the default visibility (of inheritance and declarations). Therefore all that has to be changed, aside from the keyword, is the visibility of the base classes and a possible visibility label immediately at the beginning in the body. The different cases for the switches from struct to class and the reverse are listed in Table 6.2, respectively Table 6.3. Examples for the code before and after the switch can be seen in Table 6.4 and Table 6.5.

Switch From/To	struct	class
Default Visibility	public	private
Base Class Specifier	none (default) public protected private	public public protected none (default)
Immediate Visibility Label	none (default) public: protected: private:	public: public: protected: none (default)

Table 6.2.: Struct to class switch cases

Switch From/To	class	struct
Default Visibility	private	public
Base Class Specifier	none (default) public protected private	private none (default) protected private
Immediate Visibility Label	none (default) public: protected: private:	private: none (default) protected: private:

Table 6.3.: Class to struct switch cases

From struct	To class
<pre> 1 struct A { 2 3 }; 4 5 struct B: A { 6 7 }; 8 9 struct C: public A { 10 public: 11 }; 12 13 struct D: protected A { 14 protected: 15 }; 16 17 struct E: private A { 18 private: 19 }; 20 21 struct F: private A, 22 protected B, 23 C, 24 public D { 25 private: 26 }; </pre>	<pre> 1 class A { 2 public: 3 }; 4 5 class B: public A { 6 public: 7 }; 8 9 class C: public A { 10 public: 11 }; 12 13 class D: protected A { 14 protected: 15 }; 16 17 class E: A { 18 19 }; 20 21 class F: A, 22 protected B, 23 public C, 24 public D { 25 26 }; </pre>

Table 6.4.: Examples of switching from struct to class

From class	To struct
<pre> 1 class A { 2 3 }; 4 5 class B: public A { 6 public: 7 }; 8 9 class C: protected A { 10 protected: 11 }; 12 13 class D: private A { 14 private: 15 }; 16 17 class E: A { 18 19 }; 20 21 class F: public A, 22 protected B, 23 C, 24 private D { 25 public: 26 }; </pre>	<pre> 1 struct A { 2 private: 3 }; 4 5 struct B: A { 6 7 }; 8 9 struct C: protected A { 10 protected: 11 }; 12 13 struct D: private A { 14 private: 15 }; 16 17 struct E: private A { 18 private: 19 }; 20 21 struct F: A, 22 protected B, 23 private C, 24 private D { 25 26 }; </pre>

Table 6.5.: Examples of switching from class to struct

7. Per-Project Guideline Configuration

Currently, the guidelines can only be configured for the whole Eclipse workspace. In the guideline configuration guidelines can be enabled and disabled as well as ordered by priority. In case a rule is shared between multiple guidelines it is only reported for the highest priority guideline. See Figure 7.1 for a screenshot of the current guideline preferences page for the workspace.

Guidelines only being configurable on the workspace has multiple drawbacks. The configuration page is less discoverable, as it cannot be found in the project settings. It can also confuse users as the project and workspace settings are not consistent. Last, but not least, it limits users to a single configuration for the whole workspace and does not allow different priorities or even different sets of guidelines for different projects in the same workspace.

The goal of this task is to implement per-project configuration of guidelines. For every project it should be possible to choose whether to use workspace or project settings. When choosing project settings it should allow the same options as in the workspace settings.

The task consists first of getting acquainted with the current implementation of the guideline configuration, the relevant Eclipse configuration APIs and the existing preference pages support in ILTIS, and last devising and implementing a solution.

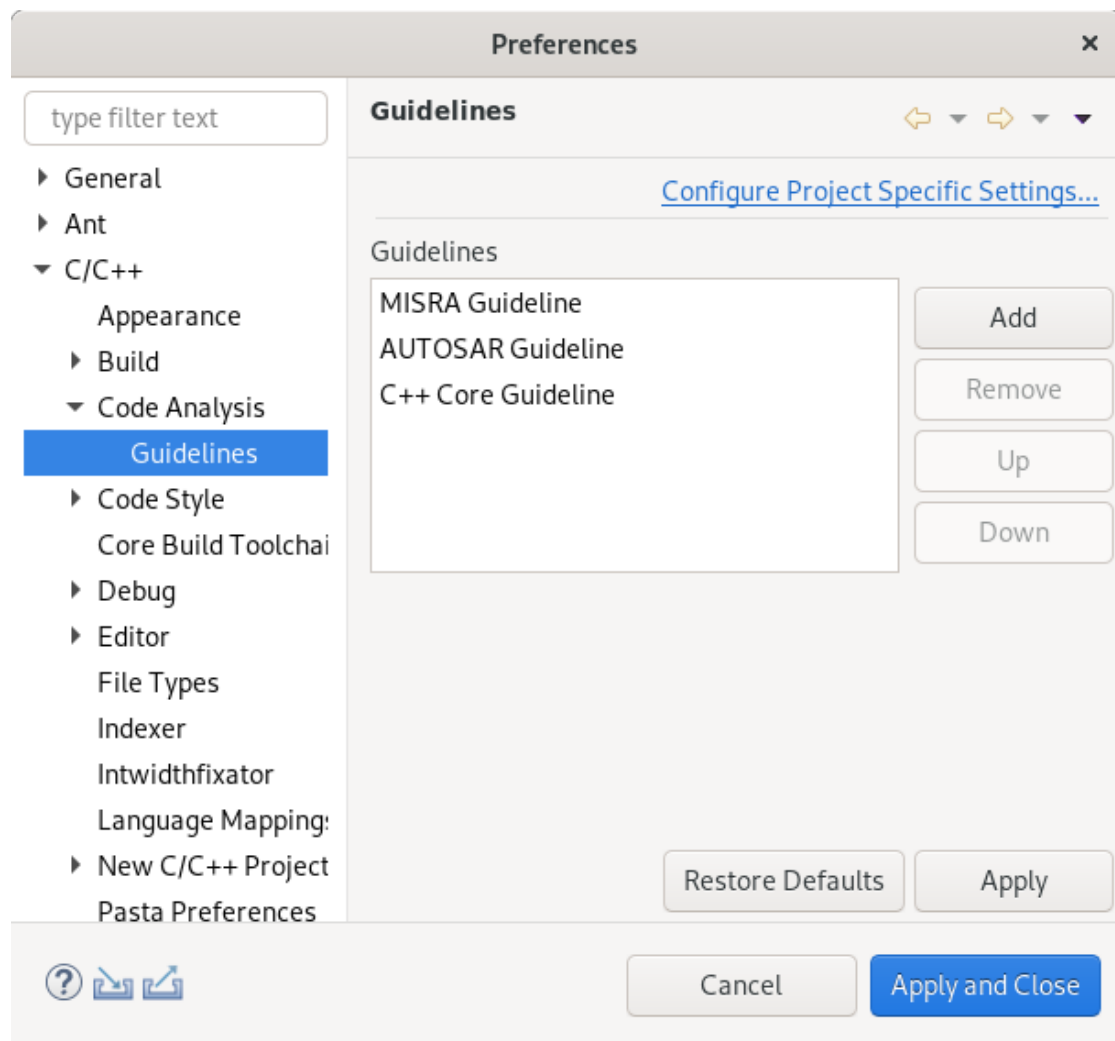


Figure 7.1.: The existing guideline preferences page for the workspace

Analysis ILTIS provides support for creating preference and property pages with `CFieldEditorPropertyAndPreferencePage`. It makes it possible to create a single page to be used for preferences and project properties. Further, it already provides linking between preference and property page, which only needs to be configured. Lastly, it provides a base class `PropertyAndPreferenceHelper` which implements accessing the active preferences values (i.e. project preferences if use project settings is enabled for the project, workspace preferences otherwise). The guideline preferences page `GuidelinePreferencePage` already derives from `CFieldEditorPropertyAndPreferencePage` and uses `PropAndPrefHelper` de-

iving from `PropertyAndPreferenceHelper`. Therefore we only need to properly register and configure the guideline preferences page as a property page to implement the per-project configuration dialog on the user-side.

For the per-project configuration to take effect we will have to adapt the `GuidelinePreferences` class to use the `PropAndPrefHelper` instead of using a `IPreferenceStore` directly. We will also need to adapt `GuidelinePriorityResolver` to compute priority orderings for every project separately.

Solution We extended `GuidelinePreferences` and `GuidelinePriorityResolver` for project preferences and registered the `GuidelinePreferencesPage` as a property page for projects with the C nature. The result can be seen in Figure 7.2. The preference query methods of `GuidelinePreferences` received a new overload taking an additional `IProject` parameter. The interface for `GuidelinePriorityResolver` did not even need to change. The main difficulty in this task laid in maintaining state in `GuidelinePreferences` and `GuidelinePriorityResolver` and keeping it consistent with the guideline preferences. For example, re-computing the guideline priorities when the preference on whether to use workspace or project settings of a project changes.

During refactoring of the solution we extracted an inner class in `GuidelinePreferences` and `GuidelinePriorityResolver` each. The inner class is responsible for managing the preferences, respectively the priorities for a single project. `GuidelinePreferences` and `GuidelinePriorityResolver` mainly delegate calls to the appropriate instance of the inner classes. We did so to avoid too deep generics nesting as a result of indexing the preferences and priority orderings by project.

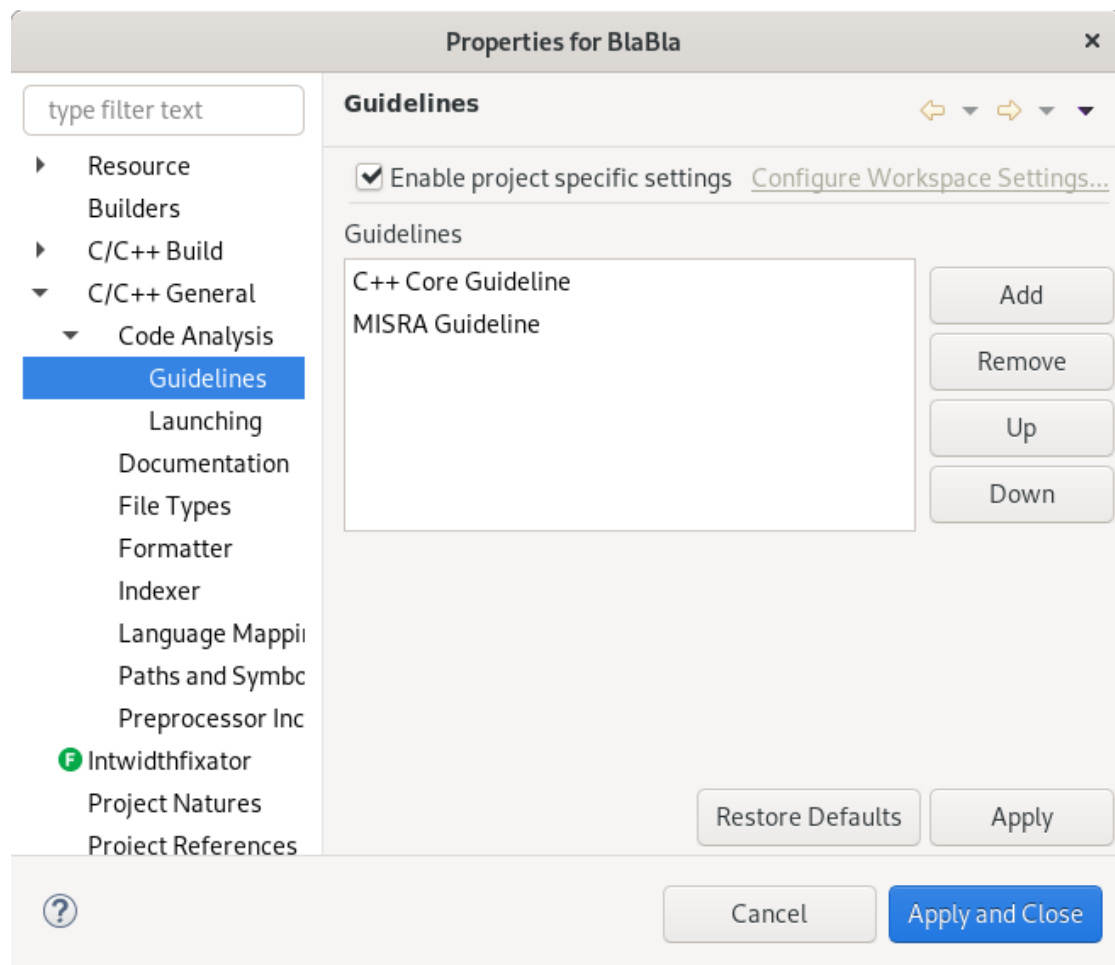


Figure 7.2.: The guideline preferences page now for projects

Conclusion Guideline preferences can now be specified for both workspace and projects. Each project can choose whether to use workspace settings or its own. The `ILTIS CFieldEditorPropertyAndPreferencePage` base class provides links to navigate between the guideline preferences of the workspace and those of the projects. Guideline preferences configuration is now more consistent and easier to discover for the user.

8. Implementation of checkers and quick fixes for further rules

Towards the end of the project we went back to our core discipline from the previous study project, which is implementing checkers and quick fixes. Some of the checkers and quick fixes planned for the study project have not been implemented. These will form the base of the list of rules for this task. The rest are new AUTOSAR rules, that we did not plan for a previous project yet [AUTOSAR 2018] [Puselja and Vlasek 2019].

8.1. Analysis

The first thing to do in this task is to decide which checkers and quick fixes should be implemented. Based on our experience in the previous study project, we will divide the rules and checkers into three different difficulty degrees as displayed in Table 8.1

	Trivial	Medium	Complex
Checker	4 h	12 h	24 h
Quick Fix	4 h	8 h	16 h

Table 8.1.: Estimated hours per task

8.1.1. Possible Rules to implement

This section contains a list of all the rules that we considered for the implementation. During analysis we created a list of all the AUTOSAR rules including those we already implemented and those we do not consider to be feasibly implementable in the scope of this project. Table 8.2 shows a list of all the rules we considered possible for us to implement. However it was not the plan to implement all of them, since there are far too many. This list can be taken as an inspiration as to which rules would make sense to implement in the future. The last column shows whether we implemented them or not. A detailed explanation of our implementation can be found in Section 8.2.

Nr	Rule	Checker	QuickFix	Implemented
A0-1-2	The value returned by a function having a non-void return type that is not an overloaded operator shall be used.	Medium	Trivial	No
A2-7-1	The character <code>\\</code> shall not occur as a last character of a C++ comment.	Medium		No
A2-10-1	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.	Medium		No
A2-11-1	Volatile keyword shall not be used.	Trivial		No
A2-13-1	Only those escape sequences that are defined in ISO/IEC 14882:2014 shall be used.	Medium		No
A2-13-5	Hexadecimal constants should be upper case.	Medium	Trivial	Yes
A2-13-3	Type <code>wchar_t</code> shall not be used.	Trivial		No
A3-3-2	Static and thread-local objects shall be constant-initialized.			No
A5-0-4	Pointer arithmetic shall not be used with pointers to non-final classes.	Medium	-	No
A5-0-3	The declaration of objects shall contain no more than two levels of pointer indirection.	Medium	-	Yes
A5-1-7	A lambda shall not be an operand to <code>decltype</code> or <code>typeid</code> .	Trivial	-	No
A5-2-1	<code>dynamic_cast</code> should not be used.	Trivial	-	Yes

A5-2-6	The operands of a logical&& or \\shall be parenthesized if the operands contain binary operators.	Medium	Medium	Yes
A5-10-1	A pointer to member virtual function shall only be tested for equality withnull-pointer-constant.	Medium	-	Yes
A6-2-2	Expression statements shall not be explicit calls to constructors of temporary objects only.	Trivial	-	No
A6-4-1	A switch statement shall have at least two case-clauses, distinct from the default label.			No
A6-5-2	A for loop shall contain a single loop-counter which shall not have floating-point type.	Medium	-	No
A6-5-4	For-init-statement and expression should not perform actions other than loop-counter initialization and modification.	Complex	-	No
A7-1-9	A class, structure, or enumeration shall not be declared in the definition of its type.	Trivial	Complex	No
A7-6-1	Functions declared with the [[noreturn]] attribute shall not return.	Complex	-	No
A8-4-4	Multiple output values from a function should be returned as a struct or tuple.	Medium	-	No
A8-5-4	If a class has a user-declared constructor that takes a parameter of typestd::initializer_list, then it shall be the only constructor apart from specialmember function constructors	Medium	-	No
A11-3-1	Friend declarations shall not be used.	Medium	-	No
A12-0-1	If a class declares a copy or move operation, or a destructor, either via“=default”, “=delete”, or via a user-provided declaration, then all others ofthese five special member functions shall be declared as well.	Medium	Medium	No

A12-1-2	Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type.	Complex	Complex	No
A12-1-4	All constructors that are callable with a single argument of fundamental type shall be declared explicit.	Medium	-	No
A12-1-6	Derived classes that do not need further explicit initialization and require all the constructors from the base class shall use inheriting constructors.	Medium	Medium	No
A12-4-1	Destructor of a base class shall be public virtual, public override or protected non-virtual.	Trivial		No
A12-8-7	Assignment operators should be declared with the ref-qualifier &.	Medium		No
A13-1-2	User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters.	Medium		No
A13-2-1	An assignment operator shall return a reference to “this”.	Medium		No
A13-2-2	A binary arithmetic operator and a bitwise operator shall return a “prvalue”.	Medium		No
A13-2-3	A relational operator shall return a boolean value.	Medium		No
A13-5-1	If “operator[]” is to be overloaded with a non-const version, const version shall also be implemented.	Medium		No
A13-5-2	All user-defined conversion operators shall be defined explicit.	Medium		Yes
A13-5-5	Comparison operators shall be non-member functions with identical parameter types and noexcept.	Medium		No
A13-6-1	Digit sequences separators ‘ shall only be used as follows: (1) for decimal, every 3 digits, (2) for hexadecimal, every 2 digits, (3) for binary, every 4digits.	Medium		No
A14-8-2	Explicit specializations of function templates shall not be used.	Medium		No

A15-1-1	Only instances of types derived from <code>std::exception</code> should be thrown.	Medium		No
A15-1-2	An exception object shall not be a pointer.	Medium		No
A15-3-5	A class type exception shall be caught by reference or const reference.	Medium		Yes
A15-4-1	Dynamic exception-specification shall not be used.			No
A18-1-1	C-style arrays shall not be used.	Trivial	Medium	No
A18-1-2	The <code>std::vector<bool></code> specialization shall not be used.	Trivial		No
A18-5-1	Functions <code>malloc</code> , <code>calloc</code> , <code>realloc</code> and <code>free</code> shall not be used.	Medium		No
A18-9-1	The <code>std::bind</code> shall not be used.			No
A18-9-3	The <code>std::move</code> shall not be used on objects declared <code>const</code> or <code>const&</code> .	Medium		No
A23-0-1	An iterator shall not be implicitly converted to <code>const_iterator</code> .	Medium		No
A26-5-1	Pseudorandom numbers shall not be generated using <code>std::rand()</code> .	Trivial		No

Table 8.2.: List of possible rules

8.2. New rules

The checkers and quick fixes for the rules in this section were added during this project.

8.2.1. A02-13-05: Hexadecimal constants should be upper case.

Rule This rule wants the developer to write hexadecimal constants in uppercase. This leads to better readability and consistency [AUTOSAR 2018].

Analysis To implement this checker, an understanding of literal expressions is needed. Literal expressions in C++ can be various different things such as numbers or strings but also `'true'` and `'false'` or the `this` pointer [User-defined literals (since C++11) - *cppreference.com* 2019]. Additionally, some literals like integers and floating point numbers can have a prefix and a suffix. Suffixes are used to indicate

the type of the literal. For example a 'u' or 'U' at the end of a literal makes it unsigned. For this rule the prefixes are more of interest. The prefixes indicate the base of the literal. For example a leading zero indicates an octal literal. For this rule a the hex prefix is more interesting though. The prefix for hex literals is 0x or capitalized 0X [*Integer literal in C/C++ (Prefixes and Suffixes) - GeeksforGeeks 2019*].

Checker Implementing the checker was not very complex but some parts of the code are done a bit inelegant. The visitor for this rule visits all expressions and only continues if the expression is a literal expression. The next thing that needs to be done is to decide, whether the literal is of a type that can have a hex prefix. This sorts out a lot of cases. This is done by checking the kind of the literal. If it is either `lk_integer_constant` or `lk_float_constant`, it might possibly be a hex literal. After it is clear that the literal can be a hex constant, the visitor needs to verify whether such a prefix exists. Unfortunately, the class `CPPASTLiteralExpression` does not offer any functionality to find this out. Because of this, the visitor looks at the value of the literal and checks whether the first two characters represent this prefix. If so it is simply a matter of verifying whether the rest of the literal has lower case letters, excluding the suffix of course. If so, the node is reported. Please refer to Listing 27 for an example.

1	<code>int</code>	<code>a = 0xF20b;</code>	<code><---</code>	Not OK
2	<code>float</code>	<code>b = 0xF20B;</code>	<code><---</code>	OK
3	<code>std::string</code>	<code>c = "0xF20B";</code>	<code><---</code>	OK

Listing 27: A02-13-05 checker example

Quick Fix The quick fix is a lot simpler than the checker. What it needs to do is simply take the reported literal and replace it with its uppercase version. It does need to take into account though that the 'x' in the prefix can be upper or lowercase and should not change with the quick fix. Additionally, there might be a suffix as well, which should not be capitalized. The quick fix removes the prefix and suffix from the literal, makes the value uppercase and reattaches the prefix and suffix. Refer to Listing 28 for a few examples.

```

1      // Before applying the quick fix.
2      float    a = 0Xf20f;
3      int      b = 0xF20f;
4      double   c = 0xf20ful;
5
6      // After applying the quick fix
7      float    a = 0XF20F;
8      int      b = 0xF20F;      // Kept lowercase x
9      double   c = 0xF20Ful;    // Kept lowercase x and suffix

```

Listing 28: An example of the quick fix for A02-13-05

8.2.2. A5-0-3: The declaration of objects shall contain no more than two levels of pointer indirection.

Rule The rationale of the rule is that more than two levels of pointer indirection may seriously impair the understandability of the code and should be avoided. The rule applies to direct as well as indirect indirections through typedefs. Arrays are also considered one level of pointer indirection in case they are used as function parameters because they are converted to a pointer to the initial element of the array. The rule generally applies to any declaration of objects. [AUTOSAR 2018, p.85]

Analysis As we are interested in declarations of objects only, we only have to concern ourselves with simple declarations (`IASTSimpleDeclaration`) and parameter declarations (`IASTParameterDeclaration`). Each object in a declaration is introduced by a declarator (`IASTDeclarator`). The type of the object can be modified by operators (`IASTPointerOperator`). Relevant for us are pointer operators (`IASTPointer`) and, in the case of function parameters, array declarators (`IASTArrayDeclarator`). The pointer operators are part of the declarator and can be applied recursively. The array of operator is represented using a distinct declarator. The base type of the objects is specified by the declaration specifier (`IASTDeclSpecifier`) as part of the declaration. Types introduced through type aliases or typedefs may consist of pointer indirection. Type alias or typedef types are specified using named type specifiers (`IASTNamedTypeSpecifier`). [*Declarations - cppreference.com* 2019]

Checker For this checker we visit each simple declaration and function definition. In case of a simple declaration we iterate through each declarator and count the

number of pointer operators and, in the case of function parameters, the array of operator as well, each. In case of a function definition or a function declarator in a simple declaration we iterate through each parameter declaration and do the same. Additionally, we test whether the declaration specifier is a named type specifier. If so, we resolve any type aliases, then count the number of nested pointer types (IPointerType). This count is added to each declarator count of the declaration. If the total number of indirection for any declaration is larger than two we report the declarator. See Listing 29 for examples violating the rule.

```
1 std::uint8_t ***pointer { }; // [A5-0-3]
2 using uint8_ptr = std::uint8_t *;
3 uint8_ptr **pointer2 { }; // [A5-0-3]
4 void (*func)(std::uint8_t ***pointer) { }; // [A5-0-3]
5 void (*func2)(std::uint8_t **pointer[]) { }; // [A5-0-3]
```

Listing 29: Examples of declarations violating rule A5-0-3

8.2.3. A05-02-01: `Dynamic_cast` should not be used

Rule Dynamic casts are used to safely cast objects along the inheritance tree. Unfortunately, dynamic casts are relatively slow and therefore not suitable for environments where high performance and low memory usage are required [AUTOSAR 2018]. Refer to Listing 30 for an example of the usage of a `dynamic_cast`.

```

1  class A {
2  public:
3      virtual void F();
4  };
5  class B: public A {
6  public:
7      void F() override {
8      }
9  };
10 void Fn(A* a) {
11     B* b { dynamic_cast<B*>(a) };
12     if (b != nullptr) {
13         // Use B class interface
14     } else {
15         // Use A class interface
16     }
17 }

```

Listing 30: Usage of `dynamic_cast`. Taken from [AUTOSAR 2018]

Analysis Besides the reasons given by the rule, the usage of `dynamic_cast` is very often a design smell. For example would extending the hierarchy tree require changes in places where `dynamic_cast` was used. Usually the same behaviour can be achieved with virtual methods. This is of course not fixable by a quick fix. This is why we decided to implement a checker, but no quick fix for this rule [dynamic_cast conversion - cppreference.com 2019] [Gibbs and Stroustrup 2006].

Checker Every `dynamic_cast` is represented in the AST with a `ICPPASTCastExpression` and is therefore an expression. Also this class has a method `getOperator` which returns the kind of cast this is. With these tools it is possible to look at every expression and if it is a `ICPPASTCastExpression` with kind `dynamic_cast` report it.

8.2.4. A05-02-06: The operands of a logical `&&` or `||` shall be parenthesized if the operands contain binary operators.

Rule This rule helps with clarity. The order of evaluation in complicated logical expressions is solved implicitly in C++. This can lead to confusion as it might not be clear on first glance. This is why this rule makes the developer add parentheses

if the operands contain binary expressions.

Checker The checker visits all expressions and filters for binary expression that have either the operator `&&` or `||`. Most cases of this rule can then be covered with the same check. A binary expression has two operands. These operands can be different things like among others variable or another expression. Relevant for this rule is when one or both of the operands are again binary expressions. In most cases, this already violates the rule. That is because if there were parenthesis, the binary expression would be wrapped in a unary expression.

```
1 void doSomething(int value){  
2     if((value > 0) && value <3){}  
3 }
```

Listing 31: Example of a binary expression

To visualize this, look at the code in Listing 31 and corresponding display of the abstract syntax tree in Figure 8.1. As can be seen, the right operand of the main binary expression has directly another binary expression as a child, while on the left side there is a unary expression first.

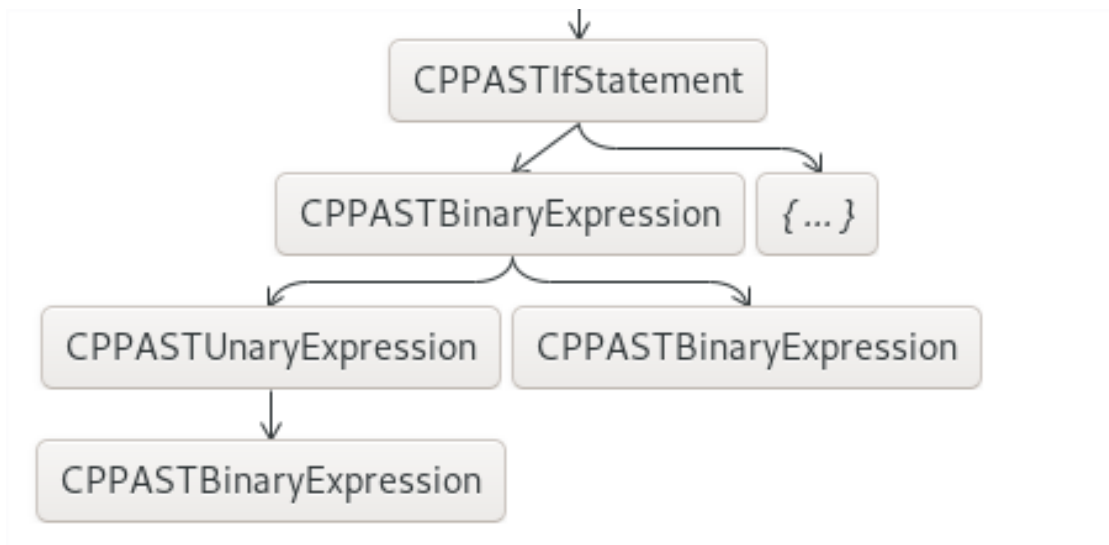


Figure 8.1.: A part of the AST for the example in Listing 31

Quick Fix Marked unnested binary expressions can be resolved by replacing the corresponding subexpressions with unary expressions that in turn contain the subexpression. It is a bit more difficult for nested subexpressions. Nested in this context means that there are binary expressions with unbracketed binary expressions as operands, which in turn are also binary expressions with unbracketed binary expressions as operands. And so on. A complicated example is shown in Listing 32.

```
1 void doSomething(int value){}
2     while ((value > 17 || 0 || value && 3 < value || true &&
3         value > 1000) && value > 0) {}
4 }
```

Listing 32:

The first problem with such cases is, that there are multiple checkers for the same rule on the same line. For each of them, there is also a proposed quick fix. There are now two possibilities to solve this.

1. Do not do anything special and let the user activate the quick fix for every occurrence.
2. Traverse the AST and find the root binary expression no matter which of the quick fixes was chosen. From there recursively parenthesize every occurrence.

Both of these approaches have advantages and disadvantages. But we think it makes more sense to resolve all the markers, even though it means, there might be a lot changing at once which might make it a bit harder to comprehend. Our opinion is that it's better to do everything at once because it will also keep the semantics of the program consistent. Look at the example in Listing 33 to see how the example in Listing 32 was resolved.

```
1 void doSomething(int value){}
2     while ((((((value > 17) || 0) || (value && (3 < value))) ||
3         (true && (value > 1000))) && (value > 0)) {}
4 }
```

Listing 33: The complicated example resolved

8.2.5. A5-10-1: A pointer to member virtual function shall only be tested for equality with null-pointer-constant.

Rule The rule specifies that pointer to virtual member functions should only be equality compared to the null-pointer constant. In general, pointer-to-members can only be compared for equality. The rationale given for this rule is that equality comparison of a pointer to a virtual member function with anything other than a null-pointer constant is unspecified. [AUTOSAR 2018, p.111] [*Comparison operators* - *cppreference.com* 2019]

Analysis Whether a pointer-to-member points to a virtual member function is not part of its type. We will therefore have to perform some kind of data-flow analysis for the checker to be useful. The `DoNotOutliveReferenceCapturedObjectsVisitor` already does something similar and its data-flow analysis could potentially be reused. Pointer-to-members can be initialized with the expression `&C::m` and we can start the analysis from there. The type of the expression would be pointer-to-member (`ICPPPointerToMemberType` and the member pointed to being of type function (`IFunction`). We will then have to check if that member function is virtual. We can resolve the name of the member in order to get a method binding (`ICPPMethod`). Checking whether a method is overriding another method is already implemented in the `VirtualHelper`. Relevant for this rule are comparisons of pointer-to-members. Comparisons are binary expressions (`IASTBinaryExpression`). There are two equality comparison operators: [*Pointer declaration* - *cppreference.com* 2019] [*Comparison operators* - *cppreference.com* 2019]

- `a == b`: equal to (`IASTBinaryExpression.op_equals`)
- `a != b`: not equal to (`IASTBinaryExpression.op_notequals`)

Checker The checker visits all expressions and tests whether they are an pointer-to-member initialization expression (`&C::m`) and that member is a virtual function. We call such expressions an address of virtual member function expression. On finding such an expression the checker tests if the expression is used in a comparison. If so, and the expression is compared with something other than a null-pointer constant, the comparison is reported. Otherwise, if the expression is used in an assignment or initialization and the variable is of type pointer-to-member, the variable is considered an alias. Then the above check is recursively repeated for every expression in the translation unit where the alias is used. The alias checking is performed with the help of the `AliasHelper` which was refactored and extracted from the `DoNotOutliveReferenceCapturedObjectsVisitor`. See Listing 34 for an example violating the rule.

```

1 struct A {
2     void non_virtual_func() { }
3     virtual void virtual_func() { }
4 };
5 bool Fn() noexcept
6 {
7     void (A::* virtual_func_pointer)() { &A::virtual_func };
8     return virtual_func_pointer == &A::non_virtual_func;
9     // ^-- [A5-10-1]
10 }

```

Listing 34: Example of comparison violating rule A5-10-1

8.2.6. A7-5-1: A function shall not return a reference or a pointer to a parameter that is passed by reference to const.

Rule According to this rule functions should return neither references nor pointers to its parameters which are passed by reference to const. The reason given is that reference to const bind to both lvalues and rvalues. Therefore the argument passed to the function as reference to const may be a temporary object, which is destroyed when the function ends. The reference or pointer returned by the function would then point to an invalid object and accessing it would lead to undefined behavior. [AUTOSAR 2018, p.147,148]

Analysis For this rule we are interested in function definitions (IASTFunctionDefinition). The return type (IType) and parameters (IParameter) of the function may be obtained from the function binding (IFunction). References are represented as a wrapping reference type (ICPPReferenceType). Constness is represented as a wrapping qualifier type (IQualifierType). Pointers are represented as a pointer type (IPointerType), which also specifies constness. To track potential references and pointers, we can use the `AliasHelper`, which implements basic data-flow analysis.

Checker The checker visits all function definitions and resolves their binding. It tests whether a function returns either a reference or a pointer. If so, it checks each parameter passed by reference to const. If the function returns a reference, the checker tests whether the parameter is returned directly or indirectly. If the the function returns a pointer, the checker tests whether the the address of the

pointer, obtained by the address-of operator (&), is returned directly or indirectly. In both cases, the return statement returning the value is reported. See Listing 35 for examples reported by the checker.

```
1 struct A {
2 };
3 A const & Fn1(A const &parameter) noexcept
4 {
5     return parameter; // [A7-5-1]
6 }
7 using const_pointer = A const *;
8 const_pointer Fn2(A const &parameter) noexcept
9 {
10     A const * variable { &parameter };
11     return variable; // [A7-5-1]
12 }
```

Listing 35: Example of comparison violating rule A7-5-1

8.2.7. A13-05-02: All user-defined conversion operators shall be defined explicit

Rule This rule defines that all user-defined conversion operators must be explicit. To make an operator explicit, the explicit keyword needs to be added. This keyword means that the operator needs to be explicitly called and cannot be called implicitly [*explicit specifier - cppreference.com* 2019] [AUTOSAR 2018].

Analysis A user-defined conversion operator is a function definition. So the checker needs to find out whether it is a function definition and if so, whether it is explicit. A function definition can be made explicit by adding the explicit keyword. This is what we could offer as a quick fix. We do not offer a quick fix though, because it might break the code. If the code already used the operator in a discouraged way, the code will not compile anymore after our quick fix.

```

1 struct A{
2     A(double value): a{value}{}
3     explicit operator double() const { return a; }
4     double a;
5 };
6
7 struct B{
8     B(double value): b{value}{}
9     operator double() const { return b; }
10    double b;
11 };
12
13 int main()
14 {
15     A a{3.141592653589793238462643383279502884197169399375105820974};
16     B b{3.141592653589793238462643383279502884197169399375105820974};
17
18     double da {a}; // <-- OK
19     double dn {b}; // <-- OK
20
21     float fa {a}; // <-- Compilation error
22     float fb {b}; // <-- Narrowing conversion but no compilation error
23
24     return 0;
25 }

```

Listing 36: The difference between an explicit and a non explicit conversion operator

The example in Listing 36 shows the problem with non explicit conversion operators. The types A and B are very similar. They both have a double value and are convertible to double. The difference is that the conversion operator of type A is declared explicit as opposed to the one from type B, which is not declared explicit. This leads to a compile error on line 21 where the code tries to implicitly convert the double value of A to float. The same conversion does not result in a compilation error for type B because it is not explicit. Instead the conversion of B to a float value causes a data loss, which might go unnoticed. This is why user defined conversion operators should be declared with the explicit keyword.

Checker The visitor reports a function declaration if the two following criteria are met. First, the function declaration needs to be a conversion operator.

This can be tested by acquiring the name of declarator. If the name is of type `ICPPASTConversionName`, the declarator is a conversion operator. The second thing to check is whether the declarator is explicit. To determine this, the visitor uses a helper method in `ASTHelper`. The declaration gets marked if it is a conversion operator but not explicit.

8.2.8. A15-03-05: A class type exception shall be caught by reference or const reference

Rule This rule states that class type exceptions must be caught by reference or const reference. The problem with catching an exception by value is that slicing might occur when a derived class is thrown but a base class is caught. [AUTOSAR 2018].

Analysis Slicing means that an object of a derived type was assigned to a variable of its base type. This causes the part of the type that was added in the derived class to be sliced off. This is why this checker is needed. Refer to Listing 37 for an example.

```
1      catch (std::runtime_error e) {} // caught by value
2      catch (std::runtime_error & e) {} // caught by reference
```

Listing 37: Examples of catching a reference by value and by reference

Checker The checker needs to find all catch statements. Then, it is simply a matter of looking at whether the exception has a `ICPPASTReferenceOperator`. If so, it was caught by reference. If not, the node gets reported.

8.3. Existing rules

Important The checkers and quick fixes for the rules in this section were added before this project and are therefore out of scope for this thesis. Their documentation is copied directly from the previous term project and were not updated for this project [Puselja and Vlasek 2019].

8.3.1. A5-0-2: Condition of if/while/for shall be bool

Checker: The checker tests all statements and continues if they are either an if, while, do or for statement. If so, their corresponding condition is inspected. If the condition is a not a boolean, the statement is marked. But there are a few exceptions. If there is a type initialization sequence, it is not marked, even though it might be something besides bool. Also, if a type has an explicit conversion to bool, it is ok. If not, it is marked. This is because it is contextually converted to bool in this case. See the example below. [AUTOSAR 2018, p. 80] [*Implicit conversions - cppreference.com* 2018]

Figure 8.2.: Example for A5-0-2 [AUTOSAR 2018]

```
1  struct A {
2      explicit operator bool() const { return false; }
3  };
4
5  struct B {
6      operator bool() const { return false; }
7  };
8
9  void F(){
10     A a{};
11     B b{};
12     if(a){} // compliant
13     if(b){} // non-compliant
14     return false;
15 }
```

The rule is not completely clearly specified. For example, there is a snippet in which the following code is non-compliant.

Figure 8.3.: Unclear example [AUTOSAR 2018]

```
1  std::uint8_t u = 8;
2  bool boolean1 = false;
3  bool boolean2 = true;
4  if (u && (boolean1 <= boolean2)); // non-compliant
```

We assume that the problem is the conversion from `u`, which is a numeral, to a boolean. This happens because the logical AND-operator needs two bool values as operands. This conversion is implicit, which is prohibited according to the rule. But technically the conversion happens when executing the AND-operation as opposed to when the condition is evaluated. According to the description, this would be compliant. But for this project we assume that the condition must not contain any implicit bool conversions. [AUTOSAR 2018]

Quick fix: As stated in the requirements chapter, it was initially planned to create a quick fix for this rule. Upon further research it became clear that it would be very complicated to create a satisfying quick fix. The rule does not specify how it should be resolved. A problem is that a quick fix might either make the code less readable or do something that is technically correct but does not really make sense. That is why we decided to not implement a quick fix for this rule.

Future possibilities: In the future a quick fix could be done but more time would be needed to to analyze the rule than is available in this project.

8.3.2. A5-1-1: No magic numbers

Checker: Recognizing a literal value in the code is achieved by checking whether an expression is a literal expression. The rule states that literal values are allowed if they are in an initialization. By checking whether the parent of the literal expression is an IASTInitializer, most exceptions are already found. There is one special case though, which has to be checked separately. This special case is when the initialization is part of a unary expression. For example, in a return or throw statement: `throw std::logic_error("LogicError")` There is also another exception which states that literal values are allowed when used for logging. But since there is no reliable way to find out whether a literal value was used for logging, we decided to leave it up to the user to suppress the warning if needed. [AUTOSAR 2018, p. 86]

Quick fix: A possible fix for this problem would be to extract the literal value into a constant or variable. This would require the user to enter a variable name. But a quick fix is not intended to use user input. For cases like this, refactorings should be used. Refactorings are not in the scope for this project, which is why this is not possible. Another possibility is to generate a name from the code, but this would lead to bad names that would have to be refactored again. That is the reason why we decided it is best to just show the problem to the user and not implement a quick fix.

Future possibilities: Working with this rule turned on can be tedious. In the future it could be made configurable, which types of literal values are to be marked and which are not. For example, `true`, `false` and `nullptr` are literal expressions which most people probably don't want to have marked every time. This customization is realizable but lies out of the scope of this project. In a future project, this could very well be done.

8.3.3. A5-1-2: Do not implicitly capture variables in lambda expressions

Checker: The checker reports all lambda expressions with capture defaults. [AUTOSAR 2018, p. 88]

quick fix: Removes all capture defaults from lambda expressions and adds explicit captures for all previously implicitly captured variables. We studied the cppreference page on lambda expressions for which variables are implicitly captured when a capture default is present. [*Lambda expressions (since C++11)* - *cppreference.com* 2018] A variable is implicitly captured by a lambda expression, if the variable is defined outside the scope of the lambda, is potentially evaluated, cannot be used without a capture and is not explicitly captured. The quick fix computes implicitly captured variables by visiting all variable uses inside the body of a lambda expression and, if the above condition holds true, adds the variable to the implicitly captured variables. [AUTOSAR 2018, p. 88] [*Lambda expressions (since C++11)* - *cppreference.com* 2018]

8.3.4. A5-1-3: Parameter list shall be included in lambda

Checker: The checker for this rule is a simplified version of the checker for rule A5-1-6. It tests whether the expression is a lambda and has a declarator. If not, there is no parameter list and the lambda is reported. [AUTOSAR 2018, p. 89]

Quick fix: This quick fix adds an empty declarator to the lambda expression, which results in an empty parameter list. An empty parameter list is always correct, as an omitted parameter list means a lambda takes no parameters. [*Lambda expressions (since C++11)* - *cppreference.com* 2018]

8.3.5. A5-1-4: Lambda expressions shall not outlive reference captured objects

Checker: The checker visits all lambda expressions and reports statements or expressions which allow the lambda expression to escape, meaning potentially allowing the lambda expression to outlive its reference captured objects. Return statements allow lambdas to escape. Except when the lambda expression is inside another lambda expression and the reference captured objects are still valid outside the lambda. Assignment expressions allow lambdas to escape if the assignment target is a global variable, a field, a reference parameter, a local variable with static storage duration or a local variable outside the scope of a reference captured object. Additionally, the checker performs simple alias checking. In case the value of a lambda expression is assigned to a variable, all uses, and aliases of that variable are recursively checked as well. [AUTOSAR 2018, p. 90]

The checker for this rule was more elaborate than our highest estimation category. There were no specific problems or unexpected difficulties, implementing the different escape and alias checks was simply more time-consuming than expected. It should have been estimated with a higher amount or a higher new complexity category.

Quick fix: There is no feasible way to implement a quick fix for this rule, as there is no general algorithm to rewrite reference captures into copy captures while preserving the behaviour of the code.

8.3.6. A5-1-6: Explicit lambda return type

Checker: The checker visits all expressions and checks whether it is a lambda expression. If so, it tests whether its return type is specified explicitly or implicitly. The return type of a lambda expression is specified implicitly, if the trailing-return-type is omitted or if the trailing-return-type is the `auto` or `decltype(auto)` specifier. If the return type is implicit and not void, the lambda is reported. [AUTOSAR 2018, p. 91] [*Lambda expressions (since C++11)* - *cppreference.com* 2018]

Quick fix: The quick fix creates a new lambda declarator with the lambda return type deduced by the CDT. It then either replaces the old declarator or simply

inserts the new one, depending on whether the lambda already had a declarator.

8.3.7. A5-1-8: Do not nest lambda expressions

Checker: The checker visits all lambda expressions and checks whether they are enclosed by another lambda expression inside the same function. If so, the nested lambda expression is reported. [AUTOSAR 2018, p. 93]

Quick fix: There is no sensible way to automatically refactor a lambda expression away.

8.3.8. A5-16-1: Do not use the ternary conditional operator as sub-expression

Checker: The checker visits all expressions and, in the case of a ternary conditional operator, checks whether it is used as a sub-expression. If the conditional operator is used as a sub-expression, the conditional operator is reported. If the super expression is a parentheses unary expression, the check is instead performed for the parentheses expression. [AUTOSAR 2018, p. 111]

Quick fix: There is no sensible way to automatically refactor a nested expression out.

8.3.9. A6-4-1: Switch shall have at least two cases

Checker: The checker visits all switch statements and counts the number of case-clauses. If there are less than two case-clauses the switch statement is reported. [AUTOSAR 2018, p. 117]

Quick fix: This quick fix turned out to have more edge cases than expected. After a thorough study of the article on switch statements on cppreference, we came to the following solution. Switch statements with less than two case-clauses can be replaced with a construct consisting of an optional init-part, an optional if-part, an optional else-part and an optional unconditional following-part. The quick fix is not applicable for switch statements containing statements outside of clauses. The init-part contains a possible init-statement, a controller declaration or in the case of no if-part, a controller expression. The if-, else- and following-parts contain the statements of the clauses. The clauses are assigned to a part depending on which clauses exist, whether they are fall-through and their order. Lastly, the quick fix checks whether the replacement code declares an identifier conflicting

with an existing identifier in the surrounding scope. If there are no conflicts, the quick fix replaces the switch statement with the replacement code directly, otherwise, the switch statement is replaced with a compound statement containing the replacement code. The following table shows how the clauses of the switch statement are assigned to the if-, else- and following-parts. [*switch statement - cppreference.com* 2018]

Case	If-part	Else-part	Following-part
No clauses	-	-	-
Only a default-clause	-	-	default-clause
Only a case-clause	case-clause	-	-
Case-clause and default-clause	case-clause	default-clause	-
Fall-through from case to default	case-clause	-	default-clause
Fall-through from default to case	default-clause	-	case-clause

Table 8.3.: Assignment of clauses to if-, else- and following-parts

8.3.10. A7-1-4: The register keyword shall not be used

Checker: It is deprecated, and some compilers even ignore it. The use of the register keyword makes the code less portable because it doesn't work in the same way on every machine. Also, the functionality of it is just a suggestion to the machine to store the value in a register. Even on machines that support the keyword, it is not guaranteed that it has any effect. The checker marks all occurrences of the keyword. [AUTOSAR 2018, p. 129] [*C++ keywords: register - cppreference.com* 2018]

Quick fix: The quick fix removes the register keyword.

8.3.11. The auto specifier shall not be used

Checker: This checker visits all declarations. If a declaration is a simple declaration, the checker tests whether the declaration is declared with the auto declaration specifier and if the variables are not initialized with a function call or initializer of non-fundamental type. If so, the declaration is reported. [AUTOSAR 2018, p. 129] [*auto specifier (since C++11) - cppreference.com* 2018] [*Fundamental types - cppreference.com* 2018] Otherwise, if a declaration is a function definition, the checker tests whether the function is defined with the auto declaration specifier and if the function is not a template function using trailing return type syntax. If

so, the function definition is reported. [AUTOSAR 2018, p. 129] [*auto specifier (since C++11)* - *cppreference.com* 2018]

Quick fix: In case of a variable declaration, the quick fix iterates through all declarators in the declaration and inserts for each declarator a new declaration with the deduced type and declarator, before removing the auto declaration. If the declarator is an rvalue reference declarator and the deduced type is an lvalue reference, the declarator is changed to an lvalue reference, according to auto specifier deduction rules. [*auto specifier (since C++11)* - *cppreference.com* 2018]

In case of a function defined with the auto keyword, the quick fix removes an eventual trailing return type and replaces the auto declaration specifier with a specifier of the deduced type. [*auto specifier (since C++11)* - *cppreference.com* 2018]

NOTE: Due to limitations in Eclipse CDT, declaration specifiers cannot be correctly constructed for template instances with `std::char16_t`, `std::char32_t`, reference (&), pointer (*), pointer to member (T::*) and enumeration non-type template parameters, because the argument used to instantiate the template is not available. As a fallback the template instance is constructed using the types instead of the values as arguments. This does not compile but should be sufficient as a hint to which arguments are required.

This quick fix took significantly longer than estimated, mainly due to two reasons. Unexpected changes aside from replacing the auto keyword with the deduced type (const- and volatile-qualifiers, auto&& reference being an rvalue or lvalue reference depending on assigned value) and non-existing functionality to generate an AST type specifier from a semantic type, requiring its implementation for this quick fix.

8.3.12. A7-1-6: The typedef specifier shall not be used

Checker: Implementation of this rule was trivial. We copied the visitor that simply checks declarations for the keyword typedef and marks every occurrence. [AUTOSAR 2018, p. 131]

Quick fix: The quick fix replaces the typedef with one or multiple using statements.

8.3.13. A7-2-2: Enumeration underlying base type shall be explicitly defined

Checker: This checker visits all enumeration specifiers and reports them, if they do not have an explicitly specified base type. [AUTOSAR 2018, p. 137] [*Enumeration declaration* - *cppreference.com* 2018]

Quick fix: The quick fix has only been partially implemented, due to missing support in CDT to insert enum base types.

Future possibilities: In the future, with added support in CDT to insert the base type between name and enumerators, the quick fix could be completed.

8.3.14. A7-2-3: Declare enumerations as scoped enum classes

Checker: The checker visits all enumeration specifiers and reports enums which are not scoped. [AUTOSAR 2018, p. 138] [*Enumeration declaration - cppreference.com* 2018]

Quick fix: A quick fix changing the enum to an enumeration class would have to additionally change all implicit conversions from the enum to numeric values to explicit casts and qualify all unqualified references to the enumeration values. Therefore no quick fix has implemented. [*Enumeration declaration - cppreference.com* 2018]

8.3.15. A7-2-4: Initialize none, the first or all enumerators

Checker: This checker visits all enumeration specifiers and reports them, if a single enumerator is initialized and it is not the first or if more than one enumerator is initialized but not all. [AUTOSAR 2018, p. 139] [*Enumeration declaration - cppreference.com* 2018]

Quick fix: The quick fix has only been partially implemented, due to missing support in CDT to rewrite enumerators.

Future possibilities: In the future, with added support in CDT to rewrite enumerators, the quick fix could be completed.

8.3.16. A7-4-1: The asm declaration shall not be used

Checker: Using the asm keyword, it is possible to run assembler code in C++. But this makes the code very machine dependent and is discouraged. The checker marks every occurrence of the asm keyword. [AUTOSAR 2018, p. 142] [*asm declaration - cppreference.com* 2018]

Quick fix: Since writing a parser that translates the assembler code into C++ would greatly exceed our time budget, we decided to not implement a quick fix for this rule.

8.3.17. A8-5-2: Braced-initialization {}, without equals sign, shall be used for variable initialization

Checker: This checker visits all simple declarations (variable declarations). The checker reports non-braced-initializers in declarations unless non-braced-initialization is required. Parentheses initialization is required when calling a constructor which would be overshadowed by an initializer-list constructor with braces initialization. An exception is made for auto declarations, as braced-initialization of auto variables would violate the following rule A8-5-3. [AUTOSAR 2018, p. 170] [*list initialization (since C++11) - cppreference.com* 2018] [*direct initialization - cppreference.com* 2018]

NOTE: The checker has not been implemented for temporary object, object with dynamic storage duration (new-expression) and constructor member initialization, due to time constraints and a possible constructor resolution bug in CDT.

Quick fix: The quick fix replaces constructor initializer (parentheses) with initializer lists (braces) when parentheses initialization is not required. [*list initialization (since C++11) - cppreference.com* 2018] [*direct initialization - cppreference.com* 2018]

8.3.18. A8-5-3: A variable of type auto shall not be initialized using {} or ={} braced-initialization

Checker: There are multiple ways to initialize a variable in C++ using auto;

Figure 8.4.: Different ways of initializing an auto variable[*Initialization - cppreference.com* 2018]

1	<code>auto x1 (10);</code>	<code>// constructor initializer</code>
2	<code>auto x2 = 10;</code>	<code>// equals initializer</code>
3	<code>auto x3 = int{};</code>	<code>// typed initializer list</code>
4	<code>auto x4 {10};</code>	<code>// initializer list</code>
5	<code>auto x5 = {10};</code>	<code>// equals initializer list</code>

C++ performs type deduction when initializing a variable with `auto`. In the first three cases of the example the expected type will be deduced. But if an initializer list is used, the deduced type will be initializer list. This can be confusing so if it is used, it will be flagged. [AUTOSAR 2018, p. 173] [*auto specifier (since C++11)* - *cppreference.com* 2018]

Quick fix: The same quick fix as for rule A7-1-5 is used. See section 8.3.11 for more information.

8.3.19. A9-5-1: Unions shall not be used

Checker: This rule has a special exception that allows to use tagged unions before the introduction of `std::variant`. It would have been very difficult to check whether a union was used in such a way. `std::variant` was introduced in C++ 17, so starting from there, it is not allowed anymore. That is why we decided not to check for this exception. Because of this decision, the visitor was quite simple to implement. The visitor simply searches for the keyword `union` and marks it if found. [AUTOSAR 2018, p. 179] [*std::variant* - *cppreference.com* 2018]

Quick fix: There is no fix for this, that will be correct in all cases. We decided not to implement a quick fix for this rule.

8.3.20. A10-1-1: Class shall not be derived from more than one base class which is not an interface class.

Checker: The glossary of the AUTOSAR document defines an interface class as follows: A class is an interface class if there are only public pure virtual methods and public static constexpr data members. Additionally there can be a virtual destructor. The visitor looks at all the base classes and counts how many of them are not interfaces. If that number is larger than one, the class is flagged. [AUTOSAR 2018, p. 183]

Quick fix: There is no way to decide which classes should be removed, so we decided to not add a quick fix.

8.3.21. A10-2-1: Do not redefine non-virtual member functions

Checker: If a non-virtual method is redefined in a derived class, the initial function is hidden. To detect this, the base-classes of the derived class need to be checked. This also includes classes that are multiple levels higher in the inheritance

tree or that are in another file. Every non-virtual member function of each base-class is compared to the redefined function. If they are equal, the base method is shadowed. This needs to be marked. To perform this check, there is some additional functionality needed such as finding all base classes, finding out whether methods override or not and so forth. This functionality is in a separate helper class called VirtualHelper. This class is used for some other checkers as well. The functionality of it was heavily inspired by the class OverrideIndicatorManager from the CDT. [AUTOSAR 2018, p. 185] [*OverrideIndicatorManager* 2018]

Figure 8.5.: B.F() shadows A.F()

```

1  struct A
2  {
3      public:
4          void F() noexcept {}
5  };
6  struct B : public A
7  {
8      public:
9          void F() noexcept { } // hides F() from A
10 };

```

Quick fix: There is no fix for this, that will be correct in all cases. We decided not to implement a quick fix for this rule.

8.3.22. A10-3-1: Virtual function declarations shall have exactly one of the specifiers virtual, override or final

Checker: This rule can be violated in two ways. Either there are too many specifiers, or there is none when there should be one. The visitor counts how many VirtualSpecifiers are present on the declarator. Since the virtual keyword is not contained in the VirtualSpecifiers, this needs to be checked separately by looking at the method through the binding. If the total amount is larger than 1, the declarator is marked. If the total amount is 0, but the function is virtual, it is marked as well. To find out whether it is virtual, the VirtualHelper is used. [AUTOSAR 2018, p. 186]

Quick fix: As stated in the checker section, there are multiple ways in which this rule can be violated. Here are the different cases. The rule specifies that virtual

should only be used to declare new virtual methods.

1. If there is a final specifier, we assume, that it is supposed to be final. In that case all the other specifiers are removed.
2. If there is an override and virtual specifier, and the method does override a method in the base class, the virtual specifier is removed.
3. If a keyword is missing, but there is supposed to be one because it overrides a method in its base class, the override keyword is added.

8.3.23. A10-3-2: Each overriding virtual function shall be declared with the override or final specifier.

Checker: The checker checks every function that overrides another function in a base class, whether it has one of the two virtual specifiers `override` and `final`. If it does not have one, the declarator is flagged. A special case are destructors. If a base class has a virtual destructor, the destructor of the derived class overrides the base destructor, even though they have a different signature. Hence, it must have an `override` keyword as well and is marked if it does not. [AUTOSAR 2018, p. 188]

Quick fix: There are two solutions for this problem. The user must choose which one they prefer. That is why there are two quick fixes. One that adds the `final` keyword and one that adds the `override` keyword.

8.3.24. A10-3-3: Do not introduce virtual member functions in a final class

Checker: Since deriving from a final class is not possible, defining a method virtual in it is inconsistent and should therefore not be done. There are two ways a method can be virtual. [AUTOSAR 2018, p. 189]

1. The first one is straight forward when the method has the `virtual` keyword. In that case the declaration is always marked when it appears inside of a final class. The problem is, that C++ allows a few special cases. For example, it is possible to introduce a pure-virtual final method. This member cannot be derived from, but it is still virtual.
2. The second possibility is, when a member overrides a virtual base class member. This cannot be recognized by traversing the AST. Instead the helper class `VirtualHelper` is used again. Note that this case is already caught by the rule A10-3-1 most of the times but it still needs to be checked, because a user might disable the other rule.

Quick fix: The quick fix is only applicable if the member is not pure-virtual. Otherwise if it is virtual because of the virtual keyword, the keyword is removed. If it is virtual because it is overriding a virtual function, the override keyword is replaced with the final keyword. For declarations it must be handled differently. For example, if the function is virtual, you cannot remove the virtual keyword and add the final keyword. That would cause a compile error, so it is not applicable.

8.3.25. A10-3-5: User defined assignment operators shall not be virtual

Checker: It is possible to override an assignment operator of a base class A in a derived class B. This allows to call the operator on B with an argument of type B. This can lead to undefined behaviour. It is possible to identify a method as an operator if the name is an instance of ICPPOperatorName. But this class does not offer any functionality to find out what kind of operator it is. Internally the operator is created by appending the operator (i.e. *=, = or —=) to the word operator. That is how we check now as well. There is a list of operator names and if it is contained in there, the operator is flagged. [AUTOSAR 2018, p. 190]

Quick fix: There is no fix for this, that will be correct in all cases. We decided to not implement a quick fix for this rule.

8.4. Conclusion

We implemented eight new checkers and two new quick fixes during this project which is less than we initially wanted. This is mainly due to the fact that other parts of the project took more time than expected. But it is not too bad, since we planned this part of the project to be some kind of buffer that can be easily expanded or made smaller. We are still content with what we achieved and consider the newly implemented rules to be worthy additions.

9. Quality Measures

The goal of this chapter is to list the different measures we took to ensure a good quality of the project. This includes the technical report as well as the written code.

- All checkers and quick fixes are covered with various automated tests. These tests are run locally regularly. After each commit, they are also run on the CI Server. This way problems are immediately recognized.
- A large part of this thesis consists of performance testing and functional testing with a real project. This will further improve the quality and robustness of the plugin. Using the insight from these tests, we will improve the code.
- Some parts of the code that we find to be particularly critical are created by applying pair-programming.

10. Conclusion

In this chapter we will summarize what we did and achieved in the different parts of this project. We also give a brief outlook on further work that could be done.

10.1. Retrospective

The goals of this thesis were to improve upon the CodeAnalysator plug-in and extend the functionality with quick assists and new checkers as well as quick fixes for further rules of the AUTOSAR guidelines. In a preceding term project we implemented many checkers and quick fixes and gained experience with the plug-in. Based on this experience, we started out our thesis with refactoring the code in order to improve the architecture of the CodeAnalysator plug-in. Our main objectives were to improve upon the many mappings in the `GuidelineConflictResolver` and to simplify the large number of classes responsible for rule suppression. We believe we were quite successful at accomplishing that. We were able to refactor the need for many of those mappings away and split the `GuidelineConflictResolver` according to its various responsibilities. We were also able to significantly reduce the number of classes in suppression while also simplifying the suppression logic. Together with many smaller refactorings throughout our thesis we improved the quality of the code and the architecture of the CodeAnalysator plug-in.

Our other undertaking to improve the CodeAnalysator plug-in consisted in testing the functionality and the performance of the plug-in on actual projects. For this purpose we used the two open source projects LevelDB, a key-value storage library, and Jucipp, a lightweight C++ IDE, with the projects containing from 20'000 (LevelDB) to 40'000 (Jucipp) lines of code. During the functional testing we were able to confirm the correctness of most of our checkers and quick fixes, with the exception of a few errors mostly resulting from not considered edge cases. Exceptions being thrown were very rare. We fixed the errors encountered and improved the reliability of the checker and quick fixes.

In the second part we ran performance tests. While most checkers had a negligible running time, we discovered two major bottlenecks. Firstly, there was a group of checkers, which took a very long time to run and accounted for the vast majority of the running time. This bottleneck was caused by the checkers compiling related files and accessing the AST instead of querying the index, which already contained

all the required information. The second bottleneck was caused by the applicability checking of quick fixes. The checks required compiling the AST and as the checks are run every time a problem marker comes into view, this resulted sluggish scrolling. Fixing these two bottlenecks greatly improved the performance of the CodeAnalysator plug-in and made it usable for realistic projects.

We are very satisfied with the improvements we were able to make and performing the functional and performance testing was definitely worth while. While the functional testing improved the correctness and reliability of the checkers and quick fixes, the performance testing made them usable for realistic projects. Together with the refactorings performed in the first task we achieved our goal of improving the CodeAnalysator plug-in.

Following the code improvement and testing we extended the functionality of the CodeAnalysator plug-in with two quick assists, Override and Struct/Class-Switcher. The quick assists offer small refactorings to be performed, like replacing a struct with a class definition. We were able to implement both planned quick assists and they offer additional useful functionality for the IDE.

Another requested additional functionality was to be able to configure guideline preferences per-project as opposed to only for the whole workspace. The reasons behind were twofold. One, for the actual functionality of being able to configure the preferences on a finer granularity. Two, to make the user interface more consistent and easier discoverable. We implemented the per-project configuration and herewith extended the functionality and improved the usability of the CodeAnalysator plug-in.

Lastly, we implemented new checkers and quick fixes for the AUTOSAR guidelines. With code improvement and testing being rather large tasks, the scope of new checkers and quick fixes was smaller than in the preceding term project. Nonetheless we extended the CodeAnalysator plug-in with additional checkers and quick fixes and refactored some of the existing ones.

As a whole we have achieved the goals of thesis. We improved the code and architecture of the CodeAnalysator plug-in, discovered and fixed errors in its checkers and quick fixes, resolved performance bottlenecks and extended the plug-ins functionality with quick assists, a per-project preferences page and new checkers and quick fixes.

10.2. Considerations for future projects

In the respective chapters considerations for future projects have already been mentioned. Here we will only summarize the two most significant ones. Firstly, implementing a mono checker. Currently, one checker is run per guideline. This results in redundancy and makes the design more complicated by requiring these

checkers to be coordinated. Having a single checker running all visitors would mostly simplify the code while requiring additional code to register problems and visitors of guidelines. Complexity, where not reduced, would be moved from run time to initialization. Secondly, making some checkers configurable. Currently, checkers can be either disabled or enabled but none can be configured. For some checkers configuration would be very useful. For example A05-01-01, which disallows using literals in general, except some hard-coded cases, could have its exceptions made configurable, letting the user decided which kinds of literals were permissible and which prohibited.

Bibliography

Gibbs, Michael and Bjarne Stroustrup (2006). **Fast dynamic casting**. In: *Software - Practice and Experience* 36.2, pp. 139–156. ISSN: 00380644. DOI: 10.1002/spe.686.

AUTOSAR (2018). **Guidelines for the use of the C++14 language in critical and safety-related systems**. AUTOSAR.

Deicha, Andreas and Pascal Bertschi (2018). **Safe C++ Guidelines Checkers und Quick Fixes**.

Kormanyos, C.M. (2018). **Real-Time C++: Efficient Object-Oriented and Template Microcontroller Programming, Third Edition**. Springer, Heidelberg.

MISRA C++ : 2008 Guidelines for the use of the C++ language in critical systems (2018). June 2008. ISBN: 9781906400033.

Puselja, Viktor and Gabriel Vlasek (2019). **Safe C++ Guidelines**.

C++ Core Guidelines (2019). URL: <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines> (visited on 06/11/2019).

C++ Operator Precedence - cppreference.com (2019). URL: https://en.cppreference.com/w/cpp/language/operator%7B%5C_%7Dprecedence (visited on 06/05/2019).

Comparison operators - cppreference.com (2019). URL: https://en.cppreference.com/w/cpp/language/operator%5C_comparison (visited on 06/05/2019).

cppit / jucipp · GitLab (2019). URL: <https://gitlab.com/cppit/jucipp> (visited on 05/28/2019).

dynamic_cast conversion - cppreference.com (2019). URL: https://en.cppreference.com/w/cpp/language/dynamic%7B%5C_%7Dcast (visited on 06/04/2019).

explicit specifier - *cppreference.com* (2019). URL: <https://en.cppreference.com/w/cpp/language/explicit> (visited on 06/06/2019).

GitHub - ckormanyos/real-time-cpp: Real-Time C++ Companion Code (2019). URL: <https://github.com/ckormanyos/real-time-cpp> (visited on 05/28/2019).

GitHub - google/leveldb: LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values. (2019). URL: <https://github.com/google/leveldb> (visited on 05/28/2019).

id-Software/DOOM-3: Doom 3 GPL source release (2019). URL: <https://github.com/id-Software/DOOM-3> (visited on 04/12/2019).

Integer literal in C/C++ (Prefixes and Suffixes) - GeeksforGeeks (2019). URL: <https://www.geeksforgeeks.org/integer-literal-in-c-cpp-prefixes-suffixes/> (visited on 06/03/2019).

Java Profiler - JProfiler (2019). URL: <https://www.ej-technologies.com/products/jprofiler/overview.html> (visited on 04/30/2019).

PeterSommerlad: Anforderungen An Studienarbeiter - (2019). URL: <https://wiki.hsr.ch/PeterSommerlad/wiki.cgi?AnforderungenAnStudienarbeiter> (visited on 03/05/2019).

Pointer declaration - *cppreference.com* (2019). URL: https://en.cppreference.com/w/cpp/language/pointer#Pointers_to_members (visited on 06/07/2019).

STAN - Structure Analysis for Java (2019). URL: <http://stan4j.com/> (visited on 03/12/2019).

User-defined literals (since C++11) - *cppreference.com* (2019). URL: https://en.cppreference.com/w/cpp/language/user%7B%5C_%7Dliteral (visited on 06/03/2019).

asm declaration - *cppreference.com* (2018). URL: <https://en.cppreference.com/w/cpp/language/asm> (visited on 12/17/2018).

attribute specifier sequence(since C++11) - *cppreference.com* (2019). URL: <https://en.cppreference.com/w/cpp/language/attributes> (visited on 04/27/2019).

auto specifier (since C++11) - *cppreference.com* (2018). URL: <https://en.cppreference.com/w/cpp/language/auto> (visited on 12/17/2018).

Statements - *cppreference.com* (2019). URL: https://en.cppreference.com/w/cpp/language/statements#Compound_statements (visited on 03/16/2019).

Declarations - *cppreference.com* (2019). URL: <https://en.cppreference.com/w/cpp/language/declarations> (visited on 03/16/2019).

direct initialization - *cppreference.com* (2018). URL: https://en.cppreference.com/w/cpp/language/direct%5C_initialization (visited on 12/17/2018).

Bug 309760 - [checker][quick fix] Provide checker for name resolution problems found by indexer (2019). URL: https://bugs.eclipse.org/bugs/show_bug.cgi?id=309760 (visited on 05/04/2019).

Enumeration declaration - *cppreference.com* (2018). URL: <https://en.cppreference.com/w/cpp/language/enum> (visited on 12/17/2018).

Fundamental types - *cppreference.com* (2018). URL: <https://en.cppreference.com/w/cpp/language/types> (visited on 12/17/2018).

Implicit conversions - *cppreference.com* (2018). URL: https://en.cppreference.com/w/cpp/language/implicit%7B%5C_%7Dconversion (visited on 12/17/2018).

Initialization - *cppreference.com* (2018). URL: <https://en.cppreference.com/w/cpp/language/initialization> (visited on 12/17/2018).

Lambda expressions (since C++11) - *cppreference.com* (2018). URL: <https://en.cppreference.com/w/cpp/language/lambda> (visited on 12/17/2018).

list initialization (since C++11) - *cppreference.com* (2018). URL: https://en.cppreference.com/w/cpp/language/list%5C_initialization (visited on 12/17/2018).

OverrideIndicatorManager (2018). URL: <http://git.eclipse.org/c/cdt/org.eclipse.cdt.git/tree/core/org.eclipse.cdt.ui/src/org/eclipse/cdt/internal/ui/editor/OverrideIndicatorManager.java?id=71ed78fcbef9ef8e620ab9a2d043763981bcaeaf> (visited on 11/05/2018).

C++ keywords: register - *cppreference.com* (2018). URL: <https://en.cppreference.com/w/cpp/keyword/register> (visited on 12/17/2018).

std::variant - *cppreference.com* (2018). URL: <https://en.cppreference.com/w/cpp/utility/variant> (visited on 12/17/2018).

switch statement - *cppreference.com* (2018). URL: <https://en.cppreference.com/w/cpp/language/switch> (visited on 12/17/2018).

Glossary

AUTOSAR Guidelines for the use of the C++14 language in critical and safety-related systems [AUTOSAR 2018]. 13, 24, 132, 133

checker Checks C++ code for a given rule. 132, 137

Codan A static analysis framework in Eclipse CDT.. 6, 19, 34

CodeAnalysator Eclipse static code analysis plugin. 131

Eclipse Eclipse is an integrated development environment. 129, 131

guideline A set of rules for programming. 129

rule A single regulation of a guideline regarding a specific programming concept or element. 129

Appendix

A. Developer Guide

The goal of this chapter is to provide information and guidance on how to implement a new rule in the CodeAnalysator plug-in. It is intended to be read by developers. Any reader should be familiar with terms like checker, visitor, marker and quick fix. For detailed information on Eclipse plug-in development, please refer to the ILTIS documentation. [Deicha and Bertschi 2018]

Implementing a rule can be roughly divided into the tasks checker, quick fix and testing. These tasks are covered in the following subsections. It is not necessary to tackle these tasks in any given order, but there are some limitations i.e. a quick fix is not testable without a corresponding visitor.

Note In the documentation to the previous project, there already exists a developer guide, written by the same authors. Over the course of this project, we changed the architecture of the plug-in significantly. This invalidated parts of the existing developer guide. Therefore this chapter with an updated developer guide exists. Some parts of the old guide are still valid and are used here again [Puselja and Vlasek 2019].

A.1. Checker

This section explains how to create a checker for a given rule. In the CodeAnalysator plug-in, there are not multiple checkers. Instead there is one checker for every guideline with many different visitors each.

A.1.1. Visitor

This section covers the creation of a visitor.

Step 1: Register your problemId in AutosarIdHelper The problemId is used in several places to refer to the rule. It consists of the default qualifier plus ".problem." plus a name of your choosing. The name should make it easy to link to the corresponding AUTOSAR rule. Check the snippet below for an example. See listing 38.

```
1 package com.cevelop.CodeAnalysator.autosar.util;
2
3 public class AutosarIdHelper {
4     public static final String DEFAULT_QUALIFIER =
5         "com.cevelop.CodeAnalysator.autosar";
6     ...
7     public static final String ExampleProblemId =
8         DEFAULT_QUALIFIER + ".problem.example";
9     // Your new problemid
10 }
```

Listing 38: AutosarIdHelper

Step 2: Create the visitor This step consists of adding the visitor itself. Add a class extending `CodeAnalysatorVisitor` to the visitor package of the corresponding guideline. If the visitor is supposed to be used by multiple guidelines, put it into the visitor.shared package in the core plug-in. Next, add the proposed constructor and the method `setShouldVisit()` which should be proposed by eclipse. In `setShouldVisit()` you can set the different types of nodes you want to visit. The example in listing 39 visits all expressions. For every type you want to visit, you need to add a visit method as well. After initialization, your visitor could look like in Listing 39.

```

1 public class ExampleVisitor extends CodeAnalysatorVisitor{
2
3     protected ExampleVisitor(Rule rule, RuleReporter ruleReporter) {
4         super(rule, ruleReporter);
5     }
6
7     @Override
8     protected void setShouldVisit() {
9         shouldVisitExpressions = true;
10    }
11
12    @Override
13    public int visit(final IASTExpression expression) {
14        return super.visit(expression);
15    }
16 }

```

Listing 39: AutosarIdHelper

Step 3: Register the problem in the fragment.xml To make a rule appear in the settings panel in eclipse, you need to register the rule in the file `fragment.xml` which can be found in the root folder of the AUTOSAR bundle. You need to add your rule to the `codan.core.checkers` extension point. You need to provide a category, a severity a description, a message pattern a name and a marker type. Additionally, you need to specify whether the rule is enabled by default. There is also an id which refers to the problem id created in the last section. Check the code snippet in Listing 40 to see how to format.

```
1 <extension point="org.eclipse.cdt.codan.core.checkers">
2     <checker>
3         <problem
4             category="com.cevelop.CodeAnalysator.core.autosar"
5             defaultSeverity="Warning"
6             defaultEnabled="true"
7             description="AXX-XX-XX: Example description"
8             id="com.cevelop.CodeAnalysator.autosar.problem.
9             exampleproblemid"
10            messagePattern="AXX-XX-XX: Example Message"
11            name="AXX-XX-XX: Example name"
12            markerType="com.cevelop.CodeAnalysator.core.autosar.
13            marker">
14        </problem>
15        <problem>
16            <!-- your new problem -->
17        </problem>
18    </checker>
19 </extension>
```

Listing 40: fragment.xml

Step 4: Add a rule to the guidelines In every fragment of the CodeAnalysator except for the core fragment, there is a guideline class. These guidelines already have multiple guidelines registered. To add a new one, add a new member variable of type rule. In the constructor, use the registry to create and register the rule. You need to pass a number, the guideline which is usually `this` and the `problemId` that was defined in step 1. If the rule uses a visitor that is used in multiple guidelines you should also enter a `sharedID`. This will enable priority checking and improve performance. See Listing 41 for an example.

```

1 public class AutosarGuideline implements IGuideline {
2     public static Rule A02_13_05_HexValuesMustBeUppercase;
3     public static Rule A04_07_01_AvoidLossyConversions;
4     private static final String autosarSuppressTag = "autosar";
5
6     private AttributeSuppressionStrategy suppressionStrategy =
7     new AttributeSuppressionStrategy(autosarSuppressTag, "autosar");
8
9     public AutosarGuideline() {
10         RuleRegistry registry =
11         CodeAnalysatorRuntime.getDefault().getRuleRegistry();
12
13         //No shared id, used only in this guideline
14         A02_13_05_HexValuesMustBeUppercase =
15         registry.createAndRegisterRule("A2-13-5", this,
16         AutosarIdHelper.HexValuesMustBeUppercaseProblemId);
17
18         // With shared id, used in multiple guidelines
19         A04_07_01_AvoidLossyConversions = registry.createAndRegisterRule
20         ("A4-7-1", this, AutosarIdHelper.AvoidLossyConversionsProblemId,
21         CoreIdHelper.AvoidLossyConversionsSharedProblemId);
22     }
23     //...
24 }

```

Listing 41: fragment.xml

Step 5: Add the visitor to the checkers Every guideline has exactly one checker in its checker package. For AUTOSAR this would for example be `com.cevelop.CodeAnalysator.autosar.checker`. In this class all the visitors of this guidelines are listed. In the method `initVisitor`, add your newly created visitor like in Listing A.1.1.

```

1  @Override
2      protected Collection<CodeAnalysatorVisitor> createVisitors
3          (RuleReporter ruleReporter) {
4
5          Collection<CodeAnalysatorVisitor> visitors = new ArrayList<>();
6          visitors.add(new ExampleVisitor(ExampleGuideline.
7              AXX_XX_XX_Example, ruleReporter));
8      }

```

Listing 42: AutosarChecker.java

A.1.2. Testing

This section describes the steps to take when writing checker tests.

Step 1: Add a new test class Add a class that extends the checker test base class for the corresponding bundle to the package `com.cevelop.CodeAnalysator.X-tests.checker`. The base class for AUTOSAR for example is `AutosarCheckerTestBase`. This class does not contain the actual tests, which makes the file very short. The only thing that needs to be done is overriding the function `getRuleToCheck`. In this method, return the Rule that you created in step 4 of the create visitor tutorial above. Look at the code in Listing A.1.2 to see an example test class.

```

1  public abstract class ExampleCheckerTest extends AutosarCheckerTestBase {
2
3      @Override
4      protected Rule getRuleToCheck() {
5          return AutosarGuideline.AXX_XX_XX_ExampleRule;
6      }
7  }

```

Listing 43: Adding the test class to the test suite

Step 2: Register the test There is a class `PluginUITestSuiteAll.java` in the package `com.cevelop.CodeAnalysator.autosar.tests`. To add your tests to the test suite of the plug-in, the visitor must be added to the list of test classes in this file as is shown in Listing A.1.2.

```
1 @SuiteClasses({
2     // ...
3     ExampleCheckerTest.class,
4     // ...
5 })
```

Listing 44: Adding the test class to the test suite

Step 3: Add a new test rts file In the tests package, there is a folder named resources. In the sub-folder checker, add a file with the same name as your new test class. But instead of `.java` use the ending `.rts`. In this rts files, the actual tests are located. They have the following structure. To begin a new test, add a line with two slashes and an exclamation mark. Text on the same line is the title of the test. Below this you can add different files. It is advised to add a `.config` file where you can describe your settings. After that you can add a `main.cpp` where you can put your C++ code. In the `.config` section, you can use the `markerLines` attribute to say where you expect your marker to report a problem. If the reported lines and the expected lines are the same, the test passes. You can omit the `markerLines` attribute, if you want to check a case where nothing is reported. The new test file should look similar to the example in Listing A.1.2.

```
1 //! Example test
2 //@.config
3 setPreferencesEval=(GUIDELINE_SETTING_ID|AUTOSAR_GUIDELINE_ID)
4 markerLines=2
5 //@main.cpp
6 /*
7 Your C++ code here
8 */
```

Listing 45: An example rts file

A.2. Quick Fix

This section explains how to add a quick fix to the plug-in. A quick fix is associated to a visitor, but there can be multiple quick fixes per visitor. In this section it is assumed that a corresponding checker already exists.

A.2.1. Quick Fix

This section covers the creation of a quick fix.

Step 1: Add the new quick fix Add a new quick fix class that extends `BaseQuickFix` to the package `com.cevelop.CodeAnalysator.core.quickfixes.shared` if it is a shared quick fix or to the corresponding quick fix folder in the according guideline. This new quick fix needs a constructor that takes a `String` as its label. The method `handleMarkedNode` is used to handle the node reported by the checker. The method `isApplicable` is not mandatory but it is advised to use it. It is used to test whether the problem can be solved by the quick fix. Your new quick fix could look similar to the one in Listing A.2.1.

```
1 public class ExampleQuickFix extends BaseQuickFix {
2     public ExampleQuickFix(String label) { this.label = label; }
3
4     @Override
5     public String getLabel() {
6         return label;
7     }
8
9     @Override
10    public boolean isApplicable(IMarker marker) {
11        return super.isApplicable(marker);
12    }
13
14    @Override
15    protected void handleMarkedNode(IASTNode markedNode,
16        ASTRewrite hRewrite) {
17    }
18 }
```

Listing 46: An example quick fix

Step 2: Register the quick fix in the AutosarGuideline In the same class where you already mapped the visitor, `AutosarGuideline.java`, register your new quick fix. If you are not using AUTOSAR, register it in the according guideline mapper of the corresponding guideline. To do this, use the rule variable that you create before and call its method `addQuickFixes` as shown in Listing A.2.1. This method takes any number of quick fixes.

```
1 public AutosarGuideline() {  
2     // ...  
3     AXX_XX_XX_SomeRule.addQuickFixes(new SomeQuickFix("AXX-XX-XX:"  
4         "Fix something"));  
5     // ...  
6 }
```

Listing 47: Registering your quick fix

A.2.2. Testing

This section describes how to test quick fixes.

Step 1: Add a new test class Add a class to the quick fix folder of the desired guideline test project that extends the quick fix base class of the corresponding guideline. For AUTOSAR this would be `AutosarQuickFixBaseTest` for example. To make it work, two methods need to be overridden. `GetRuleToQuickFix()` returns the rule from the guideline and `getQuickFixToTest` and returns a newly created quick fix object. The new test class should look similar to the example in listing A.2.2.

```

1 public class SomeFixTest extends AutosarQuickFixTestBase{
2
3     @Override
4     protected Rule getRuleToQuickFix() {
5         return AutosarGuideline.XX_XX_XX_SomeRule;
6     }
7
8     @Override
9     protected IMarkerResolution getQuickFixToTest() {
10         return new SomeQuickFix("");
11     }
12 }

```

Listing 48: A new quick fix test class

Step 2: Register the test class in the test suite The quick fix test class needs to be registered in the test suite in the same way as the visitor test before. See Listing A.2.2 for an example.

```

1 @SuiteClasses({
2     // ...
3     ExampleQuickFixTest.class,
4     // ...
5 })

```

Listing 49: Adding the test class to the test suite

Step 3: Add a new rts file Quick fixes use .rts files to test as well. This time the markerLines attribute can be omitted. Instead you can add a new section with the header =. Here you can write what the code should look like after the quick fix has been applied. The test passes, if the quick fix applied to the code in main.h results in the code in the =-section. See the example in Listing A.2.2.

```
1 //! Example quick fix test
2 //@.config
3 setPreferencesEval=(GUIDELINE_SETTING_ID|AUTOSAR_GUIDELINE_ID)
4 //@main.h
5 /*
6 * C++ code before applying the quick fix
7 */
8 //=
9 /*
10 * C++ code after applying the quick fix
11 */
```

Listing 50: An example quick fix rts file