Studien Arbeit

# Safe C++ Guidelines Checkers und Quick Fixes

written by Andreas Deicha, Pascal Bertschi

# Technical Report

supervised by

Peter Sommerlad

June 1, 2018

## Abstract

### Introduction

Cevelop is a C++ Integrated Development Environment based on Eclipse developed by IFS Institute for Software at HSR. In previous projects students designed and created various plug-ins for Cevelop with static analysis covering rules from the C++ Core Guidelines. Other guidelines share similar rules that overlap. Implementing static analysis for additional guidelines requires duplicated effort, using them causes multiple warnings on the same code and confuses C++ developers following such guidelines.

### Objective

The objective of our work is to implement a prioritization system across multiple sets of C++ Guidelines. The developer should enable one or more guidelines and their relative priority. These configurations should be available from the Eclipse preference menu. Our plug-in should make extending Cevelop with more guidelines and rules quicker and safer. Therefore our plug-in provides common infrastructure shared across guideline checkers and the ability to integrate multiple guideline's rules.

### Result

We developed an Eclipse plug-in acting as a managing instance for other C++ guideline plug-ins providing static analysis and quick-fixes.

We validated our concept by porting/implementing rules from AUTOSAR C++, MISRA C++, and the C++ Core Guidelines. The checking and fixing of eight rules was already implemented by an existing Core Guidelines plug-in and refactored to match identical rules from AUTOSAR and MISRA where applicable and to use our new infrastructure.

We implemented a guideline preference page to configure available guidelines. We ran two usability test rounds and improved our plug-in accordingly.

We provide a help page in Cevelop in assisting novice users in configuring guideline settings, also available as a separate user manual in our project report.

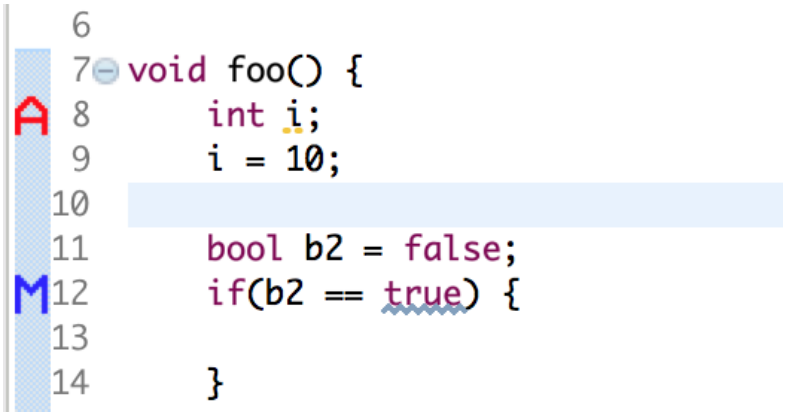


**Figure 1** Markers in code with prioritisation

We implemented a guideline preference page to configure all available guidelines. We ran two usability test rounds and improved it.

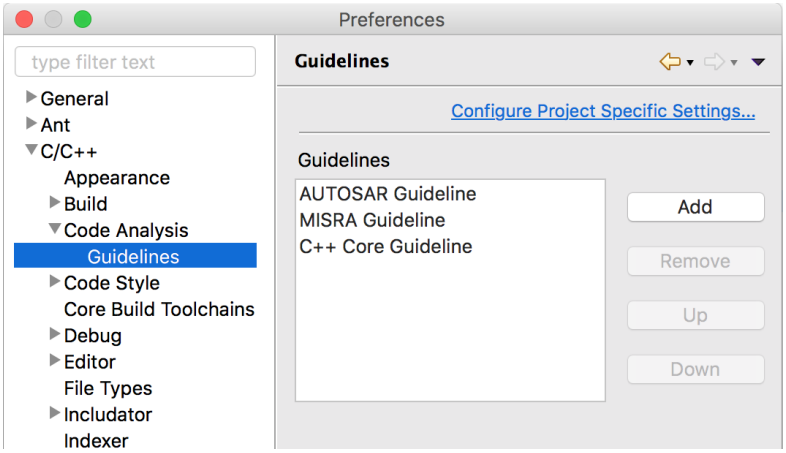


**Figure 2** Preference page

We implemented a help page in Cevelop in order to help the user to understand how to configure guideline settings.

We wrote a user and developer manual. The user manual contains the same content as the help page mentioned above and explains how to deal with our plug-in. How to add rules or a completely new plug-in is explained in the developer manual. In order to write the developer manual, we implemented a new MISRA rule used as an example to demonstrate the application of our new infrastructure.

## Management summary

Cevelop is a C++ Integrated Development Environment (IDE) based on Eclipse (Java IDE) created by IFS "Institut für Software" of HSR "Hochschule für Technik Rapperswil". In previous bachelor thesis students already designed and created various plug-ins for Cevelop to cover some C++ programming guidelines. "Cover" in this case means, when a guideline policy is violated and the plug-ins are activated, Cevelop has to warn the programmer by marking the problematic code and showing the appropriate rule with description. The guidelines we are talking about are AUTOSAR, MISRA and the C++ Core Guidelines. Each separately of these guidelines has been published and it is common that similar rules are declared by multiple guidelines. This causes multiple markers on the same C++ code violating a rule present in multiple guidelines. The markers define a problem and they are shown in a problem/warning list in the IDE. This list is very important for the programmer in order to find what part in the code produces problems/failures and is therefore essential to create clean code. Now if multiple markers for the same part exist, multiple problems get listed. This produces far too much noise and may confuse the programmer. Another issue was that most of the markers offer a quick fix. A quick fix contains the correction of a specific problem in a code. The guidelines not only marks bad code, they also offer a solution for it. Problems were not only shown multiple times for one violation, multiple quick fixes were also offered. These issues were addressed in this document. For this purpose it was planned to create another plug-in, which acts as a gateway for the guideline plug-ins. Our plug-in should manage how different guidelines handle the same problems.

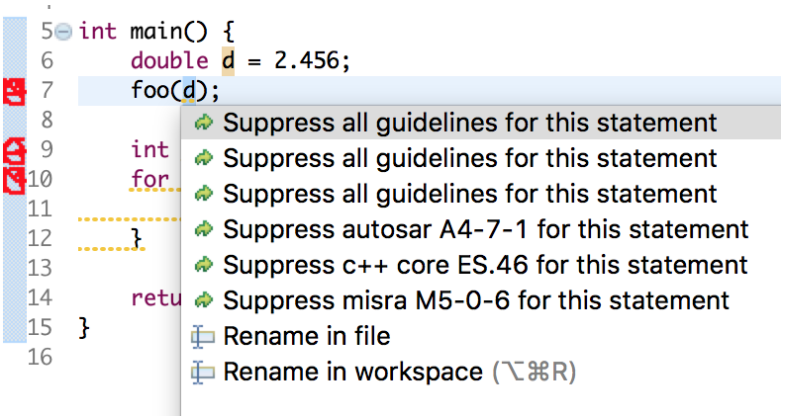


**Figure 3** Overlapping guideline rules

### Approach

Our approach can be divided into four big parts. The first quarter consisted of evaluating the architecture and to do so, we had to clarify what results we wanted to achieve. We did that by defining all functional and non-functional requirements with our supervisor. The next step was to understand how plug-ins in Cevelop work. Moreover we had to understand how problems in code get recognized and marked. We read several documentation for that and studied an older thesis where guidelines in Cevelop had been implemented. With this knowledge, we were able to create an architecture that met the requirements.

The second part consisted of implementing the planned solution. Cevelop is nothing else than a extension of Eclipse IDE. Eclipse's C/C++ development tool is the CDT project [Wika]. The CDT allows programming C/C++. Cevelop, on the other hand, extends CDT [Cev]. This means that our plug-in is basically an Eclipse extension. Eclipse is written in Java and is based on the OSGi framework "Equinox" [Wikc] since version 3.0. For code analysis, Codan is used. Codan stands for "CODe ANalysis" and is a light-weight static analysis framework that allows plug-ins to perform real time analysis on code to track down any violations of policies [Wikb].

Since our plug-in manages the guidelines, the user needs an interface where settings can be configured. Therefore, we developed a preference page. In order to check how understandable our page for the user is, we ran a usability test.

At last, we documented everything for programmers in order to later extend our plug-in or connect their guideline plug-in to ours.

### Results

The result of this work can also be divided in various partial results. On one hand we developed a logic for our plug-in that works in Eclipse and acts as a gateway or more specifically, as a managing instance for other plug-ins (guidelines).
After our supervisor approved our architecture, we implemented our logic. But in order to proof our concept, we implemented a total of eight rules of all three guidelines (AUTOSAR, MISRA, C++ Core Guidelines). The rules we used were already implemented by [RB]. We re-factored the code to match into our plug-in. Whilst doing that, we realized that in terms of performance, changes should be made to improve them. Therefore, we ran a performance analysis and improved our architecture.
Another part was the preference page for our plug-in. Once we developed it, we ran an usability test. The result showed us that there is a lack of explanation and help. To improve this, we created a help page and implemented several other changes in order to improve usability.
We then wrote a user and developer manual. The user manual contains the same content as the help page mentioned above and explains how to deal with our plug-in. How to add rules or to dock a completely new plug-in is explained in the developer manual. In order to write the developer manual, we implemented a new rule (MISRA) to show its purpose.

### Outlook

Actually, our job was to make proof of a concept. Therefore, we implemented some guidelines instead of implementing them all. A further step would be to implement the remaining rules. Furthermore, our code should be reviewed and re-factored depending on the result.

# Contents

# 1. Introduction

This work includes a concept and the correspondingly developed plug-in which addresses the problem described above in the chapter Management Summary. Subsequently, the already existing plug-ins or their overlapping guidelines should be integrated into the interface via refactoring. The guidelines should be able to be prioritized via a preference page. Therefore, a problem recognized by several guidelines is handled by the highest priority guideline. This means that the warning flag and the quick fix are displayed by the most prioritised. A special case, which also belongs to the scope, is formed in the case of a suppression annotation. If the highly prioritized guideline is suppressed by a suppression annotation, the second most important guideline that recognizes the same problem, is to be shown. Added to this are the warning logos on the various guidelines (Core, MISRA and AUTOSAR). If these tasks are solved faster than the duration of the project, the scope should be extended by implementing further guideline mappings. The scope is then summarized again in the form of a enumeration.

## 1.1. Scope Definition

Our scope for this project is divided in three sections; optimal scope, maximum scope and optional scope.

### 1.1.1. Optimal Scope

This scope defines the minimum output for our project.

- Interface for identical problems of different guidelines (concept)
- Plug-In (source code)
- Refactoring of existing guideline plug-ins towards the interface
- Preference Page for guideline prioritisation
- Correct representation of warnings and quick fixes depending on prioritisation
- Suppression-Annotation and displaying next highest prioritized guideline
- Logos for guidelines (warning-logo)

### 1.1.2. Maximum Scope

If our progress in the project goes better than expected and we have enough time left, we will extend our scope by doing the mapping of redundant rules in Core- MISRA - and AUTOSAR guideline. It isn't clear how many mappings will be done in this situation but the limitation is at the maximum of rules which are already implemented.

### 1.1.3. Optional Scope

In contrast to the other mentioned scopes, this one is optional. This means if we have spare time after finishing the maximum scope, we are obliged to continue with the optional scope but we don't guarantee to finish this scope. The optional scope consists of realizing new, by now not implemented, guidelines.

# 2. Requirement

This chapter provides the requirements which are described in user-stories.

## 2.1. Enable/Disable Guideline

The user can enable/disable different guidelines, in particular the MISRA, AUTOSAR and Core Guideline.

## 2.2. Guideline priority

It is possible for the user to prioritize his preferred guideline. This is important when two guidelines implement the same problem. In that case only the warnings and correlated quick fixes of the highest prioritized guideline should be shown.

## 2.3. Guideline suppression

If the user doesn't want to disable a guideline but still wants to hide a warning, it should be possible for him to suppress the warning. The format of the suppression annotation is: [[*guidelinename*:suppress("*guideline ID and description*")]]

## 2.4. Guideline suppression and next prioritiy

If a problem was suppressed and the same problem is recognized by the next highest prioritized guideline, then the problem/warning has to be shown with the correlated guideline message. To be more precise, if two guidelines cover the same problem and the higher prioritized guideline is suppressed, the lower prioritized guideline must then mark the problem.

## 2.5. Developer adds guideline to mapping list

The main problem that this project tries to solve, is when two different guidelines cover the same problem. Therefore a mapping is needed, meaning a list where same problems of different guidelines are indicated. This list has to be easily extendible. This means when another developer implements an additional guideline or when he implements an additional problem to an existing implementation of a guideline, he should be able to map his problem to the mapping list.

## 2.6. Nonfunctional requirements

### 2.6.1. Performance

Nowadays, every guideline has his own checker, meaning when two checkers implement the same problem, they will activate two visitors instead of one. Therefore it is an aim to find a more efficient mechanism to make the checkers only activate one visitor for the same problem.

### 2.6.2. Usability

It is very important for us to make the plug easily configurable for the user. It should be convenient to find the preference menu and the user should find the plug-in setting easily and immediately understand how to change the settings. Therefore a usability test should be performed and the UI might have to be changed.

### 2.6.3. Transferability

The plug in can be installed in 5 to 10 minutes with the manual.

### 2.6.4. Maintainability

The plug-in is programmed in a way to make it easily expandable. A developer manual will be provided.

# 3. Analysis

In this chapter we would like to analyse Eclipse and all components important for our project. In particular, all components that are important for our work are evaluated. At the end, a sequence diagram should present the logical context.

## 3.1. OSGi

The OSGI Alliance specifies a hardware independent and dynamic software platform, which makes it easy to modularize and manage your applicaton or / and service via component model (Bundle). [OSG]

### 3.1.1. Bundle

A bundle represents the implementation context of a package within the framework. The context is used to ensure method accesses from the bundle and thus to integrate the bundle into the framework. The bundles are loaded via the OSGI framework at application startup and are registered independently of each other. The Eclipse Plugin System consists of Plugin and Fragment Bundles. This loose coupling of bundles allows plugins to be extended via fragments without the original plugin knowing anything about it. [Ecle]

### 3.1.2. Extension & Extension Points

Eclipse offers an extension concept to provide functions for certain API types. The type of API is defined by the plug-in via the extension point. Extensions can be loaded via multiple plug-ins. [Voga]

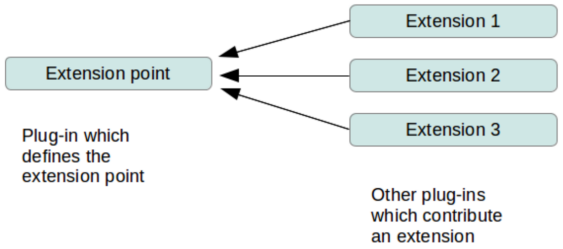


**Figure 4** Concept of Extension & Extension Points [Voga]

### 3.1.3. plugin.xml

The plugin.xml is the basis of the plugin. All extensions and extension points are defined here. When you define the extension point, an API contract is created. This allows other plug-ins to add extensions to the extension points. The plug-in is also responsible for evaluating the extensions and typically contains the code to do so. [Voga]

#### 3.1.4. fragment.xml

Sometimes it makes sense to make a part of a plug-in optional and make it de-installable and / or installable and / or updatable. A good example of this would be libraries for various operating systems or language packs. Like a plug-in expands the Eclipse framework, the fragment can be plugged into a plug-in. Unlike the plug-in, the fragment manifest must be named “fragment.xml”. The top-level element in the manifest is called fragment and has two additional attributes, plugin-id and plugin-version, which, as the name says, identify the ID and version of the plugin. Another difference to the normal plug-in is that the fragment does not need its own required elements. The fragment will automatically inherit these elements from its host plug-in. Except for these three mentioned differences, the fragment seems to be the same as the plug-in. [Eclh]

#### 3.1.5. Preference Page

To generate a preference page, we have to add an extension point in the XML plug-in. This extension points to the class that the page will create later. The preference pages are arranged in a tree structure. To place the page correctly in the UI, it must be assigned to the correct parent category. [Eclm]

### 3.2. Codan

CDT’s Code Analysis (Codan) feature assists the user by flagging possible syntax errors and other issues as problems, warnings, or not at all. [Eclg]

#### 3.2.1. CDT

CDT (9.4.3) provides a fully functional C and C++ Integrated Development Enviornment based on the Eclipse platform. [Eclf]

#### 3.2.2. Abstract Syntax Tree (AST)

The AST is the basic framework for many powerful tools in the Eclipse IDE including Refactoring, Quick Fix and Quick Assist. Simply put, the AST C++ Source Code maps into a tree structure. Each node is a subclass of ASTNode. This has the advantage that the code can be analyzed and modified more easily and reliably than with text-based source. The subclasses mentioned above are each an element of the programming language. For example, variable declaration is the subclass "VariableDeclarationFragment". For static code analysis the AST tree is traversed. During traversing, problems and improvements can be marked on individual nodes. [Ecld]

#### 3.2.3. Activator

The Activator class is the entry point for a plug-in or the plug-in itself. It is a subclass of AbstractUIPlugin (package org.eclipse.ui.plugin;). The Activator contains the start and stop method with the current context in the form of a bundle. [Eclb]

### 3.2.4. Checker

The checker is an object that expands the AST tree by multiple visitors. Visitors on the other hand deal with multiple problems (see sub chapter Visitor), which makes the checker responsible for multiple problems. [RB]

### 3.2.5. Visitor

A visitor is an object that traverse down the AST tree. If it detects a problem with a node during traversing, it creates a problem object and attaches it to it. The visitor can be responsible for several different problems. [RB]

### 3.2.6. Problem

A problem is defined by its name and unique ID. The class is called IProblem and inherits from IProblemElement. It contains a description and the marker. [RB]

### 3.2.7. Marker

A marker consists of the name and the property of whether it should be persisted. [Eclj]

### 3.2.8. AnnotationTypes

This is an extension point. It defines which category the marker is assigned to; Warning, Error or Info. [RB]

### 3.2.9. MarkerAnnotationSpecifikation

This extension point is needed to define the presentation of a marker. For example, color, label or text representations can be configured. [**?**]

### 3.2.10. MarkerResolution

To provide QuickFixes for the warnings, it is required to define marker resolutions for the problem markers. [Eclk]

### 3.2.11. QuickFix

CDT support so-called Quick Fixes. In case a marker is generated because of a warning, then a user can fix it with a click on one out of several proposed resolutions. An additional extension point needs to tells where the implementation class for the Quick Fix is.

### 3.2.12. Suppression & suppression annotation

To suppress a problem, it needs a specific Codan annotation. The annotation syntax begins with //@suppress followed by the problem name packed in brackets e.g. //@suppress("C.164: Avoid conversion operators") . It is important to make sure, that both, the code which has to be suppressed and the suppressing annotation are on the same line.

## 3.3. Workflow diagram

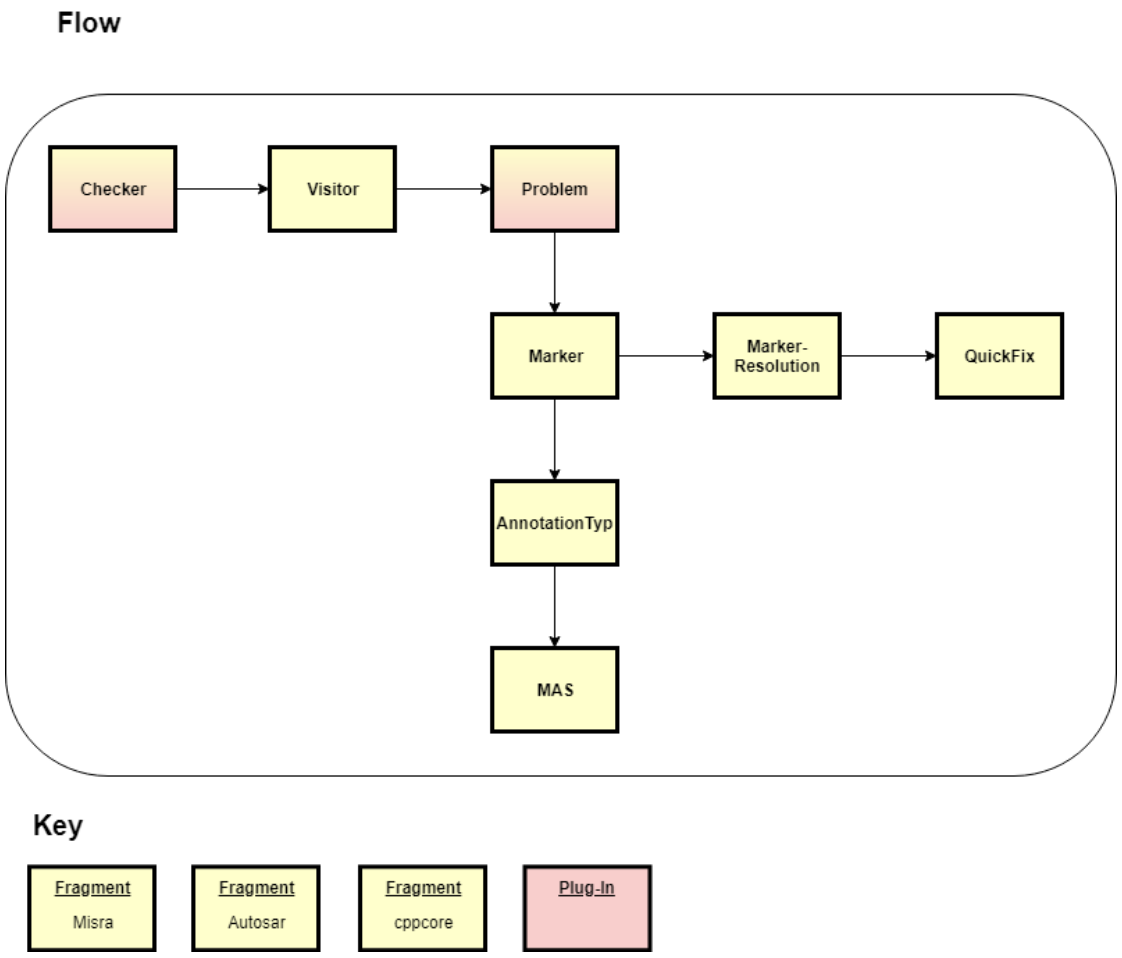


**Figure 5** Work flow diagram

# 4. Architecture

In this chapter the architecture and the correlated decisions will be presented.

## 4.1. Problem sequence diagram

In the analysis we read a lot about the Codan implementation and debugged the code to figure were we should extend our functionality. The target was that the changes on existing visitors would be small and no Codan internals had to be modified.

**Starting point** The following diagram shows a checker reporting a problem to the problem reporter. The problem reporter is an instance responsible for all problems and there markers. It detects duplicate problems and merges similar markers.

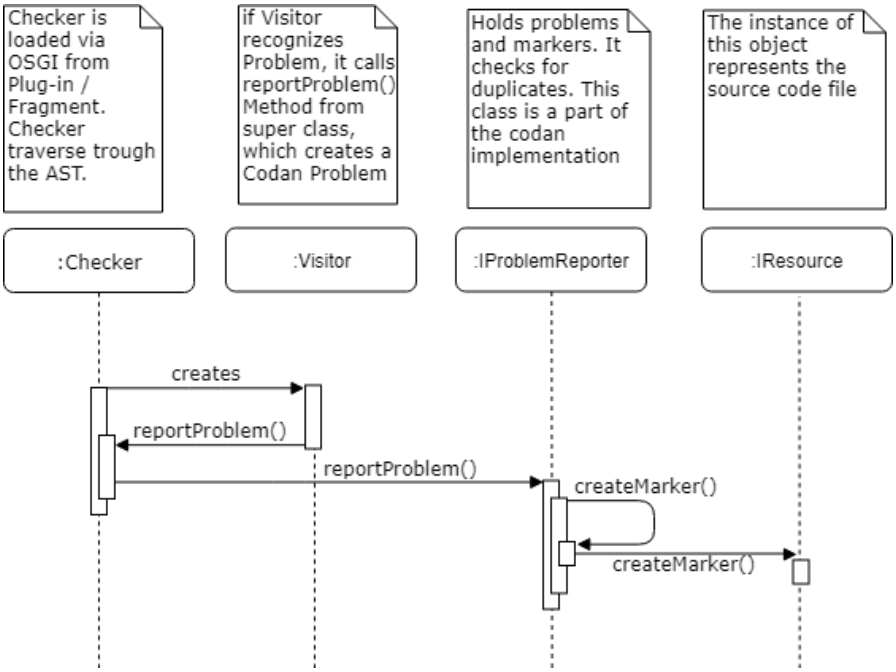


**Figure 6** sequence of a problem creation

**Solution** Our approach to solve the priority handling of guidelines with shared visitors for different guidelines, was to implement a new checker class that inherits the Codan checker and intercept the problem reporting. In the interception the available guidelines are resolved and based on the configured preferences the decision to continue or abort will be evaluated.

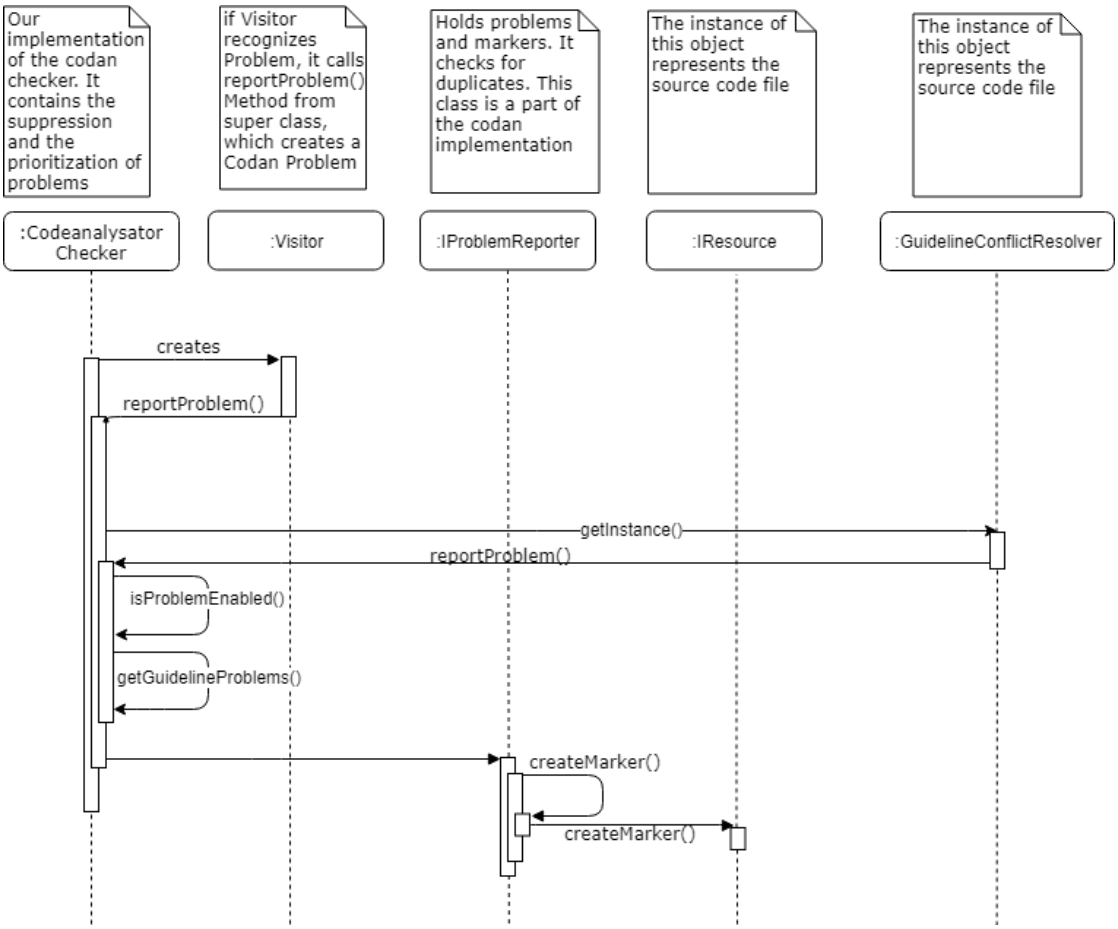


**Figure 7** sequence of problem creation with guideline suppression and prioritization

### 4.2. UML class diagram

In this sub chapter the main classes and their relationship is described. For a clearer view the diagrams are split up into smaller groups. The system architecture and the connectivity of the individual components is hard to describe, because of the given restrictions in the Eclipse framework and the way OSGI works. The components are most isolated and work independently. Because of these loosely coupling the overall program work flow is hard to describe in a single picture and we try to describe the individual parts by themselves. The reason and details about why and how we designed the architecture like it is now is mostly described in the implementation chapters 5.

#### 4.2.1. Guideline and checker

The most used class is the GuidelineConflictResolver and holds a list of guidelines. A guideline is created from a IGuidelineMapper instance that is provided by the individual fragments. The resolver instance is a singleton and is accessed by the other classes by a static instance method. The CodeAnalysatorChecker is our base implementation that uses the GuidelineConflictResolver to handle problems according to the guideline configurations. The CodeAnalysatorCompositeChecker extends those functionalities and uses a CompositeVisitor. Further details about the individual classes are described in the implementation chapter 5.

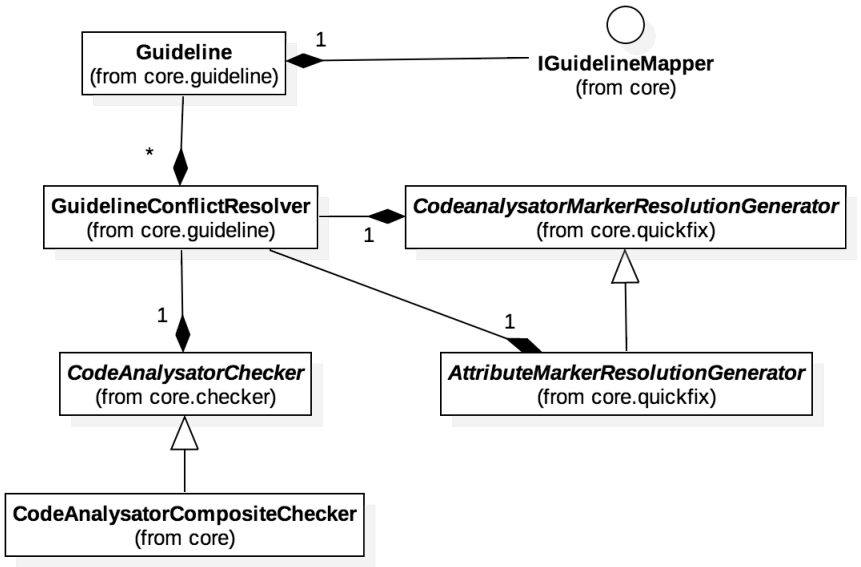


**Figure 8** Guideline and Checker

#### 4.2.2. Suppression

The ISuppressionStrategy is a interface used by the IGuidelineMapper to resolve custom suppression statements. Most guidelines use attributes to suppress code problems and therefore

we defined an AttributeSuppressionStrategy that uses SuppressionAttributes. A SuppressionAttribute holds a scope and a text value. Those can be defined by the individual guidelines or as example by the AllSuppressionAttribute. These properties define how the attribute suppression looks like in the code statement. Code examples and further implementation details are described in the implementation chapter 5.

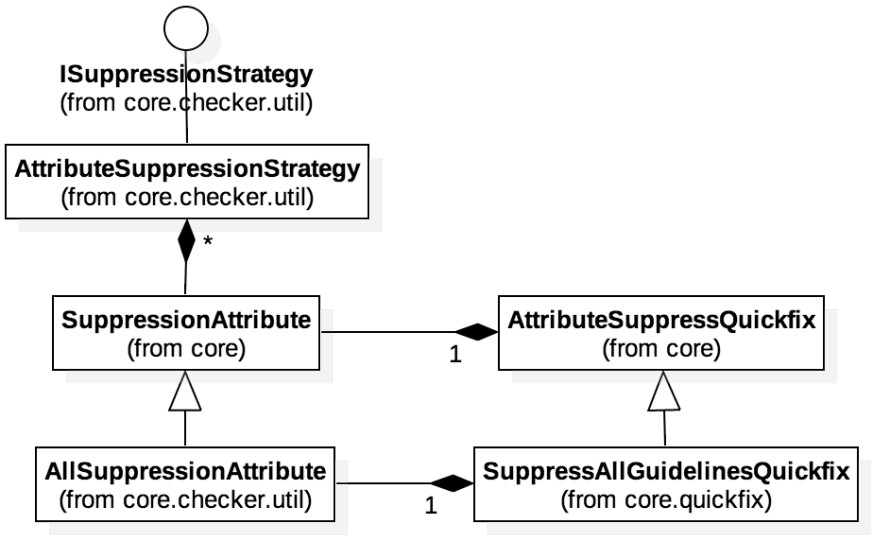


**Figure 9** Suppression

### 4.2.3. Visitor

The CodeAnalysatorVisitor is our base ASTVisitor that requires the implementing class to defined what problems they produce and what nodes they want to visit. Based on that information the checker can detect if the visitor needs to run. The VisitorComposite extends this functionality and improves performance of multiple visitors. The SharedVisitor also extends the CodeAnalysatorVisitor and implements the required methods for shared visitors to reduce boilerplate code in every shared visitor. Further details about the visitors are described in the chapters 5.2.10, 5.2.12 and 5.2.11.

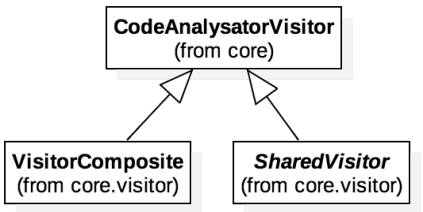


**Figure 10** Visitor

# 5. Implementation

The following chapters describe the implementation and internals of our project Code Analysator. It includes structure and flow of the system and explains how we implemented our idea described in the previous chapter on ”4 Architecture”.

## 5.1. Plug-In Structure

In order to implement our task we created a new plug-in project called ”com.cevelop.codeanalysator”. To manage the style of guideline plug-ins, we needed to put them together into the bundle of Code Analysator. Every new guideline would need to be placed in the exact same file structure. The source code of Code Analysator is located in the core folder that, on the other hand, is located in the parent folder named ”bundles”. Core also contains shared visitors. These visitors are those, covering problems that are implemented by multiple guidelines. AUTOSAR, C++ Core and MISRA contain the appropriate visitor for their style guidelines, which are only implemented by them.

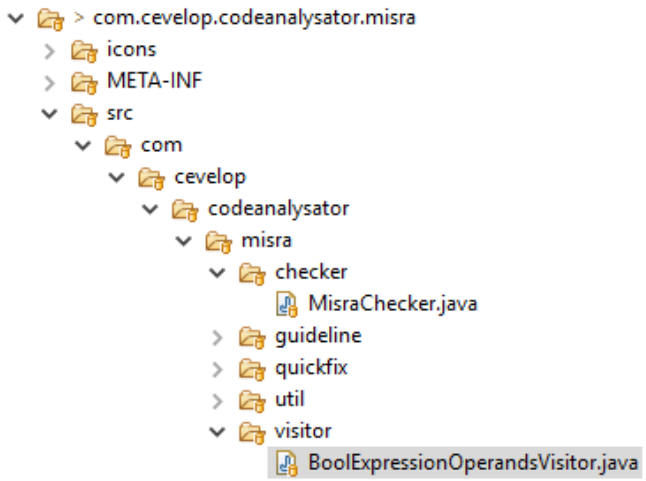


**Figure 11** plug-in file structure

## 5.2. Code Analysator Core

The Code Analysator defines the behaviour of the guidelines, declares the preference page and all its functions. It also handles visitors with shared guidelines and consists of the following classes / components:

### 5.2.1. Activator

The class "Activator" is the entry point of our plug-in and it loads the context of the guidelines with the GuidelineLoader (see sub chapter 5.2.2).

### 5.2.2. GuidelineLoader

The GuidelineLoader is responsible for registering and mounting the extensions of the guidelines. It also reads the settings on the preference page. This class contains a change listener attached to our implemented guideline preference page that resolves all changes made. The final task is to create the "GuidelineConflictResolver" (see sub chapter 5.2.3) either when Cevelop is started for the first time or when the guideline preferences are changed.

### 5.2.3. GuidelineConflictResolver

The GuidelineConflictResolver holds a list with all guidelines, their activation status (from the preference page) and their appropriate visitor and problems. It recognizes conflicts when multiple guidelines implement the same problem and then stores all information in a list sorted by priority. This information is later requested and used by the "CodeanalysatorChecker" (see subchapter 5.2.9) to decide which visitors should start.

### 5.2.4. IGuidelineMapper

Every guideline, divided in fragments, needs a guideline mapper. The Code Analysator uses the information provided by the guideline mapper to provide the list of available guidelines in the preference page and handle the priority. Implementing the "IGuidelineMapper" interface requires a guideline ID, guideline name, mapping between visitor ID and problem ID, quick fix and suppress strategy. Further details about the mapper are described in chapter B. The guideline mapper helps to reduce duplicated ID references in the plugin XML and reduces the Codan extension points to a minimum by making those references in code with constants.

The following figure 12 shows the different guideline rules and their guideline rule identification. These list shows all implemented visitors from previous theses and they refractoring status. The mapping information shown in this table should match with the implemented IGuidelineMapper classes.

| Name | AUTSAR | MISRA | C++ Core | Done |
|---|---|---|---|---|
| AvoidConversionOperators | A13-5-3 | | C.164 | Yes |
| RedundantOperations | ***A12-0-1*** | | C.20 | Yes |
| MissingSpecialMemberFunctions | ***A12-0-1*** | | C.21 | Yes |
| DestructorHasNoBody, DestructorWithMissingDeleteStatements, NoDestructor | | | C.31 | Yes |
| BaseClassDestructor | A12-4-1 | | C.35 | - |
| DestructorShouldBeNoExcept | A15-5-1 | | C.37 | - |
| NoexceptDefaultCtor | | | C.44 | - |
| InClassInitialize | ***A12-1-3, A12-7-1*** | | C.45 | - |
| DeclareSingleCtorExplicit | A12-1-4 | | C.46 | - |
| InitializeMemVarsInRightOrder | A8-5-1 | | C.47 | - |
| PreferInClassInitializerToCtors | A12-1-3 | | C.48 | - |
| NoAssignmentsInCtor | A12-6-1 | | C.49 | - |
| CopyAssignmentSignature, CopyAssignmentNonVirtual, CopyAssignmentParameterByConstRef, CopyAssignmentReturnByNonConstRef | ***A10-3-5, A13-2-1*** | | C.60 | - |
| MoveAssignmentSignature, MoveAssignmentNonVirtual, MoveAssignmentReturnByNonConstRef | ***A10-3-5, A13-2-1*** | | C.63 | - |
| MoveOperationsShouldBeNoExcept | A15-5-1 | | C.66 | - |
| ValueLikeTypesShouldHaveSwap | A12-8-2 | | C.83 | - |
| MakeSwapNoExcept | A15-5-1 | | C.84 | - |
| NamespaceLevelSwapFunction | A15-5-1 | | C.85 | - |
| AvoidALLCAPSnames | | | ES.09 | - |
| AlwaysInitializeAnObject | A8-5-0 | | ES.20 | Yes |
| DontUseVariableForTwoUnrelatedPurposes | | | ES.26 | Yes |
| AvoidLossyArithmeticConversions | A4-7-1 | M5-0-6 | ES.46 | Yes |
| IfMustUseNamedCast | A5-2-2 | | ES.49 | - |
| DontCastAwayConst | A5-2-3 | | ES.50 | - |
| DeclareLoopVariableInTheIntializer | | M3-4-1 | ES.74 | Yes |
| AvoidDoStatements | A6-5-3 | | ES.75 | - |
| AvoidGoto | A6-6-1 | | ES.76 | - |

**Figure 12** Guideline mapping table

### 5.2.5. ISuppressionStrategy

The core needs to provide a suppression strategy for the shared visitors. But the strategy can vary depending on diifferent guidelines. Therefore it needs an interface which handles specific suppression logic independent of all guidelines. In order to solve this problem we used the strategy pattern [Str]. This means that the core implementation doesn't know on which criteria the node is suppressed but can handle the logic needed to find the correct guideline.

### 5.2.6. AttributeSuppressionStrategy

The AttributeSuppressionStrategy is an implementation of the ISuppressionStrategy and handles suppression statements by C++ attributes. An attribute contains a name and an argument and is specified in code with the following syntax [[name("argument")]]. The implementation holds a list of attributes which should suppress the problems and uses them to find suppressed nodes.

### 5.2.7. CodeanalysatorMarkerResolutionGenerator

The Codan implementation loads the informations needed to suggest quick fixes for specific problems from the extension points in the plugin.xml. Another approach is to implement a MarkerResolutionGenerator dynamically resolving suggestions with quick fixes. With the implementation of the CodeanalysatorMarkerResolutionGenerator, the quick fix suggestions will be generated based on the information in the IGuidelineMapper. Because of this, the number of needed extension points can be dramatically reduced and there is no need to maintain problem IDs in multiple files. Additionally, all suggested quick fixes are filtered by their applicability. All Codan quick fixes can decide if their applicable to a specific AST node and if not, they aren't suggested in the UI menu.

### 5.2.8. AttributeMarkerResolutionGenerator

The AttributeMarkerResolutionGenerator is based on the CodeanalysatorMarkerResolutionGenerator and extends the quick fixes with the suppression statement quick fixes. The available suppression will be loaded from the IGuidelineMapper and will only suggest suppressions for the currently highest prioritized guideline.

### 5.2.9. Guideline Checker

The implementation of our CodeanalysatorChecker changes the behaviour of the inherited Codan checker sequence. As before, the visitor reports a problem to his checker and the checker hands those to the problem reporter. But before that, our implementation checks if the problem belongs to an enabled guideline. If the problem is not linked to a guideline, it gets reported. But if it is linked to a guideline and not enabled, nothing will be reported. The final step is to check if the problem belongs to a visitor that is implemented in multiple guidelines and if so, is the current guideline of the problem in first priority without counting the suppressed ones.

### 5.2.10. CodeAnalysatorVisitor

The CodeAnalysatorVisitor extends the ASTVisitor and requires the inherited class to implement the getProblemsIds method. Based on the responded problem IDs the visitor determines if the visitor is enabled. This has to be done, because we only have one checker that reports all problems with multiple visitors. If we reduce the visitors to run by checking the active problems, we can improve the checkers performance. In the chapter "5.9 Optimization" we describe the measured performance differences.

### 5.2.11. SharedVisitor

The SharedVisitor extends the CodeanalysatorVisitor and reduces the boilerplate code for visitors that are shared between multiple guidelines. Instead of creating an own visitor for each guideline with inheritance, the shared visitor accepts the guideline specific problem ID in the constructor and there is only one visitor class. The problem ID is passed in the guideline checker. The following paragraphs describe the shared visitors and the guidelines they belong to.

**Guideline mapping** In the following table, all shared visitors and their mapping to the guideline rules are listed. In some cases, one rule from a guideline can be split up into another multiple rules from another guideline .Those rules exist because the rule constraints are different the various guidelines and the rule responsibility is different

| Visitor | AUTOSAR | MISRA | C++ Core |
|---|---|---|---|
| AlwaysInitializeAnObjectVisitor | A8-5-0 | | ES.20 |
| AvoidConversionOperatorsVisitor | A13-5-3 | | C.164 |
| AvoidLossyArithmeticConversionVisitors | A4-7-1 | M5-0-6 | ES.46 |
| DeclareLoopVariableInTheInitializerVisitor | | M3-4-1 | ES.74 |
| MissingSpecialMemberFunctionsVisitor | A12-0-1 | | C.21 |
| RedundantOperatorsVisitor | A12-0-1 | | C.20 |

**AlwaysInitializeAnObjectVisitor** This visitor checks all declarations and if there is an initializer defined. If there is none, a problem gets reported. This rule is used in the AUTOSAR guideline A8-5-0 and the C++ Core Guideline ES.20. The visitor was moved from the gladiator project and integrated into our Code Analsyator Core project. The following code example is from the Core Guideline documentation [cppcoresamp] and illustrates the problem.

```
1 void use(int arg)
2 {
3     int i;  // bad: uninitialized variable
4     // ...
5     i = 7;  // initialize i
6 }
```

**AvoidConversionOperatorsVisitor** This visitor forbids the implementation of a conversion operator because in most cases the resulting behaviour is unintended. This rule is used in the AUTOSAR guideline A13-5-3 aswell as the C++ Core Guideline C.164. The visitor was moved from the gladiator project and integrated into our Code Analysator core project. The following code example is from the Core Guideline documentation [Cppb] and illustrates the problem.

```
class String { // handle ownership and access to a sequence of characters
    // ...
    String(czstring p); // copy from *p to *(this->elem)
    // ...
    operator zstring() { return elem; }
    // ...
};
```

**AvoidLossyArithmeticConversionVisitors** This visitor checks for lossy arithmetic conversions on long, double, float, int, char, signed and unsigned types. This rule is used in the AUTOSAR guideline A4-7-1, the C++ Core Guideline ES.46 and the MISRA guideline M5-0-6. The visitor was moved from the gladiator project with one modification. The visitor was configurable in the preferences, to ignore some types of conversions. To reduce the complexity of the visitor, we removed the logic and integrated the new version in our Code Analysator Core project. The following code example is from the Core Guideline documentation [Cppb] and illustrates the problem.

```
double d = 7.9;
int i = d;   // bad: narrowing: i becomes 7
i = (int) d; // bad: we're going to claim this is still not explicit enough

void f(int x, long y, double d)
{
    char c1 = x;  // bad: narrowing
    char c2 = y;  // bad: narrowing
    char c3 = d;  // bad: narrowing
}
```

**DeclareLoopVariableInTheInitializerVisitor** This visitor ensures that all loop variables are initialized in the for loop statement and are not used before or after. This rule is used in the C++ Core Guideline ES.74 and the MISRA guideline M3-4-1. This visitor was moved from the gladiator project with one modification. The original visitor marked the initializer statement in the for loop and this caused a problem with the suppression statement. There is no attribute allowed in the initializer statement. To fix this, we moved the marker to the for loop statement. The following code example is from the Core Guideline documentation [Cppb] and illustrates the problem.

```
int j; // bad: j is visible outside the loop
for (j = 0; j < 100; ++j) {
    // ...
}
// j is still visible here and isn't needed
```

**MissingSpecialMemberFunctionsVisitor** This visitor searches for special member functions which are the default, copy and move constructor, the copy assignment and move assignment operator and the destructor. This visitor was moved from the gladiator project without modification. This rule is used in the C++ Core Guideline C.21 and the AUTOSAR guideline A12-0-1. The following code example is from the Core Guideline documentation [Cppb] and illustrates the problem.

```
struct M2 {  // bad: incomplete set of default operations
public:
    // ...
    // ... no copy or move operations ...
    ~M2() { delete[] rep; }
private:
    pair<int, int>* rep; // zero-terminated set of pairs
};

void use()
{
    M2 x;
    M2 y;
    // ...
    x = y;  // the default assignment
    // ...
}
```

**RedundantOperatorsVisitor** This visitor checks for redundant constructors or operators. This rule is used in the C++ Core Guideline C.20 and the AUTOSAR guideline A12-0-1. This visitor was moved from the gladiator project without modification. The following code example is from the Core Guideline documentation [Cppb] and illustrates the problem.

```
struct Named_map {
public:
    // ... no default operations declared ...
private:
    string name;
    map<int, int> rep;
};

Named_map nm;      // default construct
Named_map nm2 {nm}; // copy construct
```

**Migration** The described visitors were moved from the gladiator source code with minimal modification. The inheritance had to be changed to the SharedVisitor class and the reported problem ID that was static before, is now passed dynamically by the constructor. During the migration, some helper functionality was moved from the visitor classes to shared static helper classes.

### 5.2.12. VisitorComposite

The VisitorComposite is our attempt to improve the checkers performance without changes to the visitors. The idea was that there is only one visitor which traverses the AST tree once, instead of multiple visitors traversing the tree on their own. This single visitor contains all other

visitors and acts as a proxy, collects the responses and returns a merged response of all visitors. The initial set up of our VisitorComposite class holds a hash map with visitor lists for each node type. During the tree traversing the map is used to access the correct visitors for each node type. Through this approach, we achieved some performance improvements as documented in the "Performance Analysis" chapter.

#### 5.2.13. Quickfixes

#### 5.2.14. AttributeSuppressQuickFix

The attribute suppress quick fix adds a C++ attribute to the reported node to suppress the problem. In case there was no attribute before, a new one will be created or the existing will be extended. Because of a bug in the ASTRewriteStore during the time of our project, we could not handle the rewrite correctly. The correct approach would be to only replace the current node, but instead we had to replace the parent node. The bug should be fixed and published soon, so the correct way could be implemented and that would improve the performance.

**DeclareLoopVariableInTheInitializerQuickFix** This is the quickfix for the DeclareLoopVariableInTheIntializerVisitor and was refactored from the gslator [RB]. Our implementation has minor changes to the original code because of the changes in the visitor we made. The marked node position changed in the visitor and therefore the quick fix had to be adjusted to that change. The quick fix also implements the *isApplicable()* method that checks if the quickfix can be applied. In this case the quick fix only works if the loop variable is not used before the loop statement. This method is validated in the CodeanalysatorMarkerResolutionGenerator **??** and filters the suggested marker resolutions according to the response.

#### 5.2.15. GuidelinePreferencePage

The GuidelinePreferencePage is attached to the Eclipse preference page under "C/C++", "Code Analysis". This page is used to enable and disable guidelines and set the desired priority. The available guidelines are dynamically loaded from the fragments. Because of our usability testing outcome (see in sub chapter C.c), we added a short cut to open the dialog via the editors context menu.

### 5.3. AUTOSAR Guideline

The AUTOSAR guideline [AUT17] is for critical and safety-related systems. This guideline is used with other guidelines to illustrate our developed logic and to handle multiple guidelines with overlapping rules.

#### 5.3.1. Implementation

The AUTOSAR GuidelineMapper implements the IGuidelineMapper and registers five visitors and their suppressions. All AUTOSAR problems reference the AUTOSAR marker that is made visible in the editor with a red A.

### 5.3.2. Suppression

The suppression of the AUTOSAR problems works by attribute and uses the AttributeSuppressionStrategy from the core implementation. The attribute suppression works with the following syntax [[autosar::suppress(”rule”)]]. The attribute can be written by hand or generated from the AutosarSuppressionQuickfix which is suggested for all AUTOSAR problems by the AutosarMarkerResolutionGenerator.

**Decision making on attributes for suppression** You might remember the chapter "3.2.12 Suppression & suppression annotation" in "Analysis" where we claimed to use line comments for suppressions. Due to MISRA and AUTOSAR not declaring a standard suppression mechanism, we, at first, thought about handling the suppressions, as mentioned, with the Codan line comments. The problem is, that the suppression annotation always has to be placed on the same code line and on bigger refactorings and these could break or move. In order to find a solution, we discussed it with our supervisor. Professor Sommerlad asked Richard Corden, a contributor to MISRA and AUTOSAR standards, if he knew more . Richard Cordan explained some details (see E-Mail traffic below) and that no standards for suppressing were defined. Professor Sommerlad then instructed us to implement suppressions with attributes. The reason for that, is that attributes are nodes in the AST, which can be easily attached or removed. It is now clear to what the suppression belongs. For instance ”int i”. Int can have an attribute and ”i” may have one. With line comment suppressions it is not clear to what element the suppression belongs. This may have impact and cause failures when reformatting the code automatically. With attribute the compiler exactly knows where to insert the suppression correctly. Due to the fact that attributes are semantic free, they also can’t cause any additional behaviour or compilation problems.

Sommerlad, Peter (2018): AW:"suppress mechanism for MISRA C++ and AUTOSAR guidelines"

> Hi Christof, hi Richard,
>
> doing a quick search, I could not find how QAC++ will need to be notified to ignore a specific guideline in an individual place. PC-Lint uses line comments for individual line warnings and configuration files or command line switches to suppress individual rules in general.
>
> What is QAC’s tools’ approach? Does MISRA C++ provide a standard suppression mechanism? Does AUTOSAR C++ GL provide a standard suppression mechanism? Core Guidelines define C++ attributes for suppression (that we already use).
>
> We have a couple of students trying to harmonize our quick fixes and thus it would be helpful, if we could generate suppression across toolings and guidelines. May be that is something to standardize within AUTOSAR and MISRA as well and C++ attributes provide a superior grammatical binding that would not fail to relate to stuff, such as comments which might be (re)moved when formatting or refactoring happens.
>
> Regards Peter.

[peter.sommerlad@hsr.ch]

Corden's answear

Corden, Richard (2018): RE:"suppress mechanism for MISRA C++ and AUTOSAR guidelines"

> Hi Peter,
>
> Regarding our approach, we have 'intrusive' suppressions that are specified using comments that list 'messages'. So there will be a small amount of work to map from a rule to the message in question. Our tools see Rules as a "display" concept only. We've also got a server based tool that allows for non-intrusive suppressions.
>
> Our simplest suppression syntax is a C or C++ style comment prefixed with "PRQA S"
>
> // PRQA S<message number>
>
> Regarding CCG, combined with "-CodeRegEx", our tools can replace specific attributes with an appropriate comment suppression:
>
> -cre 's@\[\[ *gsl *::*suppress *\(C\.133\) *\]\]@/* PRQA S 2101,1050 */@'
>
> A table of these can be auto-generated.
>
> Neither MISRA nor AUTOSAR at the moment have a standard mechanism. When this has been discussed in the past, the main issue for MISRA has been defining something that will work consistently across all tools. For example, the above suppression works for QAC++ only if the suppression is on the same line as the declaration, but it's equally valid for a tool to point to 'class A' and highlight: "this class has protected data".
>
> class A { // <— tool A issues here
> protected:
> int i [[gsl::suppress(C.133)]] ; // <— tool B issues here };
>
> It's a "sensitive thing" for MISRA to specify something that would cause a good tool to perform "worse" because of a location mismatch. The other point is that some vendors (such as ourselves) provide for external tools that allow for non-intrusive suppression support. This allows for 'sign-off' and notes. Often in domains that are heavy with process, such as automotive, they want to avoid changing code at all costs and they need the traceability for "deviations".
>
> Complete aside, one observation I had with using attributes, is that it feels very intrusive. For this case the attribute must appear somewhere with the declaration specifier and code with more than one or two of these would become ugly quickly. I suppose that this is both a blessing, in that people won't want to add them, but a curse as when they need to it will look horrible.
>
> Cheers,
>
> Richard

[richard_corden@prqa.com]

### 5.3.3. Visitors

For the AUTOSAR guideline there are five implemented visitors. Those are all shared visitors that are located in the core implementation. There are no specific AUTOSAR visitors implemented. All visitiors are described in 5.2.11 section of the core implementation under the paragraphs AlwaysInitializeAnObjectVisitor, AvoidLossyConversionsVisitor, AvoidConversionOperatorsVisitor, RedundantOperatorsVisitor and MissingSpecialMemberFunctionsVisitor.

## 5.4. MISRA

Motor Industry Software Reliability Association (MISRA) defines standards for the software development used in automotive industry [wikd]. Besides the shared visitors, that we have taken from [RB], we implemented an additional visitor which is only implemented by MISRA (but not by AUTOSAR or C++ Core).

### 5.4.1. MisraGuidelineMapper

For MISRA we have implemented three different visitors. Those visitors will be explained more specific in the sub chapter 5.4.3 "Visitor". The MisraGuidelineMapper maps the problem to the appropriate visitor to make it recognizable if one problem is covered by two guidelines. The suppression strategy gets connected as well. For further explanation on how the GuidelineMapper works, please read about it in Developer Manual chapter "B.b.2 Guideline Mapper".

```
1   private AttributeSuppressionStrategy suppressionStrategy = new AttributeSuppressionStrategy();
2   private HashMap<String, String> mappings = new HashMap<String, String>();
3   private HashMap<String, IMarkerResolution[]> quickfixes = new HashMap<String,
        IMarkerResolution[]>();
4
5   public MisraGuidelineMapper() {
6      mappings.put(CoreIdHelper.DeclareLoopVariableInTheIntializerVisitorId,
           MisraIdHelper.DeclareLoopVariableInTheIntializerProblemId);
7      mappings.put(CoreIdHelper.AvoidLossyConversionsVisitorId,
           MisraIdHelper.AvoidLossyConversionsProblemId);
8      mappings.put(MisraIdHelper.BoolExpressionOperandsVisitorId,
           MisraIdHelper.BoolExpressionOperandsProblemId);
9
10     suppressionStrategy.addSuppression(CoreIdHelper.DeclareLoopVariableInTheIntializerVisitorId,
           new MisraSuppressionAttribute("M3-4-1"));
11     suppressionStrategy.addSuppression(CoreIdHelper.AvoidLossyConversionsVisitorId, new
           MisraSuppressionAttribute("M5-0-6"));
12     suppressionStrategy.addSuppression(MisraIdHelper.BoolExpressionOperandsVisitorId, new
           MisraSuppressionAttribute("M4-5-1"));
13
14     quickfixes.put(MisraIdHelper.DeclareLoopVariableInTheIntializerProblemId, new
           IMarkerResolution[] { new DeclareLoopVariableInTheInitializerQuickFix("ES.74: Add a
           variable declaration") });
15  }
```

### 5.4.2. Suppression

The suppression implementation follows the same structure as the suppression for AUTOSAR explained in "5.3.2 Suppression". The suppression attribute annotation is the only thing that changes to:

[[misra::suppress("rule")]].

### 5.4.3. Visitors

Three visitors were implemented for MISRA guideline "M3-4-1: DeclareLoopVariableInTheIntializerVisitor", "M5-6-0: AvoidLossyConversionVisitor" and "M4-5-1: BoolExpressionOperands". The first two were already implemented by [RB] and only refactored. Both rules are shared rules, meaning they are also checked by AUTOSAR or C++Core Guideline. The last one was implemented by us and is only covered in the MISRA guideline. You will find a description for "M3-4-1: DeclareLoopVariableInTheIntializerVisitor" in chapter 5.2.11 and for "M5-6-0: AvoidLossyArithmeticConversionVisitor", check chapter 5.2.11.

**M4-5-1: BoolExpressionOperandsVisitor** MISRA describes their guideline as followed

> Expressions with type *bool* shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator. [A$^{+}$08]

Here is an example where a binary expression should fail (at line 5)

```
1 int main() {
2     bool b1 = true;
3     bool b2 = false;
4
5     if(b1 & b2){} //bad operator
6 }
```

Another example wehre the visitor should trigger is the following unary expression (at line 4)

```
1 int main() {
2     bool b1 = true;
3
4     if(~b1){} //bad operator
5 }
```

In order to implement the the "BoolExpressionOperandsVisitor" we need the methods *setShouldVisit()*, *visit()* and *getProblemIds()* from the super class "CodeAnalysatorVisitor". In *setShouldVisit()* we define what node should be checked. In our case, it's "shouldVisitExpressions"..

```
1     @Override
2     protected void setShouldVisit() {
3         this.shouldVisitExpressions = true;
4     }
```

Our *visit()* method checks whether a guideline is violated or not. Therefore, it needs to differ between two expressions. The first ones are Binary Expression, which are two operands seperated by one operator [IBMa]. The second ones are Unary Expression where one operator only acts for one operand [MSD].

```
1    @Override
2    public int visit(IASTExpression expression) {
3        if (expression instanceof ICPPASTBinaryExpression) {
4            handleBinaryExpression((ICPPASTBinaryExpression)expression);
5        } else if (expression instanceof ICPPASTUnaryExpression){
6            handleUnaryExpression((ICPPASTUnaryExpression)expression);
7        }
```

If the current node that is checked in the AST tree is a Binary Expression, following steps will be done. First, the operator needs to be checked. Logical AND (==) and logical OR (!=) are always legit and therefore if the operator is one of them, the verification can be aborted and the next node can be checked.

```
1        if (operator == IASTBinaryExpression.op_logicalAnd || operator ==
             IASTBinaryExpression.op_logicalOr) {
2            return;
3        }
```

The next thing to check is the type of the operands. At least one operand has to be a boolean.

```
1        boolean isOperand1Bool = isBooleanType(expression.getOperand1());
2        boolean isOperand2Bool = isBooleanType(expression.getOperand2());
3
4        if (!isOperand1Bool && !isOperand2Bool) {
5           return;
6        }
```

It's almost the same if the expression is unary. First we check the operator and then, instead of checking two operands, we do only one.

```
1        if (operator == ICPPASTUnaryExpression.op_star ||
2                operator == ICPPASTUnaryExpression.op_amper ||
3                operator == ICPPASTUnaryExpression.op_not ||
4                operator == ICPPASTUnaryExpression.op_bracketedPrimary ||
5                operator == ICPPASTUnaryExpression.op_throw ||
6                operator == ICPPASTUnaryExpression.op_typeid ||
7                operator == ICPPASTUnaryExpression.op_sizeof ||
8                operator == ICPPASTUnaryExpression.op_alignOf ||
9                operator == ICPPASTUnaryExpression.op_noexcept){
10           return;
11       }
12
13       if (isBooleanType(expression.getOperand())) {
14        checker.reportProblem(MisraIdHelper.BoolExpressionOperandsProblemId, expression);
15     }
```

The *getProblemIds()* method returns the problem ID which is handled by this visitor combined with the guideline ID. Code Analysator needs this value to compare with the preference page whether this guideline is activated or not and how high the prioritization is.

## 5.5. C++ Core

The C++ Core Guideline is aimed for people who use modern C++.

> The guidelines are focused on relatively higher-level issues, such as interfaces, resource management, memory management, and concurrency. Such rules affect application architecture and library design. Following the rules will lead to code that is statically type-safe, has no resource leaks, and catches many more programming logic errors than is common in code today. And it will run fast – you can afford to do things right. [Cppa]

### 5.5.1. Suppression

The suppression implementation follows the same structure as the suppression for AUTSAR explained in "5.3.2 Suppression". Only the suppression attribute annotation changes to

[[gsl::suppress("rule")]].

### 5.5.2. Visitors

For the C++ Core Guideline we refactored eight visitors in total from [RB]. Six of them are shared visitors and two are specific visitors. The shared ones are:

- Chapter 5.2.11 - "C.164: AvoidConversionOperators"
- Chapter 5.2.11 - "C.20: RedundantOperations"
- Chapter 5.2.11 - "C.21: MissingSpecialMemberFunctions"
- Chapter 5.2.11 - "ES.20: AlwaysInitializeAnObject"
- Chapter 5.2.11 - "ES.46: AvoidLossyArithmeticConversions"
- Chapter 5.2.11 - "ES.74: DeclareLoopVariableInTheIntializer"

Then, there are three rules that are packed in one rule C.31 because they go hand in hand. They are ”DestructorHasNoBody”, ”DestructorWithMissingDeleteStatements” and ”NoDestructor”. C31 is a specific rule for C++ Core Guideline, meaning it is not covered by AUTOSAR or MISRA. General rule C.31 means all resources acquired by a class must be released by the class's destructor. The reason for this is to prevent resource leaks, especially in error cases. All three rules will be shown in detail but let's give another concrete example. In the snippet below you will see that X may leak a file handle. [Cppc]

```
1 class X{
2     FILE* f; //may own a file
3     // ... no default operations defined or =deleted ...
4 };
```

The last specific rule and therefore visitor is "ES.26: DontUseVariableForTwoUnrelatedPurposes".

**NoDestructorVisitor** In this case there is no destructor at all. The marker would pop up at line 1.

```
1 struct Named_map { //bad
2 public:
3     // No special functions
4
5 private:
6     std::string name;
7     std::map<int, int> rep;
8     gsl::owner<int *> owy;
9 };
```

**DestructorWithMissingDeleteStatementVisitor** In order to release all resources the destructor should call delete on all his local variables in order to reclaim heap storage. In this case, the delete statement is missing. Line 3 should be marked.

```
1  struct Named_map {
2  public:
3      ~Named_map(){//bad
4
5      }
6
7  private:
8      std::string name;
9      std::map<int, int> rep;
10     gsl::owner<int *> owy;
11 };
```

**DestructorHasNoBodyVisitor** In order to reclaim the heap storage the destructor should call delete on every member object. To declare the destructor default is the same as an empty body, which isn't considered again.

```
1 struct Named_map {
2 public:
3     ~Named_map() = default; //bad
4
5 private:
6     std::string name;
7     std::map<int, int> rep;
8     gsl::owner<int *> owy;
9 };
```

**DontUseVariableForTwoUnrelatedPurposerVisitor** This rule means you shouldn't use a local variable for two unrelated purposes because it impairs the readability and safety . In this example, the bad code is at line 4. [Cppc]

```
1 void function() {
2     int i = 1;
3     i = 2;
4     i = 3;
5 }
```

## 5.6. Preference Page

This sub-chapter explains the implementation of the preference page. In our case, we used a ILTIS [Sta] class for the preference page. If you don't have access to the ILTIS project or if it doesn't cover your requirements, you could access further explanation through vogella's tutorial [Vogb].

**Extension** To add a page to a preference dialog, a plug-in must provide a contribution to the "org.eclipse.ui.preferencePages" extension point. This extension point is needed in order to define a class which creates a user interface and stores the preference values. [Vogb]

```
1   <extension
2      point="org.eclipse.ui.preferencePages">
3       <page
4            name="Guidelines"
5            category="org.eclipse.cdt.codan.ui.preferences.CodanPreferencePage"
6            class="com.cevelop.codeanalysator.core.preference.GuidelinePreferencePage"
7            id="com.cevelop.codeanalysator.core.preference.guideline.page">
8       </page>
9    </extension>
```

We assigned the category of the preference page (in this case the CodanPreferencePage) in order to declare where to position the page in the preference dialog.

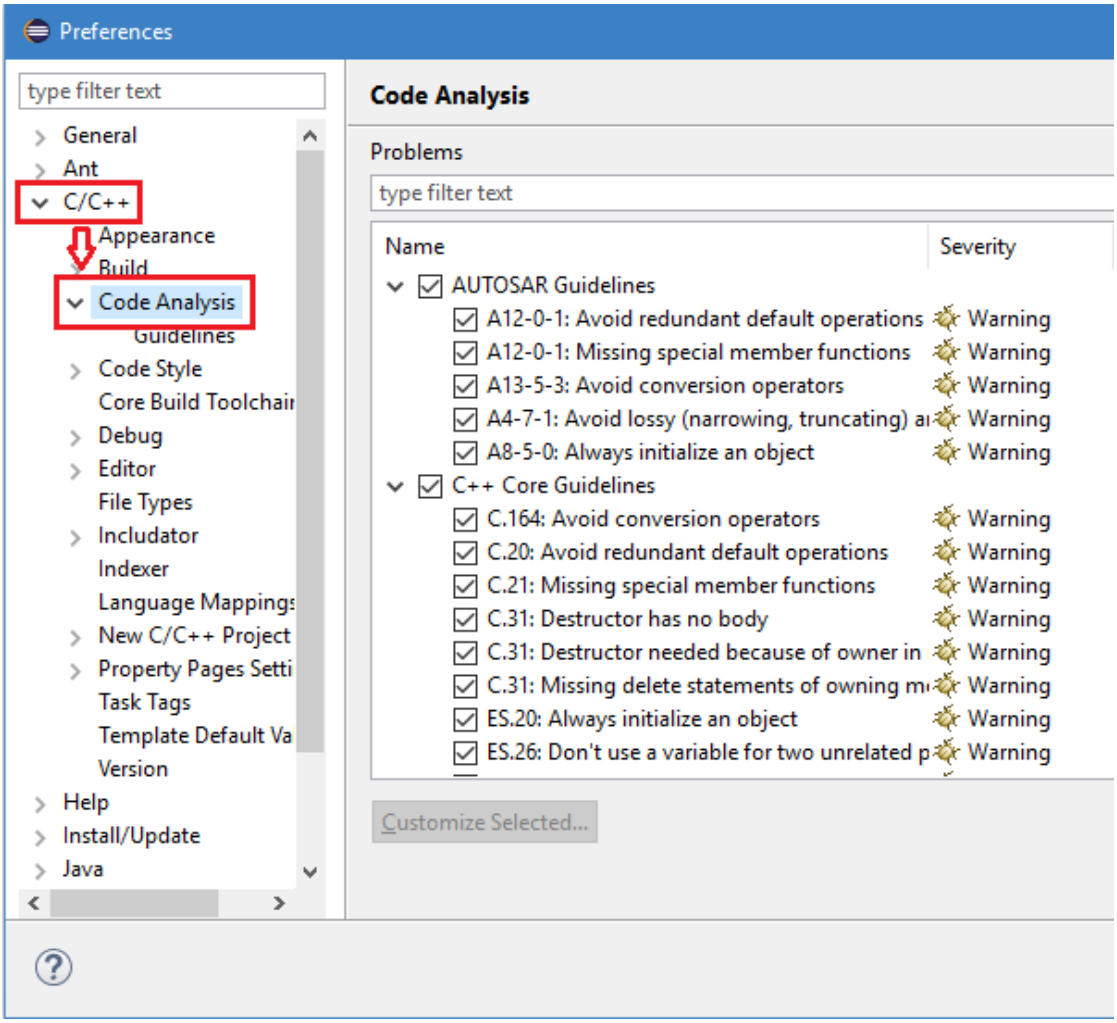


**Figure 13** Codan category for preference page

This pointed out where our preference page class (*GuidelinePreferencePage.java*) is stored.Finally, an ID attribute had to be declared in order to define search terms for the Eclipse IDE search field for preferences. [Vogb]

**GuidelinePreferencePage** For the preference page you can use the class ”PreferencePage” or extend one of its subclasses. A good subclass would be "FieldEditorPreferencePage". Your preference page class also has to implement ”IWorkbenchPreferencePage” and must have a non-parameter constructor. [Vogb] In our case, we inherited from ILTIS project ”CFieldEditorPropertyAndPreferencePage”. This class met all requirements and fulfilled all of our requirements to our preference page. The associated package is ”ch.hsr.ifs.iltis.cpp.core.preferences”.

```
1  public class GuidelinePreferencePage extends CFieldEditorPropertyAndPreferencePage {
2     public GuidelinePreferencePage() {
3        super(GRID);
4     }
5
6     @Override
7     public void createFieldEditors() {
8        addField(new GuidelineListEditor(CoreIdHelper.GUIDELINE_SETTING_ID,
               TranslationHelper.get("preferences.guidelineList"), getFieldEditorParent()));
9     }
10
11    @Override
12    protected String getPageId() {
13       return CoreIdHelper.GUIDELINE_SETTING_PAGE_ID;
14    }
15
16    @Override
17    protected IPropertyAndPreferenceHelper createPropertyAndPreferenceHelper() {
18       return PropAndPrefHelper.getInstance();
19    }
20 }
```

## 5.7. Context menu

After our first usability test, we realized that some people had problems finding the preference page. Therefore, we wrote an entry into context menu where the guideline preference page can be opened (see chapter "C.c Outcome Part 1"). In order to write this entry, we needed to add three extensions to our ”plugin.xml”.

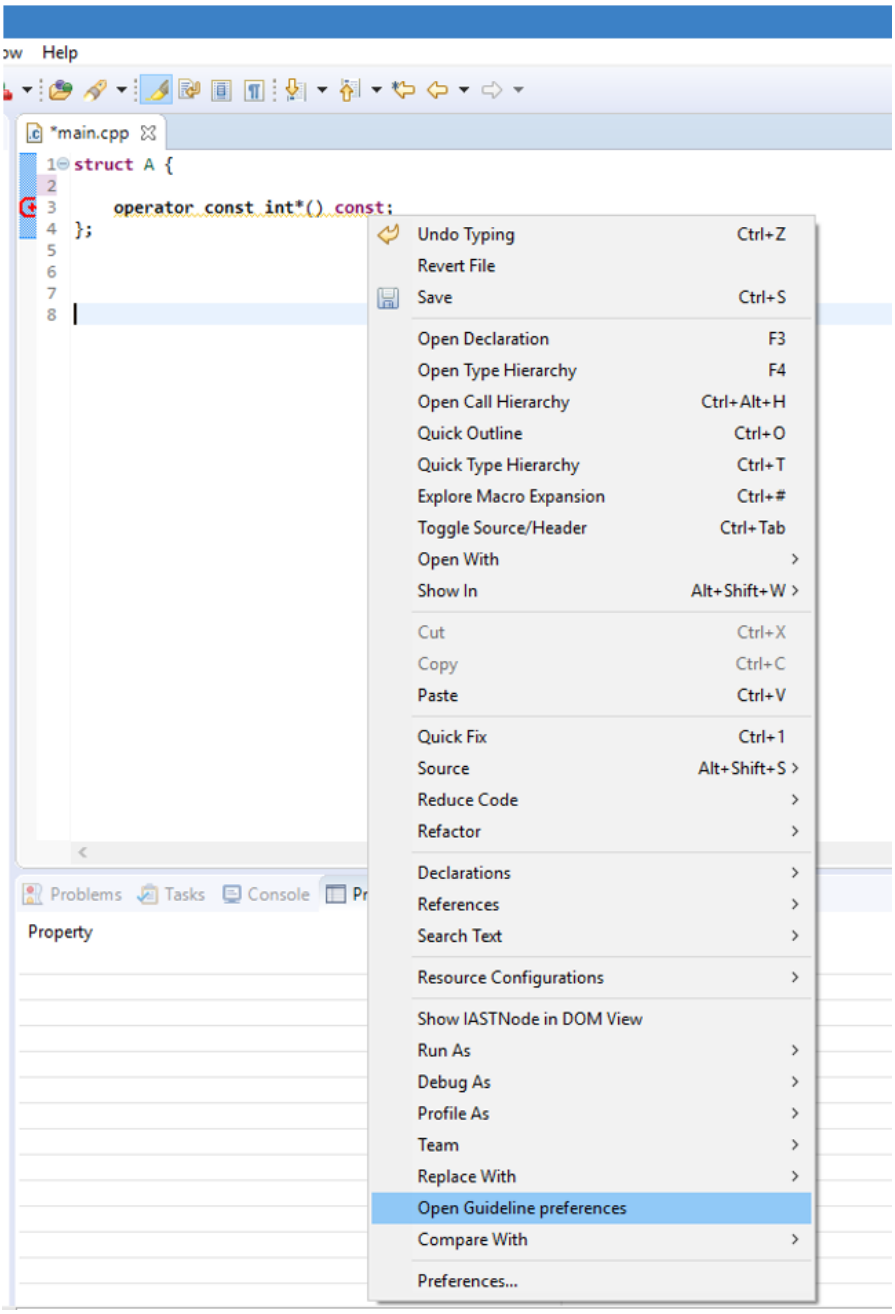


**Figure 14** Open preference page over context menu entry

**Command** First of all we needed a command that is triggered by clicking on our context menu entry. Therefore, we added the ”org.eclipse.ui.commands” extension. Since all commands have to be grouped in categories, we created a new category and assigned our command to it. [IBMb] It is important to give the command an ID in order to recognize and handle the command.

```
<extension
   name="extensionMenuCommands.name"
    point="org.eclipse.ui.commands">
     <category
           name="ActionsCategory"
           id="com.cevelop.codeanalysator.menu.command.actionscategory">
     </category>
     <command
           categoryId="com.cevelop.codeanalysator.menu.command.actionscategory"
           id="com.cevelop.codeanalysator.menu.command.guidelineOpenCommand"
           name="Open Guidelines">
     </command>
 </extension>
```

**HanIder** The handler decides what happens when the command shows up. Therefore, the extension ”org.eclipse.ui.handlers” was added. We assigned the handler to our command and defined what class to use when the command is executed. [IBMb]

```
<extension
     name="Handlers"
     point="org.eclipse.ui.handlers">
    <handler
        commandId="com.cevelop.codeanalysator.menu.command.guidelineOpenCommand"
        class="com.cevelop.codeanalysator.core.handler.GuidelineOpener"/>
</extension>
```

It is important to note that this class inherits from "AbstractHAndler" and overwrites the *execute()* method. The *execute()* method calls the *createPreferenceDialogOn()* method on a "PreferencesUtil" object with four arguments.
It tells the Shell to parent the dialog off of if it, if it has not already been created. This may be <code>null</code> and in that case, the active workbench window will be used if available.

It commands the identifier of the preference page to open; may be <code>null</code>. If it is <code>null</code>, then the preference page is not selected or modified in any way. This ID was defined in chapter 5.6 paragraph "Extension".

The IDs of the other pages will be displayed using the same filtering criteria to search. If this is <code>null</code> then the "all preference pages" are shown.

And finally, data that will be passed to all of the preference pages to be applied as specified within the page while they are created. If the data is <code>null</code> nothing will be called.

**menuContribution** This extensions adds the entry into the right place in the UI. The extension to extend is ”org.eclipse.ui.menus”. We added a menu contribution, and defined the location through the ”locationURI” attribute. Then, we assigned a command to this entry and therefore, we assigned the correct command ID defined in the ”Command” paragraph above.

```
<extension
     name="extensionMenu.name"
     point="org.eclipse.ui.menus">
    <menuContribution
        allPopups="true"
        locationURI="popup:org.eclipse.ui.popup.any?after=additions">
       <command
            commandId="com.cevelop.codeanalysator.menu.command.guidelineOpenCommand"
            label="Open Guideline preferences"
            style="push"/>
    </menuContribution>
</extension>
```

## 5.8. Help side

In order to improve the usability even more, we also created a help side which helps people to understand how to use the preference page. The help side can be reached over pressing ”F1”. The following was written by Eclipse, explaining their help system: [Eclo]

> The Eclipse Platform's help system allows you to contribute your plug-in's online help using the org.eclipse.help.toc extension point. You can either contribute the online help as part of your code plug-in or provide it separately in its own documentation plug-in. This separation is beneficial in those situations where the code and documentation teams are different groups or where you want to reduce the coupling/dependency between the documentation and code. The Platform's help facilities provide you with the raw building blocks to structure and contribute your help without dictating the structure or granularity of documentation. The Platform does however provide and control the integrated help viewers thus ensuring seamless integration of your documentation.
>
> The org.eclipse.help.toc contribution specifies one or more associated XML files that contain the structure of your help and its integration with help contributed by other plug-ins. In the remainder of this article we will create a documentation plug-in, so by the time you're done you can browse your own documentation using the Eclipse Help System.

We decided to make an own plug-in to help reduce the coupling between the documentation and code. The content of the help is the User manual (see chapter "A User manual") converted to html.

**plug-in.xml** We added our TOC (tabel of content) to the ”org.eclipse.help.toc” extension and defined the xml file. The ”primary” attribute specifies whether the TOC file is a primary table of contents and is meant to be the master, not primary. It is intended to be integrated into another table of contents. We got only one TOC and therefore, the ”primary” attribute has to be true. [Eclp]

```
  <extension
       point="org.eclipse.help.toc">
    <toc
         file="toc.xml"
         primary="true">
    </toc>
```

```
7    </extension>
```

Our TOC file defines the label text

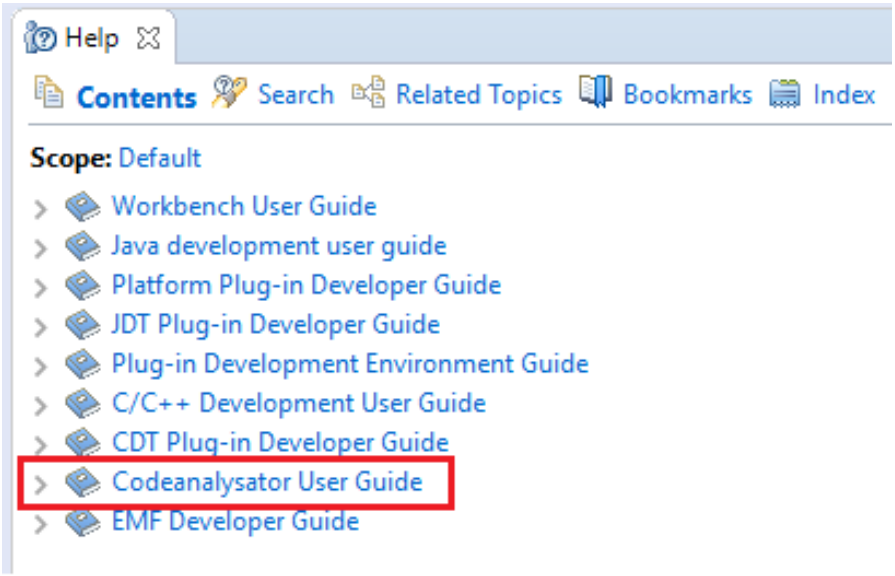


**Figure 15** Label name of help entry

Then we defined the topic name


**Figure 16** Topic name

And as a final step, we linked our html page with an ”href”-tag.

## 5.9. Optimization

For the described VisitorComposite class, we measured the checkers execution time to evaluate our optimization improvements.

### 5.9.1. Performance analysis

Due to having some extra time, we had time to analyse the performance of the plug-in Code Analysator. This chapter will describe the measurements and outcomes.

### 5.9.2. Performance measurements

For the performance measurement we took the fish shell source code in our testing Cevelop instance and measured the checker execution time to process the AST of two different source files. For the test case, we used the C++ Core Guideline checker with 9 visitors. We then measured the execution time for each file multiple times and reset the testing environment after each test. The measured execution time is always a little bit higher in the first few runs, which we think happens due to caching in the AST parsing.

The first test case is with the exec.cpp file which contains 1242 lines of code. In each run, the visitor checked 5605 AST nodes. The average execution time of the composite visitor is 524 ms and 515 ms without. For this test case, the composite visitor is slower because of the additional set up cost and the overhead while visiting the nodes.

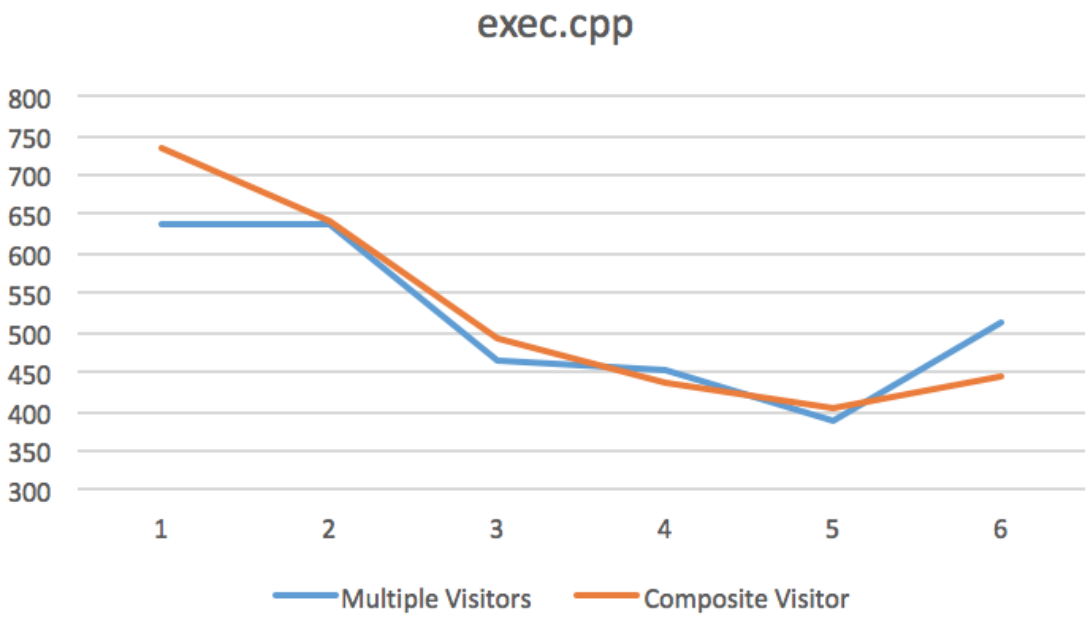


**Figure 17** Performance measurements of the exec.cpp file in the fish shell code [time in ms]

The second test case is the the reader.cpp file which contains 3373 lines of code. In each run, the visitor checked 16570 AST nodes. The average execution time of the composite visitor is 2773 ms and 2601 ms without. For this test case, the composite visitor is slightly faster, because the improvements during the visits overweight the additional set up cost.

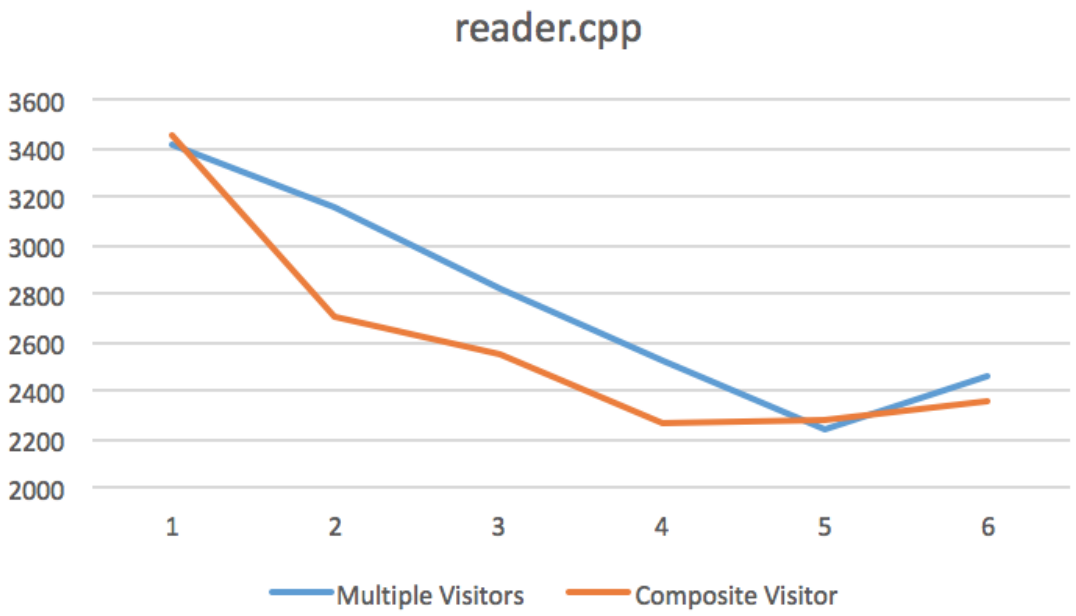


**Figure 18** Performance measurements of the reader.cpp file in the fish shell code [time in ms]

# 6. Conclusion

The last chapter discusses what results were achieved and what future work is pending for future thesis.

## 6.1. Results

Through this project we achieved several goals. All three scopes were achieved. 0ptimal scope to maximum scope and then up to optional scope. This includes creating a concept to solve conflicts between multiple implemented guidelines in MISRA, AUTOSAR and C++ Core Guidelines. The concept handles conflicts between multiple guidelines, prioritization of guidelines, suppression of guidelines and quick fixes for problems.

Another part was to program the concept. To proof the concept, we refactored eight existing guidelines from MISRA, AUTOSAR and C++ Core Guidelines. Some of them were not only refactored to match to our plug-in but also code improvement per se. We also implemented a completely new MISRA guideline with all appropriate tests needed.

Moreover, we created a preference page in Cevelop to configure and manage the guidelines (enable/disable and prioritization). The icon for the guideline (when popping up) was also designed by us.

For improvement reasons, we ran a usability test that examined the user behavior and improved our weaknesses. One of the larger weaknesses was that people couldn't find the preference page of our Code Analysator. Therefore, we implemented an entry in the context menu (right click on marked code) where the preference page can be opened directly. Another weakness was that we hadn't developed a help page where users can read how to use the Code Analysator.

One additional /not in the scope considered) point we achieved is performance analysis. We measured our solution and improved the code to make it much faster.

## 6.2. Future improvements

As far as we can tell, the Code Analysator should be stable and ready for operation. The only thing we're unsure about is the performance. There is potential to make it run faster but due to predetermined code of Eclipse and Codan, we couldn't improve the speed much. Lastly, all rules of MISRA, AUTOSAR and C++ Core Guidelines should be mapped to the Code Analysator and further new guidelines could be implemented.

# A. User manual

In this section you will find a quick overview how to enable or disable, configure and use the Code Analysator plug-in. After installing the plug-in, keep in mind that the plug-in is disabled by default.

## A.a. How to install Code Analysator

In order to install Code Analysator you need the Code Analysator source code or the repository. You will find it on the delivery stick, it is called "repository". Unzip the folder, then open a Cevelop instance and click on the "Help" tab. Choose "Install New Software"

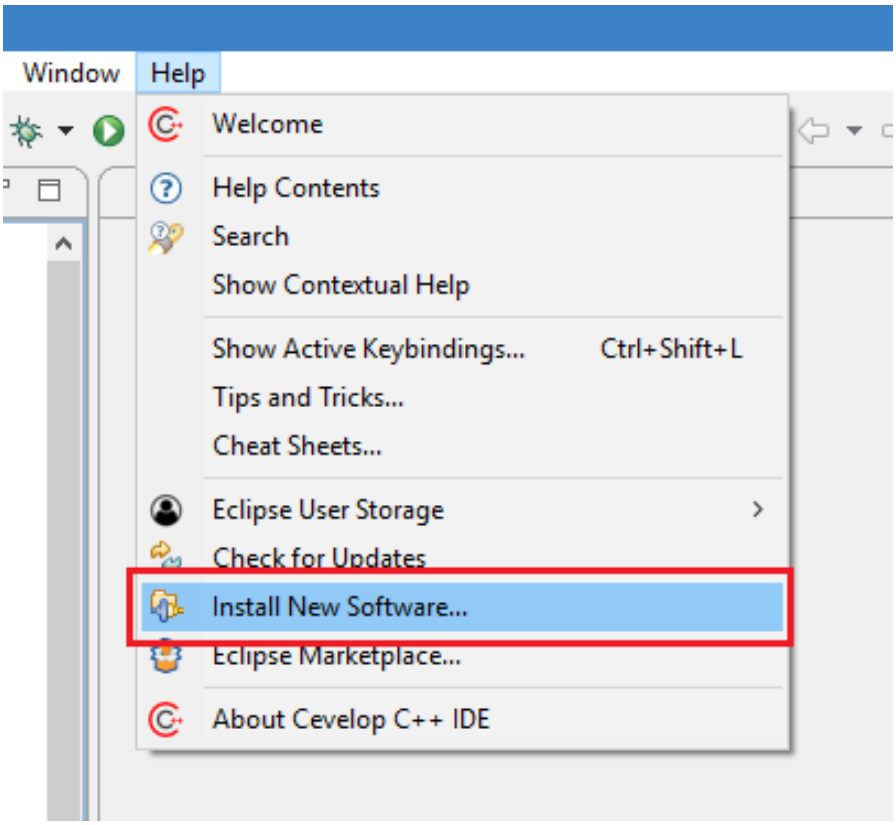


**Figure 19** Install New Software in Cevelop

Click on "Manage"

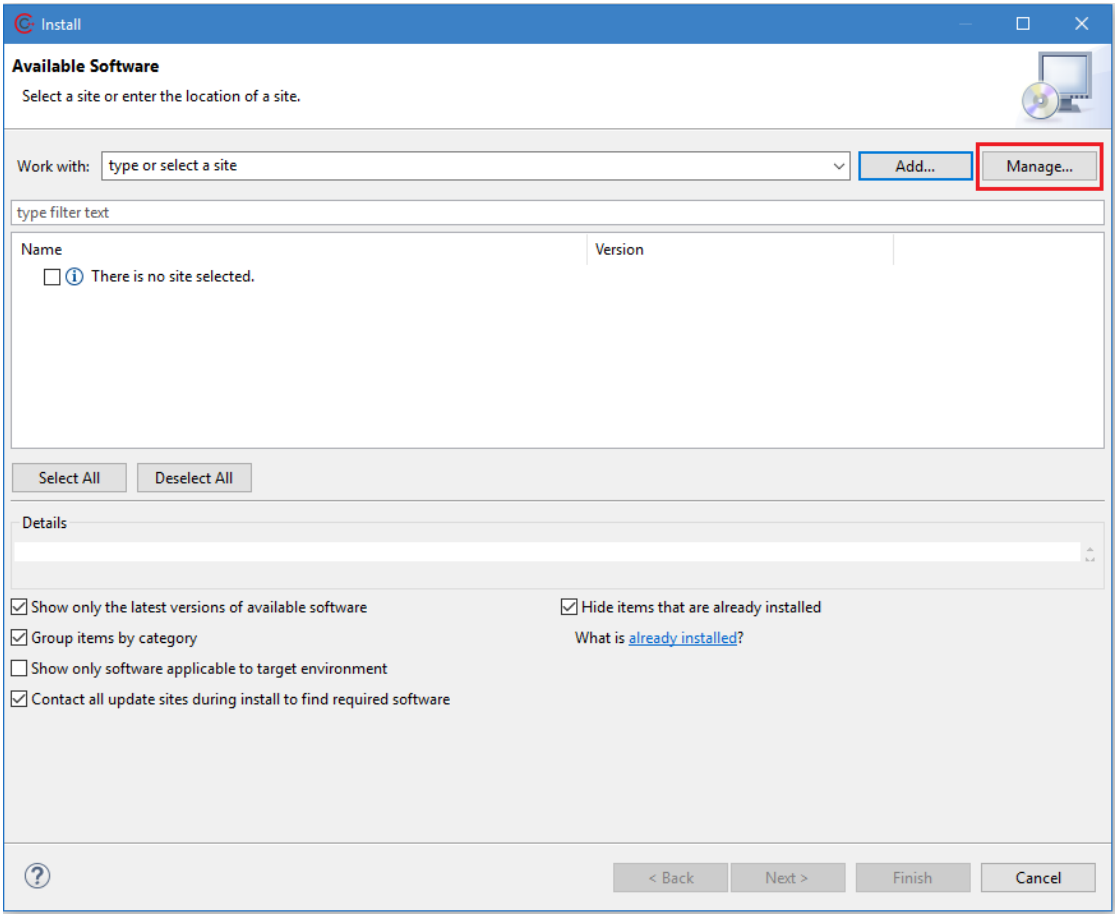


**Figure 20** Manage new Sites in Cevelop

By clicking on "Add" and selecting the previous downloaded files, you can add a new software site to Cevelop.

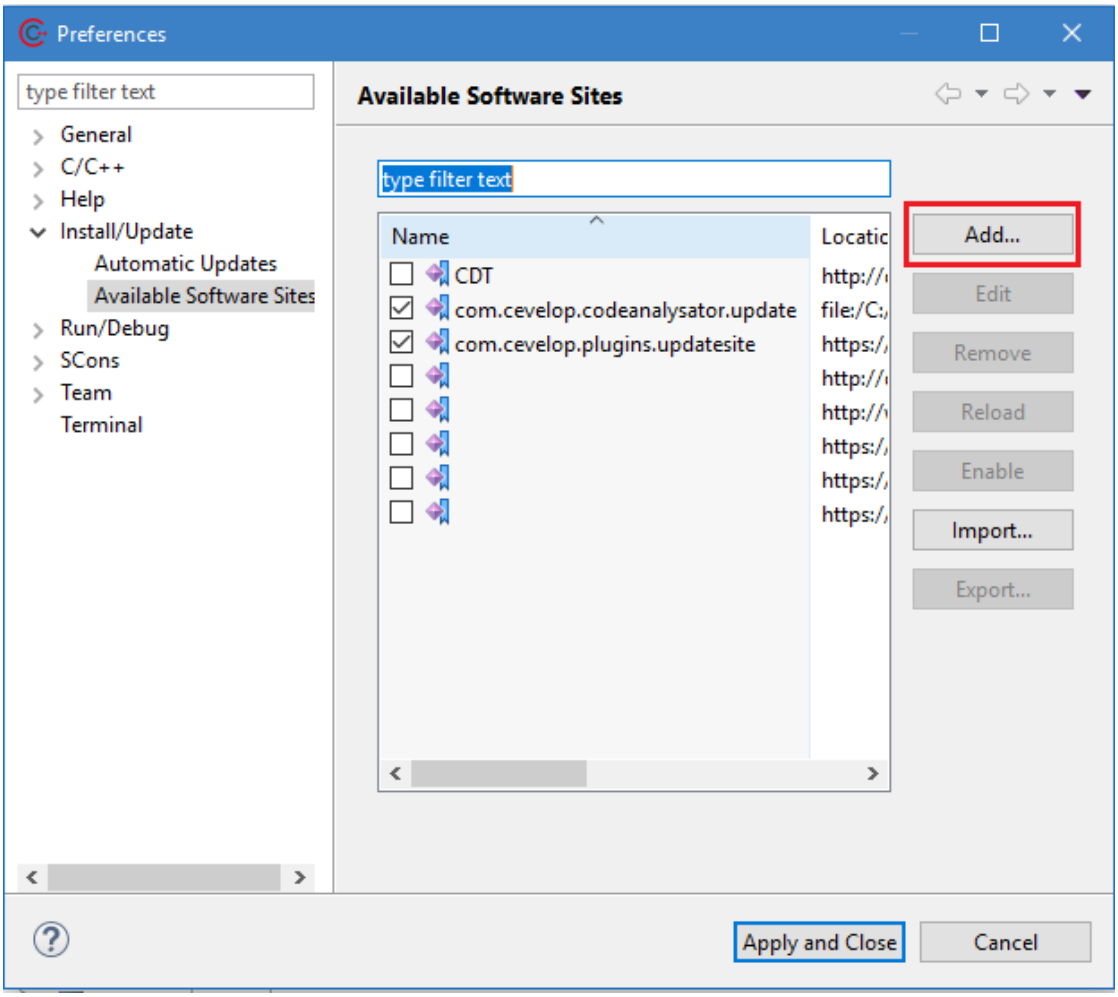


**Figure 21** Add new Sites in Cevelop

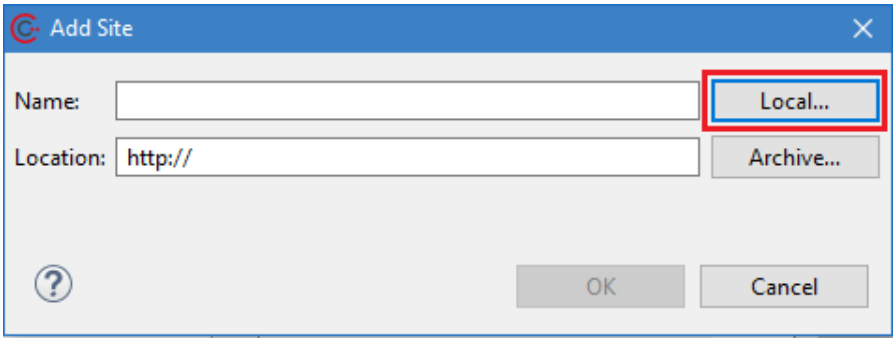


**Figure 22** Adding Code Analysator site

Choose an appropriate name and apply

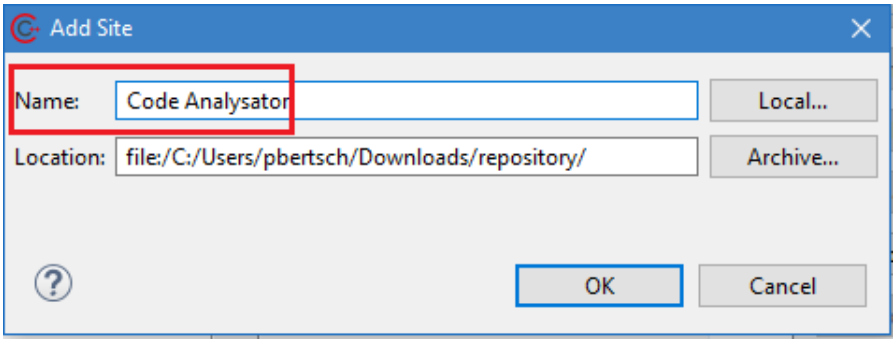


**Figure 23** Naming the site

Now, select the added site, check the box and apply

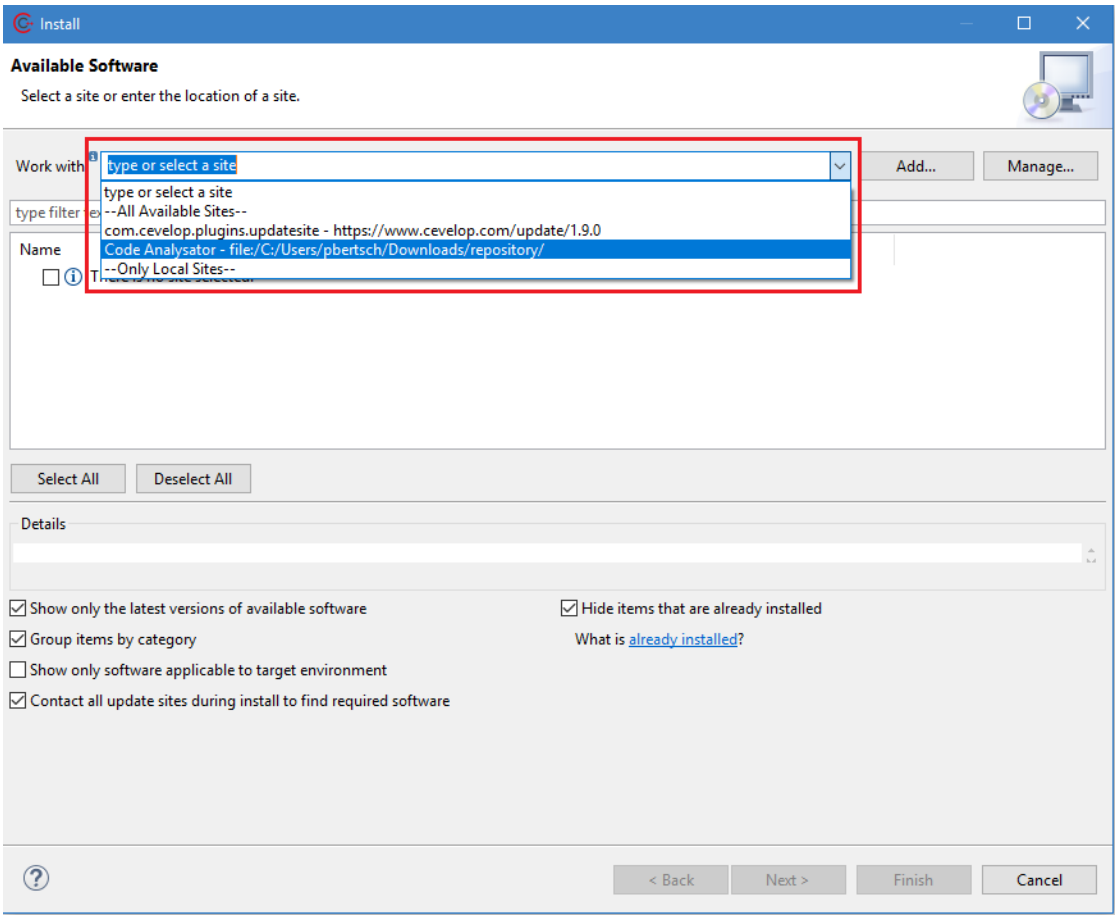


**Figure 24** Choosing the added site

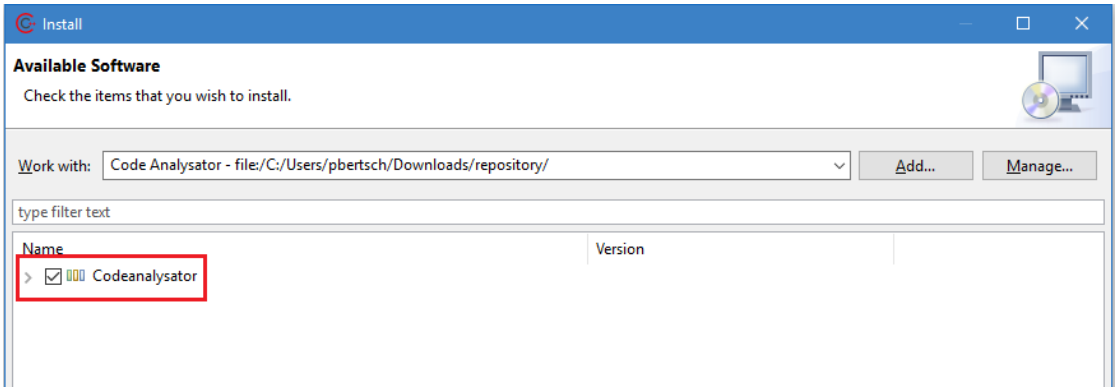


**Figure 25** Check the box

Note, in some case you will get a security warning due to unsigned content. You can ignore this warning and force the installation with clicking "Install anyway".

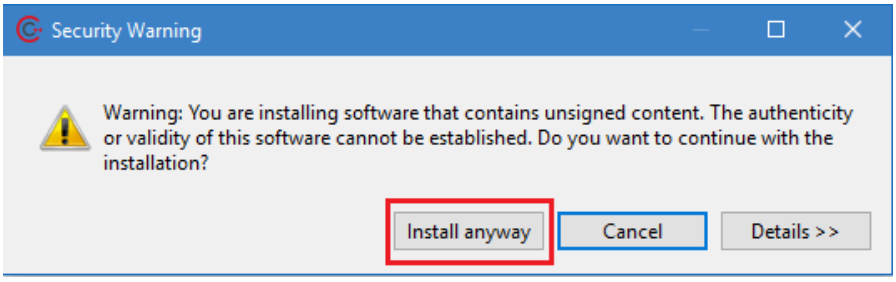


**Figure 26** Security Warning when installing Code Analysator

## A.b. How to enable/disable the plug-in

In order to enable or disable the plug-in, you have to open the preference page of the Code Analysator. On Windows and Linux go to "Window → Preferences → C/C++ → Code Analysis → Guidelines". Alternative you can look for "guideline" in "Quick Access". Then click on ”Add” and choose the guidelines you want activated. Save the changes by clicking on ”Apply changes”.

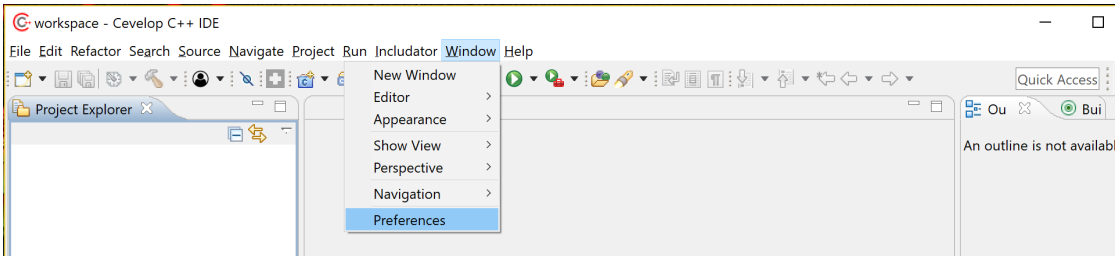


**Figure 27** Windows/Linux: access control - preferences

On Mac OS, this first step is different. Instead of using ”Window”, use ”Eclipse”. Go to "Window" use "Eclipse". Here the full path "Eclipse → Preferences → C/C++ → Code Analysis → Guidelines". Then, select ”Add” and activate all the guidelines you need.

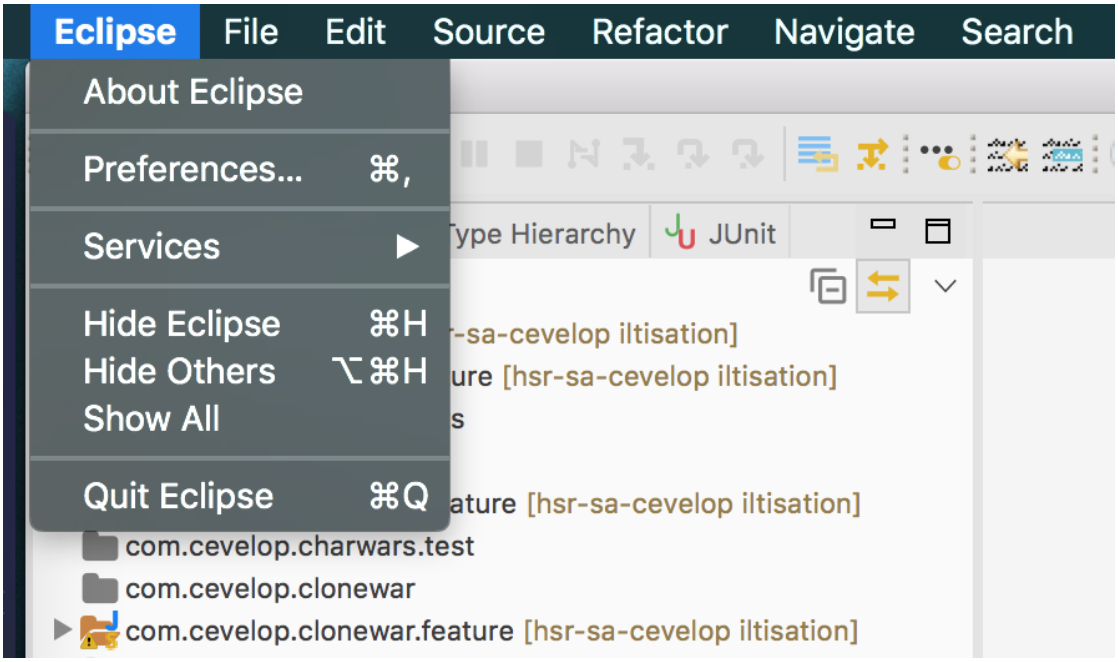


**Figure 28** Mac: access control - preferences

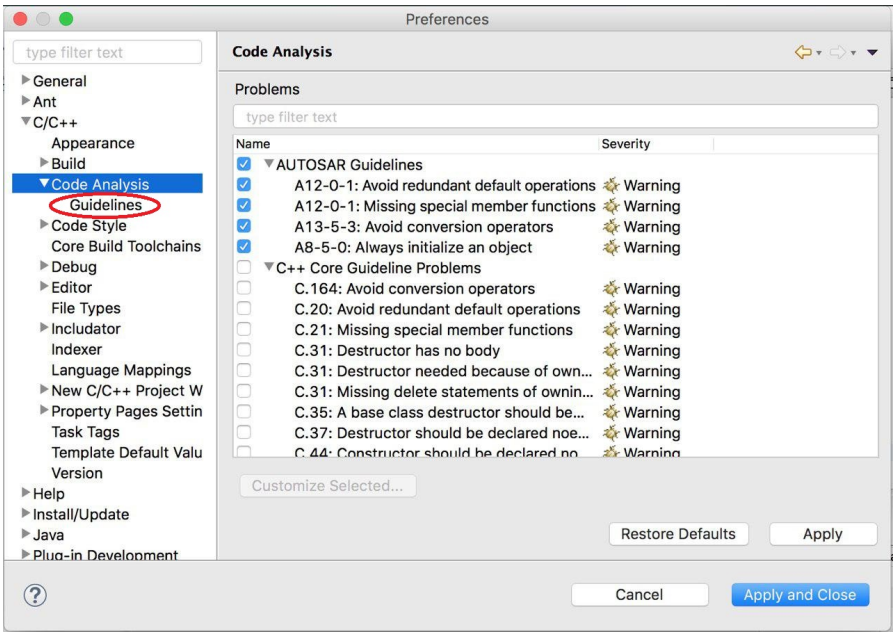


**Figure 29** access control - targeting guideline preference page

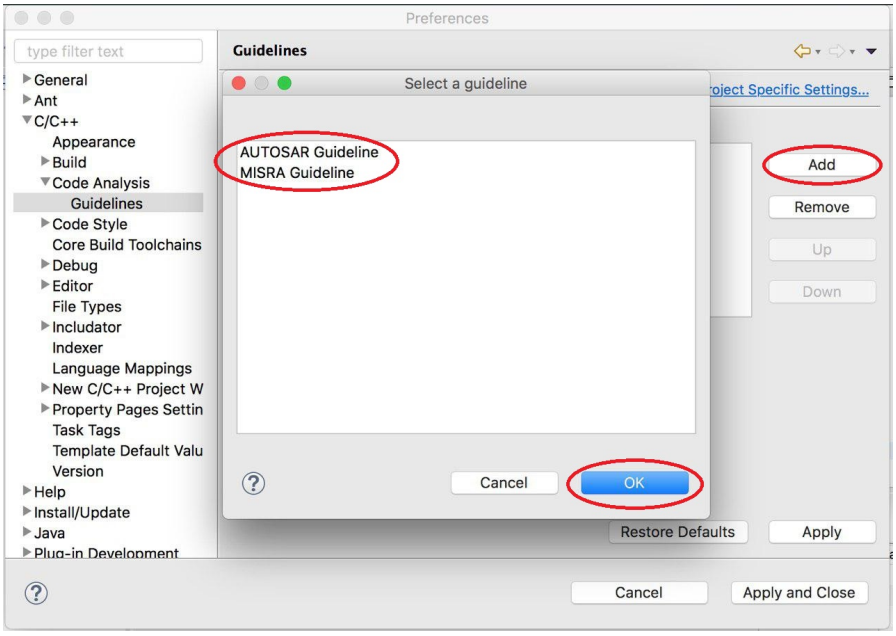


**Figure 30** access control - choosing guidelines

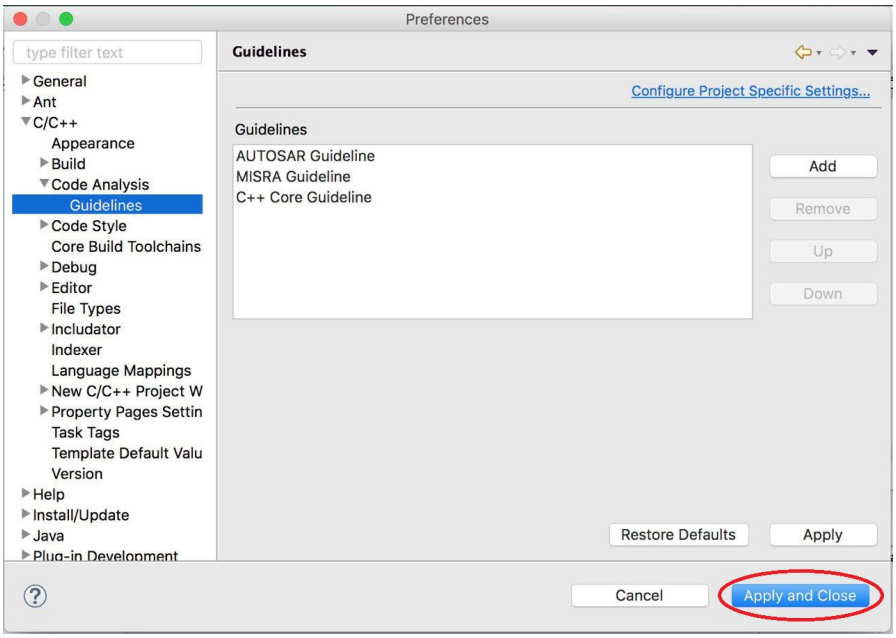


**Figure 31** access control - applying changes

### A.c. Prioritizing the guidelines

The prioritization of the guidelines is directly linked with their order at the preference page. When a problem (that is recognized by more than one guideline) shows up, the top guideline will be displayed in your editor The other guidelines will be suppressed. This also applies to quick fixes. You can change the prioritization through the preference page of the Code Analyser. To do so, open the preference page as explained in the previous chapter. Click on the activated guideline and navigate using the keys ”Up” or ”Down”.

### A.d. Usage

Any found issue with the enabled guideline will be underlined yellow in the code if it is a warning and blue if it is an info. Errors will be underlined red. To apply the quick fix you can open the quick fix list by clicking on the guideline icon on the left

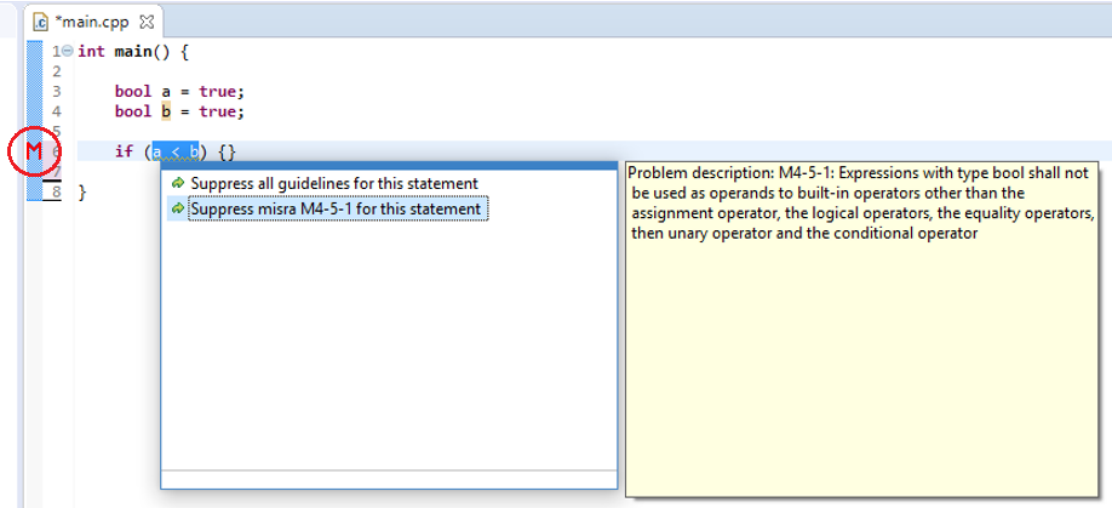


**Figure 32** open quick fix list over guideline logo

or with a right click on the underlined code.

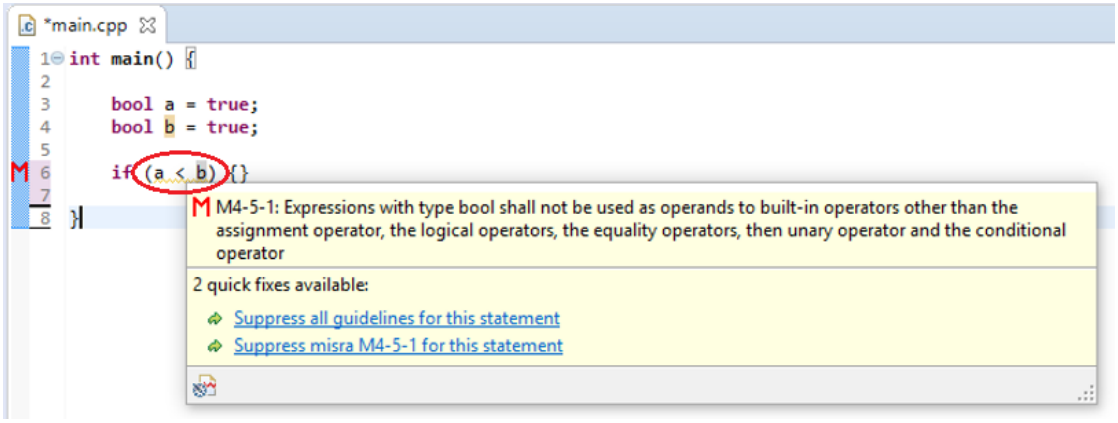


**Figure 33** open quick fix list over right click on code

### A.e. Surpressing warnings

If you want to suppress the warnings, you can do so by using the quick fix list explained above in chapter A.d. In some cases you will receive two suppress quick fixes. This happens when a warning is recognized by two or more activated guidelines. In this case you can either suppress warnings for all guidelines or only for the highest prioritized guideline. If you choose the latter, the warning disappears but then the next highest prioritized guideline activates.
This code snippet violates the "AboidConversionOperators" rule. In C++ Core Guideline "C.164 " and AUTOSAR guideline "A13-5-3"

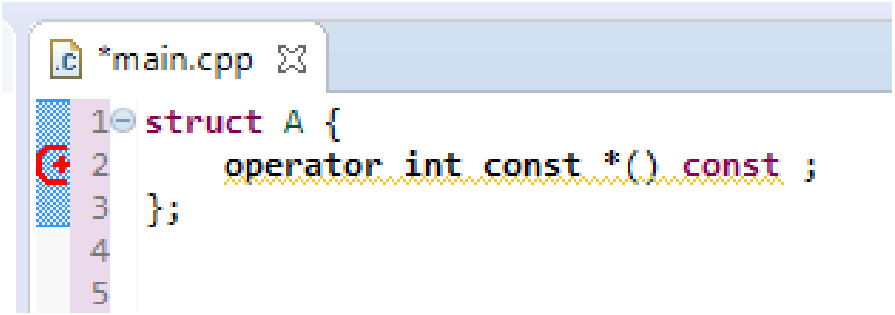


**Figure 34** suppressing the highest prioritized guideline

When suppressing only C++ Core Guidelines

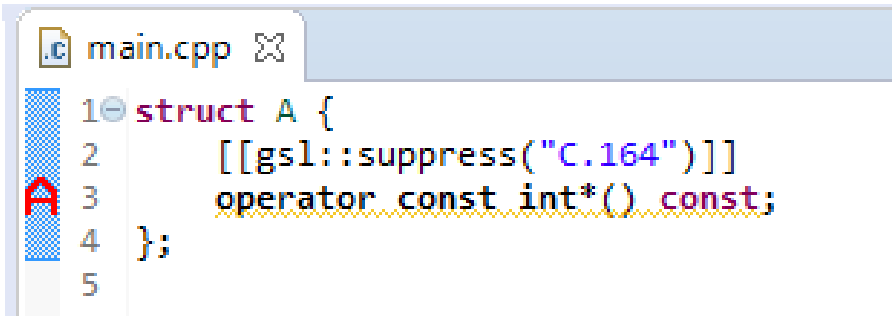


**Figure 35** suppressing the highest prioritized guideline

The "[[all:suppress("")]]" suppresses both guidelines

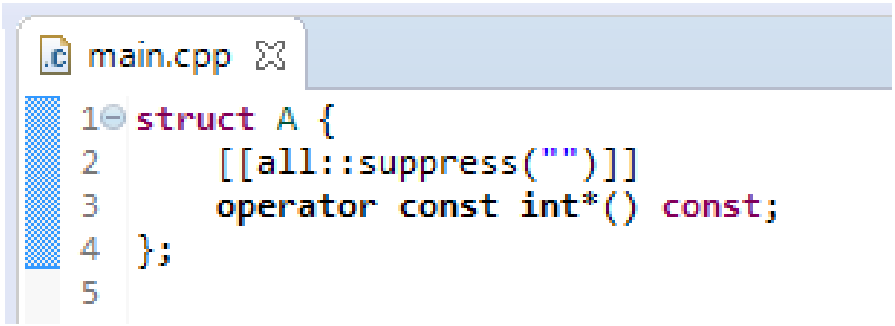


**Figure 36** suppressing the highest prioritized guideline

# B. Developer manual

The following manual is for developers, to learn how to implement a new guideline to the plug-in Code Analysator or how to extend an existing guideline. Extending existing guidelines means to add new visitors. Notice there are two different visitors. If a problem is implemented by multiple guidelines, then the visitor is shared by those guidelines. In the other case only the appropriate guideline has access to the visitor. To implement a shared or single guideline visitor differs partially on the implementation.

## B.a. Requirements

In order to extend code analyser, you need to check some requirements as follows. Before you start to develop, ensure all requirements are met.

### B.a.1. Prior Knowledge

The following text expects that you are familiar with how plug-ins and fragments with the OSGi framework work. Chapter 3 Analysis provides a corresponding introduction. Further, we expect understanding of the folder standards and guidelines for plug-in development as given in the documentation of ILTIS. [Sta]

### B.a.2. Code Analysator version

Make sure to use the latest version of Code Analysator.

### B.a.3. Making sure that the CDT testing target is set

After successfully installing the Cevelop source code `https://gitlab.dev.ifs.hsr.ch/sa-pbertsch-adeicha-2018/cevelop-plugins/tree/iltisation`, make sure that you set the right target platform for testing the cdt. This is used to build and launch your workspace plug-ins [Ecln].You need to find the tpd file in ”com.cevelop.plugins.target” and set the target with a right click. Make sure the tpd file implements two packages, ”ch.hsr.ifs.iltis.testing.highlevel.feature.feature.group” and ”ch.hsr.ifs.iltis.testing.cpp.feature.feature.group”.

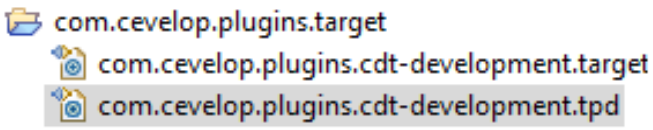


**Figure 37** tpd file location

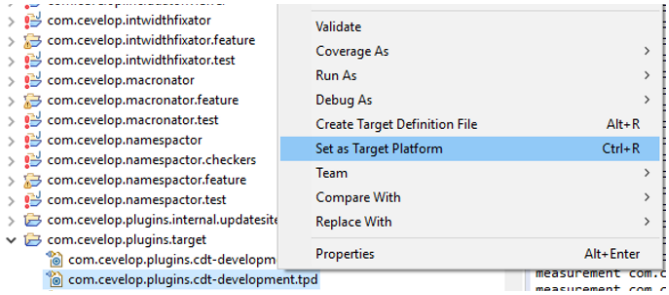


**Figure 38** set CDT target

## B.b. Implementation of a new set of guidelines

Let's first look at how new guidelines are created. To create new guidelines, you need an open Eclipse instance with the imported project "cevelop-plug-in". Than if you want to create a whole new guideline follow the next steps. But if you only want to extend a existing guideline with new rules, jump to "B.b.3 Using a checker".

### B.b.1. Adding fragments

Several guidelines like C++ Core, MISRA and AUTOSAR guidelines are already partialy implemented. Every guideline is a seperate fragment. So, adding a new guideline means adding a new fragment which contributes functionalities to the Code Analysator. Adding a new fragment is also explained in the ILTIS document [Sta] and is not here.

### B.b.2. Fragment.xml - Add Extension Points

In order to load the fragment to the Code Analysator via OSGi technique, you need to define some extension points first. To do so, you need to insert the following entries into your fragment.xml file.

**Guideline Mapper** The first extension point defines where the guideline mapper is located. Your guideline mapper class has to implement the IGuidelineMapper interface where your visitor and suppression settings are defined.

```
1  <extension point="com.cevelop.codeanalysator.core.guideline">
2    <implementation
         impl="com.cevelop.codeanalysator.misra.guideline.MisraGuidelineMapper"></implementation>
3  </extension>
```

**Using the MisraGuidelineMapper** We use the MISRA guidelines and our initial implementation of some guidelines in that set as an example on how to build your own. If you see the name "MISRA" in the following examples, change that to what your set of guidelines should be named internally. The "MisraGuidelineMapper" class maps problems to the guidelines and adds a suppression strategy and quickfixes. More specific, the *getName()* method is used to show the guideline name in the preference page. *GetId()* is used internally to identify the guideline. Then, *getMapping()* returns a hashmap showing the visitor and all of its assigned problems according to the guideline. Through this list we can find out if one problem is assigned to multiple visitors. The last method returns all quickfixes to one problem.

```
1  public class MisraGuidelineMapper implements IGuidelineMapper {
2     private AttributeSuppressionStrategy suppressionStrategy = new AttributeSuppressionStrategy();
3     private HashMap<String, String> mappings = new HashMap<String, String>();
4     private HashMap<String, IMarkerResolution[]> quickfixes = new HashMap<String,
          IMarkerResolution[]>();
5
6     public MisraGuidelineMapper() {
7        mappings.put(CoreIdHelper.DeclareLoopVariableInTheIntializerVisitorId,
              MisraIdHelper.DeclareLoopVariableInTheIntializerProblemId);
8
9        suppressionStrategy.addSuppression(CoreIdHelper.DeclareLoopVariableInTheIntializerVisitorId,
              new MisraSuppressionAttribute("M3-4-1"));
10
11       quickfixes.put(MisraIdHelper.DeclareLoopVariableInTheIntializerProblemId, new
              IMarkerResolution[] { new DeclareLoopVariableInTheInitializerQuickFix("ES.74: Add a
              variable declaration") });
12    }
13
14    @Override
15    public String getName() {
16       return "MISRA Guideline";
17    }
18
19    @Override
20    public String getId() {
21       return MisraIdHelper.GuidelineId;
22    }
23
24    @Override
25    public Map<String, String> getMappings() {
26       return mappings;
27    }
28
29    @Override
30    public ISuppressionStrategy getSuppressionStrategy() {
31       return suppressionStrategy;
32    }
33
34    @Override
35    public Map<String, IMarkerResolution[]> getQuickfixes() {
36       return quickfixes;
37    }
38
39 }
```

**Implementing a Codan category** The next thing we need to do is define a Codan category. A Codan category wraps a bunch of problems together. Our solution owns only one Codan category

and every problem is assigned to it. They could also be separated into different categories.

```
<extension point="org.eclipse.cdt.codan.core.checkers"
      id="org.eclipse.cdt.codan.core.categories">
   <category id="com.cevelop.codeanalysator.core.misra"
        name="MISRA Guidelines"/>
 </extension>
```

A Codan category is shown in the preferences in the Code Analysis page and is listed as below.

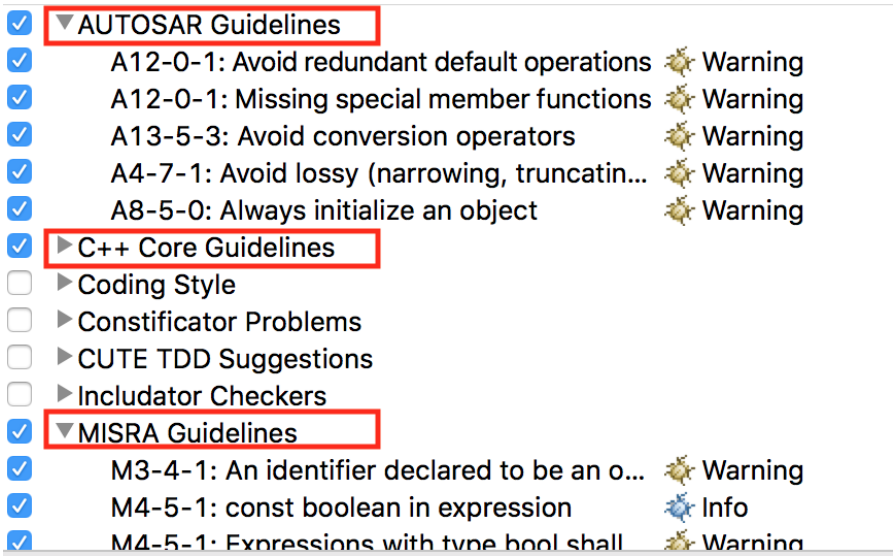


**Figure 39** storing a specific visitor

Now we need to use extensions that can define the marker. These extensions contain all the different marker types like warning, info or error, the marker resolution generator, annotation types and the marker annotation specification.

**Defining Markers** The markers may be associated with workbench resources. Workbench is defined as follow from Eclipse documentation: [Eclq]

> The Workbench aims to achieve seamless tool integration and controlled openness by providing a common paradigm for the creation, management, and navigation of workspace resources.
>
> Each Workbench window contains one or more perspectives. Perspectives contain views and editors and control what appears in certain menus and tool bars. More than one Workbench window can exist on the desktop at any given time.

Markers are used for many things in the workbench. The main uses of markers in workbenches are tasks, problems and bookmarks. Markers will be shown in the marker view or on the marker bar in the editor area [Ecll]. In our case we need the marker to handle problems (errors, warnings, information). Define the type of marker and use the super type ”codanProblem”.

```
  <extension
        id="com.cevelop.codeanalysator.core.misra.warning.marker"
        name="MISRA Guideline"
        point="org.eclipse.core.resources.markers">
     <super type="org.eclipse.cdt.codan.core.codanProblem"/>
     <persistent value="false"></persistent>
   </extension>
```

**Defining MarkerResolution** The "MarkerResolution" resolves the quick fix suggestions. It has to be defined what type of marker it is fixing.

```
<extension point="org.eclipse.ui.ide.markerResolution">
    <markerResolutionGenerator
        class="com.cevelop.codeanalysator.misra.quickfix.MisraMarkerResolutionGenerator"
        markerType="com.cevelop.codeanalysator.core.misra.warning.marker">
    </markerResolutionGenerator>
</extension>
```

**MisraMarkerResolutionGenerator** The referenced class (MisraMarkerResolutionGenerator) inherits the AttributeMarkerResolutionGenerator. This parent class knows everything about the available guidelines and resolves the matching quickfixes from the GuidelineMapper class. In order to suggest the correct guideline specific suppression, there is an abstract method to return suppression attribute.

```
public class MisraMarkerResolutionGenerator extends AttributeMarkerResolutionGenerator {

   public MisraMarkerResolutionGenerator() {
      super(MisraIdHelper.GuidelineId);
   }

   @Override
   public AttributeSuppressQuickfix getSuppressionResolution(SuppressionAttribute attr) {
      return new MisraSuppressGuidelineQuickfix(attr.getIgnoreText());
   }
}
```

Furthermore, we need to assign our markers to one annotation type, either error, warning or info. In our case we implemented a warning. This is made visible through the ”MarkerSeverity” attribute. Depending on which value, it indicates the marker as an error, warning or info. Information is 0, warning 1 and error is 2 [Ecli]. Optionally, an own marker type can be defined by tagging the ”markerType”. The ”name” attribute gives a unique identification of the own annotation type and ”super” defines the super type. [Eclc]

```
   <extension point="org.eclipse.ui.editors.annotationTypes">
      <type markerSeverity="1"
            markerType="com.cevelop.codeanalysator.core.misra.warning.marker"
            name="com.cevelop.codeanalysator.core.misra.warning.annotationType"
            super="org.eclipse.ui.workbench.texteditor.warning">
      </type>
   </extension>
```

Now you can define the style of your marker like color, icon, label ect. To do so, reference your marker through the "annotationType" attribute with your pre defined annotation type. You will find all attributes explained at [Ecla]

```
1   <extension point="org.eclipse.ui.editors.markerAnnotationSpecification">
2      <specification
3           annotationType="com.cevelop.codeanalysator.core.misra.warning.annotationType"
4           colorPreferenceKey="com.cevelop.codeanalysator.core.misra.markerAnnotation.color"
5           colorPreferenceValue="255,255,0"
6           contributesToHeader="true"
7           highlightPreferenceKey="com.cevelop.codeanalysator.core.misra.markerAnnotation.highlight"
8           highlightPreferenceValue="false"
9           icon="icons/misra_warn.png"
10          label="MISRA Problem"
11          quickFixIcon="icons/misra_warn.png">
12     </specification>
13  </extension>
```

### B.b.3. Using a checker

In order to visualize problems we need a checker (see 3.2.4).The checker and its clarified problems need to be registered, so that Codan can integrate them and use the checker on source code changes. In our architecture, every guideline got one checker which uses multiple visitors. One visitor can also cover several problems. At this point, notice that every new problem, when extending an existing guidelines, has to be defined here. Keep this in mind when moving on.

```
1  <extension point="org.eclipse.cdt.codan.core.checkers">
2      <checker
3           class="com.cevelop.codeanalysator.misra.checker.MisraChecker"
4           id="com.cevelop.codeanalysator.misra.checker.MisraChecker"
5           name="MISRA Guideline Checker">
6        <problem
7           category="com.cevelop.codeanalysator.core.misra"
8           defaultSeverity="Warning"
9           defaultEnabled="true"
10          description="M4-5-1: Expressions with type bool shall not be used as operands to
                built-in operators other than the assignment operator, the logical operators, the
                equality operators, then unary operator and the conditional operator"
11          id="com.cevelop.codeanalysator.misra.problem.boolexpressionoperands"
12          messagePattern="M4-5-1: Expressions with type bool shall not be used as operands to
                built-in operators other than the assignment operator, the logical operators, the
                equality operators, then unary operator and the conditional operator"
13          name="M4-5-1: Expressions with type bool shall not be used as operands to built-in
                operators other than the assignment operator, the logical operators, the equality
                operators, then unary operator and the conditional operator"
14          markerType="com.cevelop.codeanalysator.core.misra.warning.marker">
15       </problem>
16     </checker>
17  </extension>
```

This implementation uses the ”CodeAnalysatorCompositeChecker” as a super type. The super class implements methods to traverse through the AST tree and automatically report any problem. Our specific guideline checker, in this case the MisraChecker, needs to overwrite the initVisitor method. This method is used on each check iteration to run the defined visitors.

```
1 public class MisraChecker extends CodeAnalysatorCompositeChecker {
2
3    @Override
4    protected void initVisitor(VisitorComposite visitor) {
5       visitor.add(new DeclareLoopVariableInTheIntializerVisitor(this,
            MisraIdHelper.DeclareLoopVariableInTheIntializerProblemId));
6       visitor.add(new AvoidLossyArithmeticConversionsVisitor(this,
            MisraIdHelper.AvoidLossyConversionsProblemId));
7       visitor.add(new BoolExpressionOperandsVisitor(this));
8    }
9 }
```

### B.b.4. Visitor

As already explained, we address two types of visitors: the shared ones and the specific ones. In case your visitor implements a problem that is only covered by one guideline, check the explanation down at "Specific visitor". If, on the other hand, your visitor needs to be shared between multiple guidelines, follow "Shared visitor". However, most of the structure is similar for both types of visitors as only location and super class are different. Further visitor implementation will be explained later.

**Specific visitor** Any specific visitor has to be stored in the source folder of your plug-in under "visitor". Here an example using a MISRA specific visitor:

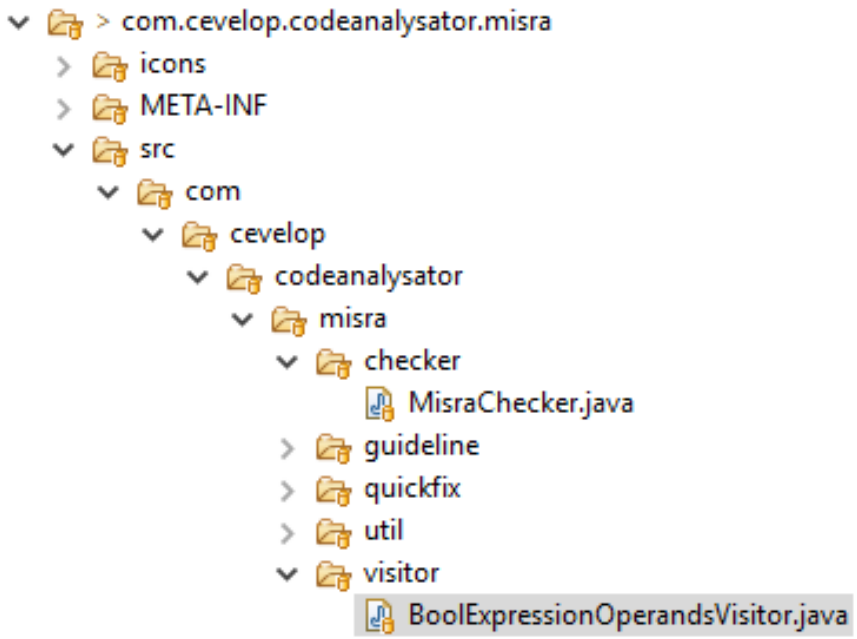


**Figure 40** storing a specific visitor

Another difference is the super class. Specific visitors inherit from ”CodeAnalysatorVisitor” using the package: ”com.cevelop.codeanalysator.core.visitor.CodeAnalysatorVisitor”.

```
public class BoolExpressionOperandsVisitor extends CodeAnalysatorVisitor<MisraChecker>
```

**Figure 41** super class of specific visitors

**Shared visitor** Shared visitors are located in the "core.visitor.shared" section of the Code Analysator.

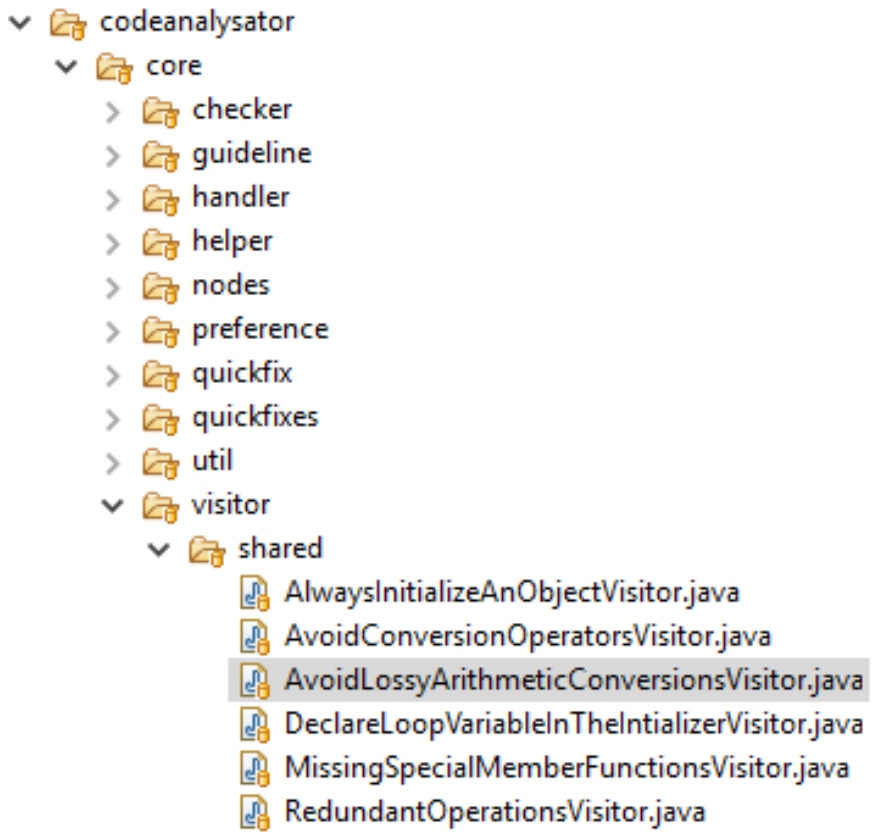


**Figure 42** storing a shared visitors

They inherit from "SharedVisitor" from package "com.cevelop.codeanalysator.core.visitor.SharedVisitor"

```
public class AvoidLossyArithmeticConversionsVisitor extends SharedVisitor
```

**Figure 43** super class of shared visitor

**void setShouldVisit()** This method defines what kind of node in the AST tree should be visited by this visitor. Choose the appropriate bool expression and set it to "true" to make the visitor check all these nodes. You can find all the available boolean flags in the super classes.

```
    @Override
    protected void setShouldVisit() {
        this.shouldVisitExpressions = true;
    }
```

**String[] getProblemIds()** This method returns all different problem id's that the visitor reports. This is used in the checker to check if the visitor needs to traverse the tree. If the guideline of the visitor is disabled or all returned problems are disabled, the visitor is skipped.

**int visit(IASTNode node)** The visit method is called by the AST tree when a matching node was found. The visitor should check the nodes for problems and report anything, if found. Depending on the return value, the visitor will either abort, skip or continue. "PROCESS_CONTINUE" will continue traversing the AST tree down. "PROCESS_ABORT" stop the traversal completely. "PROCESS_SKIP" won't traverse further down from the current node. That means the whole sub tree (under the current node) will be completely skipped. However, the traversal continues with the correct right neighbour node.

**checker.reportProblem(String problemId, IASTNode node)** Every CodeanalysatorVisitor has an instance of a checker that is used to report problems on AST nodes. The parameters of the method are problem id and the affected node. The checker instance will based on priority and suppression statements report the problem to Codan or hide it.

```
checker.reportProblem(MisraIdHelper.BoolExpressionOperandsProblemId, expression)
```

### B.b.5. Quick fix

These classes decides the appropriate solution to the warnings. Suppressions are already implemented for all checkers generally. If you want to implement quick fixes you will need to place the class in the "quickfix" folder. Again you have to choose between quick fix for shared visitor and quick fix for specific visitor.

**Quick fix for specific visitor** is stored in the appropriate source folder of the guideline.

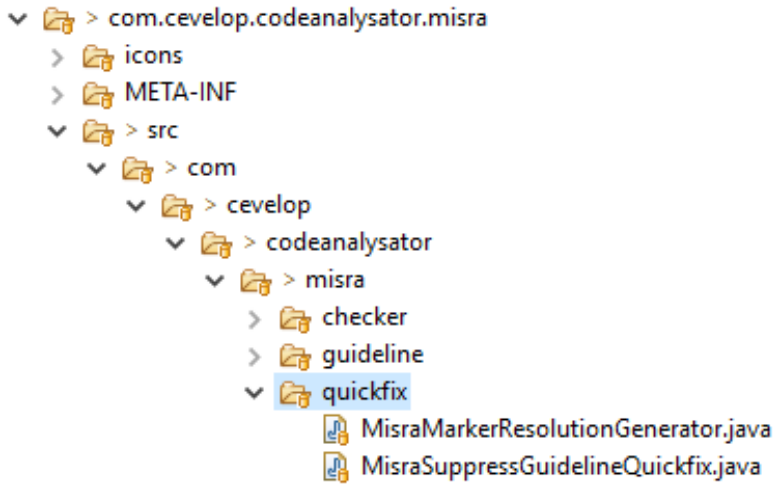


**Figure 44** storing a shared visitor

**Quickfix for shared visitor** The shared visitor differs only in where to store. Instead to use the "quickfix" folder in your guideline fragment, use "com.cevelop.codeanalysator.core.quickfixes.shared".

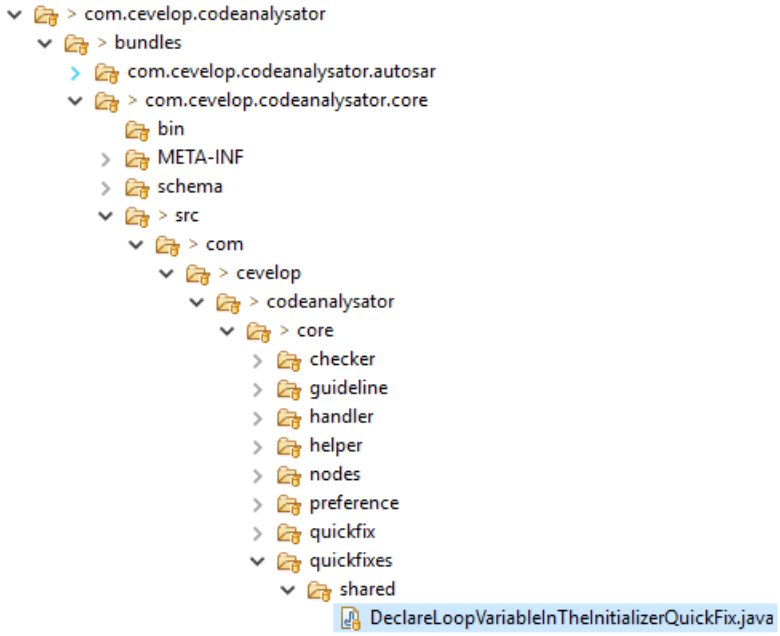


**Figure 45** storing a shared visitor

### B.b.6. IdHelper

In the chapters above there were multiple ID's mentioned. All those ID's are hard-coded references and therefore it is required to work with constant values. The IdHelper class is where all these ID's are defined and referenced from. This makes it also easy to find those ID's by others that might want to refactor the source code.

# C. Tests

This chapter outlines the results of our tests. The outcome of the tests have to match the required level described in the project plan **??** of 80% code coverage.

## C.a. Integration Test

The three guidelines MISRA, AUTOSAR and C++ Core are tested with integration tests. This tests also cover the functionality of the core of Code Analysator. You will see in the chapter below "C.b Unit Tests" that the code coverage for the core section only reaches nearly 10%. This has to do with the fact that Code Analysator interacts with Codan. In order to proof that the Code Analysator works proper, we need to test the whole system. This is also the reason why MISRA, AUTOSAR and C++ Core Guidelines are not unit tested but integration tested. Keep in mind that we only refactored the most visitors and their tests. Only one visitor was implemented by us and only for this visitor we wrote new tests. This visitor covers a MISRA problem.

**MISRA** The own implemented visitor and their tests brought the code coverage to 92% up.

| | | | | |
|---|---|---|---|---|
| com.cevelop.codeanalysator.misra.tests | 92.7 % | 164 | 13 | 177 |
| src | 92.7 % | 164 | 13 | 177 |
| com.cevelop.codeanalysator.misra.tests.quickfix | 100.0 % | 63 | 0 | 63 |
| com.cevelop.codeanalysator.misra.tests.checker | 94.6 % | 87 | 5 | 92 |
| com.cevelop.codeanalysator.misra.tests | 64.3 % | 9 | 5 | 14 |
| com.cevelop.codeanalysator.misra.tests.util | 62.5 % | 5 | 3 | 8 |

**Figure 46** MISRA integration test

**AUTOSAR** AUTOSAR and all its test were only refactored to match to our plug-in. We considered to improve the tests for AUTOSAR visitors in order to bring the code coverage to 80% up. After discussing it with our supervisor, he approved us not to do so due to more important tasks and lack of time.

| | | | | |
|---|---|---|---|---|
| com.cevelop.codeanalysator.autosar | 78.2 % | 140 | 39 | 179 |
| src | 78.2 % | 140 | 39 | 179 |
| com.cevelop.codeanalysator.autosar.checker | 100.0 % | 39 | 0 | 39 |
| com.cevelop.codeanalysator.autosar.guideline | 95.3 % | 101 | 5 | 106 |
| com.cevelop.codeanalysator.autosar.quickfix | 0.0 % | 0 | 31 | 31 |
| com.cevelop.codeanalysator.autosar.util | 0.0 % | 0 | 3 | 3 |

**Figure 47** AUTOSAR integration test

**C++ Core** C++ Core is with 96% code coverage way above the planned value.

| | | | | |
|---|---|---|---|---|
| com.cevelop.codeanalysator.cppcore | 96.6 % | 1,013 | 36 | 1,049 |
| src | 96.6 % | 1,013 | 36 | 1,049 |
| com.cevelop.codeanalysator.cppcore.checker | 100.0 % | 70 | 0 | 70 |
| com.cevelop.codeanalysator.cppcore.visitor.util | 100.0 % | 23 | 0 | 23 |
| com.cevelop.codeanalysator.cppcore.visitor | 99.0 % | 725 | 7 | 732 |
| com.cevelop.codeanalysator.cppcore.guideline | 97.4 % | 185 | 5 | 190 |
| com.cevelop.codeanalysator.cppcore.quickfix | 32.3 % | 10 | 21 | 31 |
| com.cevelop.codeanalysator.cppcore.util | 0.0 % | 0 | 3 | 3 |

**Figure 48** C++ Core integration test

**Test migration from gladiator** For every visitor or quick fix we moved from the gladiator source code we also moved the tests. Because of the new version and breaking interface changes in the cdt testing and iltis code the migration involved some changes. The rts file structure changed and some of it's commands. The most notable were the renaming of the markerPositions command to markerLines that is used in the assert statements to validate if the markers were generated. An other change was that in all tests the current guideline should be activated and this requires that every tests sets those with the setPreferencesEval command. As example the following figure shows how a refactored visitor integration test looks like.

```
10    //!ExpressionWithConstBoolFalseRight
11    //@.config
12    setPreferencesEval=(GUIDELINE_SETTING_ID|MISRA_GUIDELINE_ID)
13    markerLines=3
14    //@main.h
15 1  int main() {
16 2      bool b2 = true;
17 3      if(b1 == false) {}
18 4  }
```

**Figure 49** Refactored integration test

## C.b. Unit Tests

In this chapter we present the tests for the core of Code Analysator. As already mentioned the level of code coverage is far below the planed 80%. The core section is nearly at 10%. But the problem we had was that Code Analysator interacts a lot with Codan code. Therefore, we had to mock many functionalities. So, it is a high code coverage is nearly impossible and not necessary. But to be sure that Code Analysator works proper, we made integration tests with the guidelines, that guarantees that. Therefore, the code coverage of the core functionalities is much higher than measured with unit test.

**Code analysator** You clearly see that the guideline part "com.cevelop.codeanalysator.core.guideline" got almost 70% code coverage. This part hasn't to do much with Codan interactions that's why the value is so high.

| | | | | |
|---|---|---|---|---|
| com.cevelop.codeanalysator.core | 9.7 % | 885 | 8,275 | 9,160 |
| src | 9.7 % | 885 | 8,275 | 9,160 |
| com.cevelop.codeanalysator.core.guideline | 67.8 % | 248 | 118 | 366 |
| GuidelineProblem.java | 100.0 % | 15 | 0 | 15 |
| GuidelineConflictResolver.java | 94.2 % | 212 | 13 | 225 |
| Guideline.java | 84.0 % | 21 | 4 | 25 |
| GuidelineConflictResolverFactory.java | 0.0 % | 0 | 54 | 54 |
| GuidelineLoader.java | 0.0 % | 0 | 47 | 47 |
| com.cevelop.codeanalysator.core.visitor | 44.8 % | 419 | 516 | 935 |
| VisitorCache.java | 100.0 % | 59 | 0 | 59 |
| CodeAnalysatorVisitor.java | 76.3 % | 87 | 27 | 114 |
| VisitorComposite.java | 36.7 % | 273 | 471 | 744 |
| SharedVisitor.java | 0.0 % | 0 | 18 | 18 |
| com.cevelop.codeanalysator.core.nodes.util | 41.5 % | 51 | 72 | 123 |
| com.cevelop.codeanalysator.core.checker | 40.7 % | 88 | 128 | 216 |

**Figure 50** C++ Core integration test

## C.c. Usability Test

The following sub chapter provides the set up of the test, the tasks which the IT students have to solve and the outcome. Than we will discuss our result with the supervisor and all decisions of this discussion will be noted in the paragraph "Changes". After making the UI suitable, the usability test will be repeated with other students to test the improvement. This outcome will also be provided in this paper. At this point we want mention that the menu which contains the tab "Preference", which is important to manage the preference page of our plug-in, is positioned at two different places depending on the Operating Systems (Mac or Windows). In Windows you will find "Preferences" in the main tab "Window". If you search for it on macOS, you will find it in "Eclipse".

**Set up** The setup for the test will be as follows:

- a notebook with a started Cevelop instance will be provided
- three classes in the Cevelop instance with several code snippets
- internet access
- 10 minutes to solve all requirements
- it is not allowed to talk (questioner and proband), except to repeat the task (at any time)

**Tasks** Following will be the tasks which has to be read out for the probands:
"You got three classes with different snippets in your running Cevelop instance. Every of this class is for another company and different code guidelines are important. Therefore you want to change the warnings depending on which code you are working on. Begin with the first class. For every class your guide will give you another task. When you solve what is asked from you, ask your guide for the next task and jump to the next class in the tab."

Here the tasks for the three classes.

1) Peter Sommerlad, your C++ teacher, is walking through the class room and offers his help for the C++ task. Suddenly he stops, watches into your screen and explains, that he has found a flaw in your code. You are confused because you can't see any warning. He calls you on to find this mistake in the Core Guideline of C++. Therefore he sends you the Github link. But as always you are lazy. Thus you try to find a faster way. You also remember that somebody told you Cevelop has guideline checker implemented.
2) You are working for the auto mobile company Audi. Therefore it is very important for you to have MISRA guideline activated because it supports guidelines for the automotive industry. Your code shows you warnings but it is from the Core Guideline. Core Guidelines also implement checkers for the automotive industry, you remember. But you are told by your supervisor to work primarily with MISRA quick fixes. They also told you not to disable Core Guidelines because it provides important guidelines which MISRA does not.
3) You quit and started to work in a start up which also offers software for auto mobile. You explained your new colleges how important it is to have the style guidelines activated. Your boss is very impressed of your experience and knowledge and asks you for code review. As you search for mistakes you find a warning which obviously was ignored by your boss. As you confront him with this mistake, he explains you that the quick fix wouldn't work properly for his program. You both conclude that the warning has to be suppressed.

#### Solutions

1) Eclipse/Window → Preferences → C/C++ → Code Analysis → Guidelines → Add → C++ Core Guidelines
2) Solution 1) but instead to click on Add you need to choose MISRA and click UP
3) Hover the mouse above the code and right click or make the right click on the guideline logo which is left next to the marked code. Then choose "suppress all"

**Outcome Part 1** After testing five people (24.04.2018, see appendix) following problems crystallized out:

1) Activate plug-in
   All probands had the most difficulties with finding the preference page for the plug-in. Most of the time the probands used was to search the guideline preference menu. The most common approach was, that they expected to get the preference page through the context menu with a right click on the code or on the class. Then they checked the tabs for something appropriate. Obviously it was uncertain that "preference" could contain the management UI. Most of them tried to find the solution with search fields either in the tab "Eclipse"/"Window" or in the "Quick Access" tab and one even tried it with "Help" → "search". Another thing, which confused the probands was, even if they found the way through "Eclipse"/"Window" → "C/C++", they always first saw and clicked on the tab "Code Style" which was direct under "Code Analyse". One student also tried to come with it up above "Project" → "Properties".
2) Change prioritization
   This task took the probands second longest. But they were clearly lot faster than the at the first task. It was noticeable that the students didn't understand right away what UP does because it was not described anywhere (in preference page). It wasn't clear enough

that the order manages the prioritization. Some probands tried to find the menu for prioritization with a right click on the marked code. One even searched for "Quick Fix" in the "Quick Access" search field. Furthermore two people tried to manage this problem with the suppression.

3) Suppress warnings
Suppressing the warnings was solved by all probands immediately. But one mistake or uncertainty continued through the entire group. All probands suppressed the warnings individually instead to use the suppress all quick fix. The reason was almost always the same. They thought it will suppress all same elements in the whole document and maybe also in future, what obviously could have a negative impact, if you would have forget about it.

4) Abnormalities
One mentionable point is, that most of the probands didn't recognised the logo or when it changes. This may be related to the fact that the logos are unknown. Another possibility is, that the logos are always blue, so they won't recognize if the logo changes. Another point was, that all the probands were confused when they solved the task, e.g. activating the Core Guideline, and the warnings wouldn't appear. Before the code gets marked, you need to safe the file or at least to change one character (includes white space).

5) Recommendation of probands
We asked the students for their suggestions and one of them explained us, that he would like to have more described on the preference page when he clicks on the question mark. He also would like to be able to get informations and descriptions through the Help tab. Another proband wanted the plug-in to be activated when delivered.

6) Time
Four of five proband made it within 10 minutes. One of them needed 11 minutes. The fastest one was done in 5 minutes, the slowest as mentioned before in 11.

**First Changes** Following improvements were decided (24.04.18):

1) Activate plug-in
To make it easier to find the preference page we will add an entry in the context menu. We will also write a helper guide.

2) Change prioritization
A better description on the preference page of Code Analysator will be given to make sure that everybody understands that the order is responsible for the prioritization.

3) Suppress warnings
The text for the suppression quick fix which suppresses all guidelines for one explicit node in the AST will be improved. Instead "suppress all" the annotation will be changed to "suppress all guidelines for this warning".

4) Abnormalities
In order to make the logos more recognizable we changed the logo color to blue for infos and red for warnings. We expect the people recognize red more because it's a stronger color than dark blue.

5) Recommendation of probands
As at point 2 explained, the preference page will be better described.

**Outcome Part 2** When second time tested five different students they surprised us a lot. Not one student/proband used the improvements of our first outcome but solved the usability tests much better than the first group. Two of them had already to do with Cevelop, this maybe had an impact. But generally they were faster and found everything intuitive. Therefore, it is very hard to compare bot outcomes. In order to get good, comparable results we would have to define better what previous knowledge the proband should have and also increases the number of probands.

1) Activate plug-in
   Surprisingly this task was solved pretty quick in the second round. Only one proband couldn't find "Eclipse" tab right away. He first asked for a key short cut. He then used the Quick Access to find the guideline preference page. Another proband found a way, that even we didn't knew. He used the right click on the problem in the problem list below the coding editor and found a context menu entry "Open Guideline preferences". We found out that our implementation for the context menu entry is wrong because it is now displayed in every section.
2) Change prioritization
   No problems at this point at all. As a result of the improvement of the first task, it also had an impact of the understanding of the second task.
3) Suppress warnings
   This task also were solved perfectly. Every student solved it right away with the "suppression all" annotation.
4) Abnormalities
   No abnormalities at all
5) Recommendation of probands
   No recommendation at all
6) Time
   All probands made the test within 5 minutes. They needed significantly less time than the first five probands.

**Second Changes** Following improvements were decided (24.04.18):

1) Activate plug-in
   The context menu entry will be implemented not do be displayed in every section of Cevelop but only in the editor part.

## D. References

[A+08] Motor Industry Software Reliability Association et al. *MISRA C++: 2008: guidelines for the use of the C++ language in critical systems.* MIRA, 2008.

[AUT17] AUTOSAR. *Guidelines for the use of the C++14 language in critical and safety-related systems, Identification No. 839.* AUTOSAR, 2017.

[Cev] Cevelop website. https://www.cevelop.com/.

[Cppa] C++ core guideline. https://github.com/isocpp/CppCoreGuidelines.

[Cppb] C++ core guideline example code. https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md.

[Cppc] C++ core guideline rc-dtor. http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-dtor-release.

[Ecla] Eclipse extension point. https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fextension-points%2Forg_eclipse_ui_editors_markerAnnotationSpecification.html.

[Eclb] Eclipse foundation, activator eclipse documentation. https://www.cct.lsu.edu/~rguidry/eclipse-doc36/org/eclipse/cdt/codan/core/cxx/Activator.html.

[Eclc] Eclipse foundation, annotation types. http://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fextension-points%2Forg_eclipse_ui_editors_annotationTypes.html.

[Ecld] Eclipse foundation, ast eclipse article. http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html.

[Ecle] Eclipse foundation, bundle eclipse documentation. https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fruntime_model_bundles.htm.

[Eclf] Eclipse foundation, cdt eclipse. https://www.eclipse.org/cdt.

[Eclg] Eclipse foundation, codan eclipse documentation. https://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.ptp.pldt.doc.user%2Fhtml%2Fcodan.html.

[Eclh] Eclipse foundation, fragment eclipse wiki. https://wiki.eclipse.org/FAQ_What_is_a_plug-in_fragment%3F.

[Ecli] Eclipse foundation, luna markers. https://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fcore%2Fresources%2FIMarker.html.

[Eclj] Eclipse foundation, marker eclipse documentation. https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2FresAdv_markers.htm.

[Eclk] Eclipse foundation, marker resolution eclipse documentation. https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2FwrkAdv_markerresolution.htm.

[Ecll] Eclipse foundation, oxygen markers. https://help.eclipse.org/oxygen/index.jsp?topic=%2Forg.eclipse.platform.doc.user%2Fconcepts%2Fconcepts-11.htm.

[Eclm] Eclipse foundation, preference page eclipse documentation. https://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fpreferences_prefs_contribute.htm.

[Ecln] Eclipse foundation, target platform preferences documentation. https://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.pde.doc.user%2Fguide%2Ftools%2Fpreference_pages%2Ftarget_platform.htm.

[Eclo] Eclipse help contribution. https://www.eclipse.org/articles/Article-Online%20Help%20for%202_0/help1.htm.

[Eclp] Eclipse table of content. https://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fextension-points%2Forg_eclipse_help_toc.html.

[Eclq] Eclipse workbench. https://help.eclipse.org/oxygen/index.jsp?topic=%2Forg.eclipse.platform.doc.user%2Fconcepts%2Fconcepts-2.htm.

[IBMa] Ibm, binary expressions. https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_73/rzarg/binops.htm.

[IBMb] Ibm eclipse menu. https://www.ibm.com/developerworks/library/os-eclipse-3.3menu/index.html.

[MSD] Msdn, unary expressions. https://msdn.microsoft.com/en-us/library/hetcw0tx.aspx.

[OSG] Wikimedia foundation, osgi wikipedia. https://de.wikipedia.org/wiki/OSGi.

[RB] Kilian Diener Rolf Bislin. Ccgladiator.

[Sta] Tobias Stauber. Iltis documentation.

[Str] Strategy pattern. https://en.wikipedia.org/wiki/Strategy_pattern.

[Voga] Lars vogella, vogella tutorials. http://www.vogella.com/tutorials/EclipseExtensionPoint/article.html.

[Vogb] Vogella eclipse preferences. http://www.vogella.com/tutorials/EclipsePreferences/article.html.

[Wika] Wiki, cdt eclipse. https://wiki.eclipse.org/CDT.

[Wikb] Wiki, cdt eclipse static analysis. https://wiki.eclipse.org/CDT/designs/StaticAnalysis.

[Wikc] Wikimedia foundation, osgi wikipedia. https://de.wikipedia.org/wiki/Eclipse_(IDE).

[wikd] Wikipedia foundation, motor industry software reliability association (misra). https://en.wikipedia.org/wiki/Motor_Industry_Software_Reliability_Association.