

SA - HS2018 - Documentation

Safe C++ Guidelines

University of Applied Sciences Rapperswil

September - December 2018

Author: Viktor Puselja, Gabriel Vlasek
Supervisor: Peter Sommerlad
Technical Adviser: Hansruedi Patzen

1. Abstract

The **AUTOSAR** guidelines are a set of rules to help developers in the automobile industry write safe C++ code. In a previous project, the Eclipse plug-in **Code-Analysator** was created. It offers an infrastructure to implement different guidelines by different corporations and organizations like AUTOSAR. Before we started the project, only a few rules were implemented as a proof of concept. Using this foundation, we created **checkers** for multiple rules from the AUTOSAR guidelines. These checkers perform static code analysis to mark C++ code that violates one of those rules. By analyzing C++ code, studying examples and standards we evaluated all possible cases that are relevant to the respective rule. We also examined the Abstract Syntax Tree to see how such problems can be identified. Some of the problems can be solved automatically through refactorings. For those problems we created quick fixes. These quick fixes offer the user to automatically manipulate the AST in order to turn a violating piece of code into a compliant one. Over the course of this project we created checkers for 25 rules, 12 of which we offer at least one **quick fix** for.

2. Management Summary

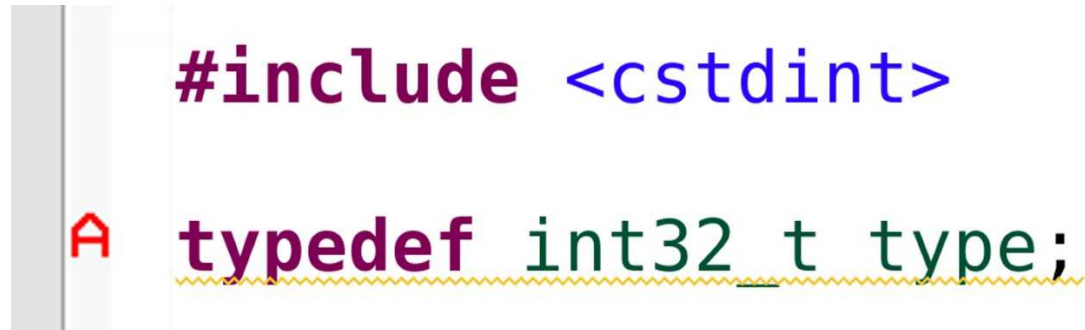
Introduction The `CodeAnalysator` plug-in supports C++ developers that work with Cevelop. It was created in another thesis at the University of Applied Sciences Rapperswil. Many different corporations and organizations define their own rules on how to write safe and robust C++ code. The `CodeAnalysator` plug-in provides an infrastructure, where many different guidelines can be implemented. Implementing in this case means the following: If a developer writes code that violates a rule in a guideline, the corresponding parts of the code are highlighted. The developer is then informed that they violated a rule. Sometimes such violations can be resolved directly by using a `quick fix`. A quick fix allows the developer to automatically transform the problematic code into compliant code. In the above mentioned thesis, the predecessor group created the `CodeAnalysator` plug-in and infrastructure but with only a few rule implementations. The topic of this thesis is to implement many additional rules in order to support C++ programmers.

Approach The main work consisted of implementing checkers and quick fixes for rules. In this project we focused on `AUTOSAR` rules. The AUTOSAR guidelines are a set of rules used to write safe C++ code in critical and safety-related projects. It is used by the automobile industry. The AUTOSAR guidelines `AUTOSAR 2018` is not a public document. As for implementing the rules, the approach was the same for each one. First, we analyzed the rule by looking at the description and examples and studying the language specification. We then wrote tests to find out all the different cases in which they apply. After we knew what cases there were, we started to analyze the C++ code and wrote the corresponding `checker`. If there was a way to solve the problem without changing the meaning of the code, we wrote a `quick fix` for the corresponding rule. To find out how such a quick fix should work, we first defined how the code should look after running our quick fix. Based on our findings from the earlier analysis we wrote our quick fix in a way that it would transform all of the bad examples into conforming code.

Results At the end of this project we had implemented twenty-five rules including corresponding tests, with the rules ranging from trivial to complex. For twelve of these rules we additionally implemented `quick fixes` to resolve the problems automatically. With these checkers and quick fixes we will hopefully help developers

write safer and more robust C++ code in the future.

To give an idea of how this works, here is an example of how these checkers and quick fixes operate.



```
#include <stdint>

typedef int32_t type;
```

Figure 2.1.: C++ code that is correct but violates a rule. The underline shows that something is amiss.

The code above is correct and compiles. It will not give an error. But our checker recognized that the usage of typedef is not best practice. It notifies the user about this by underlining the offending code.

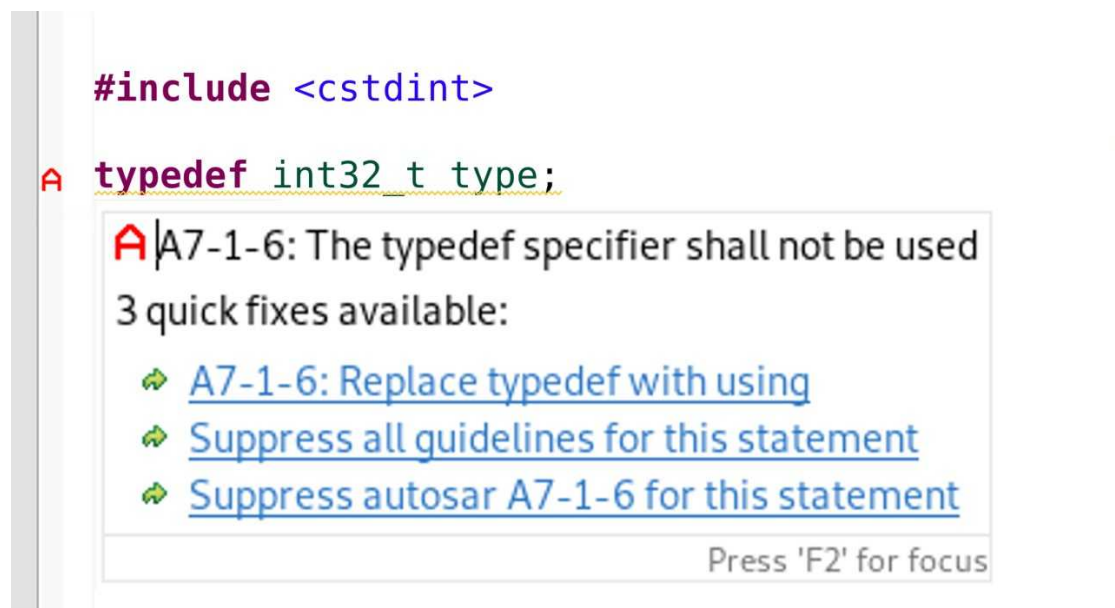


Figure 2.2.: When hovering over the bad code, the violated rule is displayed. Beneath, some solutions are proposed.

The user hovers over the notification and sees which rule he has violated. He is also shown a few possibilities on how to solve this problem. He can now click on the first one to apply the quick fix.

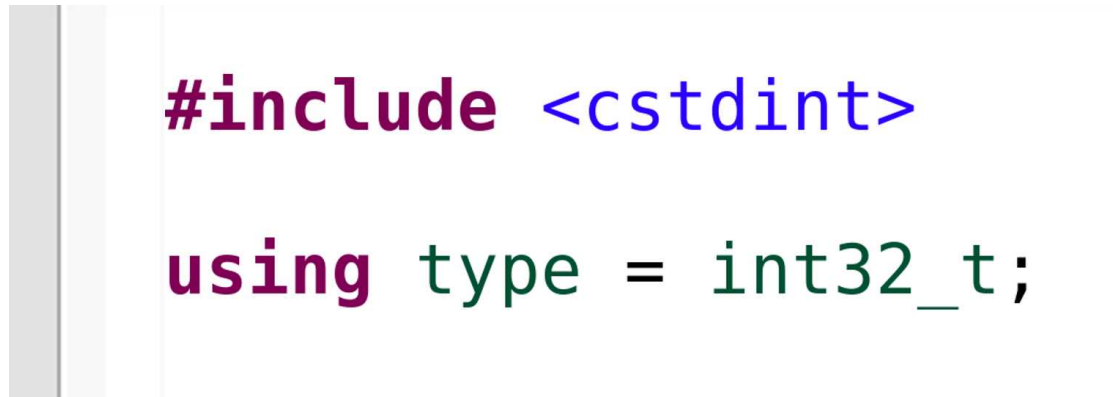
A screenshot of a code editor showing two lines of C++ code. The first line is `#include <stdint>` with `#include` in purple, `<` in blue, `stdint` in blue, and `>` in blue. The second line is `using type = int32_t;` with `using` in purple, `type` in green, `=` in green, `int32_t` in green, and `;` in green. The code is displayed on a light gray background with a vertical gray bar on the left.

Figure 2.3.: The code after the quick fix has been applied.

This is what the code looks like after the quick fix has been applied. Now no more notifications are shown and the code has become more safe and robust.

Outlook As of now there are about 30 implemented rules in the `CodeAnalysator` plug-in, including the ones that were already implemented before we started our project. `AUTOSAR 2018`. But AUTOSAR still has many more rules to implement. And there are other guidelines that can be implemented. There are definitely a lot of rules left that will provide work in the future. Also, the architecture can be refactored and maybe even improved.

Contents

1. Abstract	i
2. Management Summary	ii
3. Introduction	1
3.1. Initial Situation	1
3.2. Problem Description	1
3.3. On checkers, visitors and quick fixes	2
4. Planning	3
4.1. Elaboration	3
4.2. Documentation	3
4.3. Project Management	4
4.4. Construction	4
4.4.1. Definition of done for checker and quick fixes	4
4.5. Milestones	5
4.5.1. End of Elaboration	5
4.5.2. Mid Construction	5
4.5.3. End of Construction	6
4.5.4. End of Project	6
5. Requirements	7
5.1. Scope	7
5.2. Minimal scope	8
5.3. Desired scope	10
5.4. Optional scope	10
6. Architecture	11
6.1. Eclipse and OSGi	11
6.2. Packages	12
6.3. Relevant classes and files	13
6.3.1. fragment.xml	13
6.3.2. AutosarIdHelper	13
6.3.3. AutosarChecker	14

6.3.4. CoreIdHelper	14
6.3.5. AutosarGuidelineMapper	14
6.3.6. CheckerTest	14
6.3.7. QuickFixTest	14
6.3.8. PluginUITestSuiteAll	14
7. Implementation	15
7.1. Implementation of the checkers and quick fixes	15
7.1.1. A5-0-2: Condition of if/while/for shall be bool	15
7.1.2. A5-1-1: No magic numbers	17
7.1.3. A5-1-2: Do not implicitly capture variables in lambda ex- pressions	18
7.1.4. A5-1-3: Parameter list shall be included in lambda	18
7.1.5. A5-1-4: Lambda expressions shall not outlive reference cap- tured objects	18
7.1.6. A5-1-6: Explicit lambda return type	19
7.1.7. A5-1-8: Do not nest lambda expressions	19
7.1.8. A5-16-1: Do not use the ternary conditional operator as sub-expression	19
7.1.9. A6-4-1: Switch shall have at least two cases	20
7.1.10. A7-1-4: The register keyword shall not be used	21
7.1.11. A7-1-5: The auto specifier shall not be used	21
7.1.12. A7-1-6: The typedef specifier shall not be used	22
7.1.13. A7-2-2: Enumeration underlying base type shall be explicitly defined	22
7.1.14. A7-2-3: Declare enumerations as scoped enum classes	22
7.1.15. A7-2-4: Initialize none, the first or all enumerators	23
7.1.16. A7-4-1: The asm declaration shall not be used	23
7.1.17. A8-5-2: Braced-initialization {}, without equals sign, shall be used for variable initialization	23
7.1.18. A8-5-3: A variable of type auto shall not be initialized using { } or = { } braced-initialization	24
7.1.19. A9-5-1: Unions shall not be used	25
7.1.20. A10-1-1: Class shall not be derived from more than one base class which is not an interface class.	25
7.1.21. A10-2-1: Do not redefine non-virtual member functions	25
7.1.22. A10-3-1: Virtual function declarations shall have exactly one of the specifiers virtual, override or final	26
7.1.23. A10-3-2: Each overriding virtual function shall be declared with the override or final specifier.	27

7.1.24. A10-3-3: Do not introduce virtual member functions in a final class	27
7.1.25. A10-3-5: User defined assignment operators shall not be virtual	28
7.2. Example using A5-1-6	29
7.2.1. Checker	29
7.2.2. Quick fix	30
8. Quality measures	34
9. Results	35
9.1. Work planned / completed comparison	35
9.2. Time Management	36
9.2.1. Estimation and time spent of major areas	36
9.2.2. Estimation and time spent of checkers and quick fixes	36
10. Conclusion	41
10.1. Retrospective	41
10.2. Considerations for future projects	42
Glossary	46
A. Developer Guide	49
A.1. Overview	49
A.2. Checker	50
A.2.1. Visitor	50
A.2.2. Testing	55
A.3. Quick fix	57
A.3.1. Quick fix	57
A.3.2. Testing	60
B. Acknowledgement	62
C. Eigenständigkeitserklärung	

3. Introduction

In this chapter we will give a short introduction to our project. We will describe the situation at the beginning of our project, followed by the description of the problem to solve in this project, with an explanation of the core concepts at the end.

3.1. Initial Situation

This project has a predecessor project. Before it, another group created the `Eclipse` plug-in `CodeAnalysator`. The goal of this plug-in is to provide an infrastructure to implement `checkers` and `visitors` for different guidelines. Refer to chapter 6 for further detail on how the `CodeAnalysator` plug-in is structured. They already implemented a small number of checkers and `quick fixes` in a proof of concept way. The plug-in uses the `CDT`, the C/C++ Development Tools. The CDT is a development environment to create plug-ins for C and C++ programming in Eclipse. It offers functionality to analyze C++ code such as examining and manipulating the abstract syntax tree.

3.2. Problem Description

As stated before, the plug-in as of the beginning of the project offers the infrastructure to implement `checkers` and `quick fixes` but does not yet implement many such elements. This means that there is not a lot of architecture work to do. Instead, we need to analyze many different rules and decide how the code needs to be checked to find these problems in order to write checkers. Another part is to evaluate whether a feasible quick fix for the given problem exists. We are implementing rules for the `AUTOSAR` guidelines. The AUTOSAR guidelines are a set of rules that are used in the automobile industry. The guideline is not public. This means that some specific sections in this document will be removed for the public release.

3.3. On checkers, visitors and quick fixes

Any reader of this document will come across the words `checker`, `visitor` and `quick fix` many times. To minimize confusion, here an explanation what these terms mean in the context of this project. `Checkers` are a part of `Codan`, a static analysis framework in `CDT`. The purpose of a checker is to perform code analysis and report found problems and defects. The `CodeAnalysator` plug-in implements a single checker per guideline which runs multiple visitors, one for each rule. Visitors implement the visitor pattern for traversing the AST in `CDT`. The rule visitors in the `CodeAnalysator` check the `AST` for violations of their respective rule and report the offending node. When a node is reported, a problem marker highlighting the node is created to signal the problem to the user. Then, if available, marker resolutions are presented to the user to resolve the problem by automatically manipulating the AST. The marker resolutions for rule violations are called quick fixes in the `CodeAnalysator` plug-in. They are provided in case there is a sensible, idiomatic and automatable solution or fix for a rule violation. In short, checkers run rule visitors for a guideline, which check the AST for violations of their respective rules and report them. Quick fixes resolve rule violations, by manipulating the AST.

4. Planning

The target work amount per person for the study project is 240 hours over a span of 14 weeks. For our two men project this equals 480 hours total and 17 hours per week per person. The main working hours are Monday and Friday from approximately 8AM to 5PM.

The project itself can be divided into the following major working areas: Project Management, Elaboration, Construction and Documentation. The project time will first be allocated to the major work areas. We plan on using the following share of time on each area:

- 8%: Elaboration
- 16%: Documentation
- 10%: Project management
- 66%: Construction

4.1. Elaboration

Eight percent of the time will be spent on analysis and elaboration. In the earlier phases of the project the amount will be higher than later on. Analysis consists of the following tasks:

- Install development tools
- Clone the project and make it run locally
- Introduction to technologies like eclipse plug-in development
- Studying the AUTOSAR guideline

4.2. Documentation

This area consists simply of writing this document. Writing on anything besides the implementation section is counted towards this working area. Writing on implementation is counted towards construction. See section 4.4.1.

4.3. Project Management

Ten percent of the total time will be spent on Project Management. The following activities are part of project management:

- Weekly meetings with supervisor
- Preparation and recap of meetings
- Wiki management
- Time management

4.4. Construction

Construction takes up approximately sixty-six percent of the total time which accumulates to a total of 320 hours. We use time slots to talk about estimates. One time slot equals 4 hours of work. This means there is a total of approximately 80 slots reserved for construction. At the end of elaboration, which is planned for October 19th, we will have spent 20 slots. For the rest of the project we plan on spending another 60.

The following table shows our estimation on how many time slots we need for a given problem. The estimates are based on our experience in the elaboration phase.

	Trivial	Medium	Complex
Checker	1	3	6
Quick Fix	1	2	4

Table 4.1.: Estimated time slots per task

4.4.1. Definition of done for checker and quick fixes

A **checker** or a **quick fix** is considered complete when the following checklist is fulfilled:

- The **checker** or quick fix must be implemented and working.
- Tests must be implemented and passing.
- The changes must be committed and pushed to the **CI** server.
- The build and tests of the commit must pass.
- The implementation must be documented in the section implementation of this document.
- Javadoc documentation for helper classes must be written.

4.5. Milestones

We defined the following milestones for our project.

Project Start	17. September 2018
End of Elaboration	19. October 2018
Mid Construction	16. November 2018
End of Construction	10. December 2018
End of Project	21. December 2018

Table 4.2.: Milestones timetable

4.5.1. End of Elaboration

End of Elaboration is our first milestone. By this point we have:

- Finished project planning
- Set up our development machines and processes (CI)
- Defined our scopes (minimal, desired, optional)
- Implemented the first checkers and quick fixes
- Chosen AUTOSAR rules for at least until mid construction
- Defined the quality measures

Output:

- Project plan
- Scopes
- First checkers and quick fixes

4.5.2. Mid Construction

For Mid Construction the following has to be accomplished:

- Requirements specification completed
- All AUTOSAR rules for this project chosen
- At least 2/3 of checkers and quick fixes implemented
 - Must 2/3 of minimal scope
 - Should 2/3 of desired scope

Output:

- Requirements specification
- Most **checkers** and **quick fixes**

4.5.3. End of Construction

The following need to be completed by the end of construction:

- Architecture and implementation documentation completed
- Developer guide completed
- All **checkers** and quick fixes implemented
 - All **checkers** and **quick fixes** of the minimal scope must be completed.
 - All **checkers** and quick fixes of the desired scope should be completed but are not mandatory.

Output:

- Architecture documentation
- Implementation documentation
- Developer guide
- All **checkers** and **quick fixes**

4.5.4. End of Project

Our last milestone is the end of the project. Here we need to have completed the following tasks:

- Finish documentation including the following chapters
 - Abstract
 - Management summary
 - Introduction
 - Results and Conclusion
 - Glossary
- Create poster
- Delivery and upload of project

5. Requirements

In this chapter we will describe the requirements for the project. The requirements are defined in term of scopes. Each larger scope contains the previous scope. Each scope defines which rules to implement and whether to implement quick fixes for those rules as well, including estimations of complexity and time effort required.

5.1. Scope

We define three scopes for this project. The minimal scope defines the minimum to consider the project successful. The desired scope defines what would be expected to be achieved. The optional scope defines what would go beyond expectations. This section will focus on the construction part of the project. As introduced in chapter 4.3 Construction, we will use time slots of four hours to talk about estimates. The following table shows how many time slots we plan for each of the scopes.

	Elaboration Phase	Construction Phase	Total
Minimal Scope	20 time slots	51 time slots	71 time slots
Desired Scope	20 time slots	61 time slots	81 time slots
Optional Scope	20 time slots	68 time slots	88 time slots

Table 5.1.: Planned time slots for the different scopes

5.2. Minimal scope

Name	Checker complx.	Quick fix complx.	Time slots
A5-1-1: Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead. [AUTOSAR 2018, p. 86]	Medium	-	3
A5-1-2: Variables shall not be implicitly captured in a lambda expression. [AUTOSAR 2018, p. 88]	Medium	Simple	4
A5-1-3: Parameter list (possibly empty) shall be included in every lambda expression. [AUTOSAR 2018, p. 89]	Simple	Simple	2
A5-1-4: A lambda expression object shall not outlive any of its reference-captured objects. [AUTOSAR 2018, p. 90]	Complex	-	6
A5-1-6: Return type of a non-void return type lambda expression should be explicitly specified. [AUTOSAR 2018, p. 91]	Medium	Complex	7
A5-1-8: Lambda expressions should not be defined inside another lambda expression. [AUTOSAR 2018, p. 93]	Simple	-	1
A5-16-1: The ternary conditional operator shall not be used as a sub-expression. [AUTOSAR 2018, p. 111]	Simple	-	1
A6-4-1: A switch statement shall have at least two case-clauses, distinct from the default label. [AUTOSAR 2018, p. 117]	Medium	Complex	7
A7-1-4: The register keyword shall not be used. [AUTOSAR 2018, p. 129]	Simple	Simple	2
A7-1-5: 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. [AUTOSAR 2018, p. 130]	Medium	Simple	4
A7-1-6: The typedef specifier shall not be used. [AUTOSAR 2018, p. 131]	Simple	Medium	3
A7-2-2: Enumeration underlying base type shall be explicitly defined. [AUTOSAR 2018, p. 137]	Medium	Simple	4

A7-2-3: Enumerations shall be declared as scoped enum classes. [AUTOSAR 2018, p. 138]	Simple	-	1
A7-4-1: The asm declaration shall not be used. [AUTOSAR 2018, p. 142]	Simple	-	1
A8-5-3: A variable of type auto shall not be initialized using {} or = {} braced-initialization. [AUTOSAR 2018, p. 173]	Medium	Medium	5
A9-5-1: Unions shall not be used. [AUTOSAR 2018, p. 179]	Simple	-	1
A10-1-1: Class shall not be derived from more than one base class which is not an interface class. [AUTOSAR 2018, p. 183]	Medium	-	3
A10-2-1: Non-virtual member functions shall not be redefined in derived classes. [AUTOSAR 2018, p. 185]	Complex	-	6
A10-3-1: Virtual function declaration shall contain exactly one of the three specifiers: (1) virtual, (2) override, (3) final. [AUTOSAR 2018, p. 186]	Simple	Medium	3
A10-3-2: Each overriding virtual function shall be declared with the override or final specifier. [AUTOSAR 2018, p. 188]	Medium	Simple	4
A10-3-3: Virtual functions shall not be introduced in a final class. [AUTOSAR 2018, p. 189]	Simple	Simple	2
A10-3-5: A user-defined assignment operator shall not be virtual. [AUTOSAR 2018, p. 190]	Simple	-	1
		Total	71

Table 5.2.: Minimal Scope

5.3. Desired scope

The desired scope includes the minimal scope and additionally the following rules:

Name	Checker complexity	Quick fix complexity	Time slots
Minimal scope	-	-	71
A5-0-2: The condition of an if-statement and the condition of an iteration statement shall have type bool. [AUTOSAR 2018, p. 80]	Medium	Medium	5
A7-2-4: In an enumeration, either (1) none, (2) the first or (3) all enumerators shall be initialized. [AUTOSAR 2018, p. 139]	Medium	Medium	5
		Total	81

Table 5.3.: Desired scope

5.4. Optional scope

The optional scope includes the desired scope and additionally the following rules:

Name	Checker complexity	Quick fix complexity	Time slots
Desired scope	-	-	81
A8-5-2: Braced-initialization {}, without equals sign, shall be used for variable initialization. [AUTOSAR 2018, p. 170]	Complex	Trivial	7
		Total	88

Table 5.4.: Optional scope

6. Architecture

In this chapter we will give an overview over the architecture of the `CodeAnalysator` plug-in where relevant to our project. We neither implemented nor designed the architecture of the plug-in, as the focus of our student project was on the implementation of `checkers` and `quick fixes` for AUTOSAR rules. [\[AUTOSAR 2018\]](#) [\[Bertschi, Pascal 2018\]](#) For further information please refer to the technical report of the predecessor project.

NOTE: When mentioned or talked about, `checker` for a rule (in the context of the `CodeAnalysator` plug-in) refers to a `visitor` for that rule, instead of a `Codan` checker. Checker, in that case, is meant in the sense of performing the action of checking, as done by the rule visitors, as opposed to guideline checkers, which do not perform any checking by themselves in the `CodeAnalysator`.

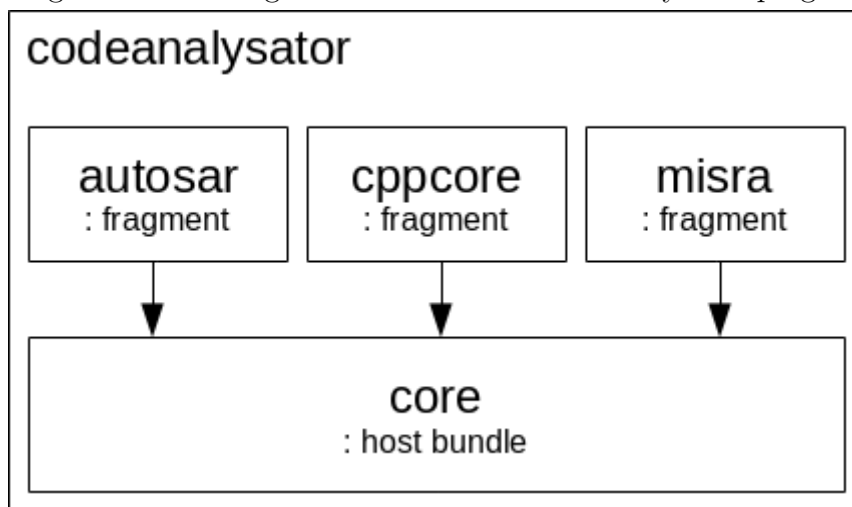
6.1. Eclipse and OSGi

Following is a short overview over OSGi as it is relevant for fully understanding the structure of the `CodeAnalysator` plug-in. `OSGi` is a module system used by `Eclipse` for plug-in management. An OSGi `bundle`, or as commonly referred to, a plug-in, is a Java archive containing a manifest explicitly describing the packages it requires (dependencies) as well as the packages it makes available to others. A bundle can be extended by `fragments`. An OSGi `fragment` is similar to a bundle, but specifies a fragment-host or host bundle to attach to in the manifest and cannot be loaded on its own. Instead, when a bundle is loaded, all fragments attached to it are loaded as well. Fragments provide a way to customize and extend bundles. [\[OSGI overview 2018\]](#)

6.2. Packages

The `CodeAnalysator` plug-in is divided into a core package and guideline packages. The core package contains the infrastructure code of the plug-in, including plug-in ceremony, base classes and utility classes, as well as rule `visitors` and `quick fixes` shared between multiple guidelines. The guideline packages contain guideline specific configuration, as well as visitors and quick fixes for rules which are specific to that guideline. The core package forms an `OSGi bundle` and the guideline packages form `OSGi fragments`, with the core package acting as the host bundle. This allows support for guidelines to be individually distributed.

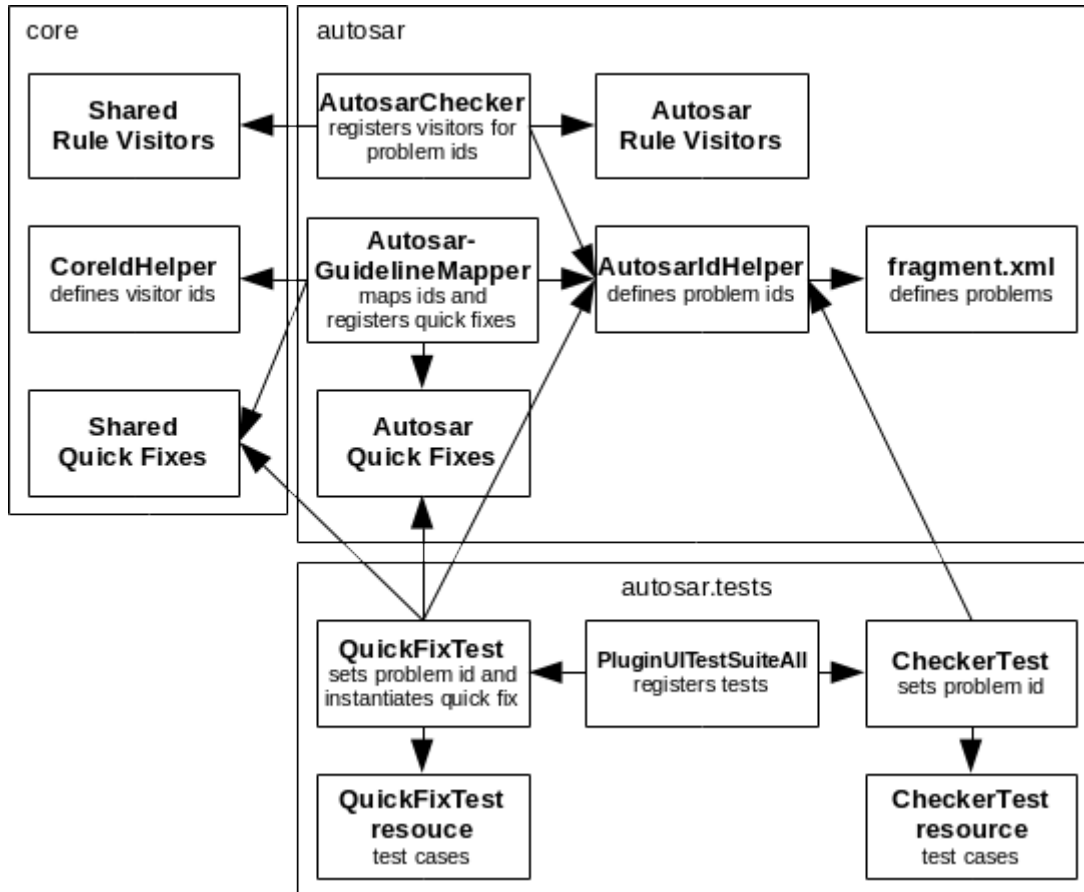
Figure 6.1.: Package structure of the CodeAnalysator plug-in



6.3. Relevant classes and files

This section shows the classes and files used to implement a single rule based on the **AUTOSAR** guidelines.

Figure 6.2.: Classes and files that implement a single rule



6.3.1. fragment.xml

The **fragment.xml** file defines all problems (rule violations) for a specific guideline. A problem definition includes name, description and a unique id.

6.3.2. AutosarIdHelper

This helper class defines constants for all ids used in a guideline, for use in Java. This includes problem ids and **visitor** ids for guideline specific rule visitors. The problem ids must match those from the **fragment.xml** file.

6.3.3. AutosarChecker

The guideline `checker` registers rule `visitors` for the problem ids. Visitors can be guideline specific or shared, but problem ids are always specific to the guideline.

6.3.4. CoreIdHelper

The CoreIdHelper defines constants for guideline independent ids. The rule `visitor` ids for shared visitors are defined here.

6.3.5. AutosarGuidelineMapper

The guideline maps specific and shared `visitor` ids to problem ids, adds suppression attributes for visitor ids, registers `quick fixes` and defines the name and id for the guideline.

6.3.6. CheckerTest

CheckerTest classes set the problem id to use for the test. The test cases themselves are defined in the corresponding CheckerTest resource file. Usually there is one CheckerTest per rule visitor.

6.3.7. QuickFixTest

QuickFixTest classes are analogues to the CheckerTest classes. The QuickFixTests set the problem id to use and instantiate the `quick fix` to test. The test cases themselves are again defined in the corresponding QuickFixTest resource file.

6.3.8. PluginUITestSuiteAll

This class is run by the build script and CI Server for testing. All test classes need to be registered here.

7. Implementation

In this chapter we describe the implementation of the different rules. We will also provide an example of how we worked when implementing a checker and a quick fix for a rule.

7.1. Implementation of the checkers and quick fixes

In this section we will talk about the implementation of the `AUTOSAR` rules. We decided not to include a separate chapter about analysis because it is tightly coupled to the implementation of the checkers quick fixes. For non-trivial rules we describe the analysis that went ahead of the implementation in the checker and quick fix descriptions. Some rules offer work for further projects. Such possibilities are also listed.

7.1.1. 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 7.1.: 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 7.2.: 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, p. 80]

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.

7.1.2. 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("Logic Error");
```

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.

7.1.3. 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]

7.1.4. 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]

7.1.5. 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.

7.1.6. 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.

7.1.7. 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.

7.1.8. 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.

7.1.9. 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. In all cases the construct is contained in a compound statement in order to preserve the additional scope of the switch statement. 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 7.1.: Assignment of clauses to if-, else- and following-parts

Future possibilities: In the future, in case the parent statement is a compound statement, the init- and following-parts could be checked for possible name-collisions with the parent scope and if there are none, the additional compound statement might be omitted, and the parts directly inserted into the parent compound statement.

7.1.10. 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.

7.1.11. A7-1-5: 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.

7.1.12. 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.

7.1.13. 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.

7.1.14. 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]

7.1.15. 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.

7.1.16. 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.

7.1.17. 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]

7.1.18. 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 7.3.: Different ways of initializing an auto variable [Initialization - cppreference.com 2018]

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

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 7.1.11 for more information.

7.1.19. 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.

7.1.20. 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.

7.1.21. 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 7.4.: 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.

7.1.22. 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.

7.1.23. 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.

7.1.24. 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.

7.1.25. 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.

7.2. Example using A5-1-6

In this section we describe the implementation of the rule A5-1-6 with a checker and a quick fix.

7.2.1. Checker

The first thing we did was look at the AUTOSAR description and the provided examples. The description says that omitting the return type of a lambda can lead to confusion. It also warns, that there might be implicit conversion from the returned element in the lambda and the specified return type. The code examples show two cases. The first one is a compliant lambda expression with explicitly defined return type. The other one is a non-compliant example without a return type. These are our first two test cases. Of course there are other cases that we need to find. One test case is to test the suppression by adding a suppression attribute to an non-compliant example. We also need to check what happens, when the return type is void i.e. there is no return statement.

The next step is to look at the **AST** of the two examples.

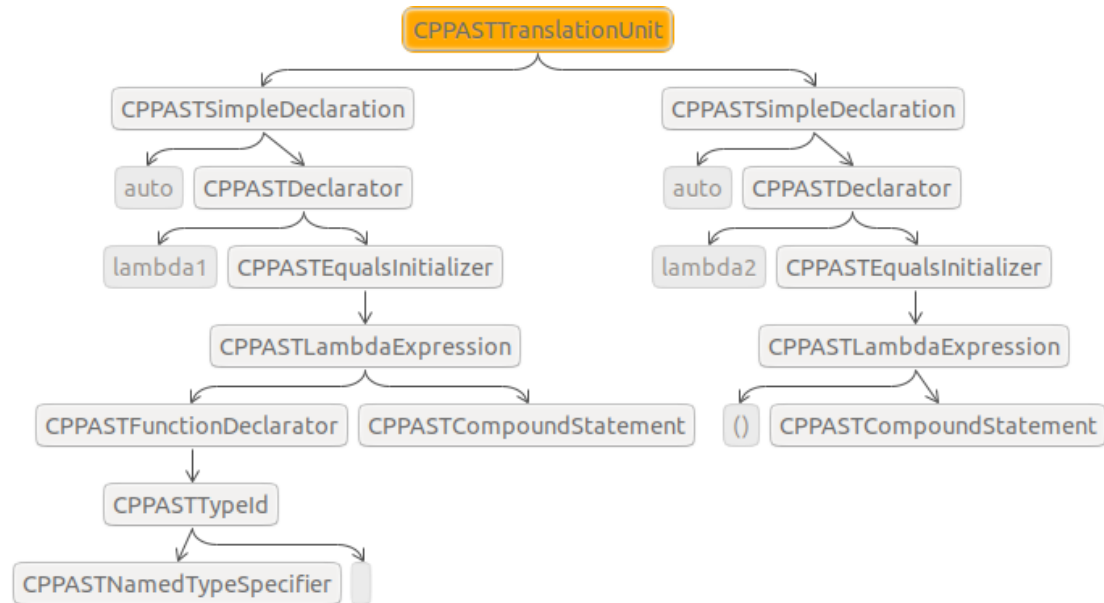


Figure 7.5.: AST that shows a lambda with (left) and without (right) explicit return type.

This is how we found out that the lambda needs a parameter list to have a return type. So another test is, what happens, when there is no parameter list. We

test this with implicit and void return type. There are a few special cases. For example what happens when the return type is `decltype(auto)`. We looked at the [cppreference page for lambda expressions \[Lambda expressions \(since C++11\) - cppreference.com \[2018\]\]](#). Here we found out, that `auto` and `decltype(auto)` also result in an implicit return type. This gives us another test case. We added test cases again for void and implicit return type because if it is void, it is compliant. [\[AUTOSAR 2018\]](#) Since we had `decltype(auto)` we also needed to test `decltype` of some type, so we added a test for `decltype(std::uint8_t)` and `decltype(void)`. At this point we were quite sure, that we had covered all the cases and started programming the [visitor](#). To cover all the tests, we visit all the expressions and then check whether they are lambda expressions. If so, it is checked whether they have an explicit return type and are not of type void. From the AST we saw, that it cannot have an explicit return type without a parameter list so we check this first. After this, we make sure that the lambda expression has a trailing return type. If so, we ensure that it is not `decltype(auto)`. If the expression fails this test and also has not a void return type, we mark it. This implementation covers all the found test cases.

7.2.2. Quick fix

We start developing the quick fix by studying the relevant [cppreference pages \[Lambda expressions \(since C++11\) - cppreference.com \[2018\]\]](#) again, in order to get a good grasp of how the return type is deduced and how it is specified. Then we need to define test cases. Generally, we start with the tests of the corresponding checker. Many of the test cases containing marker lines can be converted into suitable quick fix tests by adding a second snippet with the fixed, confirming code and removing the marker lines declaration, as shown in figures [7.6](#) and [7.7](#). Tests without marker lines can sometimes be used as fixed code snippets for above tests, otherwise they are ignored. Depending on the quick fix we may need further tests, based on our study of the relevant language definitions.

```

//! Implicit lambda return type
//@.config
setPreferencesEval=(GUIDELINE_SETTING_ID|AUTOSAR_GUIDELINE_ID)
markerLines=4
//@main.h
#include <cstdint>
void Fn() noexcept
{
    auto lambda = []() {
        std::uint8_t ret = 0U;
        return ret;
    };
}

```

Figure 7.6.: Checker test with marker lines

```


//! Implicit lambda return type
//@.config
setPreferencesEval=(GUIDELINE_SETTING_ID|AUTOSAR_GUIDELINE_ID)
//@main.h
#include <cstdint>
void Fn() noexcept
{
    auto lambda = []() {
        std::uint8_t ret = 0U;
        return ret;
    };
}
//=
#include <cstdint>
void Fn() noexcept
{
    auto lambda = []() -> uint8_t {
        std::uint8_t ret = 0U;
        return ret;
    };
}


```

Figure 7.7.: Quick fix test based on the checker test

Now, having the quick fix tests with the offending and corresponding fixed code snippets, we can inspect and compare both offending and fixed code with the ASTView and NodeView views of the PASTA plug-in for Eclipse. Based on this we can start implementing the quick fix. For the first case we can reuse the comparison between a lambda with and without explicit return type in figure 7.5 above. There we can see, that we need to set a type id on the function declarator, the trailing return type, in order to explicitly specify a return type.

As the return type is semantic information, we take a look at the interfaces implemented by the binding, that the lambda call operator name resolves to. As it turns out, it implements the IFunction interface [\[Interface IFunction 2018\]](#) which has a method getType which returns a IFunctionType interface. [\[IFunctionType 2018\]](#) The IFunctionType interface then, has a method getReturnType which returns our wanted return type. Checking the return value using debugging we can verify that the method indeed returns the deduced return type of the lambda expression.

The next step is constructing the type id corresponding to the type. The helper

class `DeclarationGenerator` already implements methods, `createDeclSpecFromType` and `createDeclaratorFromType`, for generating an abstract declarator from a type, from which we can construct a type id using the node factory. Having the type id constructed, all that is left is setting the type id as the trailing return type of the function declarator. To do this, we first have to copy the function declarator. We need to copy nodes before modifying, as they are frozen after being added to the AST and can only be modified by the creator. After copying we can set the return type on the new copied function declarator. Finally we have to call a method on the `ASTRewriter` to insert the node into the AST, usually either `insertBefore` or `replace`, depending on whether it is a new node or a replacement for an existing one, in this case `replace`, to replace the existing function declarator with the new one. Having done this, we have implemented the first test case.

Now, we need to implement the remaining test cases. A further test case would be a lambda expression with omitted parameter list, as there is no function declarator in this case. We would have to differentiate and, in case of a missing function declarator, create a new function declarator instead of copying and set the type. Then use the `insertBefore` method of the `ASTRewriter` to insert node into the AST. When all test cases have been implemented and we are confident, that the test suite is comprehensible, the quick fix is complete.

8. Quality measures

In this section, the measures taken to assure good quality are listed. There are tests for both `checkers` and quick fixes, that are run locally before each commit. The checker and quick fix tests are integration tests and are executed inside an Eclipse instance. While large numbers of tests do require some time to execute, testing the checkers and quick fixes in isolation would not be very useful and require a lot of effort, as they heavily depend on CDT and Eclipse.

The `checker` tests make sure that the correct code parts are marked at the correct time. There are also some tests that check for false-positives. If there are any exceptions or variations of the rule, they are checked here as well. The test cases themselves are defined in `".rts"`-text files, which specify code snippets to check and on which lines problems should be reported.

Tests for the quick fixes run the quick fix on offending code snippets and compare the result with the expected code snippets. The tests are specified in `".rts"`-files as well, this time without line numbers, but with an additional confirming code snippet per test.

After committing and pushing the code to the repository, the tests are rerun by a `CI-Server` to ensure that everything is correct. The `CI-Server` already existed prior to this project.

9. Results

In this chapter we will present the results of our project, with a comparison of completed and not completed checkers and quick fixes, followed by an evaluation of our planning, estimations and output.

9.1. Work planned / completed comparison

In chapter 5 we initially defined three scopes, minimum, desired and optional. In these scopes we defined which checkers and quick fixes were planned to be implemented. As we were able to fulfil all three scopes, with the exception of a few quick fixes, in the course of the project, we will contrast the completed with the not completed checkers and quick fixes.

	Checker	Quick Fixes
Completed	A5-0-2 A5-1-1 A5-1-2 A5-1-3 A5-1-4 A5-1-6 A5-1-8 A5-16-1 A6-4-1 A7-1-4 A7-1-5 A7-1-6 A7-2-2 A7-2-3 A7-2-4 A7-4-1 A8-5-2 A8-5-3 A9-5-1 A10-1-1 A10-2-1 A10-3-1 A10-3-2 A10-3-3 A10-3-5	A5-1-2 A5-1-3 A5-1-6 A6-4-1 A7-1-4 A7-1-5 A7-1-6 A8-5-2 A8-5-3 A10-3-1 A10-3-2 A10-3-3
Not Completed		A7-2-2 A7-2-4 A5-0-2

Table 9.2.: Comparison of planned and implemented checkers and quick fixes

As can be seen in the table above, we were able to complete all the 25 planned checkers. A few of them have some limitations. Refer to chapter 7 to view the details of the respective rule.

Twelve planned **quick fixes** were completed and three were not implemented. A7-2-2 and A7-2-4 cannot be feasibly implemented without changes to the **CDT**. The problem is, that the ChangeGenerator in the **CDT** does not visit enumerations. With an update to the **CDT** these two can be implemented. The A5-0-2 quick fix was not implemented because it was started at the end of the project and would have needed more time to evaluate. The rule did not specify clearly how a well-formed example should look like. With more time on hand, this rule could be implemented.

9.2. Time Management

In this section we will compare our estimations with the actual time spent. First we will compare the time estimations for the major working areas, followed by comparisons of the estimates for individual checkers and quick fixes.

9.2.1. Estimation and time spent of major areas

In chapter 4 we divided the project into major working areas and allocated the available time. Our estimations for the major working areas turned out to be rather accurate. We were below the allocated time in most areas. Documentation took longer than estimated, while elaboration, project management and implementation (construction) took slightly less than estimated.

Name	Est. [h]	Est. [%]	Spent [h]	Spent [%]
Elaboration	38h 24m	8%	32h 40m	6.7%
Project management	48h	10%	37h 35m	7.7%
Documentation	76h 49	16%	112h 53m	23.2%
Implementation	316h 48m	66%	302h 40m	62.3%
Total	480h	100%	485h 45m	100%

Table 9.3.: Time shares of major areas

9.2.2. Estimation and time spent of checkers and quick fixes

In chapter 5 we introduced the scopes and how many time slots we estimated we would need to complete them. Refer to section 4.4 to see how time slots translate to time. In this chapter the estimated time is compared to the actual spent time.

Comparison chart On the following pages is a list of all the issues with the respective estimate and time. After that follows a chart that shows this difference as well. Due to a lack of space only the issue numbers are listed next to the chart. Only the construction issues are included in the chart. Construction issues are those that are either implementing a **quick fix** or a **checker**. This is because there are a few that we did not set any estimate for, such as the issue "Create Documentation" or "Meetings". In the table below the issues not contained in the chart are listed as well.

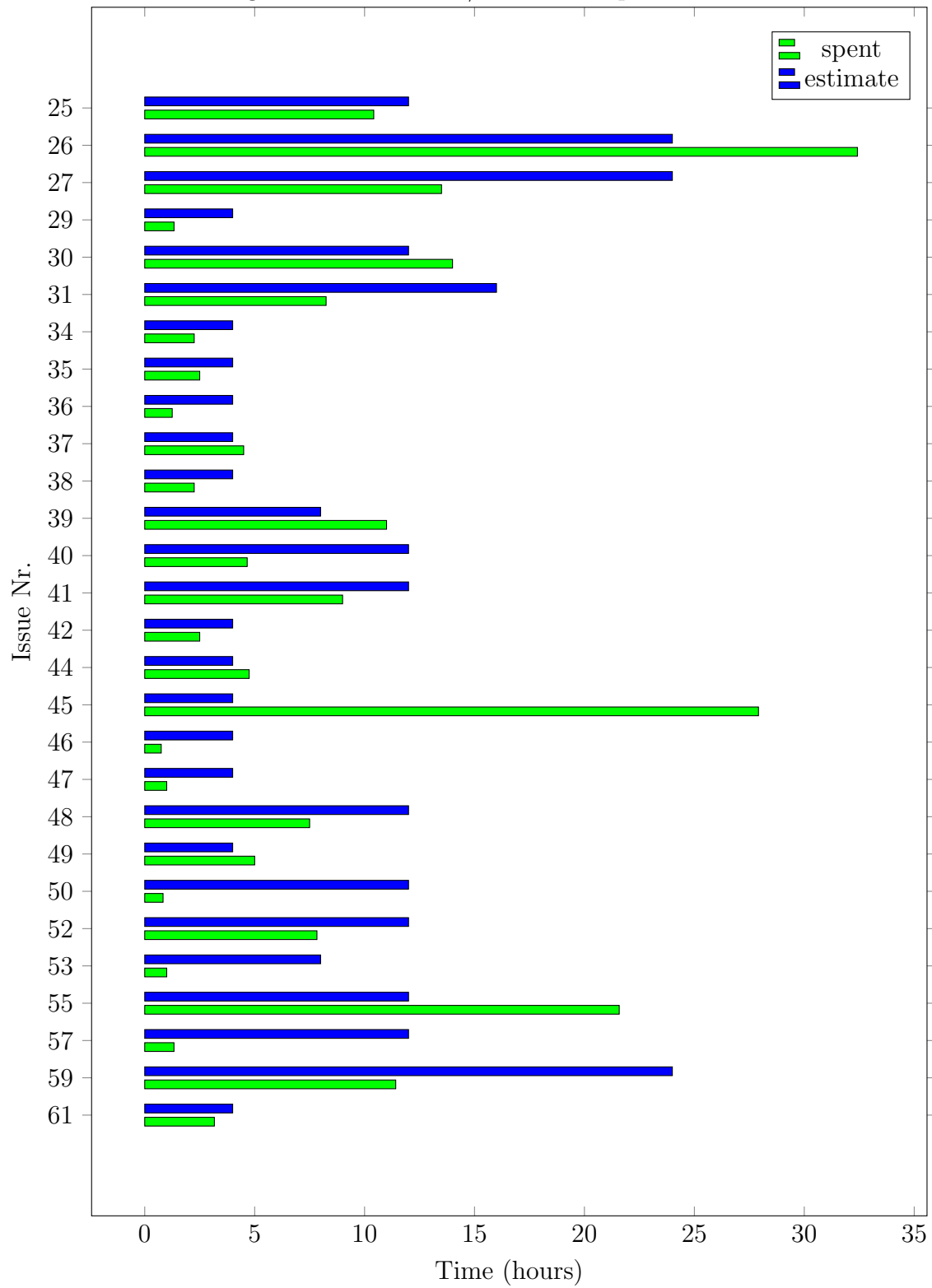
Nr	Description	Est.	Spent	Author
5	Implement checker for rule A7-1-6: Do not use typedef [AUTOSAR 2018, p. 131]	-	2d 40m	Both
10	Implement checker for rule A5-1-6: Lambda return type should be explicitly specified [AUTOSAR 2018, p. 91]	-	1d 4h 50m	Both
11	Implement quickfix replace typedef with using [A7-1-6] [AUTOSAR 2018, p. 131]	-	3h 55m	Both
12	Implement quickfix make lambda return type explicit [A5-1-6] [AUTOSAR 2018, p. 91]	-	1d 4h 10m	Both
14	Implement checker must use lambda parameter list [A5-1-3] [AUTOSAR 2018, p. 89]	-	1h 15m	Both
15	Implement checker switch must have at least two cases [A6-4-1] [AUTOSAR 2018, p. 117]	-	1h 40m	Viktor
16	Implement checker do not use union [A9-5-1] [AUTOSAR 2018, p. 179]	-	2h 30m	Gabriel
17	Implement quickfix add lambda parameter list [A5-1-3] [AUTOSAR 2018, p. 89]	-	1h	Both
18	Implement quickfix replace switch with if-else [A6-4-1] [AUTOSAR 2018, p. 117]	-	3d 1h	Viktor
25	Implement checker do not use literal values outside initialization [A5-1-1] [AUTOSAR 2018, p. 86]	1d 4h	1d 2h 25m	Gabriel
26	Implement checker do not outlive reference captured objects [A5-1-4] [AUTOSAR 2018, p. 90]	3d	4d 25m	Viktor
27	Implement checker do not hide base class member functions [A10-2-1] [AUTOSAR 2018, p. 185]	3d	1d 5h 30m	Gabriel
29	Implement checker do not use register A7-1-4 [AUTOSAR 2018, p. 129]	4h	1h 20m	Gabriel

30	Implement checker do not implicitly capture variables A5-1-2 [AUTOSAR 2018, p. 88]	1d 4h	1d 6h	Viktor
31	Implement checker do not introduce virtual functions in final class A10-3-3 [AUTOSAR 2018, p. 189]	2d	1d 15m	Gabriel
34	Implement checker do not nest lambda expressions A5-1-8 [AUTOSAR 2018, p. 93]	4h	2h 15m	Viktor
35	Implement checker do not use ternary operator as sub-expression A5-16-1 [AUTOSAR 2018, p. 111]	4h	2h 30m	Viktor
36	Implement quick fix remove register keyword A7-1-4 [AUTOSAR 2018, p. 129]	4h	1h 15m	Gabriel
37	Implement quick fix for virtual functions in final class A10-3-3 [AUTOSAR 2018, p. 189]	4h	4h 30m	Gabriel
38	Implement checker virtual functions shall either be virtual override or final A10-3-1 [AUTOSAR 2018, p. 186]	4h	2h 15m	Gabriel
39	Implement quick fix use virtual override or final for virtual function A10-3-1 [AUTOSAR 2018, p. 186]	1d	1d 3h	Gabriel
40	Implement checker use auto sparingly A7-1-5 [AUTOSAR 2018, p. 130]	1d 4h	4h 40m	Viktor
41	Implement checker for only one base class A10-1-1 [AUTOSAR 2018, p. 183]	1d 4h	1d 1h	Gabriel
42	Implement checker for no virtual operators A10-3-5 [AUTOSAR 2018, p. 190]	4h	2h 30m	Gabriel
44	Implement quick fix make implicit captures explicit A5-1-2 [AUTOSAR 2018, p. 88]	4h	4h 45m	Viktor
45	Implement quick fix use auto sparingly A7-1-5 [AUTOSAR 2018, p. 130]	4h	3d 3h 55m	Viktor
46	Implement checker use scoped enum classes A7-2-3 [AUTOSAR 2018, p. 138]	4h	45m	Viktor
47	Implement checker do not use asm A7-4-1 [AUTOSAR 2018, p. 142]	4h	1h	Gabriel
48	Implement checker use override or final when overriding A10-3-2 [AUTOSAR 2018, p. 188]	1d 4h	7h 30m	Gabriel
49	Implement quick fix use override or final when overriding A10-3-2 [AUTOSAR 2018, p. 188]	4h	5h	Gabriel

50	Implement checker explicit underlying enum base type A7-2-2 [AUTOSAR 2018, p. 137]	1d 4h	50m	Viktor
51	Implement quick fix make underlying enum base type explicit A7-2-2 (not completed) [AUTOSAR 2018, p. 137]	4h	2h 35m	Gabriel
52	Implement checker do not initialize auto using initializer list A8-5-3 [AUTOSAR 2018, p. 173]	1d 4h	7h 50m	Viktor
53	Implement quick fix use constrcutor/equals initialization for auto variable A8-5-3 [AUTOSAR 2018, p. 173]	1d	1h	Viktor
55	Implement checker Conditions must be bool A5-0-2 [AUTOSAR 2018, p. 80]	1d 4h	2d 5h 35m	Gabriel
56	Implement quickfix Conditions must be bool A5-0-2 (not completed) [AUTOSAR 2018, p. 80]	1d	3h 45m	Gabriel
57	Implement checker all one or none shall be initialized in enum A7-2-4 [AUTOSAR 2018, p. 139]	1d 4h	1h 20m	Viktor
58	Implement quick fix all one or none shall be initialized in enum A7-2-4 (not completed) [AUTOSAR 2018, p. 139]	1d	1h 45m	Viktor
59	Implement checker braced-initialization A8-5-2 [AUTOSAR 2018, p. 170]	3d	1d 3h 25m	Viktor
61	Implement quick fix use braced initialization A8-5-2 [AUTOSAR 2018, p. 170]	4h	3h 10m	Viktor

Table 9.4.: List of issues

Figure 9.1.: Estimate / Actual comparison



10. Conclusion

In this chapter we will sum up the project and decide whether we are satisfied with what we achieved. We will also write about what we learned as a team. Additionally we will give an outlook on what is to come in the future.

10.1. Retrospective

Here are a few interesting numbers about this project:

- 138 commits
- 25 implemented checkers
- 12 quick fixes
- 3 not implemented quick fixes
- 445 tests
- more than 95% Coverage

When we began the project, the `CodeAnalysator` plug-in already had a working architecture and the first `checkers` as a proof of concept. We did not change anything significant in the architecture. Instead we implemented many checkers and `quick fixes`. In the beginning of the project we set three different scopes. A minimal, a desired and an optional scope. We defined which checkers and quick fixes need to be completed to fulfil each scope. By only looking at the failed quick fixes it looks like we did not fulfil any scope because in each one there is a quick fix that we did not implement. But two of these were because of limitations in the CDT, so we do not consider them failed. This leaves us with only one unimplemented quick fix, which is one that turned out to be far more complicated than it first appeared. In conclusion, this means we have 25 checkers and 12 quick fixes that we estimated roughly accurate, even though there are a few that took longer or shorter to complete. Additionally, there are 2 quick fixes, that were not possible to implement and one that we completely underestimated. Taking these numbers into account, we consider the project to be a success. We learned to work together as a team and how to effectively communicate with each other. We both benefited from this and will hopefully be able to use this ability in the following bachelor thesis.

10.2. Considerations for future projects

The existing architecture as well as the `checkers` and `quick fixes` implemented by us work as they are. But there are some that can be improved or expanded as we already stated in the chapter 7. Additionally, a future project could consist of analyzing the current project thoroughly and refactor where appropriate and improve where possible. With an increasing number of `visitors`, the performance of the plug-in might decrease. Performance tests and improvements would be a possible activity as well.

Bibliography

Cevelop Plug-in Development. Stauber, Tobias (2016).

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

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

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

Help - Eclipse Platform. *Help - Eclipse Platform* (2016). URL: https://www.mendeley.com/reference-management/web-importer%5C#id%5C_2%20http://help.eclipse.org/neon/index.jsp?topic=%5C%2Forg.eclipse.platform.doc.user%5C%2Fconcepts%5C%2Faccessibility%5C%2Faccessmain.htm (visited on 10/05/2018).

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

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

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

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

OSGI overview. *OSGI overview* (2018). URL: <https://www.ibm.com/support/knowledgecenter/en/SSHR6W/com.ibm.websphere.wdt.doc/topics/cosgi.html> (visited on 12/14/2018).

Interface IFunction. *Interface IFunction* (2018). URL: <https://help.eclipse.org/luna/index.jsp?topic=%5C%2Forg.eclipse.cdt.doc.isv%5C%2Freference%5C%2Fapi%5C%2Forg%5C%2Feclipse%5C%2Fcdt%5C%2Fcore%5C%2Fdom%5C%2Fast%5C%2Fcpp%5C%2FCPPASTLambdaExpression.html> (visited on 12/17/2018).

IFunctionType. *IFunctionType* (2018). URL: <https://help.eclipse.org/luna/index.jsp?topic=%5C%2Forg.eclipse.cdt.doc.isv%5C%2Freference%5C%2Fapi%5C%2Forg%5C%2Feclipse%5C%2Fcdt%5C%2Fcore%5C%2Fdom%5C%2Fast%5C%2FIFunctionType.html> (visited on 12/17/2018).

Implicit conversions - cppreference.com. *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. *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. *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. *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. *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=71ed78fcbe9ef8e620ab9a2d043763981bcaeaf> (visited on 11/05/2018).

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

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

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

Glossary

AST An abstract-syntax-tree.. [2](#), [29](#)

AUTOSAR C++ guidelines for use in critical and safety-related systems AUTOSAR [2018](#). [i](#), [ii](#), [1](#), [5](#), [13](#), [15](#), [49–51](#), [53](#)

bundle A Java archive explicitly specifying its imports and exports according to the OSGi standard.. [11](#), [12](#)

CDT CDT & Eclipse C/C++ development tools. Package for creating Eclipse plug-ins for C/C++ development.. [1](#), [2](#), [22–25](#), [36](#)

checker Checks C++ code for a given rule.. [i](#), [ii](#), [1](#), [2](#), [4–6](#), [8](#), [10](#), [11](#), [14](#), [15](#), [17–28](#), [34](#), [35](#), [37](#), [41](#), [42](#), [49](#), [50](#), [52](#), [53](#), [57](#)

CI Continuous Integration. [4](#), [5](#), [34](#)

Codan A static analysis framework in Eclipse CDT.. [2](#), [11](#)

CodeAnalysator Eclipse static code analysis plug-in.. [i](#), [ii](#), [iv](#), [1](#), [2](#), [11](#), [12](#), [41](#), [49](#), [50](#)

Eclipse Eclipse is an integrated development environment.. [1](#), [11](#), [49](#)

fragment A Java archive extending an OSGi bundle according to the OSGi standard.. [11](#)

ILTIS Infrastructure-Layer plug-in, providing Tooling, Id-ioms, and Services Stauber [2016](#). [49](#)

MISRA Programming Standard used in the automobile industry. [49](#)

OSGi Standard for a modular system and a service platform for the Java programming language.. [11](#), [12](#)

quick fix Quick fixes resolve rule violations, by manipulating the AST. [i](#), [ii](#), [1](#), [2](#), [4–6](#), [8](#), [11](#), [12](#), [14](#), [16–28](#), [35–37](#), [41](#), [42](#), [49](#), [57](#), [59–61](#)

visitor Applies the visitor pattern to the AST. [1](#), [2](#), [11-14](#), [30](#), [42](#), [49](#), [50](#), [52-57](#),
[59](#), [60](#)

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. [\[Stauber 2016\]](#)

For a guide on how to use and configure the `CodeAnalysator` plug-in, please refer to chapter A: User manual in the technical report "Safe c++ guidelines `checkers` und quick fixes" by Pascal Bertschi and Andreas Deicha. [\[Bertschi, Pascal 2018\]](#)

A.1. Overview

This guide will focus on implementing `AUTOSAR` rules. Implementing rules for C++ Core Guidelines or `MISRA` works very similarly, so this guide can be still be useful. Some of the folders have to be chosen differently in such a case.

Implementing a rule can be roughly divided into the three 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`.

Important for test driven development It is strongly advised to apply test driven development because writing out the tests gives a good hint on which special cases the `visitor` or `quick fix` will have to work. It is also very convenient to write tests first, because there are no references in the code. But keep in mind, that it is advised to add all the files before testing. If not, the tests will not fail because they are not implemented, but because there is no `visitor` at all for the problem to be tested.

A.2. 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` with many different `visitors`.

A.2.1. 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.

Figure A.1.: AutosarIdHelper

```
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 }
```

Step 2: Register your rule 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 below to see how to format.

Figure A.2.: fragment.xml

```
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="AX-X-X: Example description"
8             id="com.cevelop.codeanalysator.autosar.problem.
9             exampleproblemid"
10            messagePattern="AX-X-X: Example Message"
11            name="AX-X-X: 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>
```

Step 3: Add a visitor In the package `com.cevelop.codeanalysator.core.visitor.shared`, add a new class named `<yourRule>Visitor.java`. The new class must extend `SharedVisitor`. Implement the unimplemented methods. In `setShouldVisit()`, set which parts of the code the visitor should visit. For an overview of the possibilities, check out the class `ASTVisitor`. [\[Help - Eclipse Platform 2016\]](#) `ASTVisitor` is a class higher up in the hierarchy. This class also contains a large number of `visit(...)` methods. You can override these methods in the cases that are relevant to your respective rule. I.e. if you want to visit the declarations, you override the method `visit(IASTDeclaration declaration)` and set `shouldVisitDeclarations` to `true`. In these methods you can check for violations of your rule. If any are found you can use the checker to report them. This will mark them in the code. Your new visitor should look something like this:

Figure A.3.: A newly added visitor

```

1 public class ExampleVisitor extends SharedVisitor {
2
3     public ExampleVisitor(CodeAnalysatorChecker checker, String problemId) {
4         super(checker, problemId);
5     }
6
7     @Override
8     protected void setShouldVisit() {
9         shouldVisitDeclarations = true;
10    }
11
12    @Override
13    public int visit(IASTDeclaration declaration) {
14        /*
15         * Your code
16         */
17
18        // report a problem
19        checker.reportProblem(getProblemId(), declaration);
20
21        return super.visit(declaration);
22    }
23 }
```

Step 4: Add your visitor to the checker The class `AutosarChecker.java` is in the package `com.cevelop.codeanalysator.autosar.checker`. In this class all the AUTOSAR visitors are listed. In the method `initVisitor`, add your newly created visitor.

Figure A.4.: `AutosarChecker.java`

```
1  @Override
2      protected void initVisitor(VisitorComposite visitor) {
3          visitor.add(new ExampleVisitor(this,
4              AutosarIdHelper.ExampleProblemId));
5      }
```

Step 5: Register the new visitor in CoreIdHelper To be able to map your visitor, you first need to create a constant in the file `CoreIdHelper`.

Figure A.5.: `CoreIdHelper`

```
1  public class CoreIdHelper {
2      ...
3      public static final String ExampleVisitorId = "Example";
4      // Your new visitorId
5  }
```

Steps 6 and 7: Map the new visitor and add a suppression strategy To link the visitor and the problem, the visitor needs to be mapped to the problem id. To achieve this there is the class `AutosarGuidelineMapper.java` in the package `com.cevelop.codeanalysator.autosar.guideline`. First the `visitorid` and the `problemid` must be mapped. In the same file you can add a suppression attribute. This will allow the user to suppress the given rule for his code. View the code below to see how this is done.

Figure A.6.: `AutosarGuidelineMapper.java`

```
1 public AutosarGuidelineMapper() {  
2     //...  
3     mappings.put(CoreIdHelper.ExampleVisitorId,  
4         AutosarIdHelper.ExampleProblemId);  
5  
6     suppressionStrategy.addSuppression(CoreIdHelper.ExampleVisitorId,  
7         new AutosarSuppressionAttribute("AXX-X-X"));  
8     //...  
9 }
```

A.2.2. Testing

Step 1: Add a new test class Add a class that extends `AbstractCheckerTest` to the package `com.cevelop.codeanalysator.autosar.tests.checker`. This class does not contain the actual tests, which makes the file very short. Look at the code below to see an example test class.

Figure A.7.: A new test class

```
1 public class ExampleCheckerTest extends AbstractCheckerTest {
2
3     protected IProblemId getProblemId() {
4         return IProblemId.wrap(AutosarIdHelper.ExampleProblemId);
5     }
6 }
```

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

Figure A.8.: Adding the test class to the test suite

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

Step 3: Add a new test rts file In the tests package, there is a folder named resources. In the subfolder `visitors`, 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 configure 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.

Figure A.9.: An example rts file

```
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 */
```

A.3. 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.3.1. 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`. 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.

Figure A.10.: An example quick fix

```
1 public class ExampleQuickFix extends BaseQuickFix {
2
3     String label;
4
5     public ExampleQuickFix(String label) { this.label = label; }
6
7     @Override
8     public String getLabel() {
9         return label;
10    }
11
12    @Override
13    public boolean isApplicable(IMarker marker) {
14        // check wether the quickfix is appliccable
15        return super.isApplicable(marker);
16    }
17
18    @Override
19    protected void handleMarkedNode(IASTNode markedNode,
20        ASTRewrite hRewrite) {
21        // Replace or change parts of the ast to solve the problem
22    }
23 }
```

Step 2: Register the quick fix in the AutosarGuidelineMapper In the same class where you already mapped the `visitor`, `AutosarGuidelineMapper.java`, register your new `quick fix`.

Figure A.11.: Registering your quick fix

```
1 public AutosarGuidelineMapper() {
2     // ...
3     quickfixes.put(AutosarIdHelper.ExampleProblemId,
4         new IMarkerResolution[] {
5             new ExampleQuickFix("AXX-X-X: Example resolution")
6         }
7     );
8     // ...
9 }
```

A.3.2. Testing

Step 1: Add a new test class Add a class that extends `AbstractQuickFixTest` to the package `com.cevelop.codeanalysator.autosar.tests.quickfix`. To make it work, the methods `getProblemId`, `getQuickfix` and `getSuppressionText` need to be overridden. The new test class should look like this:

Figure A.12.: A new quick fix test class

```
1 public class ExampleQuickFixTest extends AbstractQuickFixTest {  
2  
3     @Override  
4     protected IProblemId getProblemId() {  
5         return IProblemId.wrap(AutosarIdHelper.ExampleProblemId);  
6     }  
7  
8     @Override  
9     protected IMarkerResolution getQuickfix() {  
10        return new ExampleQuickFix("");  
11    }  
12  
13    @Override  
14    protected String getSuppressionText() { return "AXX-X-X"; }  
15 }
```

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.

Figure A.13.: Adding the test class to the test suite

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

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 below.

Figure A.14.: An example quick fix rts file

```
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  */
```

B. Acknowledgement

We would like to thank Hansruedi Patzen for his great support for this project. As Peter Sommerlad was ill for most of the project duration, Hansruedi acted as an adviser and attended the meetings instead. He was also very helpful as a technical adviser and in setting up our development environments.

We would also like to thank Tobias Stauber, who helped us a lot with his knowledge of the CDT and Latex.

Additionally, we would like to thank AnneMarie O'Neill for her help regarding English in our documentation.

At last, we would also like to thank Thomas Corbat, who advised us on our documentation and acted on behalf of Peter Sommerlad in some cases.



HSR

HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Eigenständigkeitserklärung

Erklärung

Wir erklären hiermit,

- dass wir die vorliegende Arbeit selbst und ohne fremde Hilfe durchgeführt haben, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass wir sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben haben.
- dass wir keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt haben.

Ort, Datum:

Rapperswil, 17.12.2018

Name, Unterschrift:

Gabriel Uvaset
[Handwritten signature]

Ort, Datum:

Rapperswil, 17.12.2018

Name, Unterschrift:

Viktor Puselja, *V. Puselja*