HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

Student Research Project

Safe C++ Guidelines

University of Applied Sciences Rapperswil
Institute for Software

Period Of Time: 16.09.2019 - 20.12.2019

Author Dario Fuoco
Jonas Pulfer

Supervisor Prof. Peter Sommerlad
Technical Adviser Hansruedi Patzen

Abstract

The main purpose of this student research project is the implementation of safe C++ guidelines. Previous teams implemented the infrastructure for the CodeAnalysator plug-in and have already implemented various coding guidelines. In this work the focus is on the AUTOSAR Guidelines. AUTOSAR is a consortium of the biggest automotive manufacturers which released a coding standard for critical and safety-related systems. The static code analysis is based on this guideline. The analysis process is done by traversing the Abstract Syntax Tree and detecting rule violations. If a violation is found, the code line will be reported with a marker. If possible, a quickfix will directly be provided. In this project 22 rules and 5 quickfixes were implemented.

Management Summary

Introduction

As C++ evolves and provides more and more complex features, the need for guidelines as to how to write safe code rises. For example, safe code is a must for the automotive industry to ensure driver and passenger safety. The AUTOSAR consortium released such C++ rules for that industry. Unfortunately it is very hard to write code following all the rules defined by AUTOSAR. To help by giving immediate feedback, previous student projects have implemented and extended the CodeAnalysator plug-in for the Cevelop IDE with some AUTOSAR rule checks. This plug-in shows a C++ developer rule violations directly in the code as written by underlining the violating code snippet. Various violations can then be directly resolved by applying a quick fix. A quick fix automatically transforms the code such that the code doesn't violate the guidelines anymore. The previous projects already implemented many rules and optimized the infrastructure surrounding the plug-in. The goal of this thesis is to implement further rules to support developers even better.

Approach

Our approach to this task was very similar to the approach the previous projects used. At the start of each week we decided which guidelines to implement next, starting with the trivial ones and slowling working our way to the more complex ones. The implementation of a guideline looked more or less the same for each individual rule: First we analysed the guideline consisting of it's description, it's rationale and the provided examples. If we still weren't sure whether we understood the guideline fully we also checked the C++ language specification. Next we wrote many test to cover the guideline as wide as possible. Once the tests were written we then continued to analyse the C++ abstract syntax tree and wrote the checker according to our findings. In these checkers we traversed the abstract syntax tree to check whether it matched the desired layout to adhere to the guideline. For some violations there is a way to fix it automatically, e.g., if only a keyword was missing. In such cases we also implemented a quick fix based on our knowledge from the earlier analysis. Finally, we also defined test cases for the application of such a quick fix. To make sure our checkers and quick fixes were performant enough to be used in real projects, we tested all of them using a profiler and two realistic open-source projects. While doing these performance checks we also tested whether the checkers reported false positives.

Result

As planned, we have implemented 22 guidelines in addition to 5 quick fixes as well as extensive unit tests for all of the implemented guidelines and quick fixes. A developer who wants to write safe C++ code can use the CodeAnalysator plug-in an will be pointed to possible rule violations and in some cases will be able to automatically fix the violation by applying a quick fix.

Acknowledgements

We would like to thank the following people for their assistance during our student research project:

- **Prof. Peter Sommerlad**
  He, as our supervisor, always provided great support. We could profit and learn a lot from his great knowledge and his many years of experience.
- **Hansruedi Patzen**
  Hansruedi was our technical advisor during our project. He provided us with a lot of technical tips. He also always had a suitable answer for our questions during our work.

Contents

Glossary and List of Abbreviations

AST Abstract Syntax Tree - The Abstract Syntax Tree is a tree representation of the structure of a source code file. Eclipse in combination with the CDT provides several visualisation tools and an API for manipulation of the abstract syntax tree. . 2, 15, 29, 30, 40

CDT C++ Development Toolkit is a set of Eclipse plug-ins. vii, 1, 40

IDE Integrated Development Environment - tool for developing software. 1

IFS Institute for Software from HSR - Institute at the University of applied Sciences Rapperswil. 2, 3

NSDMI Non Static Data Member Initialization is an alternative to non-static member initialization as it allows you to initialize a member along with it's declaration. 22

POD Plain Old Data - type without constructors, destructors and virtual member functions [ref19d]. 20

RAII Resource Acquisition Is Initialization is a C++ programming technique, that binds the life cycle of a resource that must be acquired before use to the lifetime of an object [ref19e] . 14

STL Standard Template Library is a software library for C++. 33

Listings

List of Figures

Chapter 1

Introduction

1.1 Initial situation

Cevelop is an intergrated development environment for C++ code. It's essentially an advanced version of the Eclipse IDE with the CDT and a lot of plugins to assist the developer in writing clean, safe and reliable C++ code. One such plugin is the so called "CodeAnalysator", which is used to check, whether the code adheres to the AUTOSAR- / MISRA guidelines. AUTOSAR and MISRA are two development partnerships which define coding-guidelines for C++. The main goal of these coding-guidelines is to reduce ambiguity and improve robustness in C++ programs. The plugin itself checks the code for the developer, and whenever it detects a violation of a guideline, reports the corresponding code with a marker and, where applicable, offers a quickfix to improve the code.

The plugin, the corresponding infrastructure and some guidelines have already been implemented in earlier projects.

1.2 Task description

The main goal of our student research project is to improve the code compliance checks in Cevelop. We continue the work of an earlier bachelor thesis [PV19], which provided excellent preparatory work. We extended their classification of coding guidelines to the following categories.

- Trivial - approx. 4 to 12 hours
- Medium - approx. 12 to 24 hours
- Complex - approx. 28 hours

Additionally, we define three scopes which frame our project work.

- Minimal - approx. 10 trivial rules, 8 medium rules, 1 complex rule
- Desired - approx. minimal scope + 2 medium rules
- Optimal - approx. desired scope + 1 complex rule

1.3 Code Analysator

The Code Analysator is a plug-in for Cevelop, which is developed by the IFS. This plug-in is used for performing static code analysis on C++ projects. It consists of the core component, which is the base of all static code analysis plug-ins in Cevelop. Currently the base CodeAnalysator plug-in implements static code analysis for AUTOSAR, MISRA and the C++ Core guidelines.
The Code Analysator provides the infrastructure for analyzing the source files from a project. This means that the core plug-in traverse the whole AST and provides the possibility to the specific plug-ins to analyze components of the AST. The traversal of the AST is implemented with the Visitor pattern [Gam+94]. The specific implementation could inherit from the Code Analysator and override this visitor that fulfill his needs.

1.4 Safe C++ with AUTOSAR

AUTOSAR is a partnership of some of the biggest manufacturers (Toyota, BMW, Volkswagen, etc.) in the automotive industry. AUTOSAR has published a document that specifies the usage of C++ in safety-related and critical systems. [AUT18]
The motivation for this coding guideline is to manage the complexity associated with growth in functional scope. Another important point is the achievement of non-functional requirements in the future. Their goal is to fulfill future vehicle requirements in the availability and safety area. But they want also enhance and ensure the exchangeability, maintabilty and reusability of the software. The guidelines are primary written for automotive applications and derived applications from this area such as "embedded systems". The guidelines are not applicable for ultra-hazardous activities like aviation and nuclear science. [AUT19]

Chapter 2

Infrastructure and Environment

As mentioned in the introduction we continue the work of a bachelor thesis [PV19]. They provided an excellent infrastructure for the implementation of static code analysis for Cevelop in cooperation with the IFS. We were able profit a lot from their preparatory work. The project structure looks as follows:

Codeanalysator Autosar

com.cevelop.codeanalysator.autosar
└── src
  ├── com.cevelop.codeanalysator.autosar.checker
  ├── com.cevelop.codeanalysator.autosar.guideline
  ├── com.cevelop.codeanalysator.autosar.quickfix
  ├── com.cevelop.codeanalysator.autosar.util
  └── com.cevelop.codeanalysator.autosar.visitor

Table 2.1: Autosar Project Structure

For testing the code analysis plug-in, the IFS provide a testing infrastructure. With support of this infrastructure, we are able to unit test the plug-in with C++ test cases. The testing structure looks like this:

Autosar Tests

```
com.cevelop.codeanalysator.autosar.tests
├── src
│   ├── com.cevelop.codeanalysator.autosar.tests
│   ├── com.cevelop.codeanalysator.autosar.tests.checker
│   ├── com.cevelop.codeanalysator.autosar.tests.quickfix
│   └── com.cevelop.codeanalysator.autosar.tests.util
└── resources
    └── com.cevelop.codeanalysator.autosar.tests
        ├── checker
        └── quickfix
```

Table 2.2: Testing Project Structure

The C++ test cases are located in the resource section. They are written in ".rts" files and contain the C++ code snippets, which test the checker and quickfix functionality.

Chapter 3

Implemented Rules

In this chapter we split every rule we have implemented into three or more parts:
Rule explains the rule and the reasoning for the guideline.
Analysis explains our thought process as we were implementing the rule.
Checker explains the actual checker we wrote in technical terms. The implementation of the checkers can be found in the following folder: `com.cevelop.codeanalysator.autosar.visitor`.
Quickfix explains the functionality of the quickfix for a given problem. The implementation of the quickfixes can be found in the following folder: `com.cevelop.codeanalysator.autosar.quickfix`
Problem this section is only present if we encountered difficulties during the implementation.

3.1 A00-04-02: Type long double shall not be used

Rule The width of long double type, and therefore width of the significand, is implementationdefined. The width of long double type can be either:

- 64 bits, as the C++14 Language Standard allows long double to provide at least as much precision as type double does, or
- 80 bits, as the IEEE 754 [IEE08] standard allows extended precision formats (see: Extended-Precision-Format), or
- 128 bits, as the IEEE 754 [IEE08] standard defines quadruple precision format

[AUT18]

Listing 3.1: A0-4-2 violation example

```
1  using ldouble = long double; // Non-compliant as alias declaration
2
3  int main(){
4      long double testVar{0}; // Non-compliant as declaration
5      ldouble test{0}; // Non-compliant declared with an alias
6  }
7
8  long double testFunc() {} // Non-compliant as a function return value
9
10 struct A {
11     long double test; // Non-compliant as class member
12 };
13
14 void testFunc(long double testParam) {} // Non-compliant as function parameter
```

Analysis Besides the simple declarations we will also have to check the parameter declarations to check for long double parameters which may be expected in a function. When either a simple- or parameter declaration is found, we check whether the declaration specifier is of type long double.

Checker First we check whether the declaration specifier of the declaration is of type `ICPPASTSimpleDeclSpecifier`. If that is the case, we cast the declaration specifier to the aforementioned type. Afterwards we can us the `isLong()` method provided by `ICPPASTSimpleDeclSpecifier`, and check whether the specifier's type is double.
Visitor: `TypeLongDoubleShallNotBeUsedVisitor.java`
Tests: `TypeLongDoubleShallNotBeUsedCheckerTest.rts`

3.2 A02-10-01: An identifier declared in an inner scope shall not hide an identifier declared in an outer scope

Rule This rule says, that name identifiers from an inner scope should not hide name identifiers from the outer scope. In the rule documentation inner and outer scope are defined as follows:

- Identifiers inside an anonymous namespace, could be taken in consideration as having a outer scope
- Identifiers with a block scope have an inner scope
- Nested blocks inside a scope, introduce an inner scope

Declared identifiers in a named namespaces will not hide other identifiers from an outer scope. Because they can be accessed with a fully-qualified id, for example `NameSpaceName::VariableName`. [AUT18]

Listing 3.2: A2-10-1 violation example with an anonymous namespace

```
#include <cstdint>
std::int32_t hiddenName = 0;
namespace
{
    std::int32_t hiddenName; // Non-compliant, hides hiddenName in outer scope
}
```

Listing 3.3: A2-10-1 violation example inside a lambda

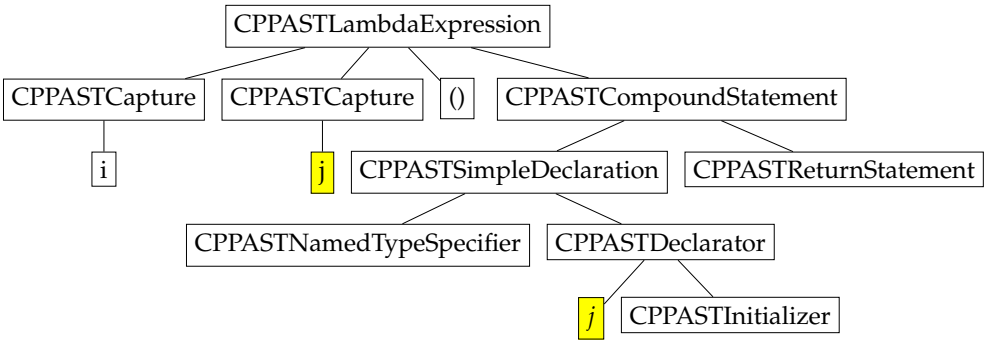
```
#include <cstdint>
int main(){
    std::int32_t a { 0 };
    std::int32_t hiddenName { 0 };
    auto lambda = [a,hiddenName]() {
        // Non Compliant - hiddenName was captured
        std::int32_t hiddenName = 10;
        return a + hiddenName;
    };
}
```

Listing 3.4: A2-10-1 violation example with a function parameter

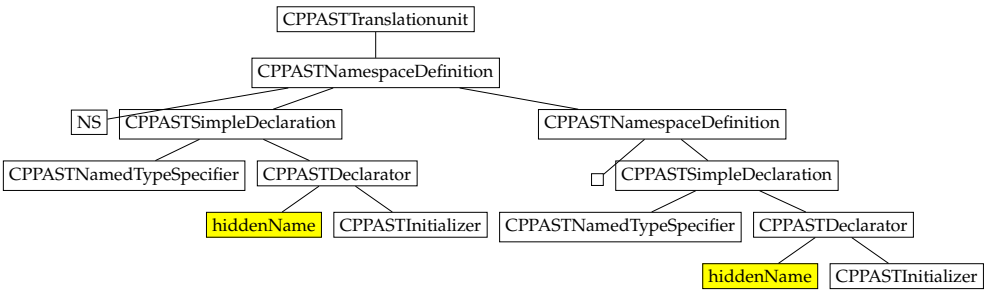
```
#include <cstdint>
std::int32_t hiddenName;

void F1(std::int32_t hiddenName)
{
    //Non-compliant, hides hiddenName in outer scope
}
```

Analysis The analysis for this rule was really complicated and took a lot of effort to implement afterwards. It is split in two parts. One visitor checks all expression for lambdas, which capture variables and re declare them with the same identifier name (see 3.2). In the other, more complicated part, we analyze the whole translation unit. In the first step, all declarations from the outermost scope are being added to a collection. In the next step, we analyze if there are inner scopes. If yes, the inner scope is being analyzed and checked for name hiding. If a hiding is detected, the declarations will be reported. These steps are repeated recursively for other nested inner scopes.

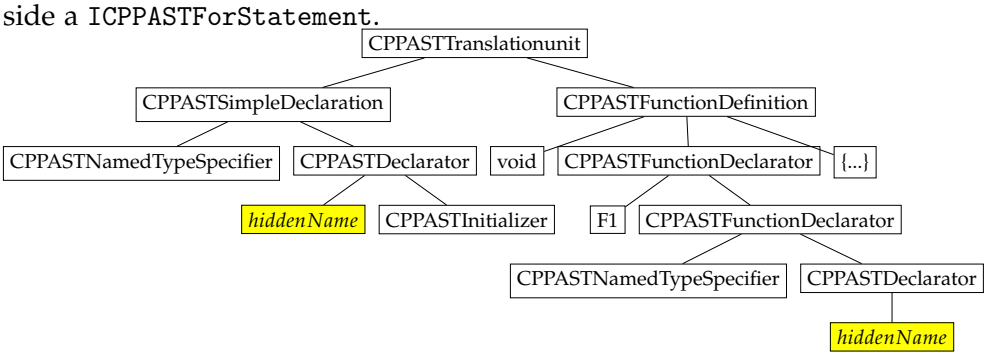
Checker The first visitor checks all `IASTExpression` to see if they are a `ICPPASTLamdpaExpression`. Inside the lambda we check if one of the `ICPPASTCapture` will be redeclared in a `IASTSimpleDeclaration`. If that is the case, we create a marker for the `IASTSimpleDeclaration` that violates the rule.



The second visitor checks the whole `IASTTranslationUnit`. Inside the translation unit, the following `IASTDeclarations` will be analyzed: `ICPPASTNamespaceDefinition`, `IASTSimpleDeclaration` and `ICPPASTFunctionDefinition`. All `IASTSimpleDeclaration` from the outermost scope are being added to a `Map<String, IASTDeclaration>`. `ICPPASTNamespaceDefinition` are being processed for violations and further nested scopes.



In a `ICPPASTFunctionDefinition` it will be checked if identifiers from the outer scope are being hidden in the `ICPPASTParameterDeclaration` or inside a `ICPPASTForStatement`.



Visitor: `IdentifierShallNotHideOuterScopeIdentifiersVisitor.java`
Tests: `IdentifierShallNotHideOuterScopeIdentifiersCheckerTest.rts`

3.3 A02-10-06: A class or enumeration name shall not be hidden by a variable, function or enumerator declaration in the same scope

Rule The C++ Language Standard allows that a class or an enumeration can be hidden by a declaration with the same name. This could be from a variable, data member, functions or a enumerator in the same scope. Using the same names for different declarations can lead to misunderstandings and should be avoided. [AUT18]

Listing 3.5: A2-10-6 violation example class type

```
1  namespace NS1 {
2      class A {
3      };
4
5      void A() {} //non-compliant, hides class A
6  }
7
8  int main(void) {
9      NS1::A();
10     class NS1::A a { }; //accessing hidden class type name
11 }
```

Listing 3.6: A2-10-6 violation example enum type

```
1  #include <cstdint>
2  namespace NS3 {
3  class A {
4  };
5  enum B
6  {
7      A = 0, //non-compliant, hides class A
8  };
9  }
10
11 int main(void) {
12     class NS3::A c { }; //accessing hidden class type name
13     std::uint8_t z { NS3::A };
14 }
```

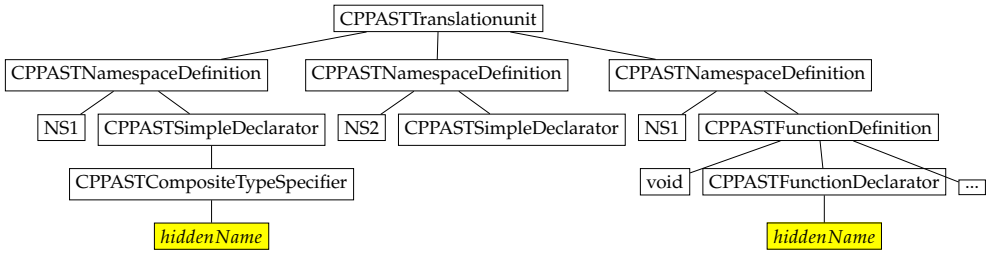
Listing 3.7: A2-10-6 violation example enum and variable

```
#include <cstdint>
namespace NS2 {
    enum class A {
        VALUE = 0,
    };
    std::uint8_t A { 17 }; //non-compliant, hides scoped enum A
}

int main(void) {
    enum NS2::A b; //accessing scoped enum NS2::A
    NS2::A = 7;
}
```

Listing 3.8: A2-10-6 violation example header and sourcefile

```
//@main.h
#ifndef MAIN_H_
#define MAIN_H_

namespace NS1 {
    class G {
    };
}
#endif

//@main.cpp
#include "main.h"
#include <cstdint>
namespace NS1 {
    void G() {} //non-compliant, hides class G inside headerfile
}

int main(void) {
    NS1::G();
    class NS1::G a { }; //accessing hidden class type name
}
```

Analysis The analysis process for this rule includes the whole translation unit. All the defined scopes of the translation unit need to be checked for rule violations. Initially it worked straight forward and the rule did not seem to cause troubles. The use cases mentioned in the Autosar [AUT18] specification could all be done with manageable effort. But with increasing knowledge in the topic, we found test cases which were more complicated to solve, for example splitting the scope in two parts or if the scope is spread over two files (header and source file). To solve these problems, we had to use the index provided by Eclipse.



Checker The checker visits all `IASTDeclaration` of a `ICPPASTTranslationUnit`. During the processing of a scope we differentiate between three possible options: `ICPPASTNamespaceDefinition`, `IASTSimpleDeclaration` and `ICPPASTFunctionDefinition`. With the `IASTSimpleDeclaration` we visit all declarations for the current scope. These names are saved into a collection. In a `ICPPASTNamespaceDefinition` or in a `ICPPASTFunctionDefinition` we check if a name inside scope has been reused. Because scopes could be nested multiple times, the whole processing logic is implemented recursively. To also get hidden variable names in the same scope but different files, we search inside the index. Inside the index we search with the function `findBindings()` for `IBindings` with the same name. As a result we receive a list with all declarations with the given name.. A hiding is given, when for example a function name and a binding of type `IPDOMCPPClassType` with the same name exists
Visitor: `ClassEnumShallNotBeHiddenInSameScopeVisitor.java`
Tests: `ClassEnumShallNotBeHiddenInSameScopeCheckerTests.rts`

3.4 A02-11-01: Volatile keyword shall not be used

Rule This rule intends to reduce incorrect usages of the volatile keyword. As the keyword disables compiler optimizations for a particular object, it is very error prone and often misused. [AUT18]

Listing 3.9: A2-11-1 violation example member variable

```
1 #include <cstdint>
2 class A {
3     volatile std::int32_t test{2}; //violation due to usage of volatile
4 };
```

Listing 3.10: A2-11-1 violation example type declaration

```
1 #include <cstdint>
2 struct A {
3     std::int32_t a;
4     const std::int32_t b;
5 };
6 volatile struct A x; //violation due to usage of volatile
```

Analysis To check for this rule we will have to look at the simple declarations. We simply have to check each declaration specifier for whether they are declared volatile.

Checker The Checker uses the `isVolatile()` method provided by the `IASTDeclSpecifier`, which checks if the volatile keyword is used. If the method returns true, we create a marker to mark the violation
Visitor: `VolatileKeywordShallNotBeUsedVisitor.java`
Tests: `VolatileKeywordShallNotBeUsedCheckerTests.rts`

3.5 A02-13-01: Only those escape sequences that are defined in ISO/IEC 14882:2014 [ISO17] shall be used.

Rule The use of undefined escape sequences leads to undefined behaviour. The defined sequences are: \', \", \?, \\, \a, \b, \f, \n, \r, \t, \v, \<Octal Number>, \x<Hexadecimal Number>.
Additionally universal-character-names (\u hex-quad and \U hex-quad) are also allowed in character and string literals.

Listing 3.11: A2-13-1 violation example

```
1 #include <string>
2    std::string testString1{"\' \" \? \\ \a \b \f \n \r \t \v"}; //compliant
3    std::string testString2{"\x"}; //violation as \textbackslash x is not
         allowed by ISO 14882:2014
4    std::string testString3{"\xGG"}; //violation as \textbackslash xGG is no
         correct hexadecimal number
5    std::string testString4{"\x2555"}; //violation as \textbackslash x2555 is
         no correct octal number
```

Analysis At first it seemed like we would have to split up each individual string and char literal. But after our first few tries we got the idea to write a regular expression which covers the above cases. We have defined the RegEx and defined extensive test cases to make sure all edge cases are covered.

Checker The checker simply checks all expression and if it is a `ICPPASTLiteralExpression` we use the precompiled pattern with the `Pattern.matches()` method provided by Java. When the pattern matches the node gets reported.
Visitor: `OnlyUseEscapeSequencesDefinedInIsoVisitor.java`
Tests: `OnlyUseEscapeSequencesDefinedInIsoCheckerTest.rts`

3.6 A02-13-03: Type wchar_t shall not be used

Rule The width of the type `wchar_t` is implementation defined. Instead of `wchar_t`, `char_16` or `char_32` should be used. [AUT18]

Listing 3.12: A2-13-3 violation examples

```
1  using charAlias = wchar_t; // Non-compliant as alias
2
3  int main(){
4      wchar_t string1[] = L"GHI"; // Non-compliant as declaration
5      charAlias string2[]{L"GHI"}; // Non-compliant declaration with alias
6      auto string3[] = L"GHI"; // Non-compliant as literal with auto
7  }
8  wchar_t testFunc(); // Non-compliant as return type
9
10 struct A {
11     wchar_t string3[10]; // Non-compliant as class member
12 };
13
14 void testFunc(wchar_t string[]) {} // Non-compliant as function parameter
```

Analysis For finding the usage of type `wchar_t` all simple declarations and parameter declarations have to be checked. Inside the simple declarations beside the simple declaration specifier and the function definition, we need to search for alias declarations and named type specifiers as well.

Checker The checker uses `getType()` for getting the type from a simple declaration specifier. Then it can be verified with the type `wchar_t` and is reported when successfully. If the usage of an alias is inspected, the binding of the named type specifier should be checked.
Visitor: `TypeWcharTShallNotBeUsedVisitor.java`
Tests : `TypeWcharTShallNotBeUsedCheckerTest.rts`

3.7 A05-07-01: A lamdbda shall not be an operand to decltype or typeid

Rule Due to each lambda expression having a different unique and unnamed classtype, the use of lambda in conjunction with `decltype` / `typeid` is heavily discouraged, as two lambda which are defined as exactly the same function still are not considered equal by if-conditions for example.

Listing 3.13: A5-7-1 violation example

```
1  #include <cstdint>
2  #include <vector>
3
4  static auto lambda1 = []() -> std::int8_t { return 1; };
5  std::vector<decltype(lambda1)> v; //non-compliant
```

Listing 3.14: A5-7-1 violation example

```
#include <typeinfo>
int testVar{0};

const std::type_info& type{typeid(testVar)}; //non-compliant
```

Analysis To check for the usage of lambda both in `decltype` and `typeid` we will have to check all expressions(typeid), as well as all `DeclSpecifiers`(decltype). Afterwards we will work with the `ICPPInternalBinding` to check the contained `ExpressionType`.

Checker We check both all expression and all declaration specifiers. Once we have found one of either, we can either use the `getExpressionType()` function of either the operand of `typeid` or of the `declTypeExpression` which is contained in the declaration specifier. This expression type then has to be cast into a `ICPPInternalBinding`, which allows us to use `getDefinition()` to check whether this definition is an instance of `ICPPASTLambdaExpression`.
Visitor: `LambdaShallNotBeAnOperandToDeclTypeOrTypeIdVisitor.java`
Tests: `LambdaShallNotBeAnOperandToDeclTypeOrTypeIdCheckerTest.rts`

3.8 A06-02-02: Expression statements shall not be explicit calls to constructors of temporary objects only.

Rule This rule defines that temporary objects should be avoided. Temporary objects are unnamed variables or implementations of the RAII. Such temporary objects only live until the end of the expression. [AUT18]

Listing 3.15: A6-2-2 violation example mutex

```
#include <cstdint>
#include <mutex>
class A {
public:
    void SetValue(std::int32_t value) {
        std::lock_guard<std::mutex> { m_mtx }; // Non-compliant: temporary
            object
        m_value = value; // Assignment to m_value is not protected by lock
    }

private:
    mutable std::mutex m_mtx;
    std::int32_t m_value;
};
```

Listing 3.16: A6-2-2 violation example temporary object

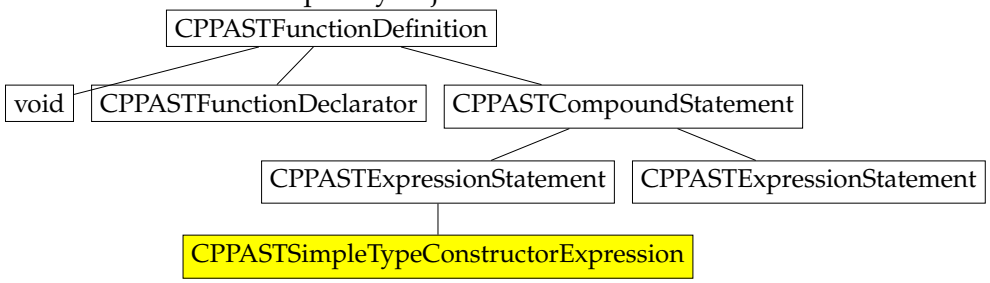
```
#include <string>
#include <fstream>

void PrintNonCompliant(std::string const &fname, std::string const &s) {
    // Non-compliant: only constructing a temporary object
    std::ofstream { fname };
}
void PrintCompliant(std::string const &fname, std::string const &s) {
    // Compliant: Not only constructing a temporary object
    std::ofstream { fname }.write(s.c_str(), s.length());
}
```

Listing 3.17: A6-2-2 violation example only construction of a temporary object

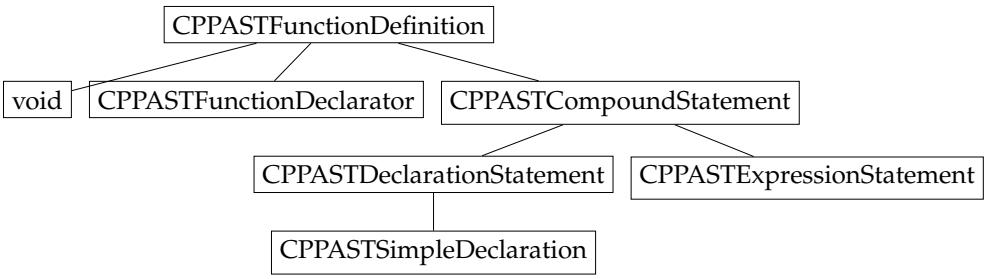
```
struct A {
    A(std::int32_t i1, std::int32_t i2) : i1(i1), i2(i2) {};
private:
    std::int32_t i1;
    std::int32_t i2;
};
int main(){
    A { 5,4}; // Non-compliant only construction of an object
}
```

Analysis To check this rule, every expression needs to be checked. The check is really straight forward, if a expression constructs a temporary object and is not a field reference, then the rule is violated and the marker will be set on the expression.

Checker The checker is straightforward and effective. All expression of type `ICPPASTSimpleTypeConstructorExpression` are potential candidates for a rule violation. To confirm the violation, it needs to be ensured that it is a autonomous expression statement and it is not a field reference. If both conditions are true, the rule is violated and the marker can be set.
The AST view of a temporary object looks as follows:



Whereas an object with a variable name looks like this:



Visitor: `ExpressionsShallNotBeCallsToTempObjectsCtorsVisitor.java`
Tests: `ExpressionsShallNotBeCallsToTempObjectsCtorsCheckerTest.rts`

Quickfix In our first attempt, we wanted to create a sort of inplace editor, in which the temporary object could be given a name. We spent a lot investigation effort, but we failed on the Eclipse API. So we decided to implement a simpler version of this quick-fix. If this quick-fix is applied, the temporary object receives a name "pleaseRename".

Listing 3.18: A6-2-2 quickfix example

```
1  //before quickfix -> line 3 is marked
2  void SetValue(std::int32_t value) {
3      std::lock_guard<std::mutex> { m_mtx };
4      m_value = value;
5  }
6
7  //after quickfix
8  void SetValue(std::int32_t value) {
9      std::lock_guard<std::mutex> pleaseRename { m_mtx };
10     m_value = value;
11 }
```

Quickfix: `ExpressionsShallNotBeCallsToTempObjectsCtorsQuickFix.java`
Tests: `ExpressionsShallNotBeCallsToTempObjectsCtorsQuickFixTest.rts`

3.9 A06-05-02: A for loop shall contain a single loop-counter which shall not have floating-point type.

Rule The rule says that a for loop shall contain only one loop-counter. If a loop without loop-counter is desired, a while loop is the more appropriate solution. This rule also states that floating-point types should not be used as loop-counter. This is due to encountering problems with equality or inequality checks. [AUT18]

Listing 3.19: A6-5-2 violation examples

```
#include <cstdint>
int main(){
    constexpr std::int32_t xlimit{20};
    constexpr std::int32_t ylimit{15};

    // Non-compliant, two loop-counters declared inside the loop
    for (std::int32_t a {0}, b{0}; (a < xlimit) && (b < ylimit) ;a++, b++)
    {
        // ...
    }

    std::int32_t c{0};
    std::int32_t d{0};

    // Non-compliant, two loop-counters but declared outside the loop
    for (; (c < xlimit) && (d < ylimit) ;c++, d++)
    {
        // ...
    }

    // Non-compliant, two loop-counters but one modified inside the loop
    for(std::int32_t e{0}, f{0}; e < xlimit;e++) {
        f++;
    }

    constexpr float glimit = 2.5F;
    // Non-compliant, float with !=
    for (float g = 0.0F; g != glimit;g += 0.1F)
    {
        // ...
    }
}
```

Analysis The analysis process for this rule was more complicated than initially thought. Because of the widely ranged origin of problems, the whole declaration of a for loop and its body must be computed for different violations. The most problematic point of the analysis process is that a loop-counter could be declared outside the for statement.

The only case which is not covered by our analysis steps is if two loop counter are declared outside the loop and are incremented or decremented inside the body of the loop.

Following case are currently being detected by our analysis process - Two loop counters:

- Two loop counters, both declared and incremented inside the for header
- Two loop counters, one declared inside the for header and one outside, but both incremented inside the for header
- Two loop counters, both declared outside, but both incremented inside the for header
- Two loop counters, both declared inside the for header, one incremented in the for header and one incremented in the body
- Two loop counters, both declared inside the for header and both incremented in the body
- Declared loop counter are being checked, if they are used inside the for header and the body. If two are declared but not used, no marking process will be started

The following case is currently detected by our analysis process - comparison of floating type:

- Loop counter declared as float inside for header

Checker The checker computes every section of the for loop. First of all `IASTDeclaration` are checked, if they contain multiple counters or a counter of the type float. All declared variables are put into a collection for further checks. Next the conditions are being tested for whether the `IASTExpression` type is float. Multiple nested `IASTBinaryExpression` are being checked as well with a recursive method. In a further step all the iteration expressions will be checked if two loop counters are being used. If two loop counters have been declared inside the loop, the iteration expression and the `ICPPASTCompoundStatement` of the loop will be checked for their usage.

Visitor: `ForLoopShallContainSingleCounterNoFloatsVisitor.java`
Tests: `ForLoopShallContainSingleCounterNoFloatsCheckerTest.rts`

3.10 A07-01-09: A class, structure, or enumeration shall not be declared in the definition of its type.

Rule This rule says, that a class, struct or enum shall not be declared right after it's definition, as combining a type definition with a declaration of another entity can lead to readability problems and can be confusing for a developer. [AUT18]

Listing 3.20: A7-1-9 violation example enum

```
1 enum class DIRECTION
2 {
3     UP,
4     DOWN
5 } dir; //non-compliant -> dir is declared right after the definition of the
      DIRECTION enumeration
```

Listing 3.21: A7-1-9 violation example class type

```
1 #include <cstdint>
2 struct Bar
3 {
4     std::uint32_t a;
5 } barObj; //non-compliant
```

Analysis As classes, structs and enumerations all appear inside of simple declarations, we check all of those. Inside of the simple declaration we can check whether it contains one of the aforementioned type specifiers. If it does, we can then check if one of the rest of the children is a declarator.

Checker Just as the analysis suggested, the checker works by visiting each `IASTSimpleDeclaration` and checking it's children. If a child is of type `ICPPASTCompositeTypeSpecifier` or `ICPPASTEnumerationSpecifier`, then all other children are checked to see if they are of type `IASTDeclarator`. If this is true for one of the children, the node gets reported.
Visitor: `DoNotDeclareTypesInTheirDefinitionVisitor.java`
Tests: `DoNotDeclareTypesInTheirDefinitionCheckerTest.rts`

Quickfix The quickfix for this rule is a bit more complicated, as it has to be able to take the declarations out of the definition and create a new declaration statement. To achieve this, we first extract all variable and / or pointer names from the `IASTSimpleDeclaration`. Once we have those, we also have to get the `IASTName` of the class/enum/struct. Now we create a new `IASTSimpleDeclaration` and an array of new `IASTDeclarators` which we combine into a new `IASTDeclarationStatement`. As there is only an `insertBefore()` method to rewrite the AST we have to first copy the old `IASTSimpleDeclaration` and reinsert it before the original one, then we replace the old declaration with our new `IASTDeclarationStatement`.

Listing 3.22: A7-1-9 quickfix example

```
1  //before quickfix -> line 4 is marked
2  struct Bar {
3      int a;
4  } barObj{10};
5
6  //after quickfix
7  struct Bar {
8      int a;
9  };
10 Bar barObj { 10 };
```

Quickfix: `DoNotDeclareTypesInTheirDefinitionQuickFix.java`
Tests: `DoNotDeclareTypesInTheirDefinitionQuickFixTest.rts`

3.11 A08-04-01: Functions shall not be defined using the ellipsis notation.

Rule This rules says that functions shall not be defined using the ellipsis notation. Usage of variadic arguments leads to bypassing of the type check from the compiler [ref19f]. When a non POD class type is used, it results in undefined behavior. Instead variadic templates should be used. They offer a type-safe alternative for the ellipsis notation. [AUT18]

Listing 3.23: A8-4-1 violation examples

```
1  void Print1(const char* fmt...) // Non-compliant - variadic arguments are used
2  {
3      // ...
4  }
5  void Print2(...) // Non-compliant - variadic arguments are used
6  {
7      // ...
8  }
```

Analysis This rule can be violated by all declarations. If a declaration is a function, it is checked more closely. The queries are made on the binding of the function and on the function type. If variadic arguments are passed, then this guideline is violated.

Checker For this rule, violations are really simple to detect. If we have a function definition, we need to resolve the binding. Resulting of the binding resolution we receive an `ICPPFunction`. On the basis of that, we could get the function type as a `ICPPFunctionType`. On the function type is defined a member function `takeVarArgs()`, which returns a boolean when variadic arguments are taken by that function.
Visitor: `FunctionsShallNotBeDefinedInEllipsisNotationVisitor.java`
Tests: `FunctionsShallNotBeDefinedInEllipsisNotationCheckerTest.rts`

3.12 A08-04-04: Multiple output values from a function should be returned as a struct or tuple

Rule This rule says that function should return multiple output values as struct or tuple. [AUT18] This means that functions parameter that are delivered as reference, should not be manipulated as a side effect inside a function. In consultation with Prof. Sommerlad, we defined that the return value should preferably be a struct instead as a tuple.

Listing 3.24: A8-4-4 violation example

```
1  // remainder is passed by reference
2  int divide(int dividend, int divisor, int& remainder)
3  {
4      if(divisor==0){
5          return 0;
6      }
7      // Non-compliant remainder is modified and returned as side effect
8      remainder = dividend % divisor;
9      return dividend / divisor;
10 }
```

Analysis In a first step all function declarations are being checked, if they have parameters passed by reference. Is that the case, they are put inside a collection. In a second step the whole function body will be checked, if something is written in the parameters passed by reference.

Checker The checker visits two different node types. First it visits all `IASTDeclaration`, with the intention of finding an `ICPPASTFunctionDefinition`. If a function definition has been found, its parameters are checked via the `ICPPASTFunctionDeclarator`. All the parameters of type `ICPPASTParameterDeclaration` that are passed by reference are put into a `Map`. The `Map` contains the names of the parameters and it's `ICPPASTFunctionDefinition`. In a second visitor all `IASTExpression` will be visited. If the expression is a `ICPPASTBinaryExpression`, then it will be checked if the `IASTExpression` is inside a function. In that case the `Map` will be checked, if it contains the the first operand of the `ICPPASTBinaryExpression` and the return-type is not of type `void`, then a marker will be set.

Visitor: `MultipleOutputBeReturnedAsStructOrTupleVisitor.java`
Tests: `MultipleOutputBeReturnedAsStructOrTupleCheckerTest.rts`

3.13 A11-01-03: Friend declarations shall not be used.

Rule The aim of this rule is to maintain encapsulation and force the programmer to produce code that is easier to maintain. There is one exception to this rule. It is allowed to declare comparison operators as friend functions, this is due to Autosar Guideline A13-05-05 [AUT18]

Listing 3.25: A11-1-3 violation example

```
1  class A {
2      public:
3          // Non-compliant
4          friend A const operator+(A const& lhs, A const& rhs);
5          // Compliant by excpetion (relational operator)
6          friend bool operator ==(A const& lhs, A const& rhs) {
7              return false;
8          }
9          void foo(int a) {}
10 };
11
12 class B {
13     // Non-compliant
14     friend void A::foo(int a);
15 };
```

Analysis Ignoring the exception to the rule, this rule is quite easy to check for. We simply have to visit all declaration specifiers. Then we simply have to check if the specifier is a friend or not. The corresponding interface offers a method to check if the specifier is a friend [ref19b]. Now for the exception it is a little bit trickier. First we have to check if any of the sibling nodes of the specifier is a function declarator, because all operator declarations are done using function declarators. Afterwards we have to extract the operator name and match it's raw signature to a regex which matches all comparison operators. If it matches the node is not reported.

Checker First we check if the current `IASTDeclSpecifier` is part of the exception case or not. We do this by first looping through all it's sibling nodes and checking if any of them are of type `ICPPASTFunctionDeclarator`. If that is the case, we then check if the declarator's name is of type `ICPPASTOperatorName`. If yes, we use `getRawSignature()` and try to match the signature with the correct regex. If the regex matches, the method returns without marking a node.

If the declaration specifier is not part of the exception case, we simply use the `isFriend()` method provided by the `ICPPASTDeclSpecifier` interface and mark the node if it returns `true`.

Visitor: `FriendDeclarationsShallNotBeUsedVisitor.java`
Tests: `FriendDeclarationsShallNotBeUsedCheckerTest.rts`

3.14 A12-01-02: Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type.

Rule According to AUTOSAR [AUT18] both NSDMI and non-static member initialization in a constructor should never be used at the same time, no matter what variable is initialized which way. However for the purpose of this checker we have adjusted the rule together with Prof. Sommerlad to only disallow non-static member initialization in a default constructor and only if the same variable is initialized with NSDMI. An example of a violation according to the original rule can be found below(see listing 3.26). The rule exists to prevent confusion as to which values are actually used to initialize a variable. [ref19c]

Listing 3.26: A12-1-2 violation according to autosar [AUT18] example

```
1  #include <cstdint>
2
3  class A {
4      public:
5      A(std::int32_t i1) : i1{10} {} //i1 is initialized using non static
           member initialization
6
7      private:
8      std::int32_t i1{10}; //i1 is also initialized using NSDMI
9      std::int32_t i2{10}; //according to the autosar rule, this is also a
           violating node
10 };
```

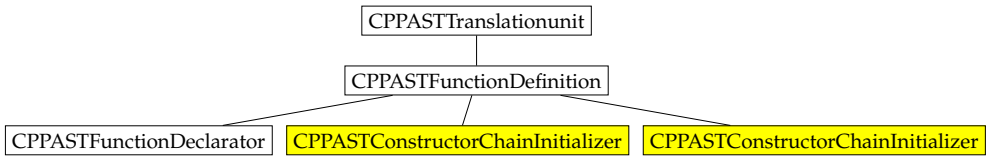
Listing 3.27: A12-1-2 violation example

```
1  #include <cstdint>
2
3  class A {
4      public:
5      A() : i1{10} {} //i1 is initialized using non static member initialization
6
7      private:
8      std::int32_t i1{10}; //i1 is also initialized using NSDMI
9  };
```

Analysis This rule proved to be a pretty hard one to analyse at first. There were multiple possibilities as to how we could solve this problem and we had to try over a few times. At first we thought visiting all initializers and grouping by their composite type specifier was a good solution, however we soon realized that without the binding from the index we would always have performance problems as there were multiple for- and while-loops in our checker. Next we tried to solve the problem by visiting each declaration specifier and searching for `ICPPASTCompositeTypeSpecifier`.

If we did that we had troubles visiting possible inline constructors. Finally we settled for checking all declarations and then simply finding all default-constructors and afterwards checking whether the binding of it's member initializers had an initial value via the binding. If the binding does have one, we create a marker for the non static member initializer in the constructor.

Checker In the checker we first try to find all `ICPPASTFunctionDefinition` by visiting all `IASTDeclaration` then we check whether the function definition is a default constructor by checking the amount of parameters the the function takes. If there are parameters we check everyone of them to see if one of them has no initializer. If that is the case, it is not a default constructor. Once we know whether our `ICPPASTFunctionDefinition` is a default constructor we can then check each of it's `ICPPASTConstructorChainInitializers`.



The first thing we do with each initializer is to save the owner, which represents the class in which the variable is defined, of their respective binding. Next we save the content of the initializer into another local variable. Now we have to check whether the move or copy constructor is used inside of the initializers. If that is the case, the node is compliant by exception. Otherwise we then try to get the binding from the index. Once we have the binding, the last step is to compare the owner, as well as the content of each initializer and it's respective binding. If both the owner and the content match, we create a marker for this node and add a flag to signal the availability of a quickfix. If only the owner matches we create a marker for the node and add a different flag to indicate that the quickfix can not be applied in this case.

Quickfix The quickfix is quite simple, thanks to our preparatory work in the visitor. We first decide whether the quickfix is even applicable by checking the flags we've set during the reporting in the visitor. If both initializers contain the same value, the quickfix is applicable, otherwise it is not. The fix itself simply removes the `ICPPASTConstructorChainInitializer` node. We only offer the quickfix if the same values are used because if there are multiple values used to initialize the variable, as we cannot decide for the user which of the two should stay.

Listing 3.28: A12-1-2 quickfix is not applicable example

```
#include <cstdint>

class A {
    public:
    A() : i1{10} {} //i1 is initialized with 10

    private:
    std::int32_t i1{20}; //i1 is initialized with 20 -> different value
};
```

Listing 3.29: A12-1-2 quickfix is applicable example

```
#include <cstdint>

class A {
    public:
    A() : i1{10} {} //i1 is initialized with 10

    private:
    std::int32_t i1{10}; //i1 is initialized with 10 -> same value
};

//after quickfix is applied

class A {
    public:
    A() {} //the constructor chain initializer is gone

    private:
    std::int32_t i1{10};
};
```

Quickfix: `EitherNsdmiOrNsmiInTheSameTypeQuickFix.java`
Tests: `EitherNsdmiOrNsmiInTheSameTypeQuickFixTest.rts`

3.15 A12-01-04: All constructors that are callable with a single argument of fundamental type shall be declared explicit.

Rule This rule is required to make sure, a constructor is not used to implicitly convert a fundamental type into a class type. According to autosar [AUT18] only constructors callable with a single argument of fundamental type shall be declared explicit [ref19a], however after a discussion with Prof. Sommerlad we have decided to include all constructors which are callable with a single argument.

Listing 3.30: A12-1-4 violation example

```
1  #include <cstdint>
2
3  class A {
4      A(std::int32_t testVar) {...} //violation
5  }
6
7  class B {
8      B(std::int32_t testVar, std::int32_t testVar1 = 10) {...} //violation
9  }
10
11 class C {
12     explicit C(std::int32_t testVar) {...} //compliant
13 }
```

Analysis This rule only concerns constructors so we visit all declarators and first check if they are function declarators without a return type only one parameter declaration and not explicit yet. If all those checks are fullfilled we then have to find out whether the function declarator is the one of the declaration or the one of the implementation. If it is the one of the implementation we return because we only the declaration should be declared explicit. If it is the one of the declaration we check the amount of parameters if the amount is one we report the node, if it is higher than one we check all parameters to see if they have default values and report the node if all other parameters have default values.

Checker First we check if the `IASTDeclarator` is of type `ICPPASTFunctionDeclarator`, if that is the case, we check the parent of the declarator to determine whether we are handling the declarator of the declaration or of the implementation. If it is the one of the declaration we then use `getDeclSpecifier()` to get the `ICPPASTSimpleDeclSpecifier` of the declarator and continue to checking the parameters. To do this we first check if the declaration specifier has an unspecified return type (means it is a constructor), if the declaration specifier is not already explicit and the amount of parameters is exactly one. If all of this is the case we report the declarator. If the amount of parameters if higher we check all parameters and see how many of them do not use `IASTEqualsInitializer` to define a default value for the parameter. Lastly we check the amount of total non-defaulted parameters, if it is exactly one we report the declarator.
Visitor: `SingleArgumentCtorsShallBeExplicitVisitor.java`
Tests: `SingleArgumentCtorsShallBeExplicitCheckerTest.rts`

Quickfix The quickfix for this rule is very simple. We simply copy the `ICPPASTSimpleDeclSpecifier` and set explicit to true. Finally we replace the old simple decl specifier with the copied one.

Listing 3.31: A12-1-4 quickfix example

```
1  //before quickfix -> line number x is marked
2  class A {
3      A(int testArg) {}
4  };
5
6  //after quickfix
7  class A {
8      explicit A(int testArg) {
9      }
10 };
```

QuickFix: `SingleArgumentCtorsShallBeExplicitQuickFix.java`
Tests: `SingleArgumentCtorsShallBeExplicitQuickFixTesr.rts`

3.16 A12-01-06: Derived classes that do not need further explicit initialization and require all the constructors from the base class shall use inheriting constructors.

Rule This rule says, that constructors of derived classes should not be re-implemented if the derived constructors initialize the object exactly the same way. Inherited constructors of the base class should be used instead. [AUT18]

Listing 3.32: A12-1-6 violation example

```
1  #include <cstdint>
2  class A
3  {
4      public:
5      A(std::int32_t x, std::int32_t y) : x(x), y(y) {}
6      explicit A(std::int32_t x) : A(x, 0) {}
7
8      private:
9      std::int32_t x;
10     std::int32_t y;
11 };
12
13 class B : public A
14 {
15     public:
16     // Non-compliant same constructor
17     B(std::int32_t x, std::int32_t y) : A(x, y) {}
18     // Non-compliant same constructor
19     explicit B(std::int32_t x) : A(x) {}
20 };
```

Analysis This rule is divided into two parts. In a first step all constructors of a base classes need to be detected. This is done with a visitor to all base specifiers.The found constructors are then saved in a collection. In a second step, another visitor checks all declarations. If this is a constructor from a derived class, it can be compared with those from the collection.

Checker This rule visits two different types. First, all `ICPPASTBaseSpecifier` are vistited, for getting all constructors of a base and derived class. The constructors as `ICPPConstructor` are detected with the binding on the `ICPPClassType`. Both collections with the constructors of the base and derived class are compared based on their signature. If the signatures are identical, they are being put in a further collection.

In a second step in another visitor, that checks every `IASTDeclaration`, the checks are more detailed. If a constructor in a derived class only calls the constructor of a base class, the rule is violated and the constructor is marked.

Visitor: `DerivedClassesShallUseInheritingCtorsVisitor.java`
Tests: `DerivedClassesShallUseInheritingCtorsCheckerTest.rts`

Quickfix For this rule a quickfix is a good feature. The main implementation of this quickfix was of medium difficulty, the more challenging part was the avoidance of the abuse cases. For example the unique usage of the base constructors.

Listing 3.33: A12-1-6 quickfix example

```
1  //before quickfix -> line 14 is marked
2  #include <cstdint>
3  class A {
4  public:
5      A(std::int32_t x, std::int32_t y) : x(x + 8), y(y) {}
6
7  private:
8      std::int32_t x;
9      std::int32_t y;
10 };
11
12 class B : public A {
13 public:
14     B(std::int32_t x, std::int32_t y) : A(x,y) {} // Non-compliant
15 };
16
17 //after quickfix
18 #include <cstdint>
19 class A {
20 public:
21     A(std::int32_t x, std::int32_t y) : x(x + 8), y(y) {}
22
23 private:
24     std::int32_t x;
```

```
25      std::int32_t y;
26  };
27
28  class B : public A {
29  public:
30      using A::A;
31  };
```

Quickfix: `DerivedClassesShallUseInheritingCtorsQuickFix.java`
Tests: `DerivedClassesShallUseInheritingCtorsQuickFixTest.rts`

Problems In a case with multiple inheritance structures (base, derived and subderived) and the derived class using the constructors of base, problems occur with the quickfix. The subderived class inherits from derived and has it's own constructor, which violate the rule.

Listing 3.34: A12-1-6 problem case multiple inheritance hierarchy

```
1   #include <cstdint>
2   class A
3   {
4       public:
5       A(std::int32_t x, std::int32_t y) : x(x + 8), y(y) {}
6
7       private:
8       std::int32_t x;
9       std::int32_t y;
10  };
11
12  class B : public A
13  {
14      public:
15      using A::A;
16  };
17
18  class C : public B
19  {
20      public:
21      C(std::int32_t x, std::int32_t y) : B(x, y) {} // Non-compliant
22  };
```

Unfortunately the constructor B(int,int) as shown in the example above, is not listed in the AST, when the class C is analyzed. In consultation with Prof. Sommerlad on this case no further efforts have to be done.

The same issue occurs, if the quick fix is applied on the following code snippet with a multiple inheritance structure. The quick fix is only possible, if it is utilised from bottom-up. Otherwise we ran into same situation as shown above in the first problem case.

Listing 3.35: A12-1-6 problem case multiple inheritance hierarchy

```
#include <cstdint>
class A
{
    public:
    A(std::int32_t x, std::int32_t y) : x(x), y(y) {}

    private:
    std::int32_t x;
    std::int32_t y;
};

class B : public A
{
    public:
    B(std::int32_t x, std::int32_t y) : A(x, y) {} // Non-compliant
};

class C : public B
{
    public:
    C(std::int32_t x, std::int32_t y) : B(x, y) {} // Non-compliant
};
```

3.17 A12-04-01: Destructor of a base class shall be public virtual, public override or protected non-virtual

Rule If a type is used as a base class, it's destructor should be either public and virtual, public and override or protected and non-virtual. This prevents the destructors for derived types from not being invoked. [AUT18]

Listing 3.36: A12-4-1 violation example

```
class Base {
public:
    ~Base() // Non-compliant destructor is not virtual
    {
    }
};
class Derived : public Base { // Marker will be set here
};
```

Problems The implementation for the checker of this rule caused us a lot of difficulties. In our first try, we parsed all declarations and had issues finding out which type is the base type. In the second attempt we parsed all the Base Specifiers. In that case we did not find a relation between the base specifier and the composite type specifier of the base specifier. In our next approach, we tried to get the information from the class type.

We could make the checks successfully, but did not have the ability to report the correct AST node for the marking process. We came to the agreement with Prof. Sommerlad that marking the Base Specifier is the best possible solution.

Analysis The easiest way to get all base classes, is to visit all `ICPPASTBaseSpecifier`. This makes the analysis of which class is derived and which is a base class, obsolete. In our case only the the base class needs to be analysed.

Checker The checker for this rule is very simple. With the Name Specifier from the Base Specifier, we get the `ICPPClassType` from the base class. After we iterate over all methods from the class. During the iteration it will be checked if the method is a destructor and if it violates this guideline. All these properties can be checked with member methods from `ICPPMethod`.
Visitor: `DestructorShallBeVirtualOverrideProtectedVisitor.java`
Tests: `DestructorShallBeVirtualOverrideProtectedCheckerTest.rts`

3.18 A13-02-02: A binary arithmetic operator and a bitwise operator shall return a "prvalue"

Rule Returning a type "T" from binary arithmetic and bitwise operators is consistent with the C++ Standard Library. [AUT18]

Listing 3.37: A13-2-2 violation example [AUT18]

```
#include <cstdint>

class A{
};

A operator+(A const&, A const&) noexcept // Compliant
{
    return A{};
}

std::int32_t operator/(A const&, A const&) noexcept // Compliant
{
    return 0;
}

A operator&(A const&, A const&)noexcept // Compliant
{
    return A{};
}

const A operator-(A const&, std::int32_t) noexcept // Non-compliant
{
    return A{};
}

A* operator|(A const&, A const&) noexcept // Non-compliant
{
    return new A{};
}
```

Analysis For this rule every declaration needs to be visited. Of all declarations, the function definitions are relevant for this rule. If a function is an operator it will be checked, whether the return type is const or a pointer and a class type.

Checker The checker vists all `IASTDeclarator` of a file. If a declaration is of type `ICPPASTFunctionDeclarator`, then it proceeds with further analysis. The next verification step is, if the function definition is a operator. This is done with the `getName()` method of `ICPPASTFunctionDeclarator`. If the result is an `instanceof CPPASTOperatorName` , then it is an operator. For getting further information from the function, we resolve it's binding of type `ICPPFunction`.

Once we have the binding, we check the `ICPPFunctionType` and it's return type with the function `getReturnType()`. A violation of the rule is reported, if `ICPPFunctionType` is `instanceof CPPPointerType` or `CPPQualifierType` and it's return type is `instanceof ICPPClassType`.

Visitor: `BinaryArithOperatorShallReturnPrValueVisitor.java`
Tests: `BinaryArithOperatorShallReturnPrValueCheckerTest.rts`

3.19 A18-01-02: The std::vector<bool> specialization shall not be used

Rule The `std::vector<bool>` specialisation should not be used, because it does not work with all STL algorithms as expected. Particularly the `operator[]()` does not return a contiguous sequence of elements as usual in the default behavior of the `std::vector<T>`. The C++ Standard guarantees the safe concurrent modification of distinct elements in STL containers. Only the `std::vector<bool>` violates this guarantee. [AUT18] To be honest, in our opinion `std::vector<bool>` shoud have never been implemented like that, because it creates more trouble than benefits.

Listing 3.38: A18-1-2 violation examples

```
1  #include <vector>
2  int main(){
3      std::vector<bool> v2{}; // Non-compliant as declaration
4  }
5  void testFunc(std::vector<bool>) {} // Non-compliant as function parameter
6
7  std::vector<bool> testFunc() // Non-compliant as return type
8  {
9      std::vector<bool> v2{};
10     return v2;
11 }
```

Analysis For this rule the declarations and parameter declarations need to be visited. Declarations can be a simple declaration or a function definition. In both cases, the system checks whether a vector of bools is used. For functions, the parameters and the return value are also checked.

Checker The checker visits all declarations and distinguishes between `IASTSimpleDeclaration` and `IASTFunctionDefinition`. For both cases it will be checked, whether the a `std::vector<bool>` is used. The same checks will be done on parameter declarations.
Visitor: `VectorBoolSpecializationShallNotBeUsedVisitor.java`
Tests: `VectorBoolSpecializationShallNotBeUsedCheckerTest.rts`

3.20 A18-05-01: Functions malloc, calloc, realloc and free shall not be used

Rule All of the functions mentioned in the rule are used for C-style memory-allocation / -deallocation. The use of these functions is discouraged because invoking them is not type safe and the class's constructors and destructors are not invoked. There are two exceptions to this rule according to the AUTOSAR document [AUT18], one of them being the use of the functions in user-defined overloads of the new and delete operators. The other exception mentioned is the use of custom implementations of the malloc and free functions. Together with Prof. Sommerlad we have decided to not implement the second exception, because it doesn't make sense for C++ and is most likely a relic from earlier C programming guidelines.

Listing 3.39: A18-5-1 violation examples

```
1 #include <cstdint>
2 class A {
3     std::int32_t* p1 =
          static_cast<std::int32_t*>(malloc(sizeof(std::int32_t)));
          //non-compliant
4     std::int32_t* array1 = static_cast<std::int32_t*>(calloc(10,
          sizeof(std::int32_t))); // non-compliant
5     std::int32_t* array2 = static_cast<std::int32_t*>(realloc(10 *
          sizeof(std::int32_t))) // non-compliant;
6     free(p1); //non-compliant
7 }
```

Listing 3.40: A18-5-1 exception example

```
1 void operator delete(void* ptr) noexcept {
2     free(ptr); //compliant by exception
3 }
```

Analysis A function call is always an expression, so we visit all expressions and check if it is a function call expression. If that is the case we then check whether a predefined array of function-names("malloc", "std::malloc", "::malloc", "calloc", ...) contains the function call expressions name-string. To account for the exception we do the above and afterwards we check all parents up to the translation unit and check if one of them is a function definition and then check that definiton's declarator if it is equal to "operator new" or "operator delete".

Checker As mentioned above we visit all `IASTExpressions` and check if they are of type `ICPPASTFunctionCallExpression`. If this is the case we then compare the `IASTName` of the function with a predefined array of strings using the `toString()` method of the `IASTName`.

Now to account for the exception we check each `ICPPASTFunctionCallExpression`'s parents up to the `IASTTranlationUnit` and if one of them is a `ICPPASTFunctionDefinition` and if they are we check if their name is of type `IASTOperatorName` if they are we check if they override the new or the delete operator.

Visitor: `DoNotUseMallocCallocReallocFreeVisitor.java`
Tests: `DoNotUseMallocCallocReallocFreeCheckerTest.rts`

3.21 A18-09-03: The std::move shall not be used on objects declared const or const&

Rule With `std::move` the programmer is able to "push" an object into another data-structure with the original object being empty afterwards. If this function is used on const or const& objects the move-constructor is never called, instead the copy-constructor is implicitly called.

Listing 3.41: A18-9-3 violation example

```
1 #include <string>
2 #include <utility>
3 #include <vector>
4 int main() {
5     std::string const& str{"Hello"};
6     std::vector<std::string> vectorTest{};
7     vectorTest.push_back(std::move(str)); // Non-compliant
8 }
```

Analysis To check for this rule we will have to check all expressions, as `std::move` is a `ICPPASTFunctionCallExpression`. Afterwards we will check the contained `IASTIdExpression` for it's raw signature. After we have made sure, that the function call is to "std::move", we can check the second `IASTIdExpression` which is the argument. We check it's type for whether it is of type `IQualifierType` and const.

Checker The checker for this rule turned out a little more complicated than expected because first we have to check whether the used expression is `std::move`. We achieve this by first comparing the expression to `ICPPASTFunctionCallExpression`, if they match we check whether the raw signature of the function expression matches `std::move`. Once we have confirmed, that the expression is `std::move` we then have to check whether the operand is defined as const. We do this by checking every child of the base expression. To be more precise, we try to find a `IASTIdExpression` and check it's expression type with the `isConst()` function.
Visitor: `DoNotUseStdMoveOnConstObjectsVisitor.java`
Tests: `DoNotUseStdMoveOnConstObjectsCheckerTest.rts`

3.22 A26-05-01: Pseudorandom numbers shall not be generated using std::rand()

Rule This rule is required due to some implementations of `std::rand()` having comparatively short cycles, which can allow exploiters to predict the outcome of the function.

Listing 3.42: A26-5-1 violation example

```
1 #include <cstdlib>
2 int main(){
3     int randNumber{std::rand() % 100}; // Non-compliant
4 }
```

Analysis std::rand() is a `IdExpression`, which allows us to compare the raw signature to `std::rand`, as the function doesn't accept any parameters.

Checker We visit all expression and check whether they are of type `IASTIdExpression`. Once we have found one we compare it's raw signature to `std::rand` by using `getRawSignature()`.

Visitor: `DoNotUseStdRandToGenerateNumbersVisitor.java`
Tests: `DoNotUseStdRandToGenerateNumbersCheckerTest.rts`

Chapter 4

Quality Measures

4.1 Performance Tests

This section covers the different measures we took to ensure the quality of our code.
Analysis To ensure that our implemented rules, would not have a bad impact on the performance of the plug-in, we decided to execute performance tests. We used the same projects, as our predecessors [PV19], juCi++ [juC19] and LevelDB [Goo19]. With the support of Hansruedi Patzen, we were able to make peformance analysis with a professional profiler [EJT19]. We let the profiler run with following settings:

- All Autosar guidelines
- Only our implemented guidelines

The results were very satisfactory. The code analysis process for our implemented rules was even slightly faster than with the already existing rules.

4.2 Quality Tests

To ensure that our checkers work correctly we executed quality inspections. We took the same projects juCi++ [juC19] and LevelDB [Goo19] and let the AUTOSAR code analysis run only with our implemented rules activated. We checked the findings of the code analysis, if they are correct or whether they are false positives. Fortunately we have done this check, we found some false positives which we could fix.

4.2.1 A8-4-4 Correction

This rule (see 3.12) says that multiple output values from a function should be returned as a struct or tuple. Function parameters passed as reference should not be processed as side effect to allow multiple return values.
Finding In this case we had an issue with a nested lambda inside a function. The checker has not realized that a further nested scope existed inside the function. This leaded to faulty markings inside the juCi++ project. The fault was, that our method, which should find the function definition of a expression, had a small issue.

Correction We corrected the method and made the code much simpler and clearer.

Listing 4.1: A2-10-1 corrected checker code snippet

```
1 public ICPPASTFunctionDefinition checkIfParentIsFunctionDefinition(IASTNode
      node) {
2         while (!(node instanceof ICPPASTFunctionDefinition) && node != null) {
3             node = node.getParent();
4         }
5         return node != null ? (ICPPASTFunctionDefinition) node : null;
6 }
```

4.2.2 A2-10-1 Correction

The rule A2-10-1 (see 3.2) says that an identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
Finding In the juCi++ project we found a strange violation of this rule. The following function with a `void *` parameter was marked as a violation.

Listing 4.2: A2-10-1 juCi++ false positive [juC19]

```
1 void log(const char *msg, void *) {
2   std::cout << "debugger log: " << msg << std::endl;
3 }
```

This was one case, which we simply did not expect during the development of the checker.

Correction The fault was that we did not check if the parameter name is empty. The fix for this problem was really simple, we just added an additional `isEmpty()` check.

Listing 4.3: A2-10-1 corrected checker code snippet

```
1 if (!(declaration.getName().toString().isEmpty()) &&
2     /* further condition checks */ ) {
3     reportRuleForNode(declaration);
4 }
```

4.2.3 A2-10-6 Correction

The rule A2-10-6 (see 3.3) says that class or enumeration names shall not be hidden by a variable, function or enumerator declaration in the same scope.
Finding When we analyzed the results from the code analysis, we realized that the rule marks false positives. The problem was in the checker inside a `if()` statement, where the conditions for the rule violation are checked. The conditions were too complex and nested and a edge cases were not covered. Resulting from a faulty checker, constructors have been marked in some non determinstic cases.
Correction To solve this problem, we reduced the complexity of the `if` condition and took the check, if it is a constructor one level higher.

Listing 4.4: A2-10-6 corrected checker code snippet

```
1 if (!(declarator.getName().resolveBinding() instanceof ICPPConstructor)) {
2     if( /* further condition checks */ ){
3         reportRuleForNode(declaration);
4     }
5 }
```

4.3 Automated Test Compilation Checker

Motivation During a meeting with our supervisor, we were confronted with the fact that one of our tests do not compile. This was a bitter issue and we saw a need for action. So we decided to write a script which compiles all test cases. **Results** We have developed a script using the Linux scripting shell `bash`. The script writes every single test into a separate file and compiles it afterwards. In this way all implemented rules are checked for compile errors.It was frightening how many test cases don't compile. As a result, developing an automated solution in this area was the only right thing to do.
File: `/com.cevelop.codeanalysator.autosar.tests/resources/com.cevelop.codeanalysator.autosar.tests/checker/compilecheck.sh`

Chapter 5

Conclusion and Outlook

5.1 Conclusion

The goal of our thesis was to implement many new checkers and quick fixes for AUTOSAR guidelines to extend the functionality of the CodeAnalysator plug-in for Cevelop. We continued the work of our predecessors [PV19] who refactored a lot of the architecture surrounding the plug-in and had already implemented a lot of rules and more importantly for us, had already looked at all AUTOSAR guideline and started to classify each rule as well as checking whether they were a candidate for future implementation. Additionally they also created a developer guide to help future developers in finding their way around the plug-in. All of those helped us immensely as we both had no previous experience developing plug-ins.
In total we implemented 22 rules and 5 quick fixed during the course of this thesis. This number at first seemed quite low to us, but we also had to take into account the fact, that we both had not learned anything about the AST before. This led to us having to use the first weeks to get to know our way around the AST as well as getting to know our programming environment consisting of Eclipse itself and the Eclipse CDT API. After these first slow weeks we were able to implement a lot of guidelines and quick fixes and we are pretty satisfied with the quantity of checkers as well as with the performance and cleanliness of our code.

To complement our checkers and quick fixes we also implemented 238 unit tests and performed an extensive performance check using two real open-source projects. The two projects used were juCi++ [juC19] and LevelDB [Goo19]. While the performance of our checkers was very satisfying, we found some false positives during our performance checks which led us to reevalute a lot or checkers and improving those which were not quite covering the AUTOSAR guidelines.

To conclude, we have achieved the goals we set for ourselves at the start of this thesis. We implemented the amount of guidelines we wanted to and the performance and quality of our code satisfies our standards. We were able to test our work with real projects and were able to fix the checkers which did not work yet.

5.2 Outlook

There are several possibilities to extend the plug-in in future projects:

- **Implement further guidelines**: As there are still a lot of guidelines left which are not yet implemented, it is a very obvious possiblity to offer a simliar term project to this one.
- **Compare AUTOSAR and MISRA guidelines**: As of now there are two separate lists of guidelines, one consisting of all AUTOSAR guidelines, with the ones considered candidates for implementation marked, and another one which contains all the new MISRA guidelines. It would be very helpful to evaluate both lists and merge them together to get an overview of the rules which both agree on.
- **Implement further MISRA guidelines**: As of now there are very few MISRA Guidelines implemented. In a further project it would be possible to extend the functionality of the plug-in with additional MISRA guidelines. Especially when the new set of guidelines will be released in the future.

5.3 What we have learned / Tips for future projects

When looking back there were several things we could have and should have done better. Firstly we should have clarified our understanding of a few documents with our supervisor Prof. Sommerlad. For example we misunderstood the purpose of a project plan and thus had to rewrite it after sending it to Prof. Sommerlad.
Another thing we definetly should have started earlier was writing compiling unit tests. Early on we just wrote unit tests which seemed correct to us but we soon realized these were not of much use, as they would not hold up in a real compiler. To ensure we would write compiling test from the on, we created a shell script which would automatically try to compile all of our unit tests and return an error message if one did not.

Overall we should have started the testing with real projects earlier as to determine checkers which were creating markers for false positives. If our performance test had had worse results we would have been in a lot of stress towards the end of the term, as we would have to fix those performance issues of course.

In contrast there were also things we are pretty happy to have done. The biggest one of those is including the documentation of a checker or quick fix in the definition of done. This forced us to continously document everything we have done, where as otherwise the documentation tends to be forgotten about.
Another great idea was to start with only one rule and to implement it using pair-programming. This way we both got to know the environment and we were able to help and bounce ideas off eachother.

Bibliography

[AUT18] AUTOSAR. *Guidelines for the use of the C++14 Language in ciritcal and safety-related systems*. 2018.

[AUT19] AUTOSAR. *AUTOSAR Homepage*. Dec. 10, 2019. URL: https://www.autosar.org/about/ (visited on 12/10/2019).

[EJT19] EJ-Technologies. *JPROFILER*. Dec. 2, 2019. URL: https://www.ej-technologies.com/products/jprofiler/overview.html (visited on 12/02/2019).

[Gam+94] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994. ISBN: 0201633612. URL: http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1.

[Goo19] Google. *LevelDB*. Dec. 2, 2019. URL: https://github.com/google/leveldb (visited on 12/02/2019).

[IEE08] IEEE. *IEEE 754-2008 - IEEE Standard for Floating-Point Arithmetic*. Second. Aug. 2008, p. 70. ISBN: 978-0-7381-5752-8. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4610935.

[ISO17] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. Fifth. Dec. 2017, p. 1605. ISBN: ???? URL: https://www.iso.org/standard/68564.html.

[juC19] juCi++. *juCi++ - C++ IDE*. Nov. 25, 2019. URL: https://gitlab.com/cppit/jucipp (visited on 11/25/2019).

[PV19] Viktor Puselja and Gabriel Vlasek. "Safe C++ Guidelines for Cevelop (AUTOSAR)". 2019. URL: http://eprints.hsr.ch/783/.

[ref19a] C++ reference. *C++ reference - explicit keyword*. Nov. 4, 2019. URL: https://en.cppreference.com/w/cpp/language/explicit (visited on 11/04/2019).

[ref19b] C++ reference. *C++ reference - Friend*. Nov. 5, 2019. URL: https://en.cppreference.com/w/cpp/language/friend (visited on 11/05/2019).

[ref19c] C++ reference. *C++ reference - NSDMI*. Nov. 20, 2019. URL: https://en.cppreference.com/w/cpp/language/data_members (visited on 11/20/2019).

[ref19d] C++ reference. *C++ reference - PODType*. Oct. 29, 2019. URL: https://en.cppreference.com/w/cpp/named_req/PODType (visited on 10/29/2019).

[ref19e] C++ reference. *C++ reference - RAII*. Oct. 18, 2019. URL: https://en.cppreference.com/w/cpp/language/raii (visited on 10/18/2019).

[ref19f] C++ reference. *C++ reference - Variadic Arguments*. Oct. 29, 2019. URL: https://en.cppreference.com/w/cpp/language/variadic_arguments (visited on 10/29/2019).