

# Stylechecker Plug-in for Cevelop

## Bachelor Thesis

Department of Computer Science  
University of Applied Science Rapperswil

Spring Term 2018

Author:	Zafer Dogan
Advisor:	Thomas Corbat
Project Partner:	Institute for Software (IFS)
External Co-Examiner:	Lukas Felber
Internal Co-Examiner:	Stefan Richter

## Contents

<b>1</b>	<b>Abstract</b>	<b>4</b>
<b>2</b>	<b>Assignment</b>	<b>5</b>
2.1	Supervisor and Expert	5
2.2	Student	5
2.3	Introduction	5
2.4	Goals of the Project	5
2.5	Documentation	6
<b>3</b>	<b>Management Summary</b>	<b>7</b>
3.1	Initial Situation	7
3.2	Approach and Technologies	7
3.2.1	Development Cycles	7
3.2.2	Implemented Technologies	7
3.3	Results	8
3.3.1	Feature Overview	9
3.3.2	UI Design	9
3.4	Conclusion	10
3.4.1	Open Features	10
3.4.2	Potential for Optimization	11
<b>4</b>	<b>Initial Situation</b>	<b>12</b>
<b>5</b>	<b>Analysis</b>	<b>13</b>
5.1	Functional Requirements	13
5.1.1	Stakeholders	13
5.1.2	Use Case Diagram	14
5.1.3	Descriptions in Brief	15
5.1.4	Descriptions in Essential Style	16
5.2	Non-functional Requirements	19
5.2.1	Performance	19
5.2.2	Stability	19
5.2.3	Usability	19
5.2.4	Installability	19
5.2.5	Maintainability	19
5.2.6	Supportability	19
5.2.7	Testability	19
5.3	Styleguide Analysis	20
5.3.1	Introduction	20
5.3.2	Abbreviations	20
5.3.3	Limitations	20
5.3.4	Discussion	21
5.3.5	Naming Conventions	24
5.4	Environement Analysis	32
5.4.1	Landscape Analysis	32
5.4.2	Conceptual Model	36
5.4.3	Class Diagram	37
5.4.4	Sequence Diagrams	39

<b>6</b>	<b>Design and Decisions</b>	<b>40</b>
6.1	Stylechecker Checking . . . . .	40
6.2	Stylechecker Resolution . . . . .	43
6.3	Stylechecker Plug-in User Interface . . . . .	45
6.3.1	User Interface conception . . . . .	45
6.3.2	Current User Interface . . . . .	49
6.4	Persistence . . . . .	53
6.5	RTS Testing . . . . .	55
6.6	Stylechecker and Codan Preferences . . . . .	56
6.7	Performance . . . . .	58
<b>7</b>	<b>Architecture</b>	<b>59</b>
7.1	Dependencies . . . . .	59
7.1.1	Checker and Quickfix . . . . .	59
7.1.2	Plug-in internals . . . . .	60
7.2	Packages . . . . .	61
<b>8</b>	<b>Conclusion</b>	<b>70</b>
8.1	Open Features . . . . .	70
8.2	Potential for Optimization . . . . .	71
8.2.1	General optimizations . . . . .	71
8.2.2	Architectural optimizations . . . . .	71
8.2.3	Rename Refactoring issue . . . . .	71
8.3	Outlook . . . . .	72
<b>9</b>	<b>Glossary</b>	<b>73</b>
<b>10</b>	<b>References</b>	<b>74</b>
<b>11</b>	<b>Appendix A</b>	<b>75</b>
11.1	Project Plan . . . . .	75
11.1.1	Project Overview . . . . .	75
11.1.2	Project organisation . . . . .	76
11.1.3	Management procedures . . . . .	77
11.1.4	Risk management . . . . .	83
11.1.5	Work packages . . . . .	83
11.1.6	Infrastructure . . . . .	84
11.1.7	Quality measures . . . . .	85
11.2	Time Analysis . . . . .	87
<b>12</b>	<b>Appendix B</b>	<b>89</b>
12.1	Installation Guide . . . . .	89
12.2	User Manual . . . . .	90

## 1 Abstract

In design work, a styleguide defines rules and guidelines to be adhered to for achieving a uniform visual look. Styleguides as such also exist for source code. Additionally, to contributing to a visually pleasing codebase, a styleguide increases understandability and readability of source code. In many projects, coding guidelines exist and are part of the development cycle ranging from a small set of rules to a broader compilation of guidelines to be followed. Cevloop is an integrated development environment (IDE) for C++. It is developed and maintained by the Institute for Software (IFS) at HSR. Cevloop itself is based on the Eclipse C/C++ Development Tooling (CDT) project. Currently, Cevloop is lacking the feature of styleguide checking.

The primary goal of the Stylechecker plug-in for Cevloop is providing the capabilities to perform styleguide checking and rule violation reporting. Furthermore, the plug-in is to offer automated resolutions wherever possible. In the scope of this project, the focus lies on covering naming conventions for three predefined styleguides: Google styleguide, Canonical styleguide and Geosoft styleguide. Additionally, users need to be able to create custom styleguides, define rules for it, change existing ones and import/export styleguides for sharing with team members.

The implemented Stylechecker plug-in realizes styleguide checking as well as automated resolutions. To achieve this, the plug-in uses the abstract syntax tree provided by Eclipse CDT. Furthermore, the Stylechecker plug-in uses CDT's code analysis (Codan) framework, an integrated Eclipse CDT plug-in, providing facilities to create markers and resolutions for C++ development. In combination, checking, reporting and initiating resolutions become available. The Stylechecker plug-in allows users to compose custom styleguide rules by joining one or more expressions, defining the language elements to check the rules for and providing custom messages on reporting.

## 2 Assignment

This section discusses the assignment for the bachelor thesis. It contains the subsections "Supervisor and Expert", "Student", "Introduction", "Goals of the project" and "Documentation".

### 2.1 Supervisor and Expert

This bachelor Thesis will be developed for the Institute for Software at HSR internally. It will be supervised by Thomas Corbat (tcorbat@hsr.ch) and Felix Morgner (fmorgner@hsr.ch), HSR, IFS. An expert independent of HSR will examine the thesis and will be present at the final presentation:

- Lukas Felber, Quatico (lukas.felber@quatico.com)

### 2.2 Student

This project is conducted in the context of the module "Bachelor-Arbeit" in the department "Informatik" by

- Zafer Dogan (zdogan@hsr.ch)

### 2.3 Introduction

Cevelop is an Eclipse CDT based integrated development environment (IDE) for C++, implemented and maintained by the Institute for Software at HSR [Cev18]. The IDE is responsible for providing various tools to ease the development of C++ software. For uniform appearance of source code developed in a software project there usually exist coding guidelines to specify conventions regarding the source code to be adhered to within the project. Such guidelines may encompass case sensitivity for names and structure of files. It is desirable that the IDE at hand automatically checks adherence of the given rules. Cdevelop currently lacks the support for such guidelines in general.

### 2.4 Goals of the Project

The goal of this term project is the development of a plug-in for Cdevelop to check coding guidelines for C++ projects. Since different projects or companies specify their own individual rules, such a plug-in needs to be flexible and configurable for the rules in each project. The first task in this bachelor thesis is the analysis of given well-known style guides:

- Google C++ Style Guide [Goo18]
- Boost Library Requirements and Guidelines [Boo18]
- One additional C++ style guide chosen by the student

From the examples above, requirements for the automatic checking and correction of violations of coding guidelines using Stylechecker plug-in have to be derived. The focus in this project lies on naming conventions. This includes:

- Casesensitivity of C++ elements like variables, functions, classes, etc.
- Camel or Pascal casing

Violations of given naming rules must be reported by the plug-in and if possible an automated resolution for such violations shall be provided, by invoking the corresponding refactoring. Eclipse CDT, on which the plug-in will be based on, already provides the corresponding facilities. Predefined (above three) and custom sets of rules must be configurable for the workspace globally and each project individually. Extended goal (optional): Beside configuring and checking of naming conventions, other structural and semantic guidelines can be added to the feature set of Stylechecker. There will be weekly meetings with the supervisors. Additional meetings might be scheduled as required by the students. All Meetings, except for the kick-off meeting, will be prepared by the students with an agenda that is sent to the supervisors at least one day before the meeting. During the meeting the current progress will be presented (What has been done? What has been achieved? How much time did it take? What is planned for the subsequent week?). Decisions of the meeting must be recorded by the students. At the beginning a project plan has to be devised for with milestones for the semester. This plan is used as a guide line to check the progress compared to the estimation. The project plan will be updated according to the actual execution, including time reports and tasks. The students get feedback for accomplished milestones. The final mark will be given based on the eventual results handed-in by the deadline at the end of the semester.

## 2.5 Documentation

This project must be documented according to the guide lines of the "Informatik" department [HSR18a]. This includes all analysis, design, implementation, project management, etc. sections. All documentation is expected to be written in English. The project plan also contains the documentation tasks. All results must be complete in the final upload to the archive server [HSR18b]. Two copies of the documentation must be handed-in:

- One in color, two-sided
- One in B/W, single-sided

---

Thomas Corbat

---

Zafer Dogan

### 3 Management Summary

The management summary gives a short overview over the entire project. It discusses the "Initial Situation", the "Approach and Technologies" used and the "Results" of the project.

#### 3.1 Initial Situation

A styleguide is set of rules and guides to be adhered to in order to achieve a uniform look and feel. The terminology styleguide is most commonly associated with design work, where similarly, it is a guideline to establish rules and guides like font size for titles, subtitles and headings, colors for buttons etc. in order to achieve a uniform look and feel across the entirety of the design. For source code, styleguides exist as well. Here they govern the look and feel of the source code by defining guides and rules for naming, casing information and formatting. A uniform look aids in readability and understandability of source code, as for example constant casing like `CONST_VARIABLE` inform the developer without knowing the definition or declarations that this is a constant, immutable variable. It's cues like these which help a developer in analyzing the code more quickly and more efficiently and therefore increases his/her productivity.

Cevelop is an integrated development environment (IDE) for C/C++ development. It is based on Eclipse C/C++ Development Tooling (CDT) and developed at the Institute for Software (IFS) at the HSR. It offers a wide range of useful plug-ins easing programming and developing in C/C++. Currently, Cdevelop is lacking the feature to check for styleguide violations. The key goal of this project is to develop a plug-in providing the Cdevelop IDE with this functionality. Additionally, the plug-in is to offer automated resolutions wherever possible. Furthermore, an import and export feature for styleguides is desired, so that a styleguide can be quickly shared among team members.

#### 3.2 Approach and Technologies

As the plug-in is developed for Cdevelop IDE, which is a tool based on Eclipse CDT, the approach in developing the plug-in is to use Eclipse's extension points. Eclipse in itself is a highly customizable framework, offering nearly all components (all except core) to be extended via plug-ins. In order to realize styleguide checking and resolution, the developed plug-in relies heavily on the Codan plug-in, an Eclipse CDT integrated code analysis plug-in, which offers functionality to report problems, mark them and provide resolution logic in the editor.

##### 3.2.1 Development Cycles

The project saw two major development cycles. In the first cycle, the necessary setup of plug-in dependencies and hooks were the major focus. Additionally, the integration of Eclipse CDT internal functionality, like rename refactoring, needed to be developed. In the second cycle, the focus shifted to the Stylechecker plug-in internals, like domain elements and UI. In combination, checker and resolution logic were implemented.

##### 3.2.2 Implemented Technologies

Implemented technologies include the following: Eclipse CDT, Codan plug-in extension, JSON and Java 8. The Eclipse CDT was used to delegate renaming logic to existing refactoring structures. Codan plug-in served as basis for editor marker generation, line marking and resolution offering. JSON was used as the main format for persisting plug-in settings. Java 8 was the development language.

### 3.3 Results

The developed Stylechecker plug-in provides styleguide checking and automated resolutions for naming conventions. Styleguide checkings can be applied to the most important C++ language elements such as variables, functions, classes etc. Furthermore, it allows import and export of styleguides to be shared among team members.

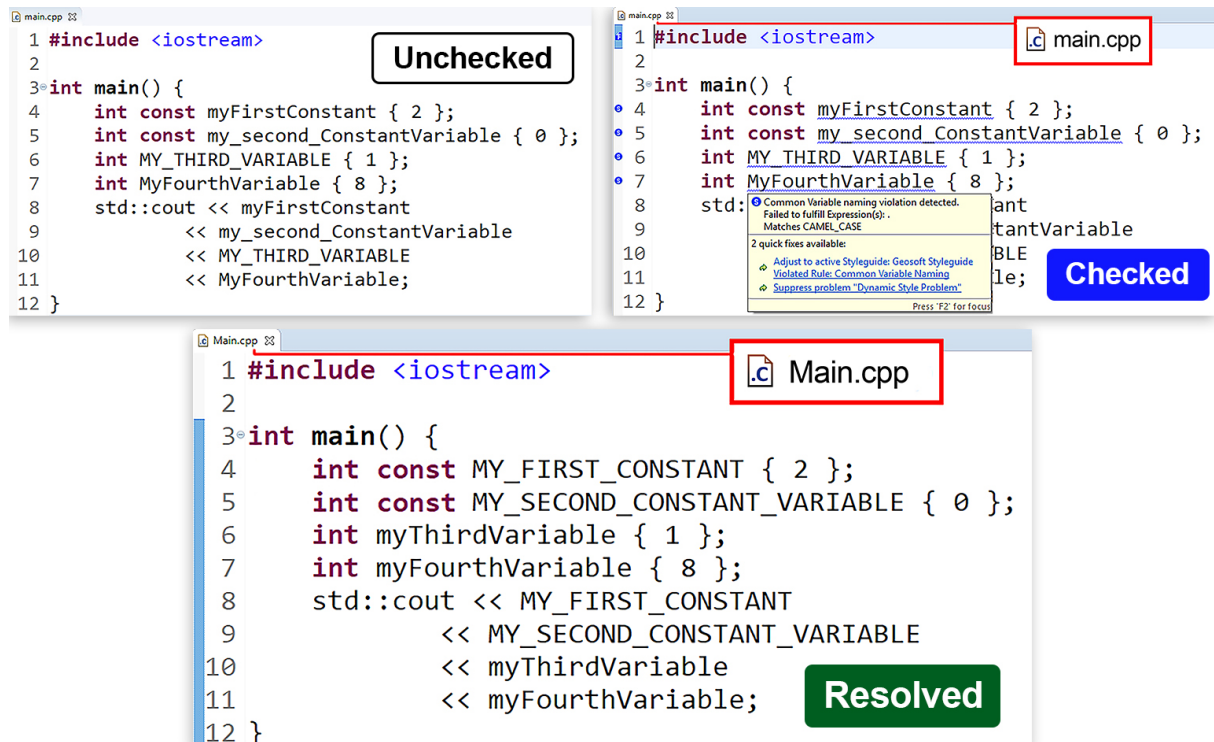


Figure 1: Unchecked, Checked and Resolved States of Stylechecker plug-in

The figure above shows the three states of the Stylechecker plug-in. In the unchecked state, the Stylechecker plug-in is turned off. In the checked state, the Stylechecker plug-in is turned on and reports styleguide violations according to the currently active styleguide. In the resolved state, the violations have been fixed via Quickfix operations and no violation is being reported anymore, the code at this point is styleguide compliant.

The Stylechecker plug-in offers a great flexibility in configuring a styleguide. A user can choose from three predefined styleguides that come with the plug-in - which are Google, Canonical and Geosoft C++ styleguides - or create his own. The plug-in preferences provide the necessary UI for these kind of operations.



### 3.3.1 Feature Overview

In the table below, the implemented features are listed.

Feature	Description
Styleguide creation, customization and removal	A styleguide definition includes rules definition and rules definition includes expression definition. This way, multiple rules with complex expression logic can be created. Rules can be limited to what concepts to check them on.
Rule and Grouping filtering	Rules and Groups can be filtered in the preferences window, in order to more quickly find rules to edit and customize.
Styleguide, Rule and Grouping Enable/Disable	A styleguide can be enabled or disabled entirely. A styleguide can be partially enabled/disabled via enabling/disabling Groupings it contains. A Grouping can be partially enabled/disabled via Rule enabling or disabling Rules it contains. This way, an existing styleguide can be flexibly adjusted to project needs.
Styleguide Checking and Reporting	Naming conventions are checked and reported, when a violation is present.
Automated Resolutions	Styleguide violations have automated resolutions. When an automated resolution fails, user input is requested.
Import & Export of Styleguides	Styleguides can be imported and exported to be shared among team members.
Codan File & Folder Inclusion / Exclusion	As the Stylechecker plug-in extends the Codan plug-in, the checker can be limited to be applied to certain files and folders via Codan settings.
Workspace & Project Scopes	On project level, the Stylechecker plug-in can be configured in three ways. To use workspace settings, to reference workspace settings or to use project specific settings.

Table 1: Feature overview table

### 3.3.2 UI Design

The design of the UI was heavily influenced by the Eclipse environment. As the plug-in had to fit within Eclipse context, it influenced the choice for the UI framework. Most of the UI was implemented with SWT. The Stylechecker settings appear within the Eclipse settings. Also, because the Stylechecker plug-in extends the Codan plug-in, it was used as a reference point.

### 3.4 Conclusion

The target features within the scope of the projects were able to be implemented. The Stylechecker plug-in can check styleguides, report them and offer automated resolutions, wherever possible. But due to time constraints and project group size, some features were implemented in the most basic way, such as File Inclusion / Exclusion feature, which basically is just a reroute to the Codan plug-in. Along the way, multiple new ideas for new features came up as well. The following sections will discuss therefore open features that could be implemented in case of a continuation of development and also will review some potentials for optimization.

#### 3.4.1 Open Features

Feature	Description
Bulk solve multiple problems	Feature to solve multiple problems at the same time.
Advanced UI Mode	An advanced UI mode feature, which would allow to use more complex settings and access more complex concepts to check against. This way new users wouldn't be overwhelmed with the amount of settings and advanced users could customize the Stylechecker to their hearts content.
Limitation of checked concepts on Expression Group basis	A limitation on basis of Expression Groups on which concepts the Expressions should be checked on would allow for more complex rule definitions and to perform a resolution in one step.
Settings versioning and auto transformation	As the plug-in might change in future, a feature that would detect changes in the settings structure and offer automatic transformation to the new structure. This way older styleguide settings wouldn't be invalidated with newer versions of the plug-in.
Special code markers for styleguide checking enable/disable	Like @Supress annotation in Java, special comments or markers in code with which style checking for a certain section of a code could be enforced or forced to skip.
AI assisted resolution	It would be interesting to have a resolution type that would be a trained AI that would determine how to best convert a string to a target casing by analyzing multiple code sources. This would for example allow for semantically correct transformations, considering the meaning of the words in the resolution procedure.
String to Regex converter	Basically the reverse procedure with regex matching. A feature that would allow a user to define a target regex, which would be used as the basis for a resolution. This way resolutions could be defined in a dynamic way.

Table 2: Open features overview table

### 3.4.2 Potential for Optimization

There is still a lot of potential for optimization in the Stylechecker plug-in. First optimization source that comes to mind is the Expressions UI part in the Rule dialog, which at the moment is a little bit overloaded with text. It could be improved by using visual cues to indicate for example whether a group matches ALL or ANY with an icon rather than with text. Optimizations like these would make reading the Expressions easier. Additionally, Expression Groups and Expressions use the same tree view, which results in many fields being empty for the Expression Groups as those settings just simply don't exist on that level. A different approach in visualizing the Expressions and Expression Groups is probably necessary if the Stylechecker is to be developed further in the future.

Another source of optimization would be a consolidation of all features within the Stylechecker plug-in itself without having to reference the Codan plug-in. There are ways to initiate Codan procedure in order to, for example, enable or disable a checker from the outside, but the implementation can get tricky due to hard to control update procedure that are controlled by the Codan UI. But a consolidation would make using the Stylechecker plug-in more convenient, even though most of the settings are one time settings anyway.

Furthermore, some UI elements still could need some proper refactoring work as some of them are still bloated due to limited time. SettingsComposite for one is still a huge block of code.

## 4 Initial Situation

A styleguide is set of rules and guides to be adhered to in order to achieve a uniform look and feel. The terminology styleguide is most commonly associated with design work, where similarly, it is a guideline to establish rules and guides like font size for titles, subtitles and headings, colors for buttons etc. in order to achieve a uniform look and feel across the entirety of the design. For source code, styleguides exist as well. Here they govern the look and feel of the source code by defining guides and rules for naming, casing information and formatting. A uniform look aids in readability and understandability of source code, as for example const casing like `CONST_VARIABLE` inform the developer without knowing the definition or declarations that this is a constant, immutable variable. It's cues like these which help a developer in analyzing the code more quickly and more efficiently and therefore increases his/her productivity.

Cevelop is an integrated development environment (IDE) for C/C++ development. It is based on Eclipse C/C++ Development Tooling (CDT) and developed at the Institute for Software (IFS) at the HSR. It offers a wide range of useful plug-ins easing programming and developing in C/C++. Currently, Cevelop is lacking the feature to check for styleguide violations. The key goal of this project is to develop a plug-in providing the Cevelop IDE with this functionality. Additionally, the plug-in is to offer automated resolutions wherever possible. Furthermore, an import and export feature for styleguides is desired, so that a styleguide can be quickly shared among team members.

As the Stylechecker plug-in builds on a previous work, sometimes it might be referred to as Cstylechecker. In this context, Stylechecker and Cstylechecker designate the same plug-in.

## 5 Analysis

This section discusses the requirements analysis for the Stylechecker plug-in. Furthermore a "Styleguide Analysis" can be found, discussing the chosen styleguides and reflecting about the implementation of them in the plug-in. It also includes the Environment analysis, analyzing the context for the plug-in to be developed in and the necessary domain elements required; including a look at important sequences in the plug-in.

### 5.1 Functional Requirements

The functional requirements contains the analysis of "Stakeholders", the Use Cases as "Descriptions in Brief" and "Descriptions in Essential Style".

#### 5.1.1 Stakeholders

The Stakeholders section discusses the primary actor and the remaining stakeholders and their interests.

##### 5.1.1.1 Primary Actor

C++ Developers

##### 5.1.1.2 Stakeholders and Interests

There are three relevant stakeholders of the Stylechecker plug-in: C++ Developers, C++ Project Managers and C++ Team Leaders.

#### C++ Developers

C++ Developers want to use existing or custom defined styleguides in their projects without much hassle. C++ Developers want styleguide violations to be detected and if possible, offered an option to automatically fix styleguide issues. If auto resolution isn't possible, C++ Developers want a convenient way to resolve the issue without much hassle so that they won't be impaired in their productivity. C++ Developers also would like to decide for themselves when styleguide checking should be performed or not and to which files and folders they should be applicable to.

#### C++ Project Managers

C++ Project Managers want to define and export styleguidelines, so that their developers can adhere to a chosen styleguide. Preferable, C++ Project Managers would like to choose from already existing styleguides to apply to their projects if a certain styleguide (like f.ex. Google C++ Styleguide) matches their styleguide needs. Furthermore, C++ Project managers would like to have a preliminary report of styleguide violations, so that they can effectively discuss styleguide issues with their team leaders.

#### C++ Team Leaders

C++ Team Leaders want to be able to define where styleguide checking is applied to, so that they can avoid unnecessary styleguide checking for f.ex. external libraries. Furthermore, C++ Team Leaders would like to generate a report of styleguide violations, so that they can distribute the work to fix styleguide issues more effectively. C++ Team leaders also want to modify existing or custom defined styleguides, in order to adjust them to the needs of their team.

### 5.1.2 Use Case Diagram

The use case diagram displays an overview of the relationships between stakeholders and the use cases.

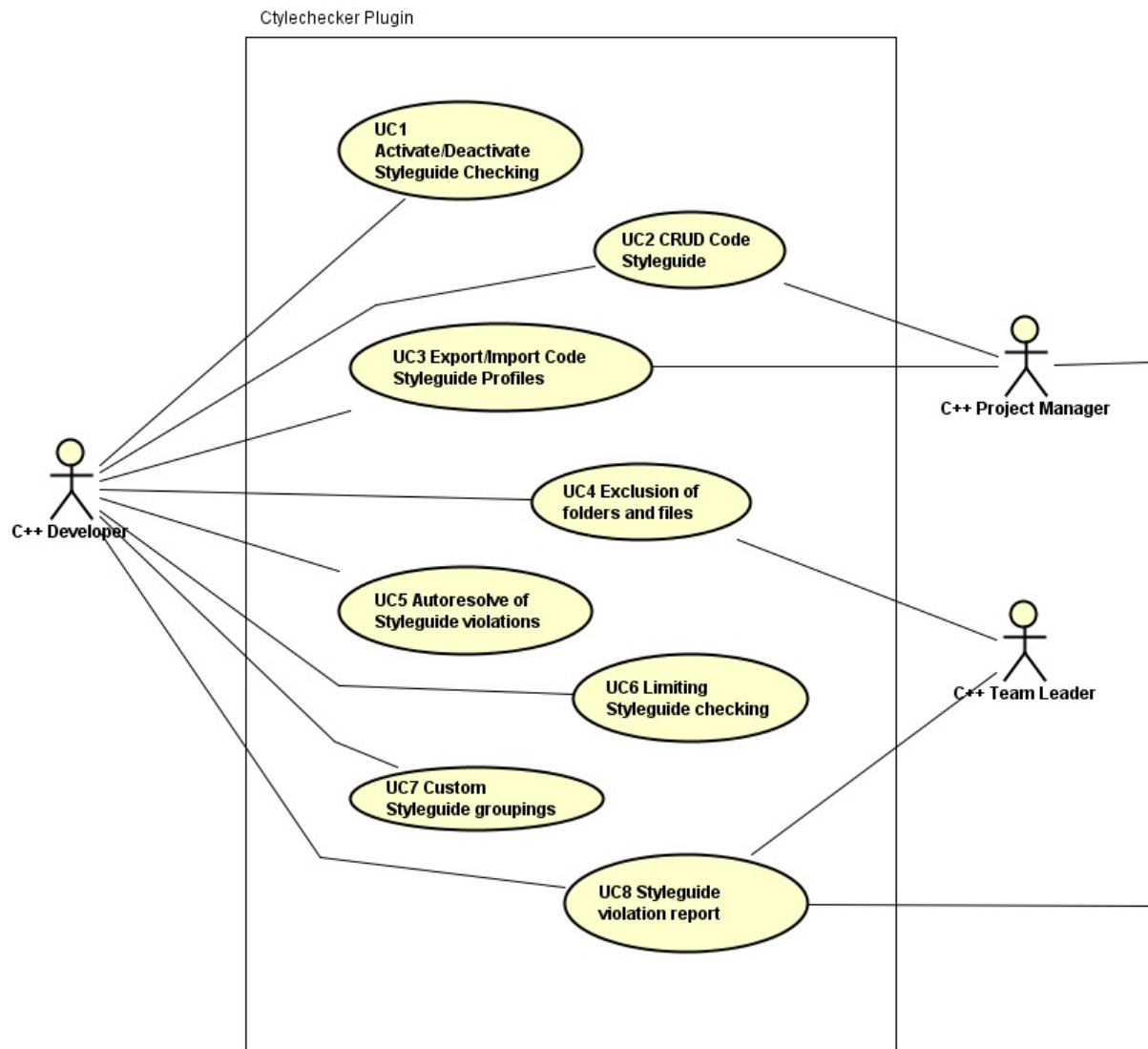


Figure 2: Use Case Diagram Stylechecker plug-in

### 5.1.3 Descriptions in Brief

All use cases in this section will be held in brief format and serve as a general overview.

#### 5.1.3.1 UC1 Activate/Deactivate Code Styleguide Checking

A C++ Code Styleguide can be activated and deactivated at will. When active, the Styleguide Checker will notify about Styleguide violations. A C++ Code Styleguide can be applied to the entire file, to the entire workspace or a specific line in the code. Styleguides can be activated on project basis or workspace basis.

#### 5.1.3.2 UC2 CRUD Code Styleguide

C++ Styleguides can be configured via preferences in the IDE. The User can quickly change between pre-defined styleguides and apply them on the fly. Additionally, a User can define a custom set of Styleguide's to be checked against. A User can save as many custom profiles as he wishes. Profiles are persisted within the IDE. The User can delete custom Code Styleguides when desired.

#### 5.1.3.3 UC3 Export/Import Code Styleguide Profiles

The User can save and export custom Code Styleguide's to share with team members and also import profiles shared by team members. Imported profiles can be edited like any other custom profiles. And like any other custom profiles, imported profiles can be used project-wide or workspace wide.

#### 5.1.3.4 UC4 Exclusion of folders and files

The C++ Styleguide checking can be activated/deactivated for specific files and folder structures. This is especially relevant when working with external dependencies.

#### 5.1.3.5 UC5 Autoresolve of Styleguide violations

Where appropriate and possible, Styleguide violations can be auto resolved with a quickfix. Where not appropriate, the user will have the option to enter a name manually to be refactored to. Auto refactoring can be applied per file or project-wide.

#### 5.1.3.6 UC6 Limiting Styleguide checking

A user can limit the type of styleguide checking that is performed according to specifically coarsely defined groupings like Naming Conventions or Comment Conventions.

#### 5.1.3.7 UC7 Custom Styleguide groupings

A user can define coarse groupings and assign new checker rules within those groupings. This way any type of C++ styleguide can be adapted to work with the plug-in and currently existing differences in grouping can be targeted with this feature. Alternatively, the user can choose from predefined groupings, a groupings catalog, to group custom checker rules. For example Naming Conventions is a prime example for a predefined grouping.

#### 5.1.3.8 UC8 Styleguide violation report

A user can run a styleguide checking analysis and produce a report. This might be especially interesting if a project contains a lot of violations and a team needs to properly distribute refactoring tasks. The output will be generated as an HTML.

## 5.1.4 Descriptions in Essential Style

This section analyses the most important use cases in essential style.

### 5.1.4.1 UC1 Activate/Deactivate Code Styleguide Checking

#### Preconditions

- C++ Project is open

#### Description

A C++ Code Styleguide can be activated and deactivated at will. When active, the Styleguide Checker will notify about Styleguide violations. A C++ Code Styleguide can be applied to the entire file, to the entire workspace or a specific line in the code. Styleguides can be activated on project basis or workspace basis.

#### Postconditions

- Styleguide violations are shown or not shown

#### Main Scenario

1. Open Project Preferences
2. Open Stylechecker register
3. Enable Stylechecker plug-in
4. Save/Apply Changes

#### Alternatives

- 3a Disable Stylechecker plug-in



### 5.1.4.2 UC4 Exclusion of folders and files

#### Preconditions

- C++ Project is open
- Project has at least one file or folder
- Cstylechecking is activated

#### Description

The C++ Styleguide checking can be activated/deactivated for specific files and folder structures. This is especially relevant when working with external dependencies.

#### Postconditions

- Styleguide violations are not shown for excluded files and folders

#### Main Scenario

1. Open Project Preferences
2. Open Stylechecker register
3. Activate File and Folder Exclusion
4. Add files and folders to exclusion list
5. Save/Apply Changes

#### Alternatives

- 3a Deactivate File and Folder Exclusion (when deactivated, all files and folders are checked)
- 4a Remove files and folders from exclusion list

### 5.1.4.3 UC5 Autoresolve of Styleguide violations

#### Preconditions

- C++ Project is open
- C++ Code is present
- Violation is autoresolveable

#### Description

Where appropriate and possible, Styleguide violations can be autoresolved with a quickfix. Where not appropriate, the user will have the option to enter a name manually to be refactored to. Autorefactoring can be applied per file or project-wide.

#### Postconditions

- Code is refactored to proper style
- Code is working (i.e. refactoring doesn't brake Code)

#### Main Scenario

1. Navigate to Styleguide violation line
2. Call Quickfix operation
3. Execute Autoresolve in file

#### Alternatives

- 1a Call context menu in file
  - 1aa Execute Autoresolve styleguide violations in file
  - 1ab Execute Autoresolve styleguide violations in project
- 1b Call context menu on project
  - 1ba Execute Autoresolve styleguide violations
- 2a Click violation marker
- 3a Execute Autoresolve in project

## 5.2 Non-functional Requirements

This section has a look at the quality attributes the Stylechecker plug-in needs to adhere to.

### 5.2.1 Performance

- Styleguide resolutions (singular refactoring) need to be executed within:
  - three seconds in singular file refactorings
  - five seconds on entire workspace refactorings

in small to middle sized projects.

### 5.2.2 Stability

- Refactorings cannot break the code. Any autoresolve or styleguide fixing refactoring needs to leave the code in a working state.
- If a refactoring breaks the code, the code can be easily restored to its working state via an Undo operation

### 5.2.3 Usability

- Settings UI: A new custom rule can be defined within three steps. Create, Write and Save
- Settings UI: Imported styleguide profiles can be used without additional configuration

### 5.2.4 Installability

- The plug-in needs to be able to be installed via an Update site. The Update site files and folders need to conform to Eclipse standards, so that it can be quickly installed via 'Install new Software'.

### 5.2.5 Maintainability

- The plug-in needs to be easily extendable without significant architectural adjustment in case additional predefined styleguides are added in the future

### 5.2.6 Supportability

- The plug-in shows helpful information respective to settings
- The plug-in shows meaningful error messages when a refactoring didn't succeed

### 5.2.7 Testability

- The plug-in Helper components don't have more than one additional external dependency

### 5.3 Styleguide Analysis

The styleguide analysis records the naming conventions for the Google, Canonical and Geosoft styleguides and was used as an aid in the development of the Stylechecker plug-in. The styleguides are grouped into categories and recorded in table form to better illustrate the differences. The styleguide analysis also includes an analysis of the chosen styleguides and a reflection about styleguides in general in the context of this plug-in

#### 5.3.1 Introduction

The categories for the naming conventions are chosen according to Google's C++ Styleguide and will serve as a general guideline for grouping. The direct comparison of the styleguides in table form can be found at the end of the styleguide analysis section. Before the direct comparison of the differences in naming conventions, the three used styleguides will be evaluated against one another and in general an analysis will be made, to what extent the styleguides go and to what extent they could have gone based on the C++ language.

#### 5.3.2 Abbreviations

Here is a list of abbreviations used in the document

- D: Description
- E: Example

#### 5.3.3 Limitations

In all of the three Guidelines analyzed here, limitations in respect to implementability in regards to project scope and possibility apply. An example of limitations are the following from the Geosoft Styleguide definition, which says under specific styleguides:

- The terms get/set must be used where an attribute is accessed directly.
- Plural form should be used on names representing a collection of objects.
- The prefix n should be used for variables representing a number of objects.
- The prefix is should be used for boolean variables and methods.

While not all of these limitations are beyond implementability, they all require major architectural extensions (i.e. AST Traverse needs to pickup variable types, which need to be parsed and made editable etc.). The current architecture supports extensions such as these (i.e. qualifiers are supported), but the implementation of types into the resolution procedure may be well beyond the project scope and the work a single student may accomplish during a three months period.

### 5.3.4 Discussion

This section has a look at the styleguide definitions and includes a discussion and analysis of them.

#### 5.3.4.1 Definition

First off, it is prudent to establish a common understanding of what exactly a styleguide is and what it entails in the scope of this project. The term styleguide isn't one specific to the IT world and can be found in various other fields, most prominently in design related works. A styleguide is a guide to a uniform look and feel of a product; it defines rules and limitations to be followed in order to achieve a common look. As the primary medium in programming is plain text, styleguide in respect to programming and more specifically to source code refers to a common look and feel of source code in order to enhance readability and understanding; it does not refer to visual styling like spacing, kerning etc. as may be the case in design related styleguides. Styleguides in the IT world are target-oriented set of rules to enhance understanding; it tries not to be visually pleasing. For example a common widespread "style" in the programming world is the use of all caps lettering for constant variables, like `const int PI_NUMBER{3.1415...}`. As programmers we immediately know by only reading the name and not even looking at the type specification that this variable is a constant variable. It's cues like these that improve readability and in effect, understanding of code. In the same way we give our functions meaningful names to indicate what it does and with it, make it easier for other programmers to understand our code, by adhering to styleguides we can ease the analysis of code and increase productivity.

In the scope of this project with styleguide we refer to the set of rules that guide and contribute to a common look and feel of C++ Code.

#### 5.3.4.2 Google, Geosoft and Canonical - A comparison

Before we begin with a comparison, the three chosen styleguides need to be elaborated in respects to from which companies they actually are. As Google is a worldwide well-known company, it needs no further explanation. Their C++ styleguide is one of the most known examples for a styleguide for C++. Canonical is a British Linux distributing company and their styleguide is very similar to Google's C++ styleguide definition in look and structure, but deviates in many points from Google's guidelines significantly, qualifying it as an additional predefined styleguide for which naming conventions are to be provided. Canonical additionally serves as replacement for the Boost Library styleguide - as defined in the project goals - as the Boost Library is poorly defined and in frictions, it was deemed it unfit as a predefined styleguide to be implemented in the scope of this project. Finally, there is the Geosoft styleguide, a predefined styleguide chosen by the student. Geosoft styleguide belongs to the company "Geotechnical Software Services" and differs in its presentation and structure from Google and Canonical. But in case of naming conventions, many overlapping areas are found, which is why it was chosen as the last of the three predefined styleguides to implement.

All three styleguides follow a certain structure and have different sections, like Header Files, Scoping, Classes, Naming and the like. Not all sections define a clear style related rule, for example, Google styleguide has a section detailing that the use of the friend keyword is OK and should be used within reason. Now for a programmatic application of a styleguide checking it is hard to determine what is within reason and what not. Other deviations exist also among the styleguides, e.g. Geosoft has styleguide definitions that go like: Use calculate in function name, if the function is performing some calculation. This kind of styleguide is unique to Geosoft and can't be found in Google or Canonical. In respect to naming rules, a lot of commonalities can be observed.

For the scope of the project, the most relevant part from all of the styleguides is the sections related to naming. Other sections, such as Classes e.g. detail in some extend dos and don'ts, like to use explicit keyword on single parameter Constructors or to provide Copy Constructors only if needed and if not, to disable them explicitly with = delete. As the naming sections show commonalities, they also serve as a good base for the Stylechecker plug-in for development.

Styleguides do have explicit and suggestive parts, a tool can enforce the explicit parts, with some Machine Learning magic maybe even cover suggestive parts to some extent, but all in all, not enforce it entirely. This is a limitation to be aware of. But this doesn't mean a tool for style checking is pointless, it serves as a safety net to cover in the very least a certain amount of the styleguide.

#### 5.3.4.3 The good and the bad

The good parts of all the chosen styleguides are the proper sectioning (which not all styleguides always adhere to, e.g. the Boost Library styleguide is scattered) and especially the section related to the naming conventions as they are the most specific and clearly defined. There are no ambiguities and are mostly illustrated with examples. The naming conventions cover the most general concepts in C++, like classes, local variables and functions and can by themselves contribute in a significant way to a common look as they are the most used elements in C++ programming.

The bad part that all the styleguides share is that they try to be more than styleguides at times, especially the Google Styleguide is a good example in this respect. It has a section on how to use the Boost library (i.e. only approved libraries). The usage of a certain library has not much to do with a common look at feel of the code base and as such should not be part of the styleguide. Furthermore all three styleguides try to generalize wherever possible, which is why for example the same format can be defined for all types of functions. In the context of a purely textual description without any tools to check, this approach, of course, makes sense, because the more rules a styleguide defines, the harder it becomes to implement it correctly.

#### 5.3.4.4 Postanalysis

In all fairness, all three chosen styleguides do a relatively good job on defining clear styleguides to achieve a common look of the codebase. They cover naming guides and also formatting guides, both of which affect significantly the look and feel of the code. All of the styleguides do one thing in common: they don't cover or distinguish more specific concepts of C++ like UDL's or template variables. In the styleguides of Google and Canonical I've seen a lack of discussion of parameters in general. Geosoft suggests naming a parameter as the type name itself, but that's it.

And that's where we have to ask ourselves, for which concepts in the C++ language styleguides make sense and at what point do we reach the limit? Generalizations with minor variations can make more sense if we are to achieve a uniform source code look. Therefore that's a basis from where to look beyond. In the context of generalizations, it is better to choose one casing for a general group of names, e.g. Pascal Casing for all types of functions makes more sense than different casings for the different function types as defining a different format for all different variations of functions might clutter the code in such a way that a uniformity couldn't be recognized in the source code anymore. This way (one casing for different types of functions) a function can be quickly identified by its casing only while keeping the visual uniform look. Could we designate member functions, free functions etc. with something other

than casing information? Yes, we could. For this purpose prefixes or suffixes could play a useful role. Although not defined in any styleguides, something along the following could help distinguish function types better:

Type	Example
Free functions	<code>void FPrintAll();</code>
Member functions	<code>void MPrintAll();</code>
Template functions	<code>void TPrintAll();</code>

Table 3: Prefix variations on function types

But in the same way, it could also make reading the functions harder as the names get polluted. Most of the function types are clear by context use alone and in most cases a special designation doesn't really add to the readability nor understanding of the code.

As we can see, the generalization approach of all of the predefined styleguides has some rationale behind it. But what about covered language elements? All three styleguides cover the most common C++ language elements like variables, function, classes etc., but all three also lack distinguished definitions for templates, be they template variables, functions or else. It would be wrong to say they're entirely absent, but usually the suggested norm is to follow the actual C++ norm, i.e. a "T" for a template parameter. Most of the time template based definitions fall into the same generalizations like any other, i.e. template functions are named like normal functions, template variables are named like normal variables etc. One C++ language element that isn't covered by all three styleguides, but which would make sense to be covered, are UDLs, i.e. user defined literals. A quick example of a UDL:

```
int operator"" _KM(int param){...}
int distance_to_school = 2_KM;
```

User defined literals provide us meaningful literals we can attach to values to make them more telling of what kind of value they're representing. As such their textual representation can influence how effectively we interpret them. Therefore a styleguide definition for UDLs makes sense. In the scope of this project UDLs weren't included for naming conventions, reasons being:

- 1) They aren't defined in the three predefined styleguides
- 2) The Abstract Syntax Tree (AST) injected into the Codan (Code Analysis) plug-in Checkers by Eclipse C/C++ Development Tooling (CDT) don't pick the user defined literal as an IASTName by itself, it registers it with the operator keyword.

Beyond this, most of the language elements or at least the most important and visible ones are already covered by the styleguides.

With tools like the Stylechecker plug-in a more distinguished approach to C++ language elements could make sense, as with it a tool would aid in adhering to the styleguide and thus freeing up developers from having to remember specific parts of the styleguide. But then again, an approach like in Table 3.

### 5.3.5 Naming Conventions

This section discusses the naming conventions for the three chosen styleguides and lays them out in a tabular fashion for easier comparison. It is important to note that not all language element variations are listed simply because in most cases a styleguide definition applies to all variations, like on type of casing for functions, be they member, free or template.

#### 5.3.5.1 File Naming

This section compares the naming conventions for "Files".

##### File Body Naming

	Google[Goo18]	Canonical[Can18]	Geosoft[Geo18]
<b>D</b>	Filenames should be all lowercase and can include underscores (_) or dashes (-). An underscore is preferred.	Same as Google	A class should be declared in a header file and defined in a source file where the name of the files match the name of the class. Names representing types must be in mixed case starting with upper case.
<b>E</b>	<code>my_useful_class.h</code> <code>class MyUsefulClass...;</code>	Same as Google	<code>MyClass.h</code> <code>class MyClass...;</code>

Table 4: File Body Naming

##### File Ending Naming

	Google	Canonical	Geosoft
<b>D</b>	C++ files should end in .cc and header files should end in .h. Files that rely on being textually included at specific points should end in .inc (see also the section on self-contained headers)	C++ files should end in .cpp and header files should end in .h.	C++ header files should have the extension .h (preferred) or .hpp. Source files can have the extension .c++ (recommended), .C, .cc or .cpp.
<b>E</b>	<code>my_useful_class.cc</code> <code>my_useful_class.h</code>	<code>my_useful_class.cpp</code> <code>my_useful_class.h</code>	<code>MyClass.c++</code> <code>MyClass.C</code> <code>MyClass.cc</code> <code>MyClass.cpp</code> <code>MyClass.h</code> <code>MyClass.hpp</code>

Table 5: File Ending Naming



### 5.3.5.2 Type Naming

This section compares the naming conventions for "Types" like classes and structs.

#### Class Naming

	Google	Canonical	Geosoft
<b>D</b>	Type names start with a capital letter and have a capital letter for each new word, with no underscores	Type names start with a capital letter and have a capital letter for each new word, with no underscores	Names representing types must be in mixed case starting with upper case
<b>E</b>	<code>class MyClass...</code>	<code>class MyClass...</code>	<code>class MyClass...</code>

Table 6: Class Naming

#### Struct Naming

	Google	Canonical	Geosoft
<b>D</b>	Same as class.	Same as class.	Same as class.
<b>E</b>	<code>struct MyStruct...</code>	<code>struct MyStruct...</code>	<code>struct MyStruct...</code>

Table 7: Struct Naming

### 5.3.5.3 Variable Naming

This section compares the naming conventions for "Variables".

#### Common Variable Naming

	Google	Canonical	Geosoft
<b>D</b>	Contains three areas of application: Common Variable names, Class Data Members and Struct Data Members	Variable names are all lowercase, with underscores between words. Class member variables follow this convention	Mixed case, starting with lowercase. Private class members with suffix <code>_</code> . Generic variables, such as parameters should have the same name as their type.
<b>E</b>	<pre>std::string table_name std::string tablename</pre>	<pre>std::string table_name std::string tablename</pre>	<pre>std::string tableName  Private Class member: std::string tableName_  Generic Variable (Param): void myFunc(Database database)</pre>

Table 8: Common Variable Naming

#### Constant Variable Naming

	Google	Canonical	Geosoft
<b>D</b>	Variables declared <code>constexpr</code> or <code>const</code> , and whose value is fixed for the duration of the program, are named with a leading "k" followed by mixed case	Name constants like other variables, using all lowercase, with underscores between words	Named constants (including enumeration values) must be all uppercase using underscore to separate words.
<b>E</b>	<pre>const int kDaysInAWeek = 7</pre>	<pre>auto const match = map.find(value); int const width1024</pre>	<pre>const int MAX_ITERATIONS = 25; const std::string COLOR_RED = "#ff0000"; const float PI = 3.1415;</pre>

Table 9: Constant Variable Naming

**Parameter Naming**

	<b>Google</b>	<b>Canonical</b>	<b>Geosoft</b>
<b>D</b>	No guide defined.	No guide defined.	Parameter names should be the same as the typename.
<b>E</b>			<code>void openConnection(Database database)...</code>

Table 10: Parameter Naming

### 5.3.5.4 Function Naming

This section compares the naming conventions for "Functions".

#### Regular functions

	Google	Canonical	Geosoft
<b>D</b>	Regular functions have mixed case; accessors and mutators may be named like variables.	Regular functions, accessors, and mutators are all lowercase, with underscores between words.	Names representing methods or functions must be verbs and written in mixed case starting with lower case.
<b>E</b>	AddTableEntry() DeleteUrl() Optional for getters and setters: int count() void set_count(int count)	add_table_entry() delete_url() open_file_or_die()	getName() computeTotalWidth()

Table 11: Regular functions

#### Member functions

	Google	Canonical	Geosoft
<b>D</b>	Accessors and mutators may be named like variables. No specific guideline for other types of member functions. Treated as regular functions.	Regular functions, accessors, and mutators are all lowercase, with underscores between words.	Names representing methods or functions must be verbs and written in mixed case starting with lower case.
<b>E</b>	Optional for getters and setters: int count() void set_count(int count)	add_table_entry() delete_url() open_file_or_die()	getName() computeTotalWidth()

Table 12: Member functions

**Free functions**

	<b>Google</b>	<b>Canonical</b>	<b>Geosoft</b>
<b>D</b>	No guide defined, treated as regular functions.	No guide defined, treated as regular functions.	No guide defined, treated as regular functions.

Table 13: Free functions

**Template functions**

	<b>Google</b>	<b>Canonical</b>	<b>Geosoft</b>
<b>D</b>	No guide defined, treated as regular functions.	No guide defined, treated as regular functions.	No guide defined, treated as regular functions.

Table 14: Template functions

**5.3.5.5 Namespace Naming**

This section compares the naming conventions for "Namespaces".

	<b>Google</b>	<b>Canonical</b>	<b>Geosoft</b>
<b>D</b>	Namespace names are all lower-case.	Same as Google	Same as Google
<b>E</b>	websearch::index websearch::index_util	-	model::analyzer io::iomanager common::math::geometry

Table 15: Namespace Naming

### 5.3.5.6 Enumerator and Enumeration Naming

This section compares the naming conventions for "Enumerators" and "Enumerations".

#### Enumeration Naming

	Google	Canonical	Geosoft
<b>D</b>	Enumeration names are Pascal Case. They correspond to type naming.	Enumeration names are Pascal Case. They correspond to type naming.	No guide defined.
<b>E</b>	<pre>enum UrlTableErrors { ... }; enum AlternateUrlTableErrors { ... };</pre>	<pre>enum class UrlTableErrors { ... };</pre>	

Table 16: Enumeration Naming

#### Enumerator Naming

	Google	Canonical	Geosoft
<b>D</b>	Enumerators (for both scoped and unscoped enums) should be named either like constants or like macros: either kEnumName or ENUM_NAME	Enumerators should be named like member variables: out_of_memory, enclosed within an enum class.	Named constants (including enumeration values) must be all uppercase using underscore to separate words.
<b>E</b>	<pre>enum UrlTableErrors { kOK = 0, kErrorOutOfMemory, kErrorMalformedInput, }; enum AlternateUrlTableErrors { OK = 0, OUT_OF_MEMORY = 1, MALFORMED_INPUT = 2, };</pre>	<pre>enum class UrlTableErrors { ok, out_of_memory, malformed_input, };</pre>	<pre>MAX_ITERATIONS COLOR_RED PI</pre>

Table 17: Enumerator Naming

### 5.3.5.7 Macro Naming

This section compares the naming conventions for "Variables".

	<b>Google</b>	<b>Canonical</b>	<b>Geosoft</b>
<b>D</b>	Macro names are defined in const case	Same as Google	No guideline defined
<b>E</b>	<code>#define ROUND(X)</code> <code>#define PI_ROUNDED 3.0</code>	Same as Google	-

Table 18: Macro Naming

5.4 Environement Analysis

This section discusses the environement analysis. It has a look at the Eclipse and Codan plug-in landscape into which the Stylechecker plug-in is to be integrated. It also contains an analysis of the domain elements and has a look at the most important sequences.

5.4.1 Landscape Analysis

The landscape analysis looks at the context elements the plug-in is using. Therefore, the sections looked at are the Rename Refactoring from the Eclipse CDT and the Checkers from the Codan plug-in.

5.4.1.1 Eclipse CDT Rename Refactoring

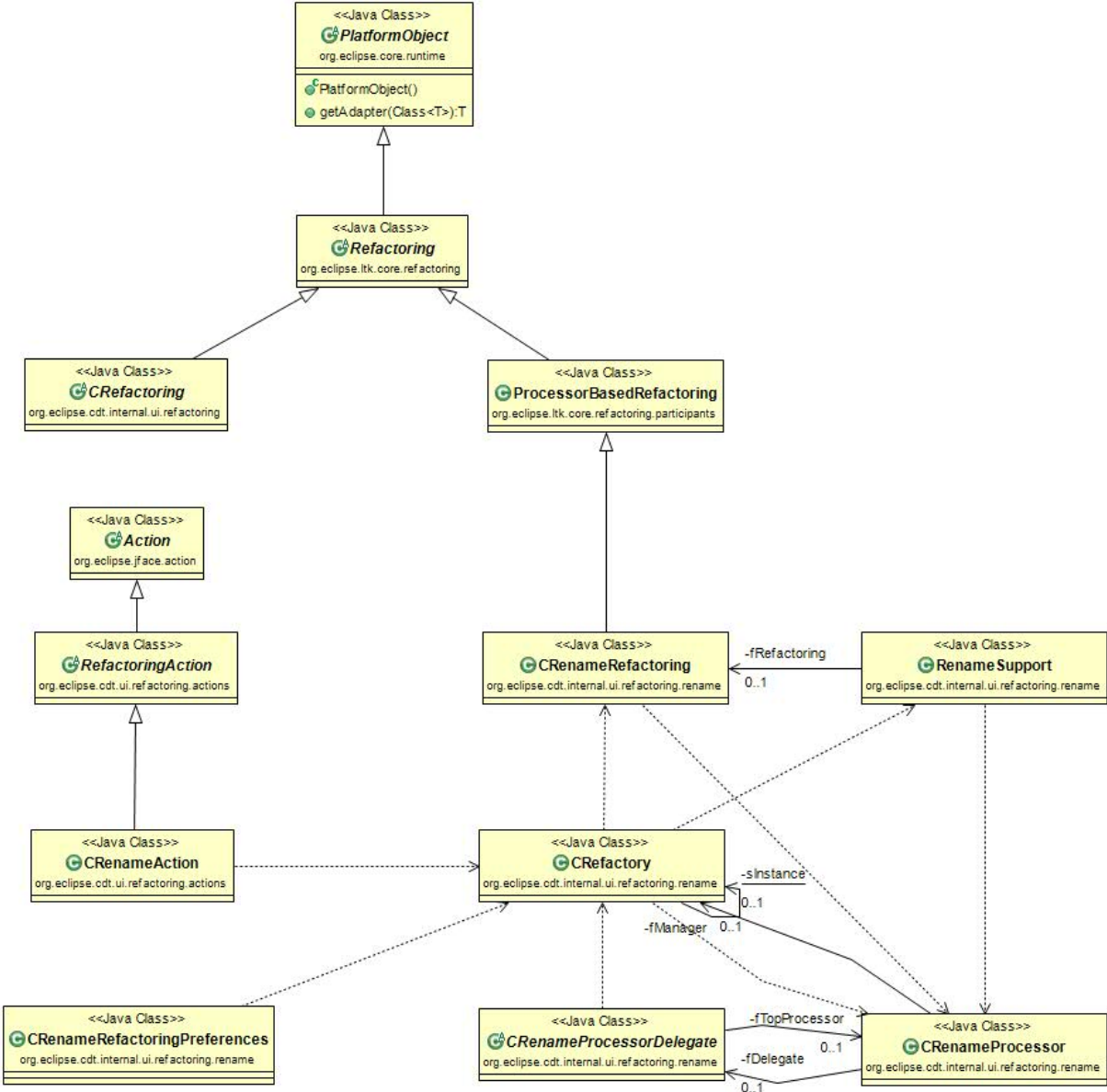


Figure 3: CRefactory Class relationships



<b>What</b>	<b>Description</b>
PlatformObject	Is used to access Eclipse's GUI
Refactoring	A Base class, which other Refactorings need to implement in order to be used with the Eclipse Refactoring
CRefactoring	Realizes the C/C++ specific refactoring tasks
ProcessorBasedRefactoring	A refactoring, that uses a special processor to execute the change
CRenameProcessor	A C/C++ specific rename refactoring processor
Action	An abstract Action class for UI related executions
RefactoringAction	A Refactoring Action. This starts a new UI in which a refactoring can be performed
CRenameAction	A C/C++ specific rename action. Starts a rename refactoring wizard.
CRenameRefactoring	A C/C++ specific rename refactoring implementation
RenameSupport	A support class for renaming procedures, used by CRenameRefactoring and also when delegating to Eclipse CDT internal rename refactoring from within the plugin
CRefractory	A Singleton class providing a singular interface for access to C/C++ related refactorings
CRenameRefactoringPreferences	As the name suggests, preferences class for the CRenameRefactoring procedure
CRenameProcessorDelegate	A delegate class that encapsulates the Eclipse CDT C/C++ Refactorings

Table 19: Conceptual Elements

5.4.1.2 Codan Extension Points

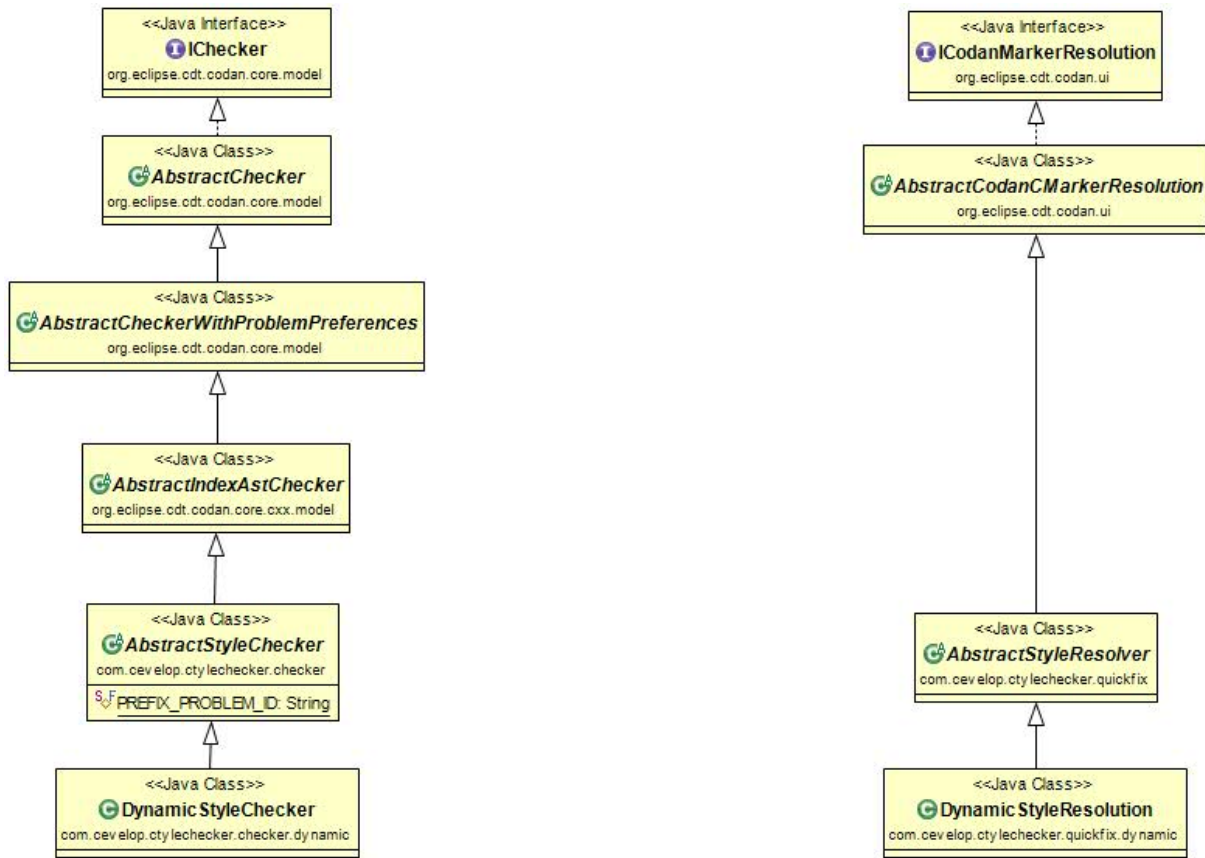


Figure 4: Relevant Codan Extension Points

What	Description
IChecker	Codan Interface class for Checkers
AbstractChecker	Convenience implementation of IChecker interface with default implementations of common methods
AbstractCheckerWithProblemPreferences	AbstractChecker that has extra methods to simplify adding problem preferences
AbstractIndexAstChecker	Convenience implementation of checker class that works on an index-based AST of a specific C/C++ program
AbstractStyleChecker	A special convenience class to gather common functionalities across user implemented checkers
DynamicStyleChecker	A Checker implementation for dynamic style checking for names

Table 20: Left Side, Checker Extension Point

<b>What</b>	<b>Description</b>
ICodanMarkerResolution	Codan Interface for Resolution implementations
AbstractCodanMarkerResolution	Generic class for Codan Marker resolution. Used as base class for Codan's marker resolution extension.
AbstractStyleResolver	Convenience inbetween class to provide common functionalities across user implemented resolution implementations
DynamicStyleResolution	A Resolution implementation for DynamicStyleChecker violations

Table 21: Right Side, Resoluttion Extension Point

### 5.4.2 Conceptual Model

The conceptual represents the conceptual ideas of the Stylechecker plug-in. Based on this, a more specific class diagram mapping will evolve. The model shows the basic relationships between the conceptual entities.

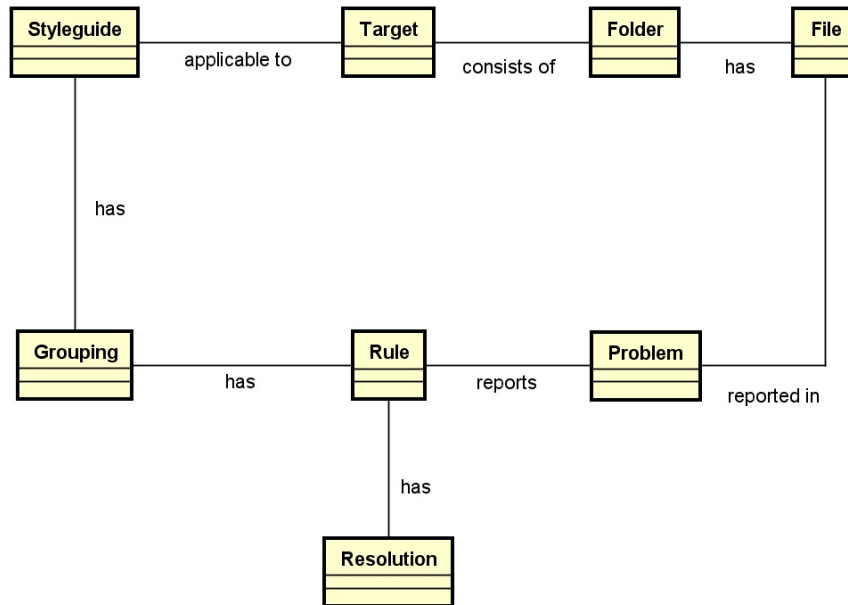


Figure 5: Conceptual model Stylechecker

What	Description
Styleguide	A Styleguide represents the sum of all settings of Styleguide profile
Target	An exclusion Target
Folder	Self explanatory
File	Self explanatory
Grouping	A grouping represents a specific set of rules that belong to the same domain, f.ex. Naming Conventions is an example of a Grouping
Rule	A rule is a specific set of checking configuration, according to which problems are reported
Problem	A problem is a specific type of styleguide violation
Resolution	A resolution is a specific type of solution to a reported styleguide violation problem

Table 22: Conceptual Elements

### 5.4.3 Class Diagram

The class diagram shows the relationships of the concrete domain elements used to model the Stylechecker plug-ins core business logic.

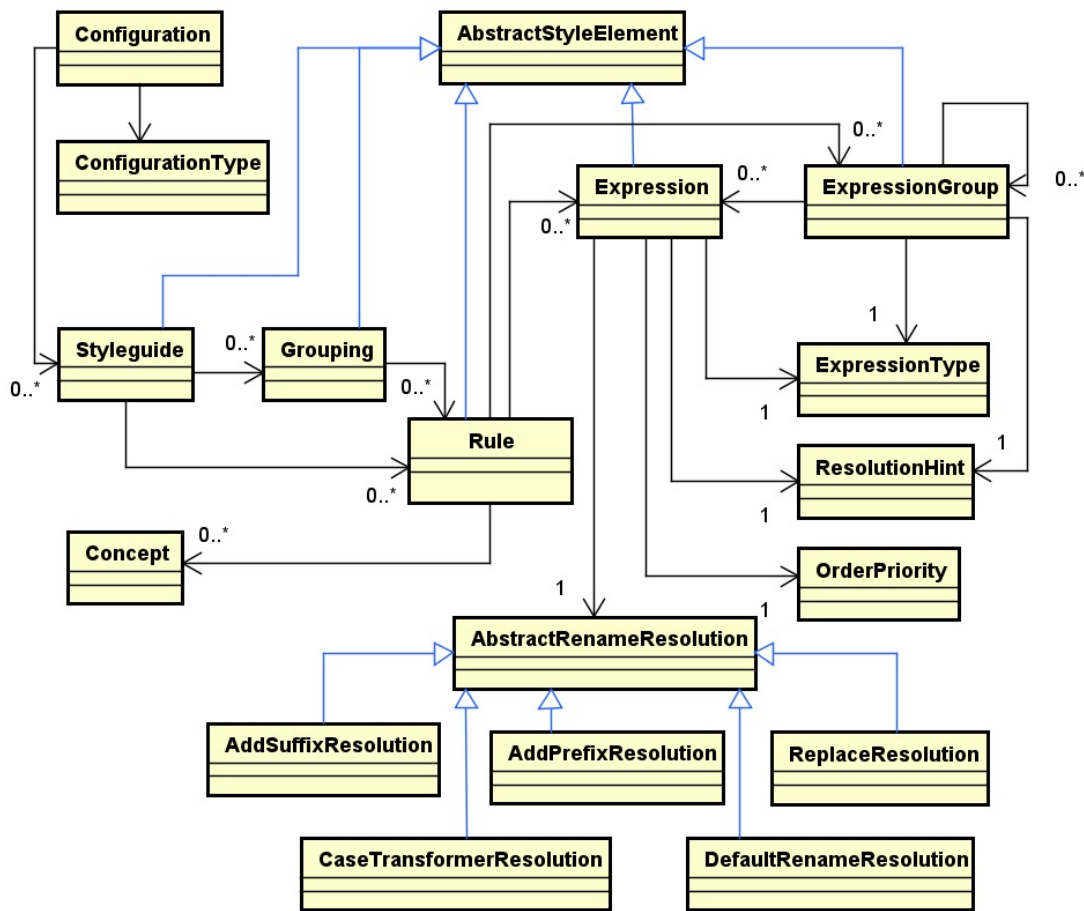


Figure 6: Class Diagram Stylechecker

What	Description
AbstractCtyle Element	Abstract class that provides common fields and methods to most of the domain elements. Specifically, the AbstractCtyleElement provides facilities that are used when persisting domain objects, like ID information.
Styleguide	Represents a styleguide and consists of groupings and rules
Grouping	A grouping is a class to group rules
Rule	A rule is next to an expression the heart piece of the Stylechecker plug-in, it defines the checking logic, resolution logic and scope of language elements.
Concept	A Concept models C++ language elements in the DOM. e.g. CPPVariable is a Concept, that models C++ Variables.

Table 23: Class Diagram Elements, part 1

What	Description
ExpressionGroup	An ExpressionGroup models a set of Expressions and how they're checked against. ExpressionGroup's can contain nested Expression-Groups.
Expression	An Expression models what to check for when controlling source code. They reference a resolution, with which a violation can be solved.
ResolutionHint	A ResolutionHint is used to manage resolution procedures. Different resolutions react differently to types of ResolutionHints. This is an enum specifier.
OrderPriority	An enum defining which Expressions should be processed first or last.
AbstractRename Resolution	Abstract base class that contains common functionality for all resolution types.
AddSuffix Resolution	Resolution, that adds a suffix.
CaseTransformer Resolution	Resolution that performs a CaseTransformation.
AddPrefix Resolution	Resolution that adds a prefix.
DefaultRename Resolution	Resolution that indicates a resolution by user input.
Replace Resolution	Resolution that replaces a certain string according to matching regex capture group.
Configuration	Represents Stylechecker plug-in configuration and holds all currently available styleguides.
ConfigurationType	An enum modeling what kind of Configuration is currently active. There exist three types, Workspace, Workspace reference and Project.
ExpressionType	Represents an expression type. Currently, there are two expression types: Single, which is a singular expression and Group, which is a group consisting of multiple expressions and expression groups.

Table 24: Class Diagram Elements, part 2

### 5.4.4 Sequence Diagrams

There are two important sequences in the Stylechecker plug-in. The first one is the loading of the configurations during each checking to access the active styleguide. The active styleguide is used to perform the styleguide checking within the DynamicStyleChecker.

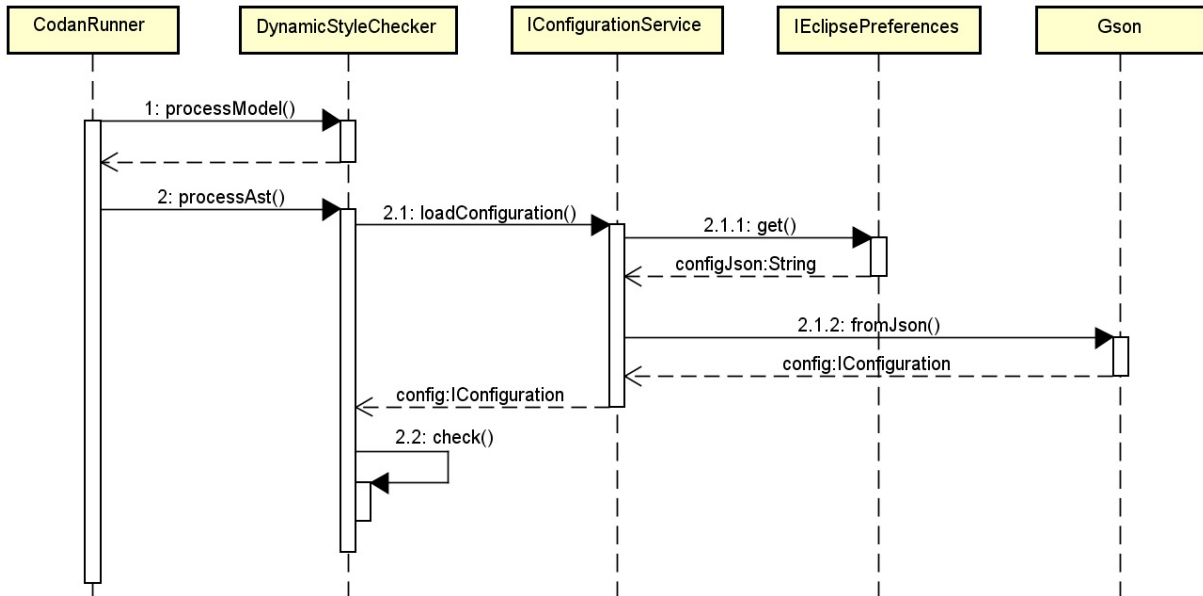


Figure 7: SSD Configuration loading when checking

The second sequence is the Quickfix procedure. When a quickfix operation is kicked off, DynamicStyleResolution class, the Quickfix class for the Stylechecker plug-in, first retrieves the rule via the JSON string delivered with the Marker when reported. After the rule is retrieved, it is passed to the internal fix procedure(applyFix), where, depending on the reporting type (file, ast) the corresponding refactoring is called to apply the necessary transformations.

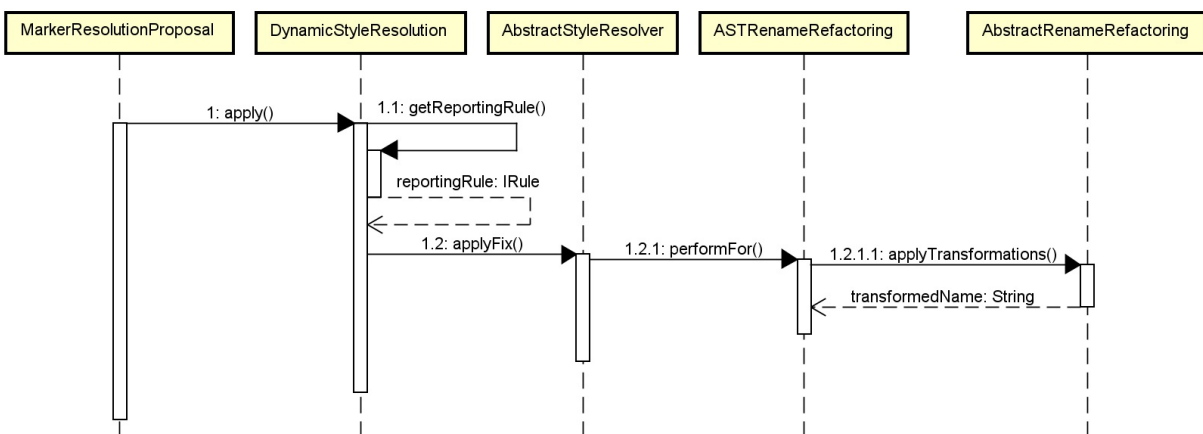


Figure 8: SSD Quickfix with IRule retrieval

## 6 Design and Decisions

This section discusses the design choices of the Stylechecker plug-in and the decisions that went with them. The aim is to provide a better insight into certain aspects of the Stylechecker plug-in and its structure; whether it be user interface related choices or architectural designs.

### 6.1 Stylechecker Checking

The Stylechecker plug-in extends the Codan (Code Analysis) plug-in and uses a singular Checker (DynamicStyleChecker). This way the facilities to check, report and provide quick-fixes are automatically given. It was decided to use only one Checker (DynamicStyleChecker) for the entire styleguide checking rather than to implement separate checkers for all different cases (File, Variables, Functions etc.).

```

1 #include <iostream>
2
3 int main() {
4     const int kMyFirstConstant { 2 };
5     const int kMySecondConstantVariable { 0 };
6     int my_third_variable { 1 };
7     int my_forth_variable { 8 };
8     std::cout << kMyFirstConstant
9                 << kMySecondConstantVariable
10                << my_third_variable
11                << my_forth_variable;
12 }

```

Figure 9: Simple code excerpt

The Codan Checkers use the Abstract Syntax Tree (AST), a representation of the source code in tree form, provided by the Eclipse CDT to perform Checker logic. In the same way, the DynamicStyleChecker relies on the AST as well. Figures 9 and 10 show how a source code is parsed into the AST. The AST is realized via the Composite Pattern [EGea94].

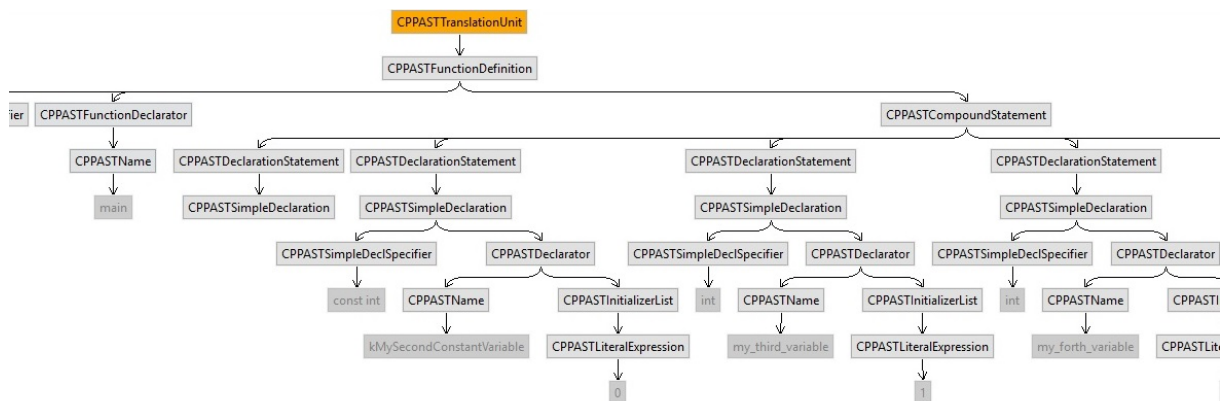


Figure 10: AST representation excerpt of Code from Figure 9



As the Stylechecker plug-in's scope required to check for naming conventions, only name nodes of the AST are considered when performing a checking as shown in Figure 11.

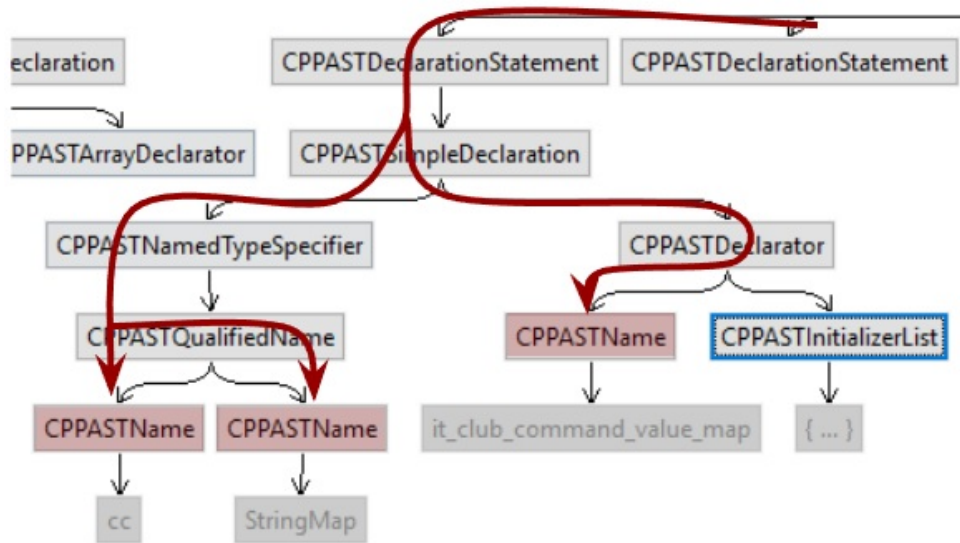


Figure 11: AST name traversal example

With the single Checker approach the AST needs to be traveled only once rather than multiple times. When reporting a violation, the DynamicStyleChecker embeds the reporting rule into the Marker for later retrieval at quickfix time (i.e. when a quickfix is called on a Marker).

```
private void reportFileProblem(IRule pRule, ResourceType pType) {
    reportProblem(POSITION_FILE_PROBLEM_ID, createProblemLocation(getFile(), 0, 0, 1), pRule.getMessage(),
        mapper.toJson(pRule), pType);
}

private void reportProblem(IASTName name, IRule pRule) {
    reportProblem(POSITION_PROBLEM_ID, name.getOriginalNode(), pRule.getMessage(),
        mapper.toJson(pRule), ResourceType.AST);
}
```

Figure 12: DynamicStyleChecker problem reporting excerpt

Via the ResourceType the type of reported resource is given as an additional information when reporting a problem. This was necessary for the checking of names that weren't covered by the Abstract Syntax Tree during traversal. An example of that would be Macro definitions and File names. Depending on the resource type, the corresponding Quickfix (DynamicStyleResolution) determines what kind of refactoring to use (i.e. rename on Resource or on language concepts). This way the Stylechecker can be easily extended to different types of refactoring should the need arise. An example could be formatting related checkings and fixes that might need a different refactoring approach (e.g. comments or control structures, that only need to be reformatted rather than replaced, could have a refactoring logic that uses some custom logic to achieve the necessary target representation via ASTRewrite or something similar).

The primary design goal of the DynamicStyleChecker was to provide a solid basis for flexible checking logic. Although the checking procedure itself covers that or at least offers the flexibility to easily extend towards it, the domain specific logic still needs adjustment. They currently work for naming conventions, i.e. they fulfill the requirements of the project's main goal of implementing naming conventions. But they still need to be tweaked a little more to be fully independent and extensible. More on this subject can be found in the section 8.2 relating to optimization potentials.

On every checking, the active styleguide is loaded from the configuration files.

```
@Override
public void processAst(IASTTranslationUnit ast) {
    IConfiguration config = configService.loadConfiguration(getProject());
    if (config.isEnabled()) {
        check(ast, config.getActiveStyleguide());
    } else {
        try {
            getFile().deleteMarkers(PPOSITION_PROBLEM_ID, true, IResource.DEPTH_INFINITE);
            getFile().deleteMarkers(PPOSITION_FILE_PROBLEM_ID, true, IResource.DEPTH_INFINITE);
        } catch (CoreException e) {
            CstylecheckerRuntime.log(e);
        }
    }
}
```

Figure 13: DynamicStyleChecker configuration loading excerpt

This approach was chosen in order to always work with the newest configurations. Rather than manage a global object, the configuration is loaded from the configuration file when it is needed and doesn't exist beyond its usage scope. As the Stylechecker plug-in only uses one Checker for styleguide checking, retrieving the configuration from the configuration file isn't causing too much overhead.

## 6.2 Stylechecker Resolution

Similar to the checking logic, the Stylechecker resolution implementation, `DynamicStyleResolution`, is a singular Quickfix. It delegates a Quickfix request to the internal logic, where depending on the reported rule the appropriate resolutions are applied. The `DynamicStyleResolution` decides which type of refactoring to use depending the `ResourceType` passed to the Marker at checking time and applies all violated expression resolutions to the corresponding name before the refactoring.

```
@Override
public void apply(IMarker marker, IDocument document) {
    Optional<IRule> oRule = getReportingRule(marker);
    if (oRule.isPresent()) {
        IRule rule = oRule.get();
        try {
            ResourceType checkType = ResourceType
                .valueOf(getProblemArgument(marker, MarkerArg.RESOURCE_TYPE.ordinal()));
            if (checkType.equals(ResourceType.AST)) {
                applyFix(marker, document, rule);
            }
            if (checkType.equals(ResourceType.FILE) || checkType.equals(ResourceType.FILE_ENDING)) {
                applyFix((IFile) marker.getResource(), rule, checkType);
            }
        } catch (ArrayIndexOutOfBoundsException ex) {
            CstylecheckerRuntime.Log(ex);
        }
    }
}
```

Figure 14: `DynamicStyleResolution` apply excerpt

Figure 14 shows the `DynamicStyleResolution.apply` implementation. The `ResourceType` approach was chosen due to the different types of refactorings needed for different aspects of styleguide checking like AST and file name. But this design choice also allows to easily extend for future refactorings that go beyond simple name modifications. All refactorings extend `AbstractRenameRefactoring`, which contains common logic to apply multiple resolutions to a reported name violation. It basically iterates over the reported rule and picks out the expressions that are violated by the name and applies their resolution in order before starting the refactoring process. Figure 15 shows a delegation to the AST refactoring procedure.

```
public void applyFix(IMarker pMarker, IDocument pDocument, IRule pRule) {
    try {
        ASTRenameRefactoring refactoring = new ASTRenameRefactoring(pMarker, pDocument);
        if (!refactoring.performFor(pRule)) {
            CstylecheckerRuntime.Log("Refactoring failed");
        } else {
            CodanRuntime.getInstance().getBuilder().processResource(pMarker.getResource(),
                new NullProgressMonitor());
        }
    } catch (Exception e) {
        CstylecheckerRuntime.Log(e);
    }
}
```

Figure 15: `DynamicStyleResolution` applyFix excerpt

The `applyTransformations` method is used internally by all refactoring types and provides the dynamic resolution applications. Figure 16 shows the basic `applyTransformations` method.

```

protected String applyTransformations(IRule pRule) {
    transformedName = "";
    Optional<String> oOriginalName = getOriginalName();
    if (oOriginalName.isPresent() && pRule != null) {
        transformedName = oOriginalName.get();
        executeTransformations(pRule, pRule.getPredefinedExpressions());
        executeTransformations(pRule, pRule.getCustomExpressions());
    }
    return transformedName;
}

```

Figure 16: AbstractRenameRefactoring applyTransformations excerpt

Through this design approach, the amount of resolutions can be dynamically applied to a reported violation. New features only need to define a new resolution (IResolution) that fixes the according violation to the required target format.

Although this approach allows for multiple resolutions to be applied at once, it can be problematic, meaning: A previous resolution's fix could be destroyed by a following resolution. Therefore, to manage the multiple resolutions, a specific ordering approach was implemented as shown in Figure 17.

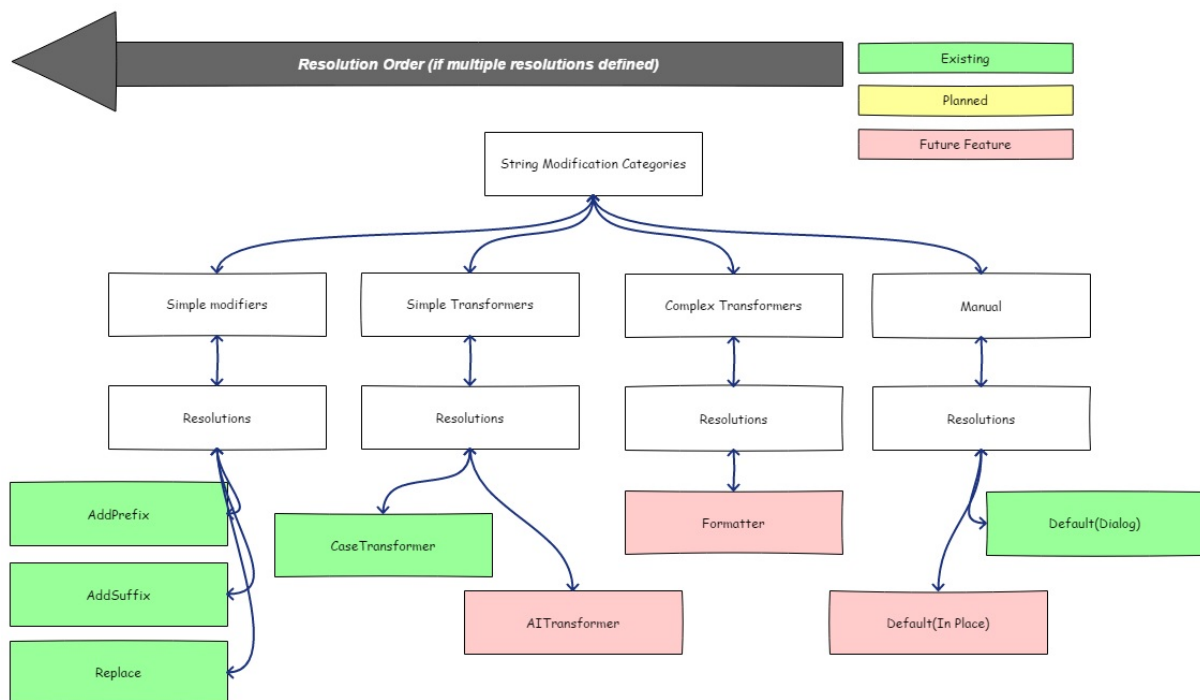


Figure 17: Resolution ordering excerpt

With this approach the order of the resolutions is known beforehand and the Quickfix procedure can react accordingly, resulting in multiple resolution application without destructive side effects.

### 6.3 Stylechecker Plug-in User Interface

This section discusses the design and conception of the Stylechecker plug-in's user interface.

#### 6.3.1 User Interface conception

The Stylechecker user interfaces underwent several iterations, each with a short usability test with the advisers. The initial concept is represented in Figures 18 and 19.

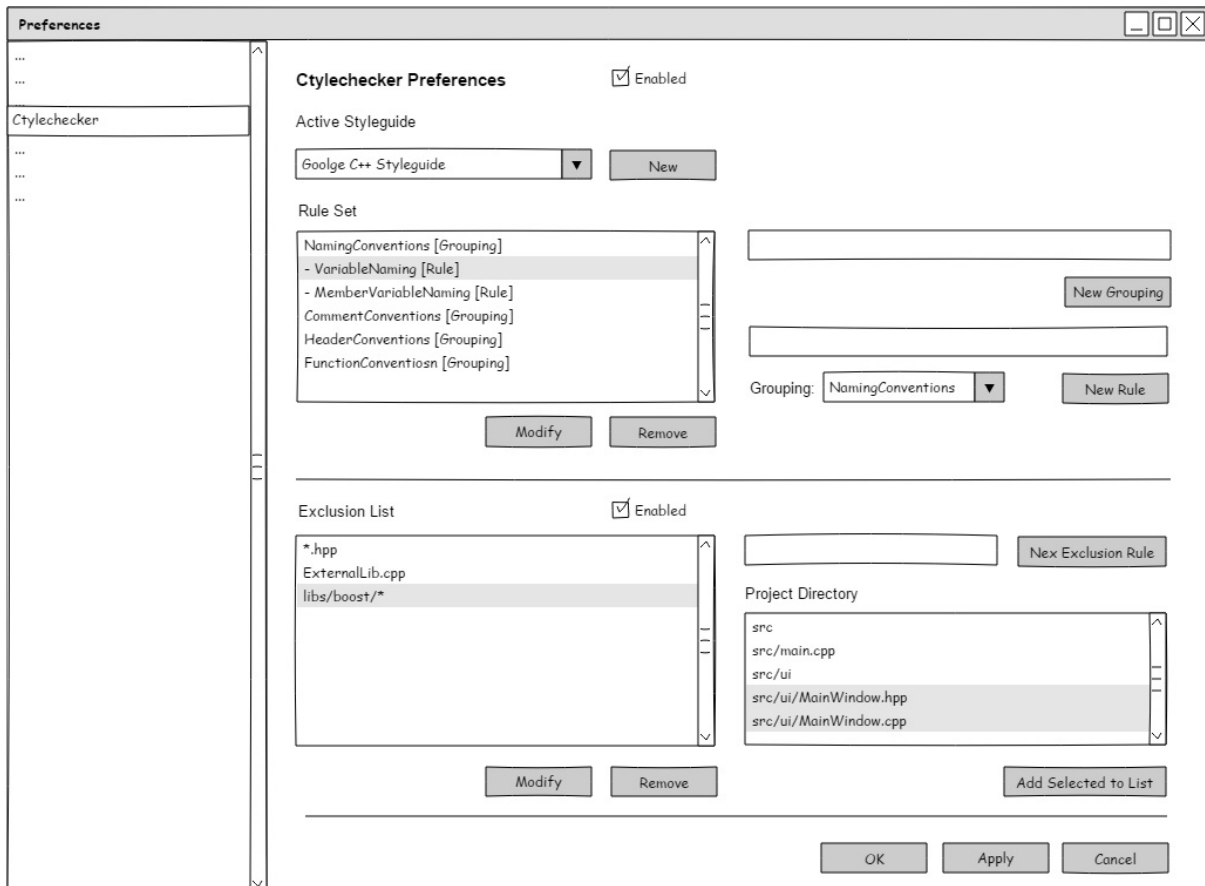


Figure 18: Initial UI design Stylechecker preferences page

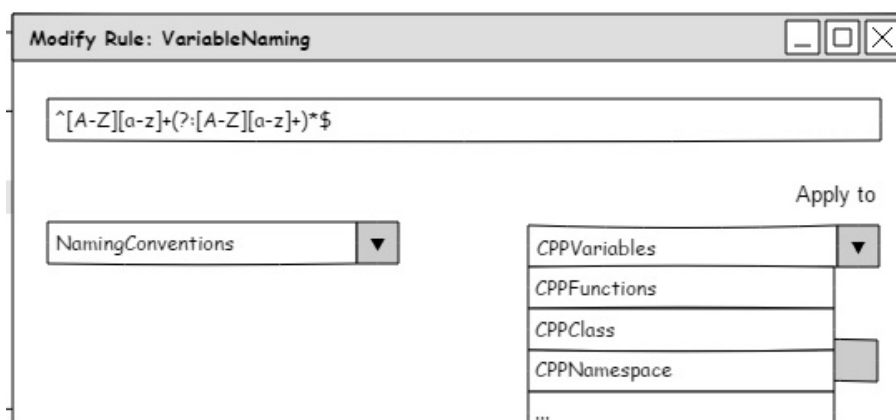


Figure 19: Initial dialog design for Rules

The first iteration had an overloaded UI for the Stylechecker preferences page. Additionally, the rules dialog was still very simple. In the next iteration, the main entrance point to the Stylechecker plug-ins settings, the Stylechecker preferences page, was tried to be simplified.

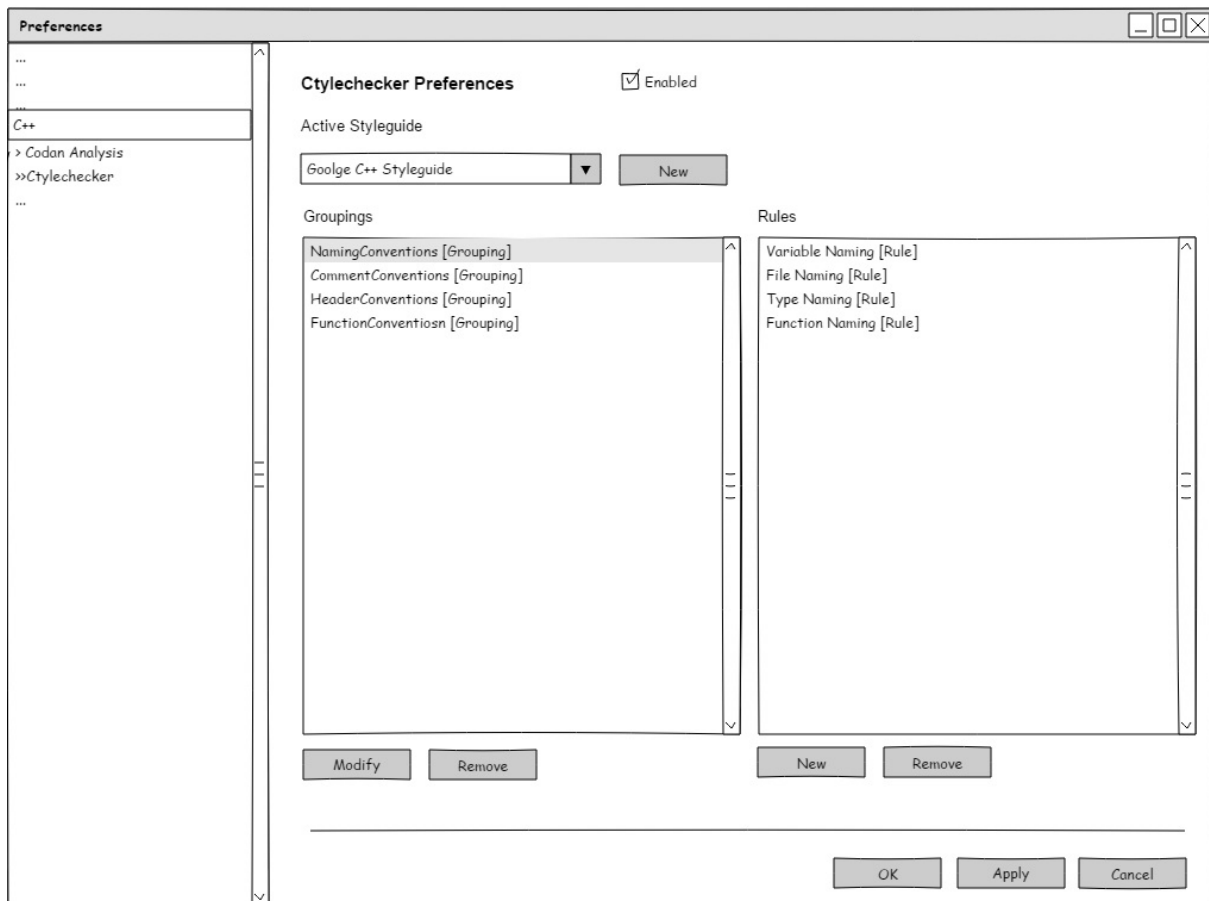


Figure 20: Second iteration design Stylechecker preferences

Additionally a new dialog was introduced for grouping of rules. In the initial concept for the dialog, there was a category element to the groupings. The idea was to limit available options for the rules, like to provide certain resolutions in rules that were grouped within. This idea was abandoned in later iterations.

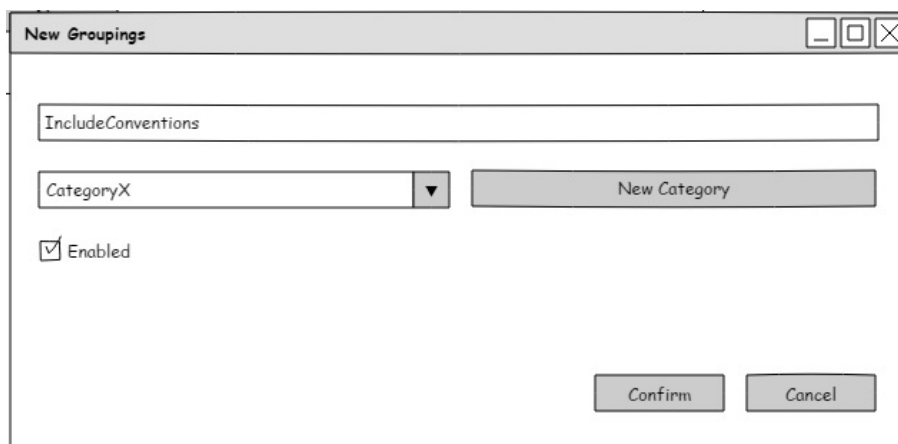


Figure 21: Second iteration, grouping dialog

The second iteration brought new elements to the rule dialog as well. As it became apparent that a rule needed a lot more flexibility in the configurations, the rule dialog was adjusted accordingly. At this stage, the plug-in had closer ties to the Codan plug-in, the Severity field being an indicator of that.

The image shows a 'New Rule' dialog box with the following fields and controls:

- Rule Name: SelfIncludeForbidden
- Check on: CPPInclude
- Severity: Warning
- Add button
- Check with (Regex): ^file\$
- file: CPPFile
- variables: variables
- CPPVariable: CPPVariable
- Define Reference button
- Message: Cannot include same file as header
- Autoresolveable:
- Rename Refactoring: Rename Refactoring
- Confirm and Cancel buttons

Figure 22: Second iteration, Rule dialog

In later stages, the UI developed more and more according to the needs of the domain. After it became apparent that a rule needed to allow for alternative valid forms, expression groups were introduced. Figures 23 and 24 show the redesign of the UI to accommodate for the new domain element. Instead of a table representation for expressions a tree view was chosen, so that expressions and expression groups could be displayed at the same time and in relation. Due to the new UI structure, additional dialogs for defining expressions and expression groups needed to be designed as well, Figure 24 shows the conception of these dialogs.

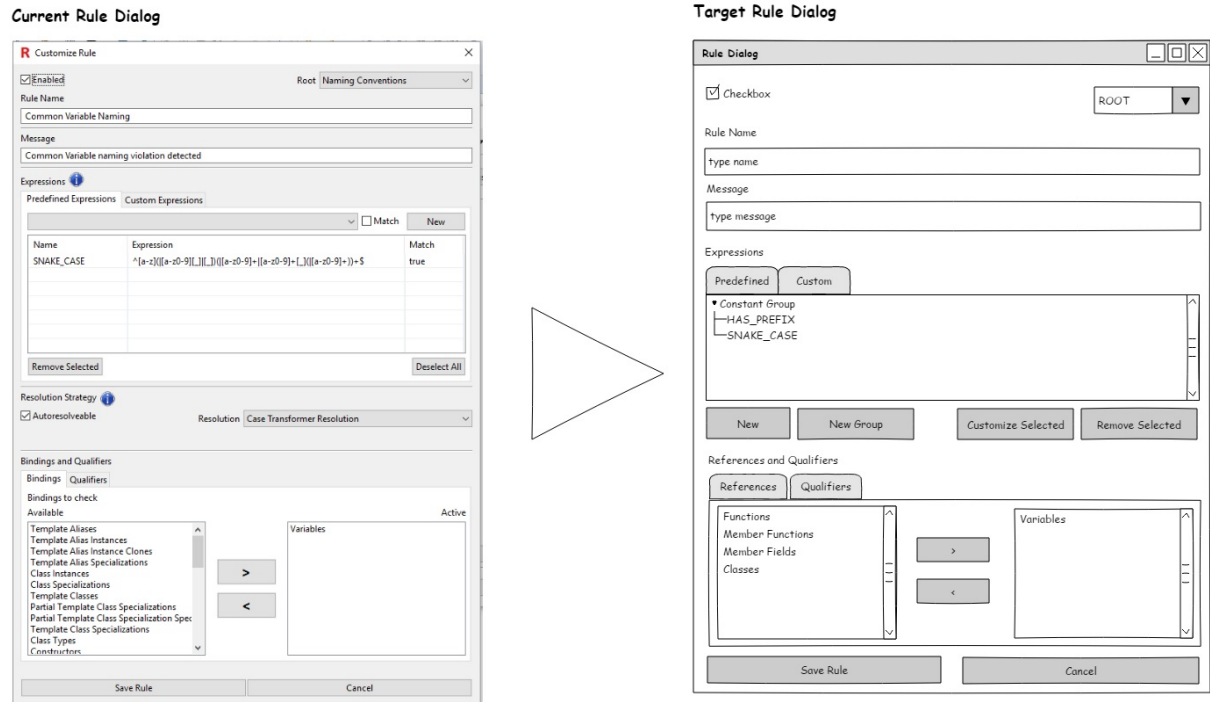


Figure 23: Third iteration, rule dialog

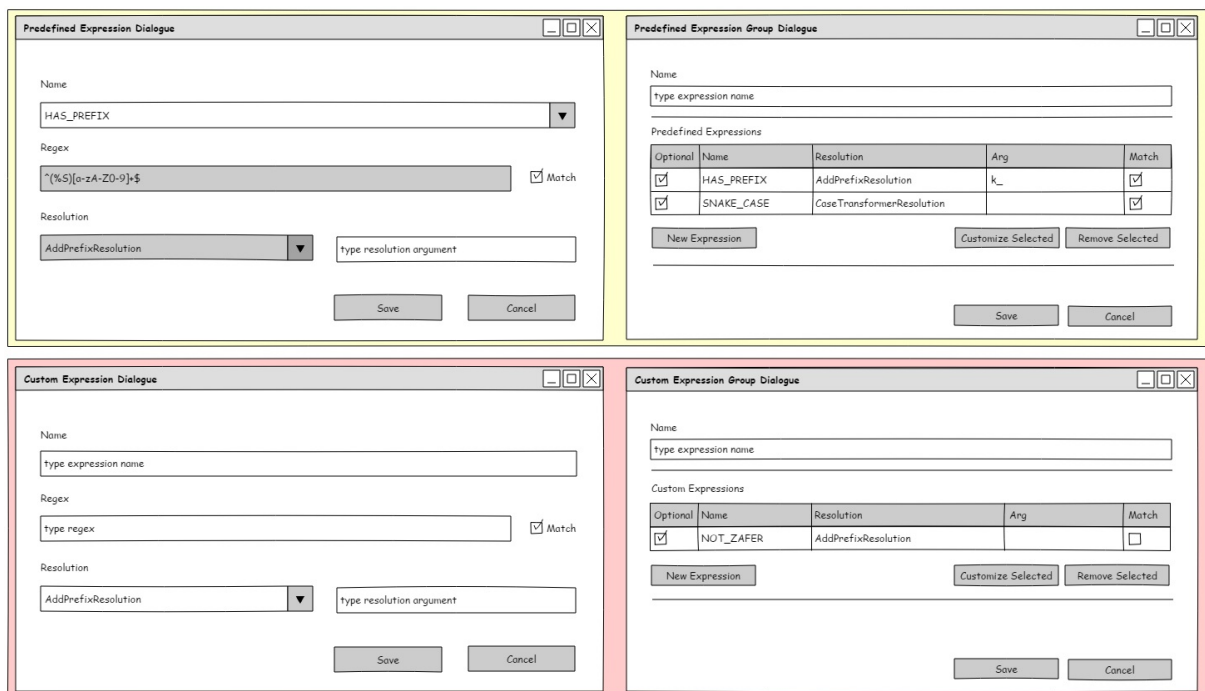


Figure 24: Third iteration Expression dialogs

The final design of the UI will be discussed in more detail in the following section. The only major deviation from the last iteration is that for the displaying of the expression and expression group values the additional dialogs were dropped and the values were displayed directly in the main tree view, which reduced the total dialog count from four to three.



### 6.3.2 Current User Interface

The active styleguide with the defined rules can be viewed in the Stylechecker plug-in settings. As shown in Figure 25, the preferences page contains a list of the currently defined rules. From here, new rules can be defined. Rules can be also grouped in order to better distinguish certain rule sets. Additionally, styleguides can be exported and imported in the Stylechecker preferences pages. An imported styleguide can be directly activated and used without any further configurations. As the Stylechecker plug-in is connected to the Codan plug-in (Stylechecker extends Codan), a quick access to Codan settings can be found here as well. Codan settings control to what files and folders a checker is applied. Furthermore, Codan settings manage whether the checker is considered in code analysis procedure or not. It differs from the Stylechecker plug-in enabling in one way. The Stylechecker enabling controls internal checking, whereas the Codan controls whether the checker is at all considered.

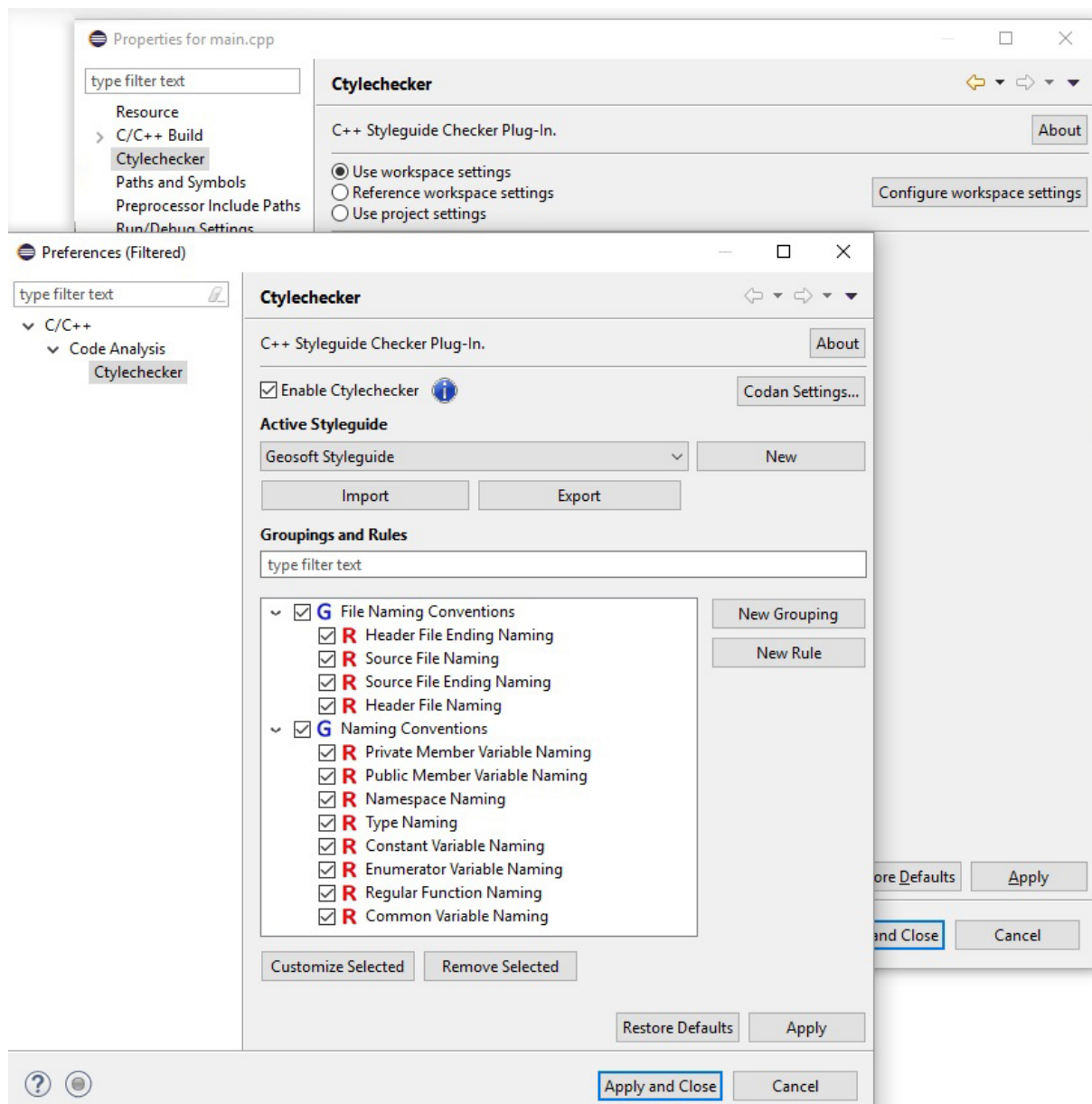


Figure 25: Stylechecker plug-in Preferences page

New rules can be added via the preference page as well as customized. For new rules as well as customization of existing rules, a new dialog is opened as shown in Figure 3. The rule dialog contains various configurable fields, such as rule name, rule message, expressions and checked concepts. The rule message is shown, when a violation is found. The expressions fields are used to define constraints on what is a valid styleguide and what is not via regular expressions. There exist two kind of expressions, predefined and custom expressions. Predefined expressions contain pre-configured expressions for common expression types such as pascal casing, snake case, prefix, suffix and the like. Custom expressions allow for, as the name suggests, defining own expressions. The expressions fields have two types of expression objects. One is the plain Expression, the other is the Expression Group. Expression group offer the definition of valid alternatives and can be set to match either ALL expressions it contains within or ANY expressions. With these two types, complex checking expressions can be constructed to be used on names. And finally, the checked concepts fields define on which C++ Concepts, like variables, functions, member functions etc. the rule will be applied to. In Figure 4, you can also see the qualifiers available for some concepts in order to limit the application of a rule even more.

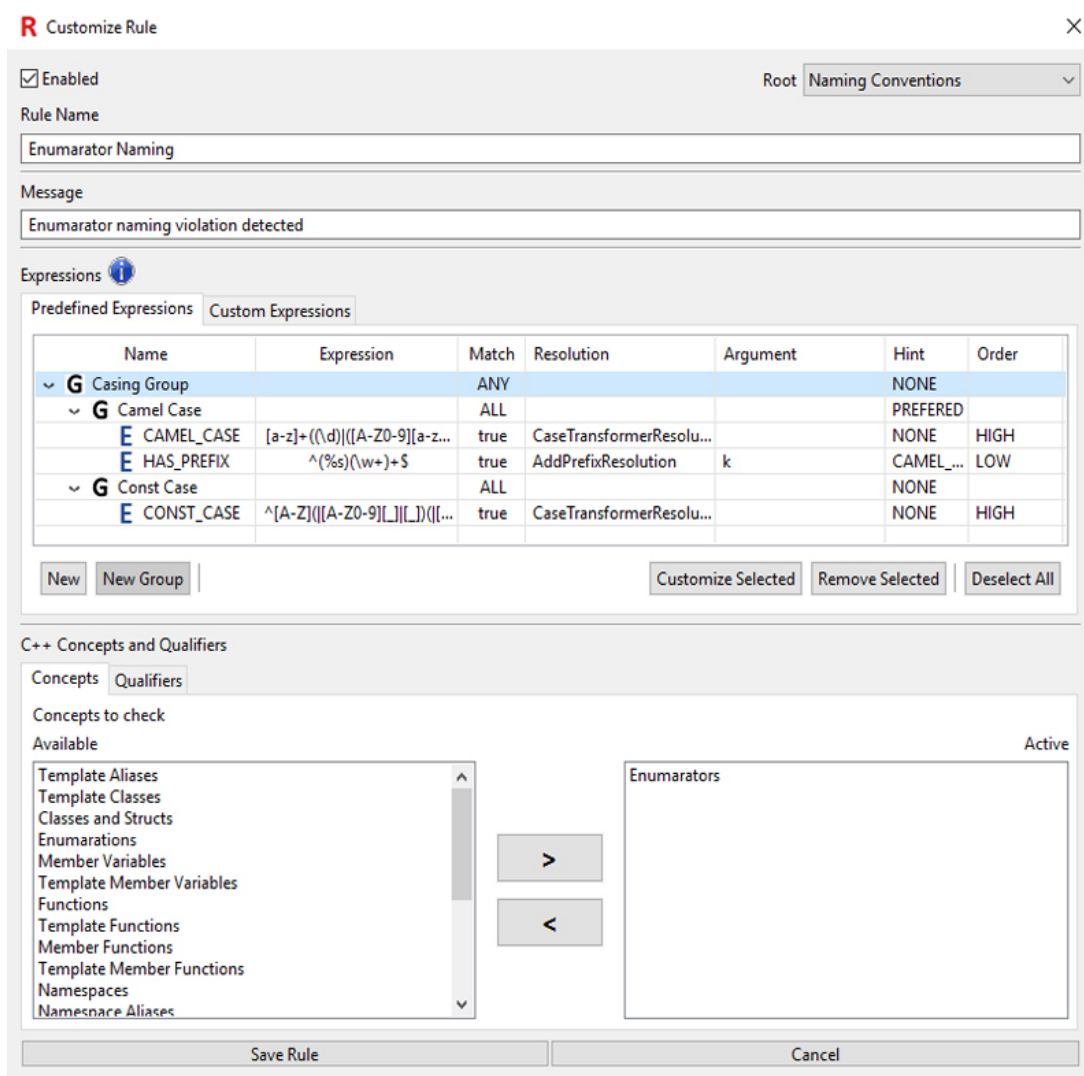


Figure 26: Rule dialog

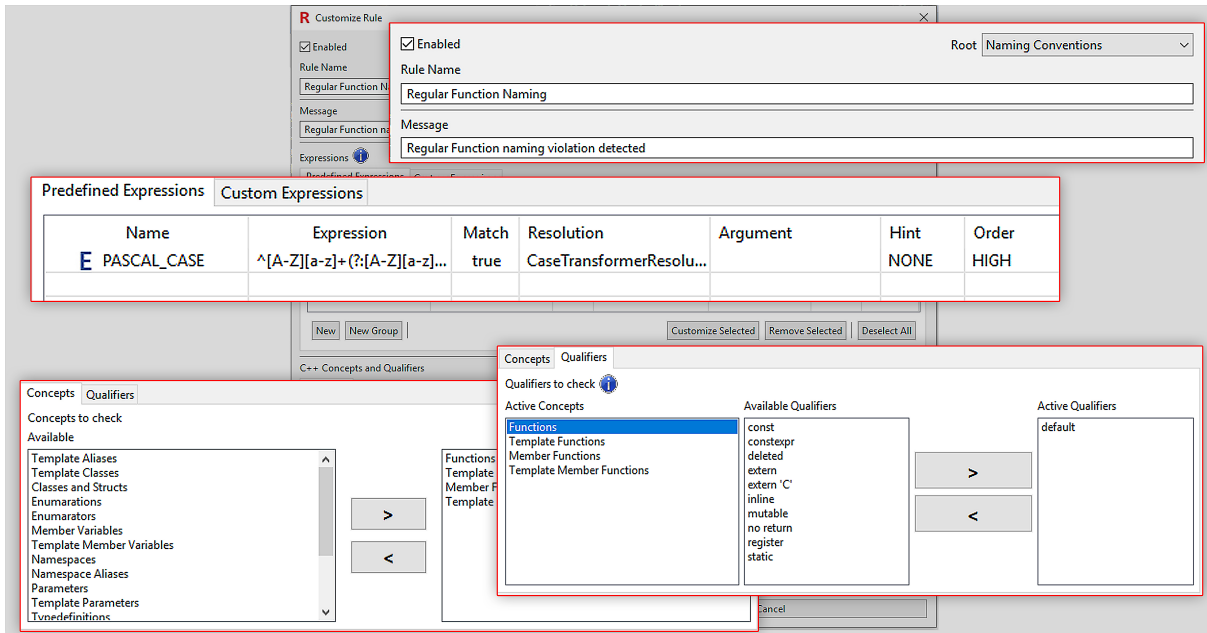


Figure 27: Different rule dialog with close up

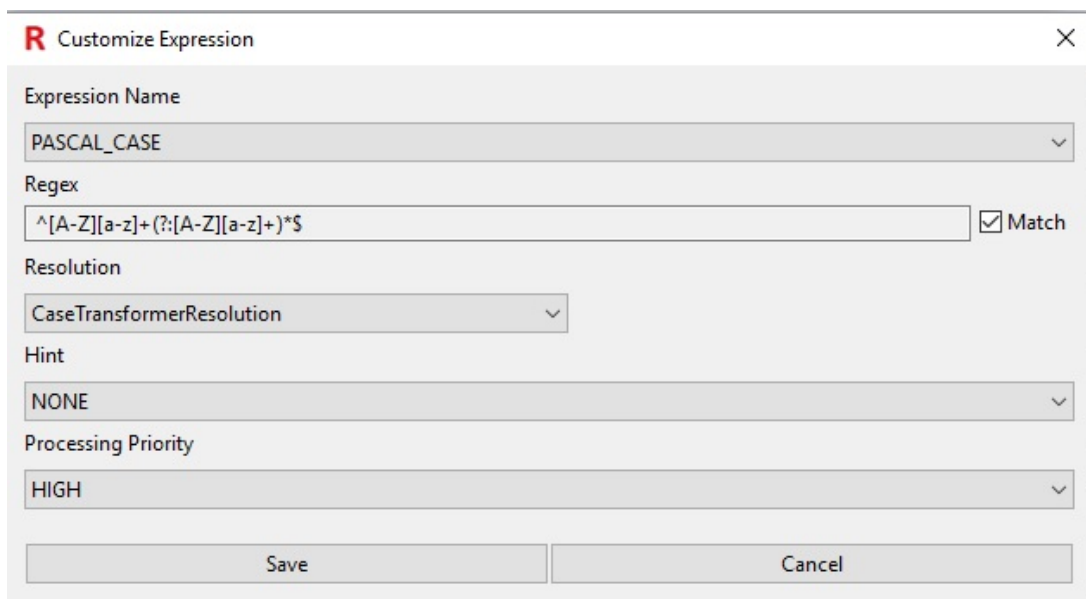


Figure 28: Predefined expression dialog

Figures 5 and 6 show the dialogs for defining a new Expression, Figure 5 representing a Predefined Expression dialog, Figure 6 a Custom Expression dialog. Expressions have a name and a regex field. The regex field can be controlled with a match flag, indicating whether a check should match or not match the given regex. Expressions also contain information on how it can be solved in case of violation in the resolution field. For this, there is a set of existing resolutions that can be chosen from. Furthermore, there is a hint field that can be set. Hint's are used by the resolution process in order to determine how to apply a resolution. In an Expression Group for example, if there are multiple alternative expressions a check can match, like a variable, that can be snake case or pascal case, with a hint, the user can define which of the alternative cases should be resolved to (via Hint: PREFERRED). Different resolutions have different hints. The field for processing priority defines the processing order of the expressions at resolution time. This way, resolutions can be optimized so that they don't conflict all to much with one another.

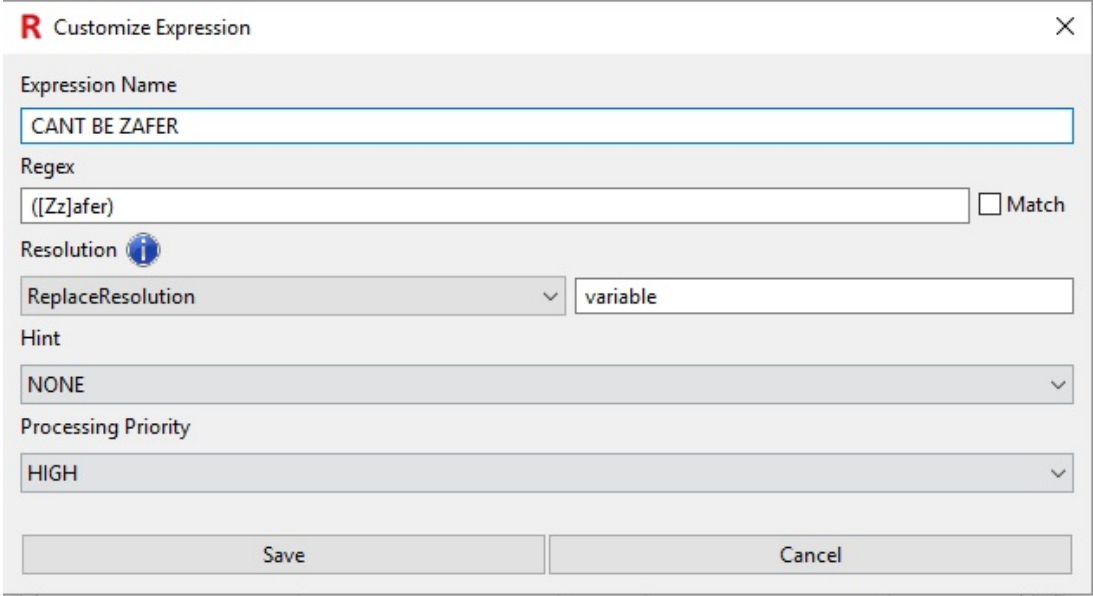


Figure 29: Custom expression dialog


## 6.4 Persistence

Due to the requirement of persisting plug-in configurations in Eclipse fashion, Eclipse persistence utilities were used; specifically in the Stylechecker plug-in's case, the persistence interfaces `IEclipsePreferences` and `IPreferenceStore` served as primary gateways to Eclipse internal persistence. Eclipse persistence files usually consist of key-value pairs. Because Stylechecker plug-in configuration can become quite extensive, it was decided to persist the configuration data in JSON format. This way only a few key-value pairs are necessary, which makes persistence management easier and more convenient. An additional benefit to JSON format persistence is that the configuration is human readable.

```
com.ovegap.stylechecker.prefs [2]
1 | ctylechecker.config="styleguides":{"097fe7ff-7565-4a6b-828f-b5fc407483bb":{"groupings":{"5f465e7b-c81d-41eb-b4de-f64956d4215a":{"rules":{"84
{"checkedConcepts":{"type":"CPPVariable","qualifiers":{"default":true},"predefinedExpressions":{"type":"Expression","properties":{"order":"Hi
sformerResolution"},"properties":{"argument":"","expression":"[a-z]([a-z0-9]_|_|)"}},("message":{"type":"Expression","properties":{"order":"NO
9a6-60ef-4121-84f8-e69360a02304"},"enabled":true},"customExpressions":{"type":"Common Variable naming violation detected","name":"Common
Naming","id":"84e3baea-d21c-41f7-a546-311alcc2632e"},"enabled":true},"8bb133c-e5a4-46b4-ace9-0e8a175634d4":{"checkedConcepts":{"type":"CPPNa
CPNNamespacesAlias"},"qualifiers":{"type":"Expression","properties":{"order":"HIGH","resolution":{"type":"CaseTr
}},"argument":"","expression":"^[a-z]u00E4u00F6u00FC0-9_$"},"hint":"NONE","name":"IS_ALL_SMALL","id":"adf553b2-dd38-44ef-8325-75aa8ceb161
}},"message":"Namespace naming violation detected","name":"Namespace
Naming","id":"8bb133c-e5a4-46b4-ace9-0e8a175634d4"},"enabled":true},"ade9f7a7-ffe7-4281-b65b-f85c3632047b":{"checkedConcepts":{"type":"Object
efinedExpressions":{"type":"Expression","properties":{"order":"ransformerResolution"},"properties":{"argument":"","expression":"^[A-Z][a-z]+(?
)+$"},"hint":"NONE","name":"SNAKE_CASE","id":"edcf29a6-60ef-4121-84f8-e69360a02304"},"enabled":true},"customExpressions":{"type":"Type
sage":{"Macro naming violation detected","name":"Macro
Naming","id":"ade9f7a7-ffe7-4281-b65b-f85c3632047b"},"enabled":true},"a6177c81-1082-4556-8d9c-a53563a22c41":{"checkedConcepts":{"type":"CPPTy
PClassType","qualifiers":{"type":"CPPEnumeration","qualifiers":{"type":"Expression","properties":{"order":"ransformerResolution"},"properties":{"argument":"","expression":"^[A-Z][a-z]+(?
)+$"},"hint":"NONE","name":"SNAKE_CASE","id":"edcf29a6-60ef-4121-84f8-e69360a02304"},"enabled":true},"customExpressions":{"type":"Type
Naming","id":"a6177c81-1082-4556-8d9c-a53563a22c41"},"enabled":true},"0bd7b0bc-1369-436f-8135-alb82c46930d":{"checkedConcepts":{"type":"CPPFur
PPFunctionTemplate","qualifiers":{"type":"CPPMethod","qualifiers":{"type":"CPPMethodTemplate","qualifiers":{"type":"Expression","properties":{"order":"HIGH","resolution":{"type":"CaseTransformerResolution","properties":{"argument":"","expression":"^[a-z]([a-z0-9]_|_|)"}},("message":{"type":"Regular Function
Naming","id":"0bd7b0bc-1369-436f-8135-alb82c46930d"},"enabled":true},"3ac650c-1f50-45d1-8a25-12851a9d07b1":{"checkedConcepts":{"type":"CPPEm
nedExpressions":{"type":"Expression","properties":{"order":"HIGH","resolution":{"type":"CaseTransformerResolution","properties":{"argument":"","expression":"^[a-z]([a-z0-9]_|_|)"}},("message":{"type":"Regular Function
Naming","id":"3ac650c-1f50-45d1-8a25-12851a9d07b1"},"enabled":true},"debl08b8-806a-49e0-bab6-3bdc6bab4b5c":{"checkedConcepts":{"type":"CPPVa
expr"},"predefinedExpressions":{"type":"Expression","properties":{"order":"HIGH","resolution":{"type":"CaseTransformerResolution","properties":{"argument":"","expression":"^[a-z]([a-z0-9]_|_|)"}},("message":{"type":"Constant Variable naming violation detected","name":"Constant Variable
Naming","id":"debl08b8-806a-49e0-bab6-3bdc6bab4b5c"},"enabled":true},"aeb5b95d-b762-4811-bcf2-918b6fdb8a9e6":{"checkedConcepts":{"type":"CPPFie
s"},"predefinedExpressions":{"type":"Expression","properties":{"order":"HIGH","resolution":{"type":"CaseTransformerResolution","properties":{"argument":"","expression":"^[a-z]([a-z0-9]_|_|)"}},("message":{"type":"Member Variable naming violation detected","name":"Member Variable Naming","id":"aeb5b95d-b762-4811-bcf2-918b6fdb8a9e6"},"enal
Conventions","id":"5f465e7b-c81d-41eb-b4de-f64956d4215a"},"enabled":true},"48a57310-16ff-47d8-9ce8-cf3feef2b421":{"rules":{"82a482c-6659-423c-f
":{"type":"Header File","qualifiers":{"File
Body"},"predefinedExpressions":{"type":"Expression","properties":{"order":"HIGH","resolution":{"type":"CaseTransformerResolution","properties":{"argument":"","expression":"^[a-z]([a-z0-9]_|_|)"}},("message":{"type":"Header File naming violation detected","name":"Header File
Naming","id":"82a482c-6659-423c-bbaf-7ef409cf661"},"enabled":true},"2f190a2e-a4b-453c-9ba4-60ef89ee041c":{"checkedConcepts":{"type":"Header
Ending"},"predefinedExpressions":{"type":"Expression","properties":{"order":"LOW","resolution":{"type":"ReplaceResolution","properties":{"
```

Figure 30: Stylechecker plug-in excerpt

For serialization and deserialization the Gson library was used. Due to its Apache License 2.0 licensing it is ideal to use in the Stylechecker plug-in. The Gson library is delivered as an additional dependency within the Stylechecker plug-in and is downloaded automatically upon installation via Updatesite.



google/gson is licensed under the **Apache License 2.0**

A permissive license whose main conditions require preservation of copyright and license notices. Contributors provide an express grant of patent rights. Licensed works, modifications, and larger works may be distributed under different terms and without source code.

Permissions	Limitations	Conditions
<span style="color: green;">✓</span> Commercial use	<span style="color: red;">✗</span> Trademark use	<span style="color: blue;">🔗</span> License and copyright
<span style="color: green;">✓</span> Modification	<span style="color: red;">✗</span> Liability	<span style="color: blue;">🔗</span> notice
<span style="color: green;">✓</span> Distribution	<span style="color: red;">✗</span> Warranty	<span style="color: blue;">🔗</span> State changes
<span style="color: green;">✓</span> Patent use		
<span style="color: green;">✓</span> Private use		

Figure 31: Gson license [Gso18]

Project and workspace settings differ in the amount of key-value pairs they manage. Both configuration files, in project and in workspace, manage the `ctylechecker.config` key, which persists the entire configuration in JSON format as its value. The project settings additionally manage two keys: `ctylechecker.config.reference` and `ctylechecker.config.setting`. With the help of these additional informations, a project can use workspace settings, reference styleguides from the workspace or use entirely unique configurations on project level. An example of this can be seen in Figure 32.

Report

53

HSR FS 2018

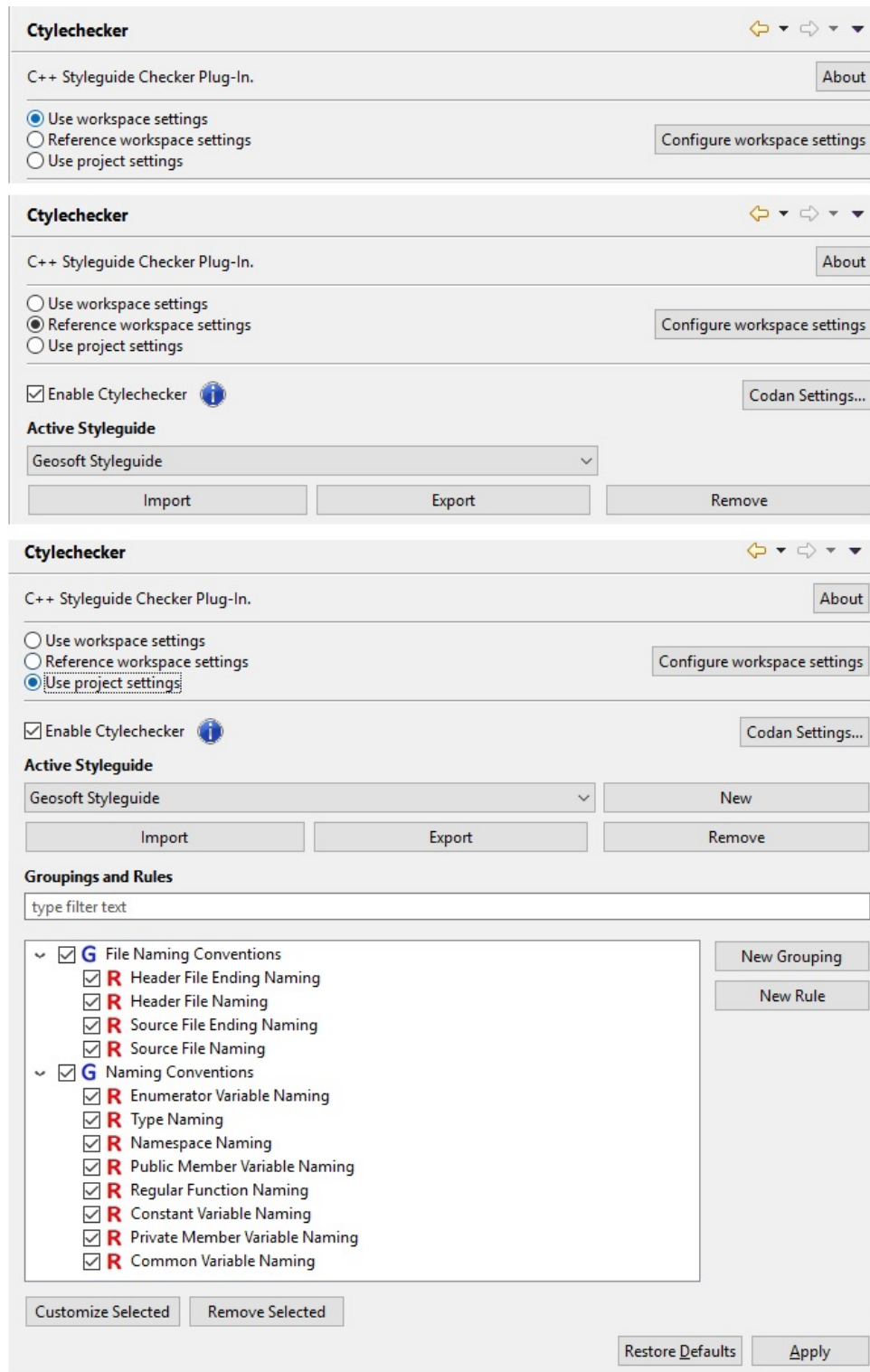


Figure 32: Different setting configurations on project basis

## 6.5 RTS Testing

In order to test the Checkers and Quickfixes, RTS Tests were used. RTS Tests are a special form of integration tests provided by the IFS's testing plug-in (`ch.hsr.ifs.cdtesting.testingPlugin`), which allows to define TestCases as before and after file states, specifically file state after a Checker has been applied or before and after file states of a Quickfix application. TestCases look like in Figures 33 and 34.

```

1  //!Geosoft:Naming violation with simple int variable found
2  //@.config
3  markerLines=4
4  activeStyleguide=predefined:Geosoft
5  //@foo.cpp
6  1 #include <iostream>
7  2
8  3 void main() {
9  4     int test_number{0};
10 5     std::cout << testNumber;
11 6 }
```

Figure 33: DynamicStyleCheckerTest Checker TestCase

```

1  //!Test Predefined Styleguide Google, Class fix
2  //@.config
3  markerLines=1
4  activeStyleguide=predefined:Google
5  //@main.cpp
6  1 class my_value{};
7  //main.cpp
8  1 class MyValue{};
9  2
```

Figure 34: DynamicStyleResolutionTest Quickfix TestCase

For Stylechecker plug-in tests, an additional test property was introduced in the form of `activeStyleguide=[OPTION]:[VALUE]`. Over this property a TestCase can run with specific styleguides. There are three options for the `activeStyleguide` property: `predefined`, `custom` and `file`. `Predefined` option allows to load a styleguide from the set of predefined styleguides (Google, Canonical and Geosoft). The `custom` option allows to define a styleguide in JSON format directly in the TestCase. The third option `file` allows to reference a styleguide file to be used for that TestCase. Referenced files are loaded from the styleguides folder as shown in Figure 35.

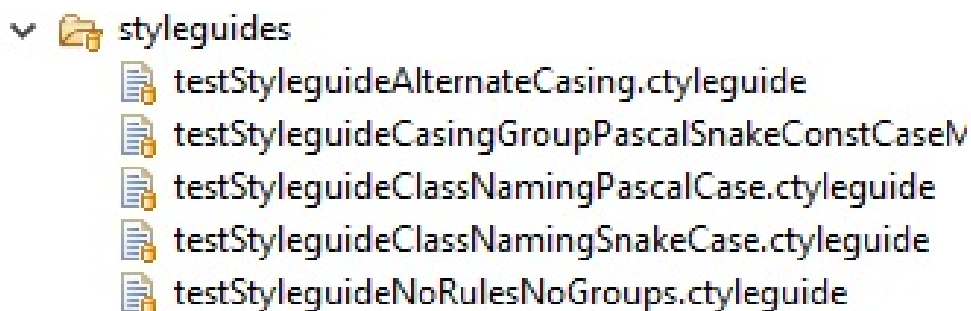


Figure 35: Styleguides folder excerpt

These styleguide files can be generated via the styleguide dialog in the Stylechecker preferences (export styleguide). This way there exists a convenient way to define styleguide files and use them as TestCases without having to manually type the JSON representation of a styleguide directly.

```
testStyleguideNoRulesNoGroups.styleguide 1 [{"groupings": {}, "rules": {}, "name": "TestCase1", "id": "a5237c46-56a1-469e-a7d6-971eae6fba41", "enabled": true}]
```

Figure 36: Styleguide file excerpt

With these three options, the Stylechecker plug-in can be extensively tested.

## 6.6 Stylechecker and Codan Preferences

Because the Stylechecker plug-in extends the Codan plug-in, some of its settings are dependent on Codan settings. These settings are mostly Checker specific settings, which determine whether a Checker is to be considered at all in the code analysis procedure, for which files to consider a Checker and when to run the specific Checker. A Checker's Codan settings look like in Figure 37.

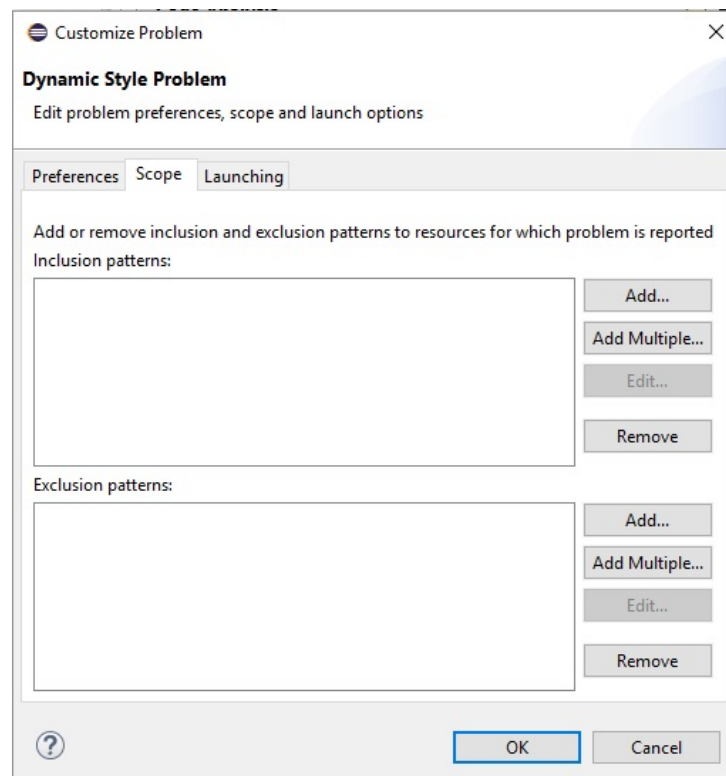


Figure 37: Codan Checker settings excerpt

The decision to delegate to Codan settings where Checker specific configurations are concerned was made due to the many complications that would arise otherwise. Manipulating the Codan settings from outside of Codan plug-in is very tricky and relies on unsafe operations. That is why there are info messages in the Stylechecker preferences where Codan settings are concerned and a quick access is implemented so that the Codan settings can be reached quickly from the Stylechecker plug-in.



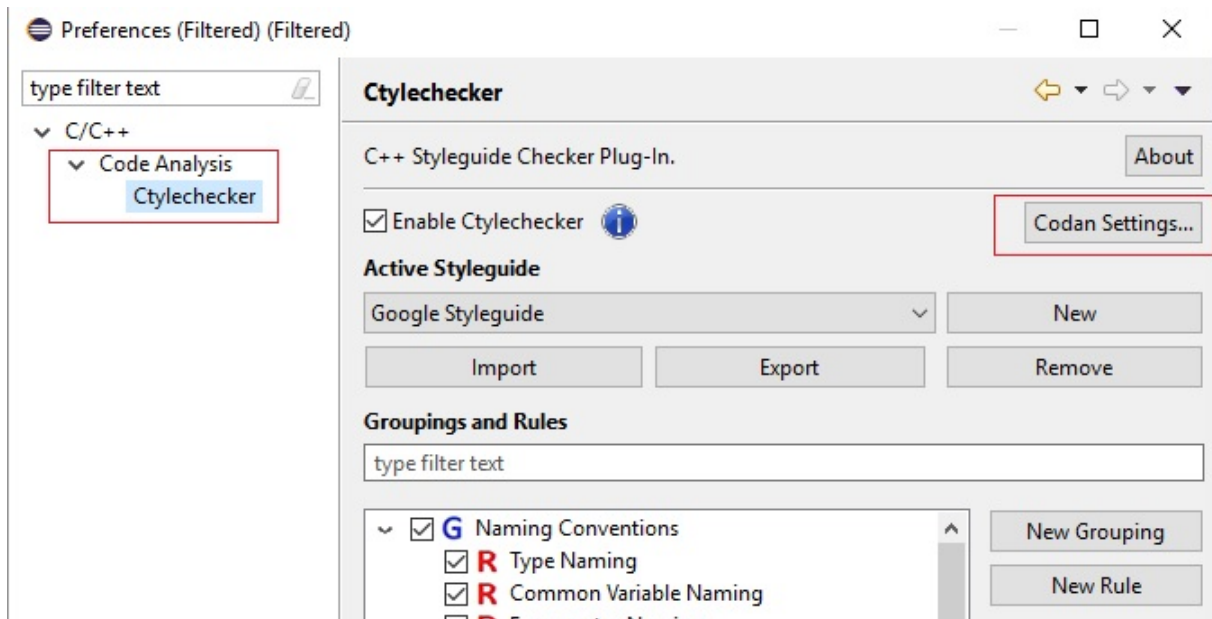


Figure 38: Stylechecker Codan relationship

Due to its strong relation to the Codan plug-in, the Stylechecker plug-in is also grouped under the Codan settings page.

## 6.7 Performance

The Stylechecker plug-ins performance related non-functional requirements are defined as follows:

- Styleguide resolutions (singular refactoring) need to be executed within:
  - three seconds in singular file refactorings
  - five seconds on entire workspace refactorings

in small to middle sized projects.

For Stylechecker plug-in internal operations, these goals hold true. But because the Stylechecker plug-in delegates rename operations to the Eclipse CDT's Rename Refactoring, these goals might not hold true for very large projects where a name is referenced in multiple files and across projects. This might be especially relevant in larger projects where custom C++ libraries are referenced in multiple projects.

Nonetheless, the decision to use the internal Rename Refactoring from the Eclipse CDT is still a sound one: Relying on the Eclipse CDT's Rename Refactoring brings with it the benefit of optimized and working rename refactoring logic. Therefore the tradeoff for non-functional requirements not exactly holding true for very large projects is acceptable.

Another performance boost is delivered by the CDT's Abstract Syntax Tree (AST) approach. Rather than searching all files manually for a specific character string, using the AST from the Eclipse CDT improves name lookup significantly by relying on the visitor pattern [EGea94] to travel over all name nodes of the AST.

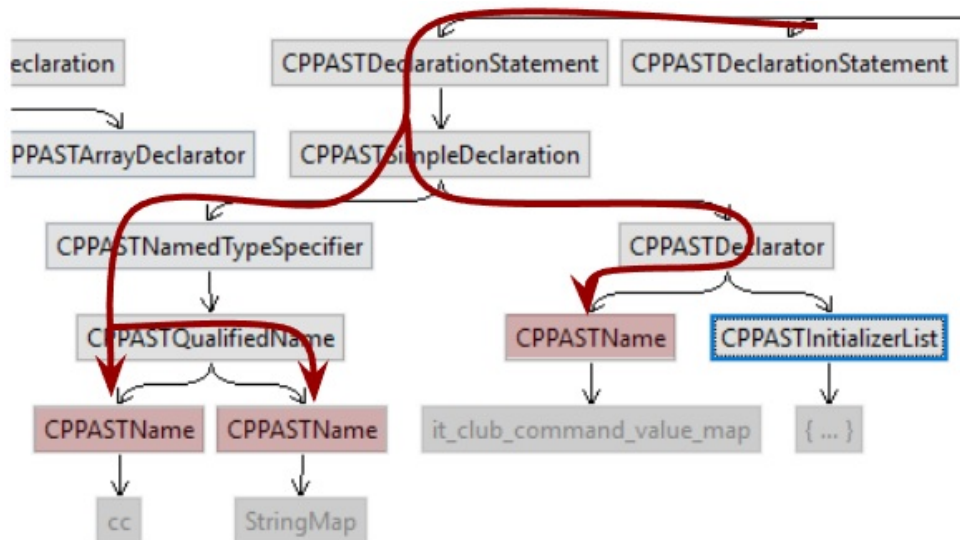


Figure 39: AST name traversal example

The additional benefit of using this approach is that via a found name, all references of the same name can be automatically retrieved from the name provided by the AST (IASTName interface).

## 7 Architecture

This section analyzes the Stylechecker plug-in's architecture and explains the most important relationships.

### 7.1 Dependencies

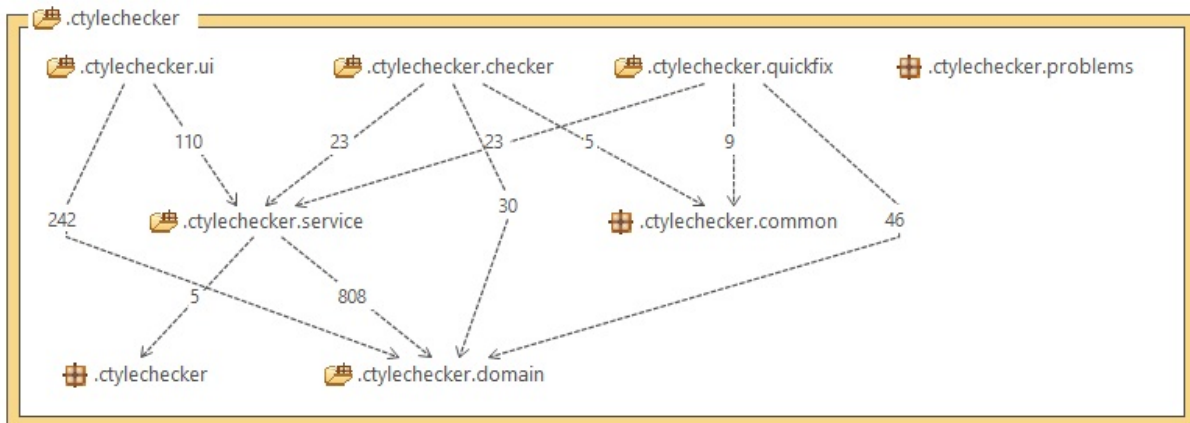


Figure 40: General overview packages

Figure 40 shows a general overview of the packages of the Stylechecker plug-in and their dependencies. The ui, checker and quickfix packages use the service package to access the corresponding models for styleguide operations. The persisted data are called once per Checking and Quickfix, to ensure access to the most recent configuration.

#### 7.1.1 Checker and Quickfix

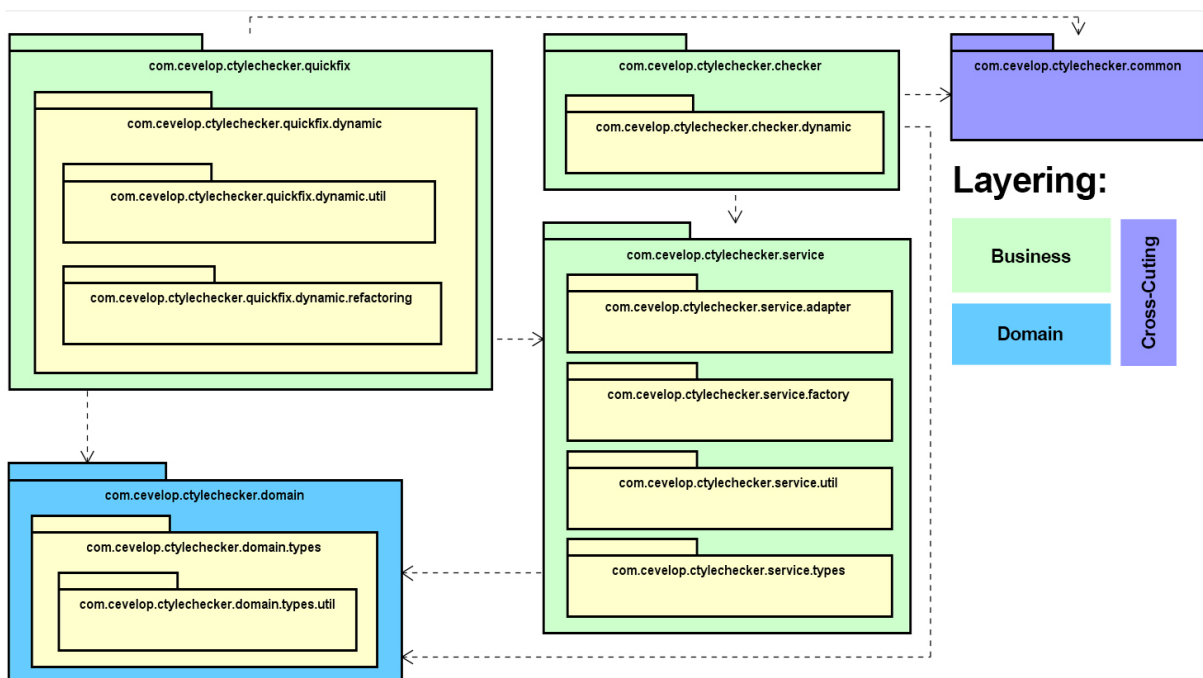


Figure 41: Checker and Quickfix overview

### 7.1.2 Plug-in internals

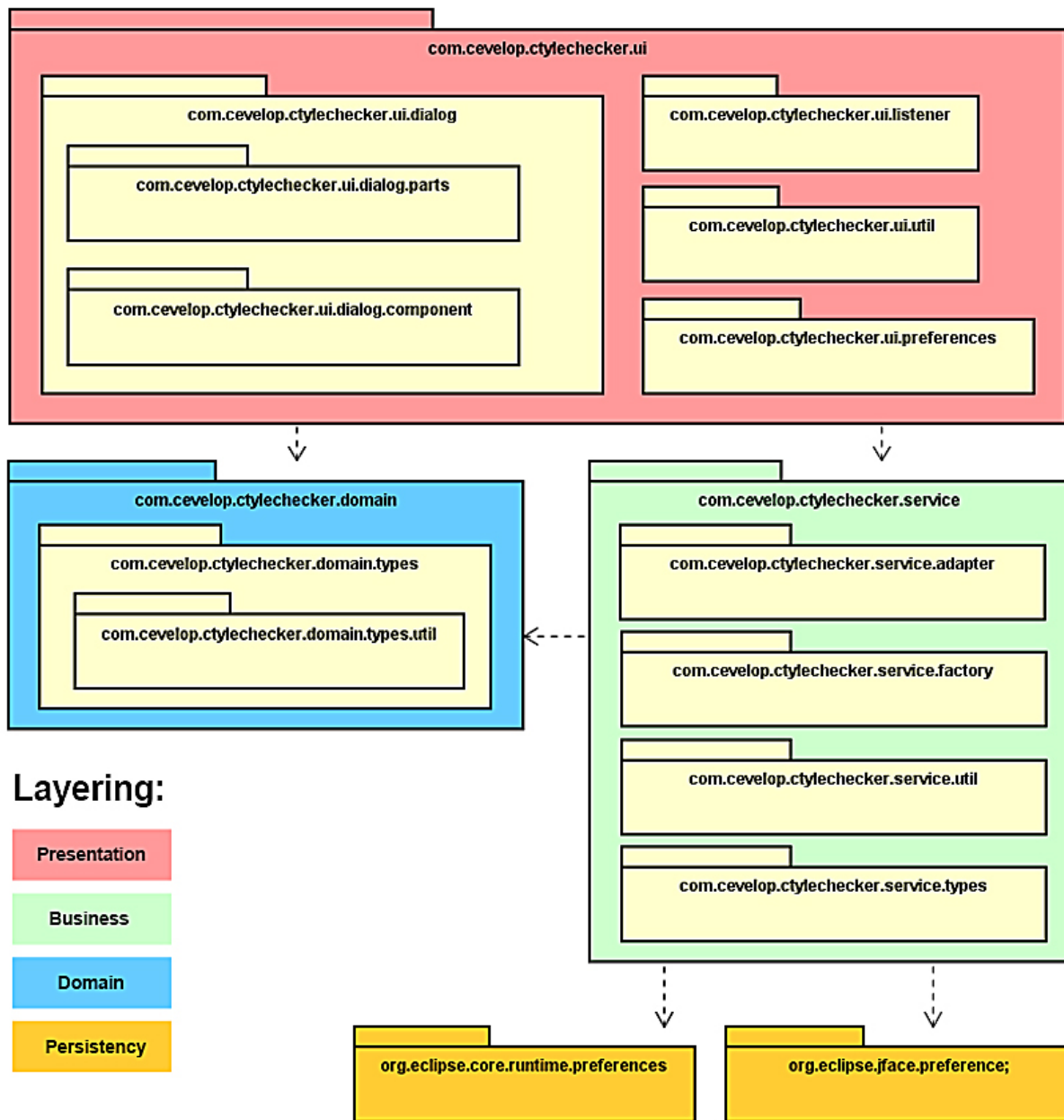


Figure 42: Plug-in internals overview

## 7.2 Packages

This section discusses the packages of the Stylechecker plug-in and provides a quick summary about the packages and their roles. The domain package is looked at in-depth due to its central role in the Stylechecker plug-in.

### com.cevelop.ctylechecker.domain

The com.cevelop.ctylechecker.domain package contains all styleguide relevant objects, such as Styleguide, Grouping, Rule, Expression and many more. The dependencies among the domain elements can be seen in Figure 43.

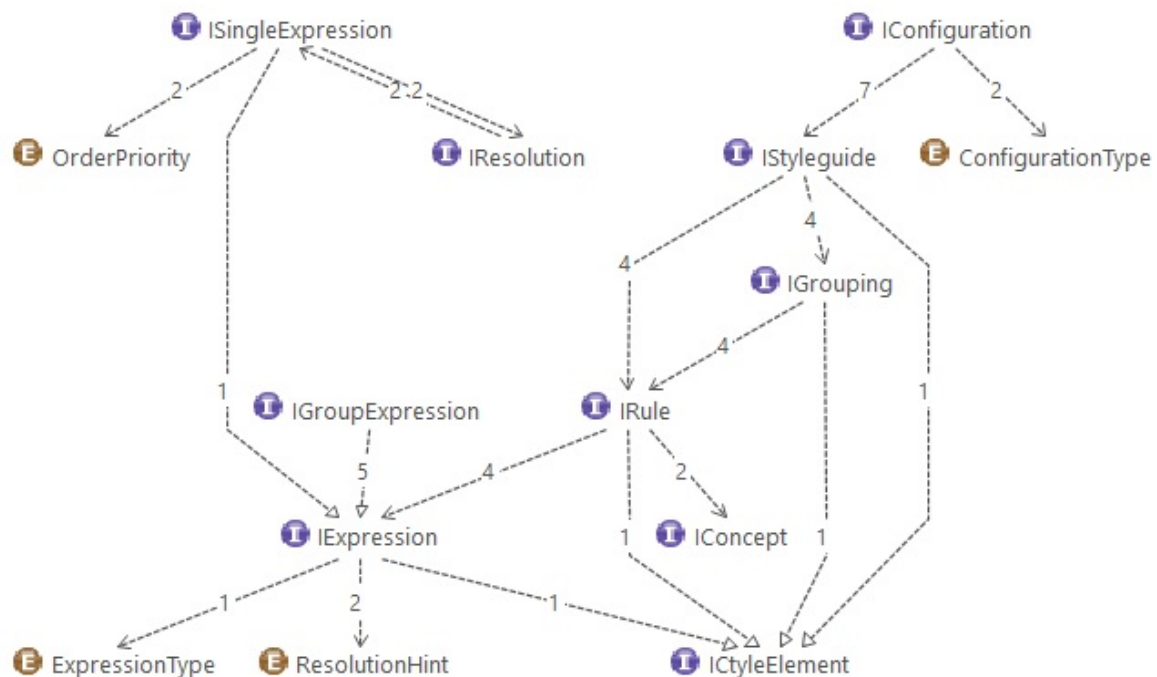


Figure 43: com.cevelop.ctylechecker.domain overview

The com.cevelop.ctylechecker.domain package contains the interfaces and enumerations of the concrete implementations. Beyond the service level, only the interfaces are referenced, which is why in the packages overview ui has references to the domain layer.

Although a better separation could be reached with DTOs on the business layer (where packages service, checker and quickfixes live), it is an intermediary solution due to time constraints not allowing to implement an additional in-between layer to decouple ui and domain more properly. The same goes for com.cevelop.ctylechecker.checker and com.cevelop.ctylechecker.quickfix packages. They also reference only the interfaces from the domain layer. Calls to the domain layer are realized via the service package. This approach was deemed acceptable, as Codan plug-in uses the same approach. See the discussion in the Conclusion section 8.2 for more information.

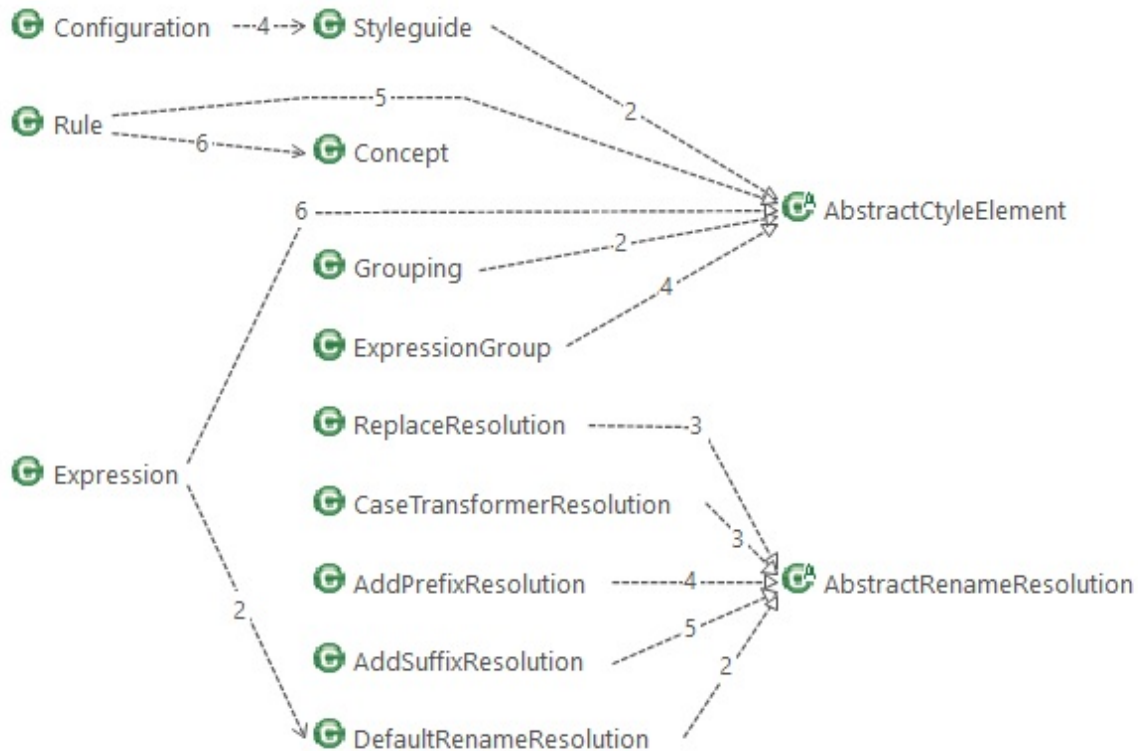


Figure 44: com.cevelop.ctylechecker.domain.types overview

The com.cevelop.ctylechecker.domain.types package contains the concrete implementations, Figure 44 shows the relationships between the classes. It can be useful to know that AbstractCtyleElement implements ICtyleElement and AbstractRenameResolution implements IResolution; both provide common logic for concrete classes implementing these interfaces.

Noteworthy interfaces and their methods are discussed in more detail below. Simple getters and setters were excluded. Figure 45 shows an overview of the interfaces to be discussed.

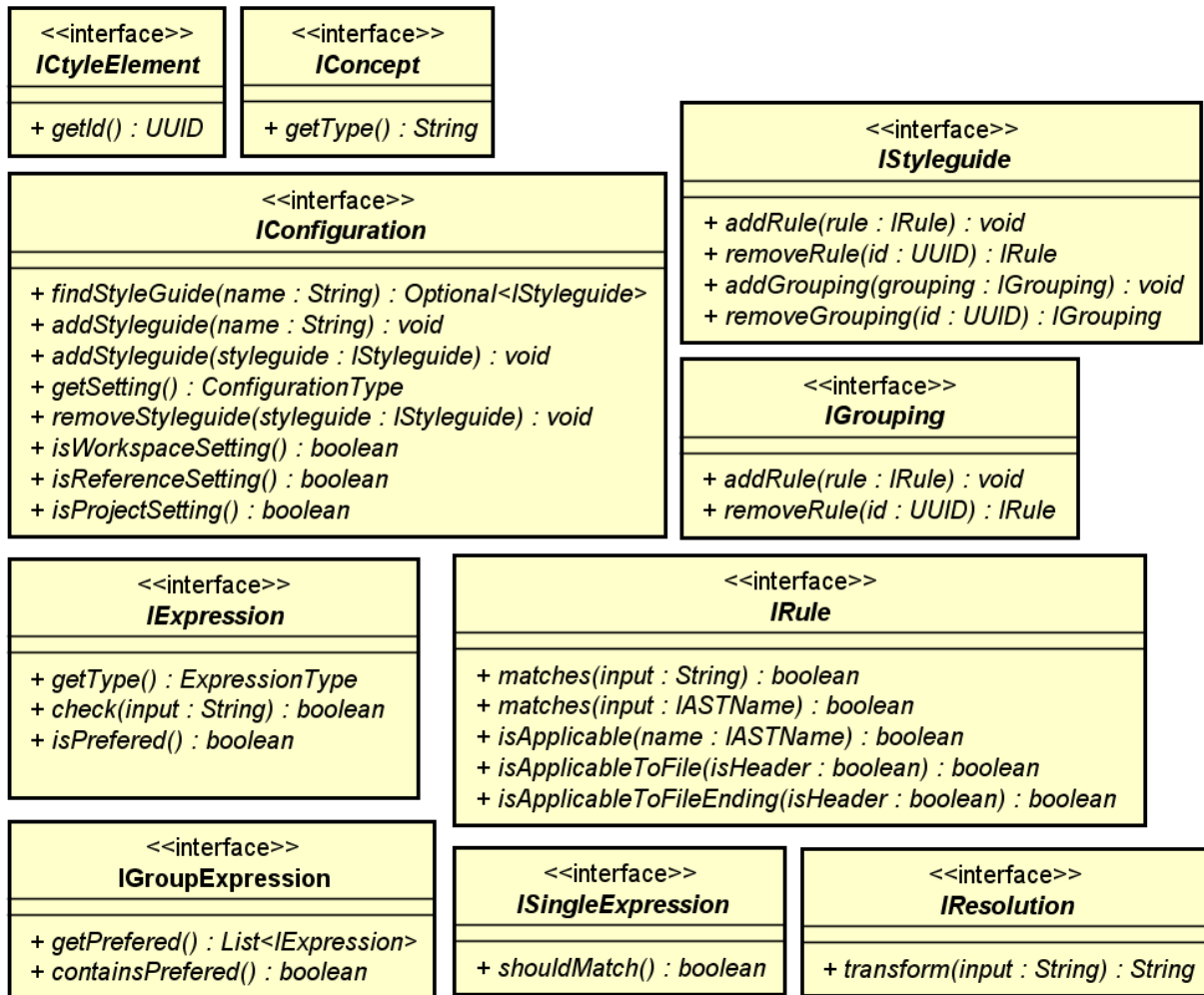


Figure 45: Important domain interfaces and methods

### ICTyleElement

The interface ICTyleElement provides common methods for all domain elements that are to be uniquely persisted.

What	Description
getId	Method that retrieves a unique ID. The idea is generated at random when a new object is created and serves to uniquely identify persistence relevant domain objects.

Table 25: ICTyleElement methods

## IConcept

The interface IConcept describes an implementation of a language element, a language concept.

What	Description
getType	Returns the type a concept is representing. A type can be e.g. CPPVariable, CPPFunction etc., each representing a C++ language element.

Table 26: IConcept methods

## IConfiguration

The interface IConfiguration represents a contract for configuration implementations. It models Stylechecker plug-in configurations.

What	Description
findStyleGuide	Method to find a styleguide by name from existing styleguides in configuration object.
addStyleguide	Styleguides can be added via a name string, which creates an empty styleguide, which is added to the list of available styleguides in the configuration. Alternatively, a styleguide can be created from the outside and added to the configuration.
getSetting	There are three types of settings for a configuration: Workspace (ConfigurationType.WORKSPACE), referencing workspace from project (ConfigurationType.REFERENCE) and project settings (ConfigurationType.PROJECT). This method returns an enum designating the type of configuration this object represents.
removeStyleguide	In order to remove a styleguide from the configuration, the styleguides is passed as an argument. Internally, the styleguide ID is used to uniquely identify the relevant styleguide.
isWorkspaceSetting	Returns true if configuration type is ConfigurationType.WORKSPACE.
isReferenceSetting	Returns true if configuration type is ConfigurationType.REFERENCE.
isProjectSetting	Returns true if configuration type is ConfigurationType.PROJECT.

Table 27: IConfiguration methods



## IStyleguide

The IStyleguide interface defines a contract for styleguide implementations. It models the entirety of a styleguide definition.

What	Description
addRule	Rules can be added to a styleguide via this method. Every added rule is saved in a map, where the ID of the rule is the key.
removeRule	Rules can be removed from the styleguide via their ID.
addGrouping	Groupings can be added to a styleguide via this method. Every grouping, like rules, are saved via their ID.
removeGrouping	Groupings can be removed from the styleguide via their ID.

Table 28: IStyleguide methods

## IGrouping

The interface IGrouping defines a contract for grouping implementations. A grouping implementation is used to group rules in a styleguide to provide better overview.

What	Description
addRule	A Grouping groups rules. Via this method, rules can be added to the Grouping. Like with styleguides, rules are stored with their ID in an internal map.
removeRule	Rules can be removed via their ID from a Grouping.

Table 29: IGrouping methods

## IExpression

The IExpression interface defines a contract for base Expression implementations. It is extended by ISingleExpression and IGroupExpression, more concrete contracts for more concrete implementations.

What	Description
getType	There are two types of Expression, one is Expression and the other ExpressionGroup. The ExpressionType enum models this with ExpressionType.SINGLE and ExpressionType.GROUP. This method offers a quick identification.
check	The check method checks whether a given input string is matched by the Expression or Expressions in an ExpressionGroup.
isPreferred	A simple indicator, whether the current Expression/ExpressionGroup is preferred or not. This is a special kind of ResolutionHint that indicates which Expressions/ExpressionGroups to reference when resolving a violation.

Table 30: IExpression methods

## IRule

The IRule interface defines a contract for Rule implementations. A rule describes what is considered a violation and what isn't.

What	Description
matches	The matches method provides business logic to check whether a string or a IASTName conforms to the constraints set by the rule.
isApplicable	Checks whether an IASTName is applicable to this rule. With this method the checker decides whether a rule is relevant for this type of IASTName.
isApplicableToFile	This method is used to check whether a rule is applicable to a file name (body).
isApplicableToFileEnding	Like isApplicableToFile, this method checks whether a rule is applicable to a file ending.

Table 31: IRule methods

## IGroupExpression

The IGroupExpression interface defines a contract for expression group implementations. Expression groups model a set of expressions, which can be matched against. This is especially important for allowing variations within a rule.

What	Description
getPrefered	Retrieves all expressions and expression groups with the ResolutionHint.PREFERED.
containsPrefered	Checks whether an expression group contains expressions or expression groups with ResolutionHint.PREFERED.

Table 32: IGroupExpression methods

## ISingleExpression

The ISingleExpression interface defines a contract for singular expressions. They model the smallest unit of checking logic.

What	Description
shouldMatch	A boolean value indicating whether an expression's regex is supposed to be matched or not.

Table 33: ISingleExpression methods

## IResolution

The IResolution interface defines a contract for resolution implementations.

What	Description
transform	This method defines the transformation logic a resolution implements when solving a violation.

Table 34: IResolution methods

### com.cevelop.ctylechecker.ui

The com.cevelop.ctylechecker.ui package contains the classes for the Stylechecker plug-in user interface. It contains the necessary dialogs and preference pages. It also contains all the composites that make up the internals of the UI elements. As the Stylechecker plug-in's user interface is mainly situated in the Eclipse preferences, the user interface elements consist mainly of composites, which are used in the plug-in hooks for the preference and property pages. Composites that need to access the business logic layer extend the AbstractCtylecheckerComposite (see Figure 13), which provides access to the services.

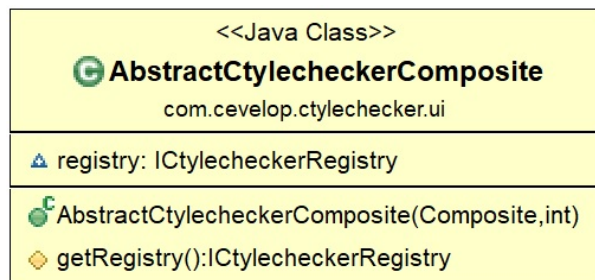


Figure 46: AbstractCtylecheckerComposite class

In the figure below, an overview of the ui package is illustrated.

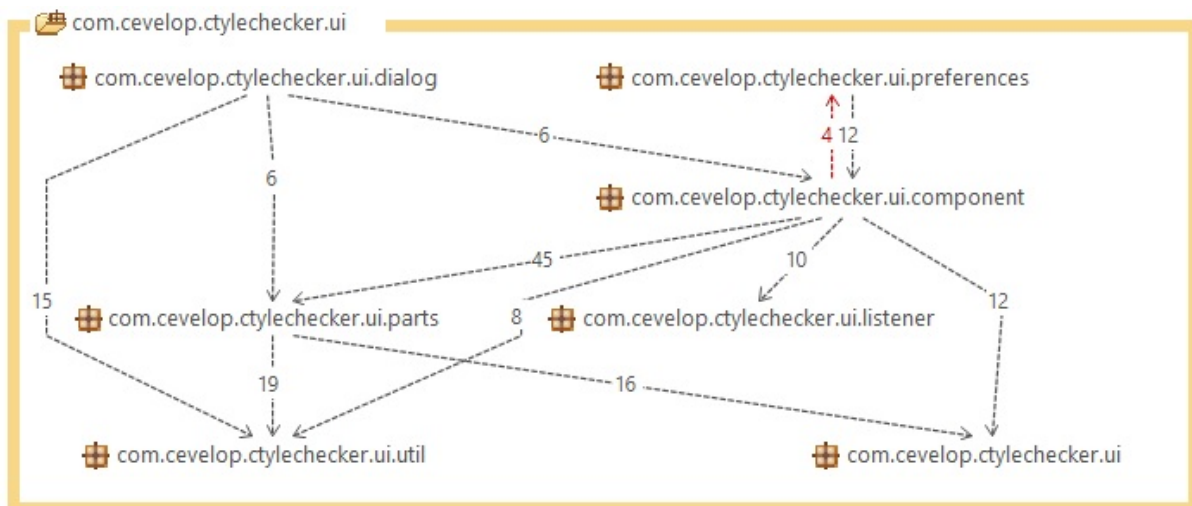


Figure 47: Overview com.cevelop.ctylechecker.ui package

Due to special options for the Stylechecker plug-ins project preferences, the property page needs to be updated from within the SettingsComposite, which is why there exists a circular dependency between the ui.preferences and ui.component packages.

### com.cevelop.ctylechecker.service

The com.cevelop.ctylechecker.service package contains services like IRuleService, IStyleguideService etc. and provides business logic in the business logic layer. The com.cevelop.ctylechecker.service package also contains the necessary facilities to persist Styleguide plug-in settings. It uses persistency entry points, such as IEclipsePreferences or IPreferenceStore to persist data in the Eclipse environment. For this, the ConfigurationService converts domain objects into JSON representation and persists them in the corresponding configuration files (workspace and project level).

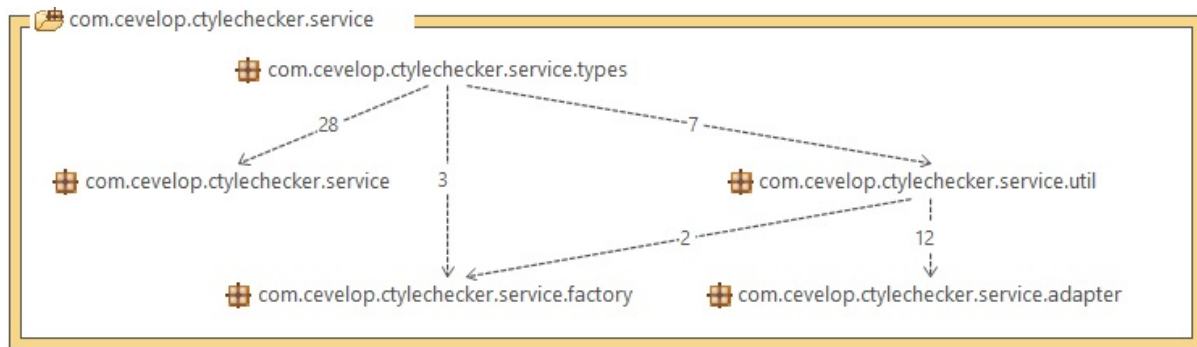


Figure 48: Overview com.cevelop.ctylechecker.service package

### com.cevelop.ctylechecker.checker

The com.cevelop.ctylechecker.checker package contains the DynamicStyleChecker checker class and is the main source for checker operations. It is a singular entry point to all styleguide related checker operations.

### com.cevelop.ctylechecker.quickfix

The com.cevelop.ctylechecker.quickfix package contains the DynamicStyleResolution quickfix class and is the main source for quickfix operations of the Stylechecker plug-in. It uses helper classes like ASTRenameRefactoring and FileNameRefactoring to delegate refactoring logic to Eclipse internal refactoring facilities.

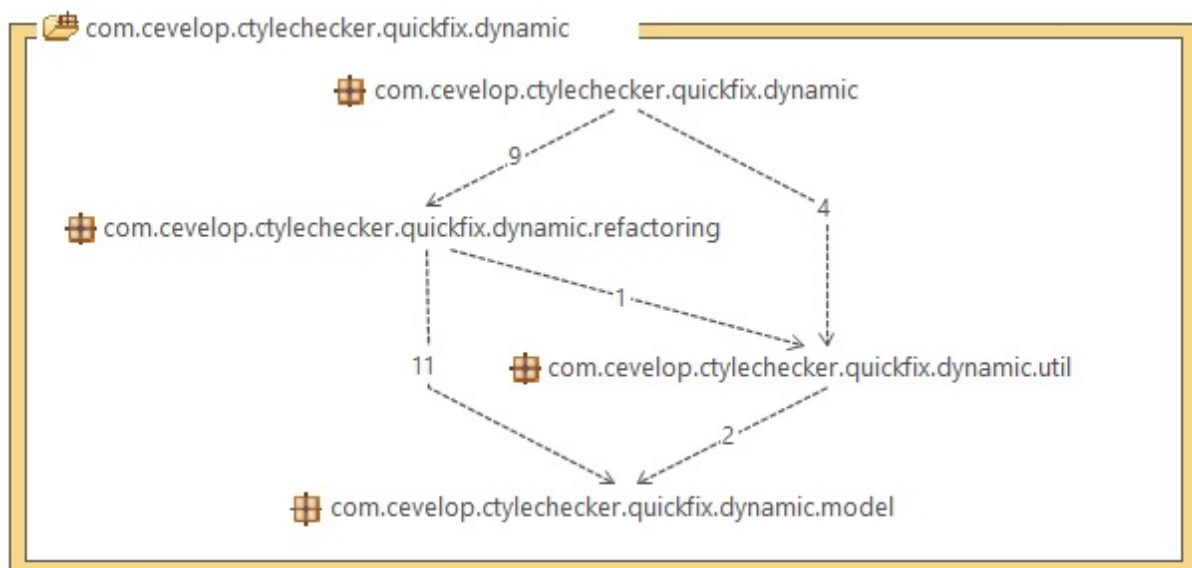


Figure 49: Overview com.cevelop.ctylechecker.quickfix package

### com.cevelop.ctylechecker.common

The com.cevelop.ctylechecker.common package contains common logic to presentation, business and domain layer. It is situated in the corss-cutting layer.

## 8 Conclusion

The target features within the scope of the projects were able to be implemented. The Stylechecker plug-in can check styleguides, report them and offer automated resolutions, wherever possible. But due to time constraints and project group size, some features were implemented in the most basic way, such as File Inclusion / Exclusion feature, which basically is just a reroute to the Codan plug-in. Along the way, multiple new ideas for new features came up as well. The following sections will discuss therefore open features that could be implemented in case of a continuation of development and also will review some potentials for optimization.

### 8.1 Open Features

Feature	Description
Bulk solve multiple problems	Feature to solve multiple problems at the same time.
Advanced UI Mode	An advanced UI mode feature, which would allow to use more complex settings and access more complex concepts to check against. This way new users wouldn't be overwhelmed with the amount of settings and advanced users could customize the Stylechecker to their hearts content.
Limitation of checked concepts on Expression Group basis	A limitation on basis of Expression Groups on which concepts the Expressions should be checked on would allow for more complex rule definitions and to perform a resolution in one step.
Settings versioning and auto transformation	As the plug-in might change in future, a feature that would detect changes in the settings structure and offer automatic transformation to the new structure. This way older styleguide settings wouldn't be invalidated with newer versions of the plug-in.
Special code markers for styleguide checking enable/disable	Like @Supress annotation in Java, special comments or markers in code with which stylechecking for a certain section of a code could be enforced or forced to skip.
AI assisted resolution	It would be interesting to have a resolution type that would be a trained AI that would determine how to best convert a string to a target casing by analyzing multiple code sources. This would for example allow for semantically correct transformations, considering the meaning of the words in the resolution procedure.
String to Regex converter	Basically the reverse procedure with regex matching. A feature that would allow a user to define a target regex, which would be used as the basis for a resolution. This way resolutions could be defined in a dynamic way.

Table 35: Open features overview table

## 8.2 Potential for Optimization

In this section the optimization potentials for the Stylechecker plug-in are discussed. The section is divided into two subsections: "General optimizations" and "Architectural optimizations".

### 8.2.1 General optimizations

There is still a lot of potential for optimization in the Stylechecker plug-in. First optimization source that comes to mind is the Expressions UI part in the Rule dialog, which at the moment is a little bit overloaded with text. It could be improved by using visual cues to indicate for example whether a group matches ALL or ANY with an icon rather than with text. Optimizations like these would make reading the Expressions easier. Additionally, Expression Groups and Expressions use the same tree view, which results in many fields being empty for the Expression Groups as those settings just simply don't exist on that level. A different approach in visualizing the Expressions and Expression Groups is probably necessary if the Stylechecker is to be developed further in the future.

Another source of optimization would be a consolidation of all features within the Stylechecker plug-in itself without having to reference the Codan plug-in. There are ways to initiate Codan procedure in order to, for example, enable or disable a checker from the outside, but the implementation can get tricky due to hard to control update procedure that are controlled by the Codan UI. But a consolidation would make using the Stylechecker plug-in more convenient, eventhough most of the settings are one time settings anyway.

Furthermore, some UI elements still could need some proper refactoring work as some of them are still bloated due to limited time. SettingsComposite for one is still a huge block of code.

### 8.2.2 Architectural optimizations

The current architecture provides a special service layer to access domain elements. Furthermore the UI only references the interfaces from the domain layer like the Codan plug-in. Still, this leads to a strong coupling between the UI and domain layer. Although the elements for a proper architectural separation are provided, the separation is incomplete. Additionally, some domain elements still have business logic within them, which could make extending the Stylechecker plug-in problematic for future features (bloating of domain objects). Therefore two optimizations are suggested.

The first is to decouple the UI even stronger by providing DTOs to communicate with the service layer. This way the UI would not need to reference the domain layer directly and the UI would become more resistant to change. The second optimization is to implement business objects in the business logic loayer using domain elements to provide business logic rather than having business logic directly in the domain elements. Through this approach new business logic could be introduced easier and domain objects would become thinner.

### 8.2.3 Rename Refactoring issue

During development a bug in the rename refactoring for C++ template functions was found. The bug [Ecl18] was reported on the Eclipse Bug Report site.

### 8.3 Outlook

The Stylechecker plug-in now offers the functionality to check and fix naming conventions. But naming conventions only make up a part of a styleguide. In order for the Stylechecker plug-in to comprehensively aid in the adherence to a styleguide, multiple additional features still need to be implemented; some of them are mentioned in the open features section.

Currently other "style" checkers exist for C++. They often focus on more standard related adjustments than a true styleguide approach. For example Vera++ [Ver18] reports "style" violations like "trailing whitespace", "if not followed by a single space" etc. In this respect existing "style" checkers are most often more like linters, although some of them, e.g. Vera++, do also provide transformations in certain situations. Furthermore, only a few use an Abstract Syntax Tree (AST) approach like for example the "Clang-format" tool.

With the Stylechecker plug-in now a solid basis for a more comprehensive styleguide checking tool is provided. The Stylechecker plug-in can benefit from previous work done in various different C++ style checker implementations and in future maybe even collect the most interesting angles to style and styleguide checking within one powerful tool. Vera++'s Script API approach is one additional feature that directly comes to mind that could be interesting to have inside the Stylechecker plug-in. But it is important to properly distinguish what kind of tool the Stylechecker plug-in is or is to be when considering features like these. Is it to be a style checking tool only, i.e. only checking the look and feel of source code or is it to offer additional features as well, like what is proper coding style in order to avoid problems at execution time. Questions like these are important when considering to extend the Stylechecker plug-in even further in the future.

A comprehensive styleguide checking tool could make styleguides and properly styled source code more popular and increase readability and understandability of source code; ultimately increasing productivity of developers. Therefore it will be interesting to follow the development of this plug-in in the future.



## 9 Glossary

The glossary contains all abbreviations and terms that need further elaboration.

Term	Description
Codan	<b>Code Analysis.</b> The Codan plug-in is an internal plug-in of the Eclipse CDT providing code analysis capabilities.
Checker	A Checker is an implementation that "validates" the contents of a source code and upon violations reports those via Markers in the editor.
Quickfix	A quickfix is the counterpart to a Checker and allows to apply a fix/a resolution to the reported violation by the Checker
Marker	A special indicator in a file, reporting a problem/violation found by a Checker.
Styleguide	A styleguide is a set of rules that guide the look and feel of source code.
Grouping	A construct to "group" rules.
Rule	A rule is a specific guide definition within the styleguide that controls whether something is OK or not. The Stylechecker Checkers use rules as their basis when reporting violations.
Expression	An expression in the context of the Stylechecker is a domain element containing a regular expression and additional information like name, resolution, hints etc. An expression therefore is a beefed up regex construct used by rules to determine validity of source code.
IDE	Integrated development environment
Eclipse	An open-source, highly extendable IDE developed by IBM and available for multiple languages.
CDT	C/C++ Development Tooling.
AST	Abstract Syntax Tree. A representation of source code in tree form.
IFS	Institute for Software
JSON	JavaScript Object Notation
Gson	JSON Library from Google
AI	Artificial intelligence
DTO	Data Transfer Object

## 10 References

- [Boo18] Boost. Boost requirements. <http://www.boost.org/development/requirements.html>, Accessed on: 9th June 2018.
- [Can18] Canonical. Canonical C++ Style Guide. <http://people.canonical.com/~msawicz/guides/c++/cppguide.html>, Accessed on: 12th March 2018.
- [Cev18] Cevloop. Cevloop. <https://www.cevloop.com>, Accessed on: 9th June 2018.
- [Cor18] Thomas Corbat. Assignment bachelor thesis "Cevloop C++ Stylechecker Plugin, Assignment.pdf. E-Mail, 26th February 2018.
- [Dog18] Zafer Dogan. Session protocol. [https://gitlab.dev.ifs.hsr.ch/zdogan/cevloop-stylechecker/blob/master/docs/02\\_protocols/Protokoll\\_26022018.pdf](https://gitlab.dev.ifs.hsr.ch/zdogan/cevloop-stylechecker/blob/master/docs/02_protocols/Protokoll_26022018.pdf), Accessed on: 26th February 2018.
- [Ecl18] Eclipse. Eclipse Bug with ID 536160. [https://bugs.eclipse.org/bugs/show\\_bug.cgi?id=536160](https://bugs.eclipse.org/bugs/show_bug.cgi?id=536160), Accessed on: 22th June 2018.
- [EGea94] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
- [Geo18] Geosoft. Geosoft C++ Styleguide. <http://geosoft.no/development/cppstyle.html>, Accessed on: 4th March 2018.
- [Goo18] Google. Google C++ Styleguide. <https://google.github.io/styleguide/cppguide.html>, Accessed on: 4th March 2018.
- [Gso18] Gson. Gson github page. <https://github.com/google/gson>, Accessed on: 22th June 2018.
- [HSR18a] HSR. HSR Allgemeine Infos Diplom Bachelor. <https://www.hsr.ch/Allgemeine-Infos-Diplom-Bach.4418.0.html>, Accessed on: 9th June 2018.
- [HSR18b] HSR. HSR Archiv Website. <https://archiv-i.hsr.ch>, Accessed on: 9th June 2018.
- [Lar05] Craig Larman. *Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Pearson Education inc., third edition edition, 2005.
- [Ver18] Vera++. Vera++ Bitbucket page. <https://bitbucket.org/verateam/vera/wiki/Home>, Accessed on: 22th June 2018.

## 11 Appendix A

### 11.1 Project Plan

The project plan lays out the plan for the bachelor thesis. It also includes a record of all the meetings, used tools and arrangements to ensure code quality.

#### 11.1.1 Project Overview

The stylechecker plug-in for Cevelop, a C++ IDE developed by the IFS at the HSR, is a plug-in to check different C++ coding styleguides and is developed for the IFS within the scope of the bachelor thesis by Zafer Dogan.

#### Purpose and Objective

The goal of the project is to develop a C++ coding styleguide's stylechecker plug-in for the IFS developed Cevelop IDE. The plug-in is to support two major styleguides, in the scope of the bachelor thesis these being the Google and Boost C++ styleguides, out of the box with a third one chosen by the student. The plug-in also needs to allow for custom styleguides to be defined, so company internal styleguides can be defined and used with the plug-in.

#### Delivery Scope

The project will be delivered as an installable plug-in. Additionally, following documents will be provided: Project plan, domain model, use cases and non-functional requirements, sequence diagrams, contracts, SAD, test definitions, test protocols, final version of the plug-in, final report and the presentation of the project.

#### Assumptions and Limitations

One of the major limitations of the project is the extent of the stylechecker plug-in that can be realistically developed. As the bachelor thesis initially was designed for 2-3 students, the limitation of providing naming convention checking as a must have was set due to the project being executed by only one student. Additionally, the following limitations have been set: The documentation needs to be in English [Cor18] and written in LaTeX [Dog18]. Furthermore, after analysis of the Boost Styleguide, it was decided to opt for a more convenient guideline due to the nature of the Boost Styleguide definition being not very clear about its structure and providing a common interface by using it would prove too difficult. This doesn't mean the Boost Styleguide won't be able to be defined in the plug-in itself, it just won't be one of the predefined Styleguides provided by default.

### 11.1.2 Project organisation

Only one person is participating in this project and accordingly, is filling out all roles that usually arise during the development life cycle. Tasks are split into work packages and distributed to iteration. The development style for the project is agile. The assurance of information and time management are responsibilities of the project leader, which in this case is the same person as the project developer and other roles. The supervisor observes the project flow and is responsible for the grading. The final grade will be determined after the review of the final project by an external expert. The project will be partitioned into the four RUP phases[Lar05] inception, elaboration, construction and transition, but do only serve as a coarse grouping as the project is developed in agile manner.

#### Organisation structure

Project leader & developer:

Zafer Dogan, *zdogan@hsr.ch*

#### External interfaces

Supervisor:

Thomas Corbat, *thomas.corbat@hsr.ch*

Technical Advisor:

Felix Morgner, *felix.morgner@hsr.ch*

Expert:

Lukas Felber, *lukas.felber@quatico.com*

### 11.1.3 Management procedures

This section details the plans for management of the project, including estimations and separations into development phases.

#### Estimate of costs

For the execution of the project 16 weeks are available. In this period, every student has an investment budget of 360 hours (12 Credits). This results in a time budget of total 360 hours as only one student is working this project. The project starts on the 19.02.2018 and ends on 29.06.2018.

#### Time plan

The detailed planning and management of work packages will be managed and tracked with GitLab’s Issue tracking system. The plan will be iteratively adjusted during the projects life cycle according to the needs of the projects circumstances and is coarsely partitioned into RUP Phases.

Issues page:

<https://gitlab.dev.ifs.hsr.ch/zdogan/cevelop-stylechecker/issues>

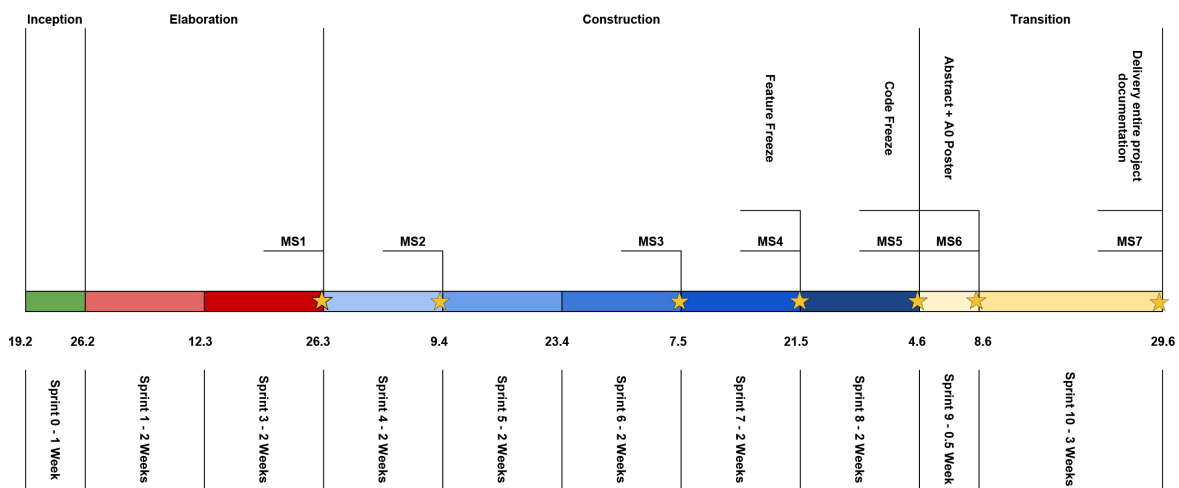


Figure 50: Iteration plan

#### Phases/Iterations

The development will be partitioned into the RUP phases inception, elaboration, construction and transition and serve as coarse grouping for a general reference of the project’ state. The individual iterations carry the names of the phases they belong to and are numbered in an ascending order. The iterations are managed with GitLab’s Issue tracking system. The project is being developed in agile manner.

Start	End	Name	Details
19.02.2018	26.02.2018	I1	<ul style="list-style-type: none"> <li>- Setup of <ul style="list-style-type: none"> <li>– Workspace</li> <li>– Document structure</li> <li>– Code repository</li> </ul> </li> <li>- Researched <ul style="list-style-type: none"> <li>– Code Styleguides</li> <li>– Eclipse PDE</li> <li>– Stylechecker Plug-in</li> </ul> </li> </ul>
26.02.2018	12.03.2018	E1	<ul style="list-style-type: none"> <li>- Preliminary requirements acquired</li> <li>- Use Cases in brief format defined</li> <li>- Working prototype exists</li> <li>- Domain analysis performed and model created</li> <li>- Rough drafts of Settings UI prepared</li> </ul>
12.03.2018	26.03.2018	E2	<ul style="list-style-type: none"> <li>- Non-functional requirements defined</li> <li>- SAD started</li> <li>- Use Cases in brief and essential style defined</li> <li>- First working UI prototype implemented (Settings UI) and basic usability test performed</li> </ul>
26.03.2018	09.04.2018	C1	<ul style="list-style-type: none"> <li>- Naming convention feature finished for Google and Canonical</li> <li>- SAD extended</li> </ul>

Table 36: Iteration table, part 1

Start	End	Name	Details
09.04.2018	23.04.2018	C2	<ul style="list-style-type: none"> <li>- Naming convention feature finished for additional student styleguide</li> <li>- First draft of custom styleguide definition for naming conventions finished</li> <li>- SAD extended</li> </ul>
23.04.2018	07.05.2018	C3	<ul style="list-style-type: none"> <li>- Comment convention feature finished for Google and Canonical</li> <li>- SAD extended</li> </ul>
07.05.2018	21.05.2018	C4	<ul style="list-style-type: none"> <li>- Comment convention feature for student styleguide finished</li> <li>- SAD extended</li> <li>- Feature freeze</li> </ul>
21.05.2018	04.06.2018	C5	<ul style="list-style-type: none"> <li>- Bugfixes and refactorings performed</li> <li>- SAD extended</li> <li>- Code freeze</li> </ul>
04.06.2018	08.06.2018	T1	<ul style="list-style-type: none"> <li>- Delivery of A0 Poster and Abstract</li> <li>- Code freeze</li> </ul>
08.06.2018	29.06.2018	T2	<ul style="list-style-type: none"> <li>- All project documentation finished and packaged to delivery format</li> <li>- Bachelor thesis presentation finished</li> </ul>

Table 37: Iteration table, part 2

**Milestones**

<b>Date</b>	<b>Milestone</b>	<b>Delivery of</b>
26.03.2018	MS1 - Documentation	<ul style="list-style-type: none"> <li>- Project plan</li> <li>- Domain model</li> <li>- Use cases</li> <li>- Non-functional requirements</li> <li>- UI Draft for plug-in settings</li> </ul>
09.04.2018	MS2 - Alpha version	<ul style="list-style-type: none"> <li>- Contracts</li> <li>- Sequence diagrams</li> <li>- SAD Prototype</li> <li>- Settings UI in plug-in</li> <li>- Naming convention feature for Google and Canonical</li> <li>- Test definitions</li> </ul>
07.05.2018	MS3 - Beta version	<ul style="list-style-type: none"> <li>- Beta version</li> <li>- Naming convention feature completed</li> <li>- Comment feature for Google and Canonical</li> </ul>
21.05.2018	MS4 - Release Candidate	<ul style="list-style-type: none"> <li>- Release Candidate</li> <li>- Test protocols</li> </ul>
04.08.2018	MS5 - Release version	<ul style="list-style-type: none"> <li>- Release version</li> <li>- Test protocols</li> </ul>

Table 38: Milestones table, part 1



08.06.2018	MS6 - Abstract and A0 Poster	<ul style="list-style-type: none"> <li>- Abstract</li> <li>- A0 Poster</li> </ul>
29.06.2018	MS7 - Final Delivery	<ul style="list-style-type: none"> <li>- Entire project documentation</li> <li>- Delivery folder</li> <li>- Presentation</li> </ul>

Table 39: Milestones table, part 2

### Meetings

This section contains a listing of all the meetings held during the project development.

### Discussions

Once a week there is a meeting with the project supervisors. The meetings ordinarily will be held at Mondays at 10 o'clock in building 8, room 8.261. At these meetings, the current state of the project will be discussed.

### Reviews

Date	Review	Details
19.02.2018	R0	Initial meeting, assignment discussion
26.02.2018	R1	Discussion project structure, GitLab setup and GitLab Pipelines and Plug-in delivery
05.03.2018	R2	Discussion first use cases, setup Updatesite
12.03.2018	R3	Discussion Updatesite location (GitLab instead of own server), use cases, preliminary domain analysis
19.03.2018	R4	Discussion UI Prototype, discussion finished requirement analysis, discussion domain analysis
26.03.2018	R5	TBD
02.04.2018	R6	No meeting

Table 40: Review table, part 1

<b>Date</b>	<b>Review</b>	<b>Details</b>
09.04.2018	R7	Discussion of persistence implementation, settings UI
16.04.2018	R8	Discussion GSON for persistence, settings UI
23.04.2018	R9	Discussion domain element resolution implementation, check against logic
30.04.2018	R10	Discussion of Prefix and Suffix resolution, autoresolve, adjustment rule ui
07.05.2018	R11	Discussion Expression conflict resolution and new UI elements for Expression/ExpressionGroup
14.05.2018	R12	Discussion Resource Rename refactoring, file marker, refactoring rule composite and adjustment of Expression domain (moved resolutions from Rule to Expression)
21.05.2018	R13	No meeting
29.05.2018	R14	Discussion Expression UI, RTS properties for styleguide checking, import and export feature and styleguide documentation
07.06.2018	R15	Discussion Documentation, Bugfixing, Abstract, A0 Poster and Unit Tests
15.06.2018	R16	Discussion Documentation, Abstract and A0 Poster adjustments
21.06.2018	R17	Discussion Documentation

Table 41: Review table, part 2

#### 11.1.4 Risk management

It was determined with the supervisor that no risk management will be needed for this particular project.

#### 11.1.5 Work packages

Work packages will be defined and tracked in GitLab's issue tracking system. GitLab provides multiple views on the issues, i.e. in list or board form and also allows partitioning into milestones. Furthermore it is possible to filter according to milestones and also to define time estimates and time spent values on issues themselves.

Link: <https://gitlab.dev.ifs.hsr.ch/zdogan/cevelop-stylechecker/issues>

#### Updatesite

The most recent installable plug-in via Updatesite can be queried via following link:

[https://gitlab.dev.ifs.hsr.ch/zdogan/cevelop-stylechecker/-/jobs/artifacts/master/raw/com.cevelop.ctylechecker.updatestite/target/repository/?job=deploy\\_updatesite](https://gitlab.dev.ifs.hsr.ch/zdogan/cevelop-stylechecker/-/jobs/artifacts/master/raw/com.cevelop.ctylechecker.updatestite/target/repository/?job=deploy_updatesite)

This is a special URL that resolves to the latest master branch build. To get the current Updatesite URL resolve the URL and use the resolved URL as the Updatesite. You can ignore the 404 message on the resolved URL; this is a view permission issue with GitLab.

### 11.1.6 Infrastructure

- Development language
  - **Java 8**
- Development environment
  - **Eclipse PDE, Oxygen 2**
- Libraries
  - **Eclipse environment libraries**
    - See Manifest.mf for further detail on versions
  - **Google GSON Library 2.7.0**
    - Used for saving and loading configuration files in JSON format. Distributed under Apache License 2.0
- Tools
  - **WindowBuilder 1.9.0, Eclipse plug-in**
    - For UI design of SWT UI's
  - **ObjectAid 1.2.2, Eclipse plug-in**
    - For UML design
  - **STAN IDE 2.2.1, Eclipse plug-in**
    - For structure analysis and dependency optimization
  - **PASTA 9.4.1, Eclipse plug-in**
    - Abstract Syntax Tree analyzer for C++
- Testing
  - **JUnit 4**
  - **IFS CDT Testing plug-in 9.4.0**
- Documentation
  - **TexWorks**: For writing documentation in LaTeX
  - **GitLab**: LaTeX documentation source files
  - **Astah** for diagrams
  - **draw.io** for diagrams
  - **Pencil Evolus** for diagrams, UI prototyping, UML etc.
- Management
  - **GitLab**: Code repository, Issue tracking system and Continuous Delivery (Pipelines)
  - **SourceTree**: Version management tool
- Operating systems
  - **Windows 10**: As the plug-in is developed in Java, cross platform compatibility is automatically provided.

### 11.1.7 Quality measures

The quality measures section lays out steps to ensure code and project quality.

#### Documentation

The documentation and its version are managed on GitLab in the form of LaTeX files (.tex), PDF outputs and Wiki entries. The corresponding invites to relevant persons, like the supervisors, have been performed. Access to relevant sites:

GitLab:

<https://gitlab.dev.ifs.hsr.ch/zdogan/cevelop-stylechecker/tree/develop/docs>

GitLab Wiki:

<https://gitlab.dev.ifs.hsr.ch/zdogan/cevelop-stylechecker/wikis/home>

#### Project management

GitLab will be used for the execution of this project. GitLab provides facilities for issue tracking, CD and documentation.

GitLab:

<https://gitlab.dev.ifs.hsr.ch/zdogan/cevelop-stylechecker>

No login information needs to be provided, as the GitLab server is on the internal HSR server and the relevant persons are already present on the platform. They have been granted "Reporter" level permissions to the repository.

#### Development

For development, Git is used as a versioning tool. The code is managed on GitLab. The corresponding permissions have been granted to the relevant persons (i.e. supervisors). See the link in Project management section for GitLab - Repo link. The code quality will be guaranteed through code reviews.

#### Procedure

The project is developed in agile manner in iterations of two weeks in general. The agenda of iterations will be determined on a weekly basis and adjusted if the need arises.

#### Continuous Delivery

The GitLab platform provides facilities for Issue tracking, documentation and Continuous delivery (Pipelines). Through Pipelines, CD will be provided for this project in the form of installable plugins.

#### Code Reviews

Code reviews will be performed according to need and will be scheduled by the student with the supervisors. Code reviews will be performed with the supervisors as in this projects case they are also stakeholders.

#### Code Style Guidelines

No particular Code Style Guideline will be followed for the development with Java, though compliance with previous plug-in code will be tried to be reached.

**Unit Tests**

Unit tests will be written for all relevant parts of the code with JUnit. As the plug-in is mostly a "invisible" functionality, the tests will be mostly focused on the code itself (i.e. no UI automation tests etc.). For proper unit tests, the code will need to be properly isolated where possible.

**Integration Tests**

After a certain point in the development of the project, integration tests will be written as well. This certain point will be reached when significant component interaction can be observed and the need to ensure their proper cooperation arises (i.e. they aren't operating in isolation anymore). Some integration tests are already available as Stylechecher plug-in is used as basis for this project. Integration tests will be written as RTS Tests using the IFS CDT Testing plug-in.

**Usability Tests**

Minor usability tests will be performed to ensure the creation of custom styleguides is intuitive and easy to use. Beyond that, there isn't much UI interaction for the plug-in.

**System Tests**

Due to the scope of the project (i.e. a plug-in), no system tests will be performed.

## 11.2 Time Analysis

This section lists the analysis of time spent during the project, including budgeted and planned time. The time analysis is presented in two forms. The first is in tabular format and the second as a column chart. The target hours for the bachelor thesis amounts to 360, 12 credits, per credit 30 hours,  $12 * 30 = 360$ .

Iteration	Inception	Elaboration 1	Elaboration 1	Elaboration 2	Elaboration 2
Time	19.02 - 26.02	26.02- 05.03	05.03 - 12.03	12.03 - 19.03	19.03 - 26.03
Budgeted	21	21	21	21	21
Planned	13.3	19	19	18	18
Actual	23	14.5	18	9.5	15.5
Iteration	Construction 1	Construction 2	Construction 2	Construction 3	Construction 3
Time	26.03 - 09.04	09.04 - 16.04	16.04 - 23.04	23.04 - 30.04	30.04 - 07.05
Budgeted	42	21	21	21	21
Planned	32	18	18	39	39
Actual	46	19	14	47	8
Iteration	Construction 4	Construction 4/5	Construction 5	Transition 1/Transition 2	Transition 2
Time	07.05 - 14.05	14.05 - 29.05	29.05 - 07.06	07.06 - 15.06	15.06 - 21.06
Budgeted	21	42	21	21	24
Planned	12	32	16	16	16
Actual	19	41	22	29	37.5
<b>Total Budgeted</b>	360				
<b>Total Planned</b>	325.3				
<b>Total Actual</b>	363				

Table 42: Time analysis table

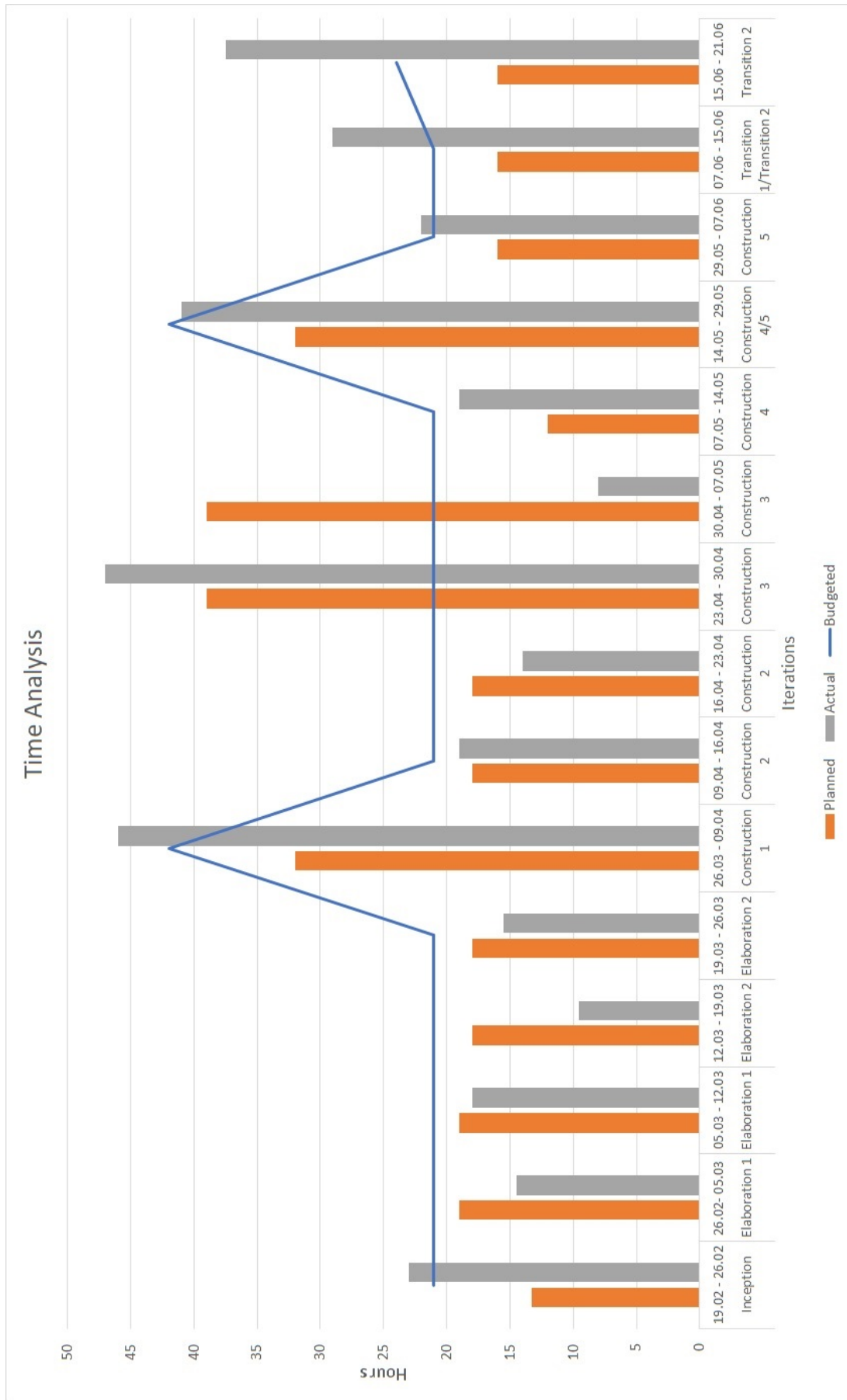


Figure 51: Time analysis diagram



## 12 Appendix B

### 12.1 Installation Guide

The most recent installable plug-in via Updatesite can be queried via following link:

[https://gitlab.dev.ifs.hsr.ch/zdogan/cevelop-stylechecker/-/jobs/artifacts/master/raw/com.cevelop.ctylechecker.updateite/target/repository/?job=deploy\\_updateite](https://gitlab.dev.ifs.hsr.ch/zdogan/cevelop-stylechecker/-/jobs/artifacts/master/raw/com.cevelop.ctylechecker.updateite/target/repository/?job=deploy_updateite)

Once the resolved URL is retrieved, it can be used in Eclipse directly. For this use the URL in the "Help > Install New Software..." dialog as shown below.

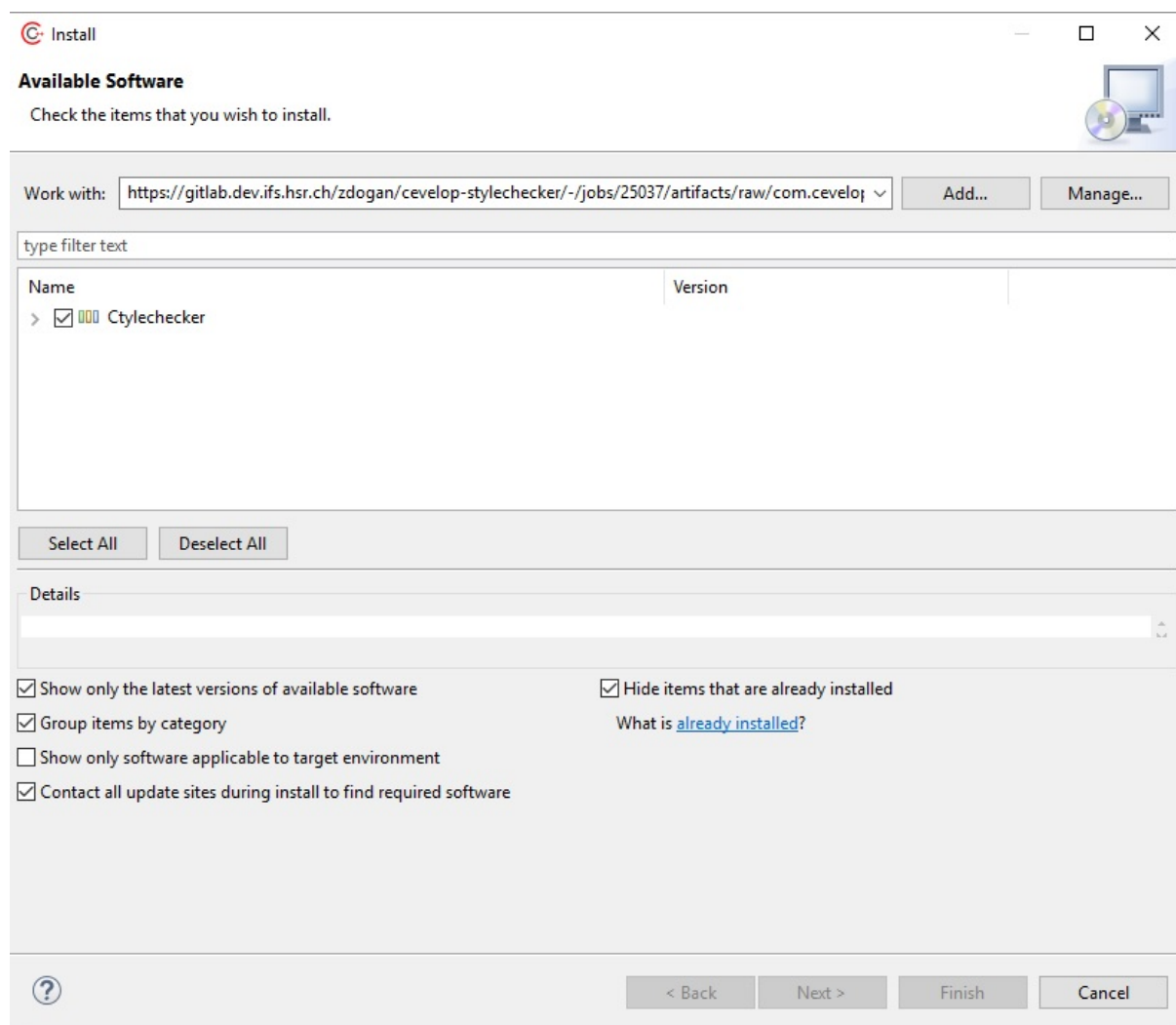


Figure 52: Install New Software dialog excerpt

Follow the instructions. After the installation is complete, Cevelop will restart and the plug-in will become available under Codan Analysis in the preferences.

## 12.2 User Manual

The user manual contains information about the use of the plug-in and focuses on the most important aspects of it.

### Starting point

The starting point of the plug-in settings is the preference page. In Eclipse, there are two variations of this page, one being the preference page on the workspace level, the other being the preference page on project level, usually called property page.

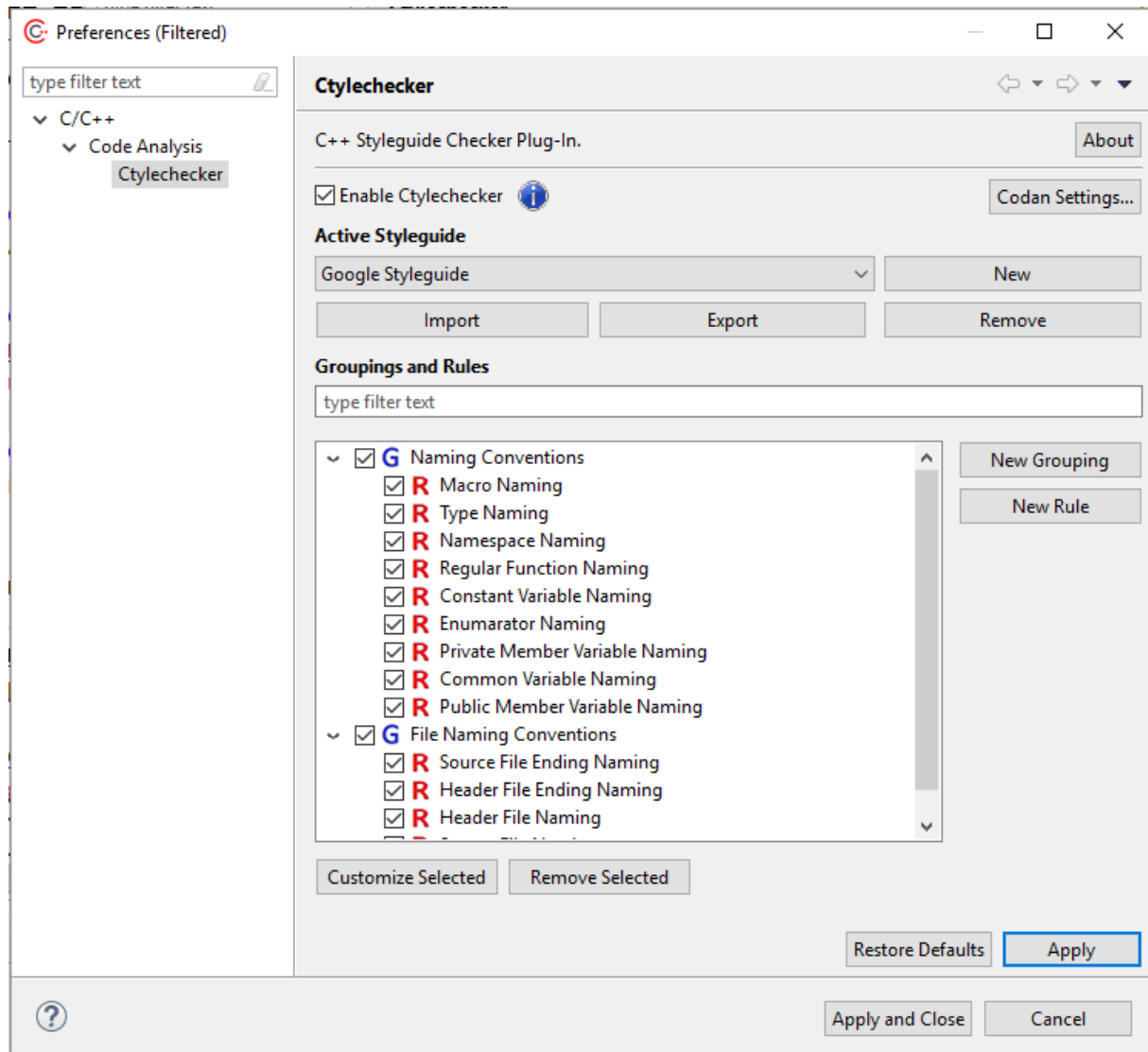


Figure 53: Preference page Stylechecker

The preference/property page of the Stylechecker plug-in shows a list of groupings and rules of the currently active styleguide. The active styleguide can be changed via the "Active Styleguide" combo. From here as well, a new styleguide can be created.

## Creating a Styleguide

In order to create a new styleguide, you can click on the "new" button next to the "Active Styleguide" combo box. A simple dialog appears, where you can enter the name of the new styleguide. After pressing "OK", the new styleguide is available under the list of styleguides.

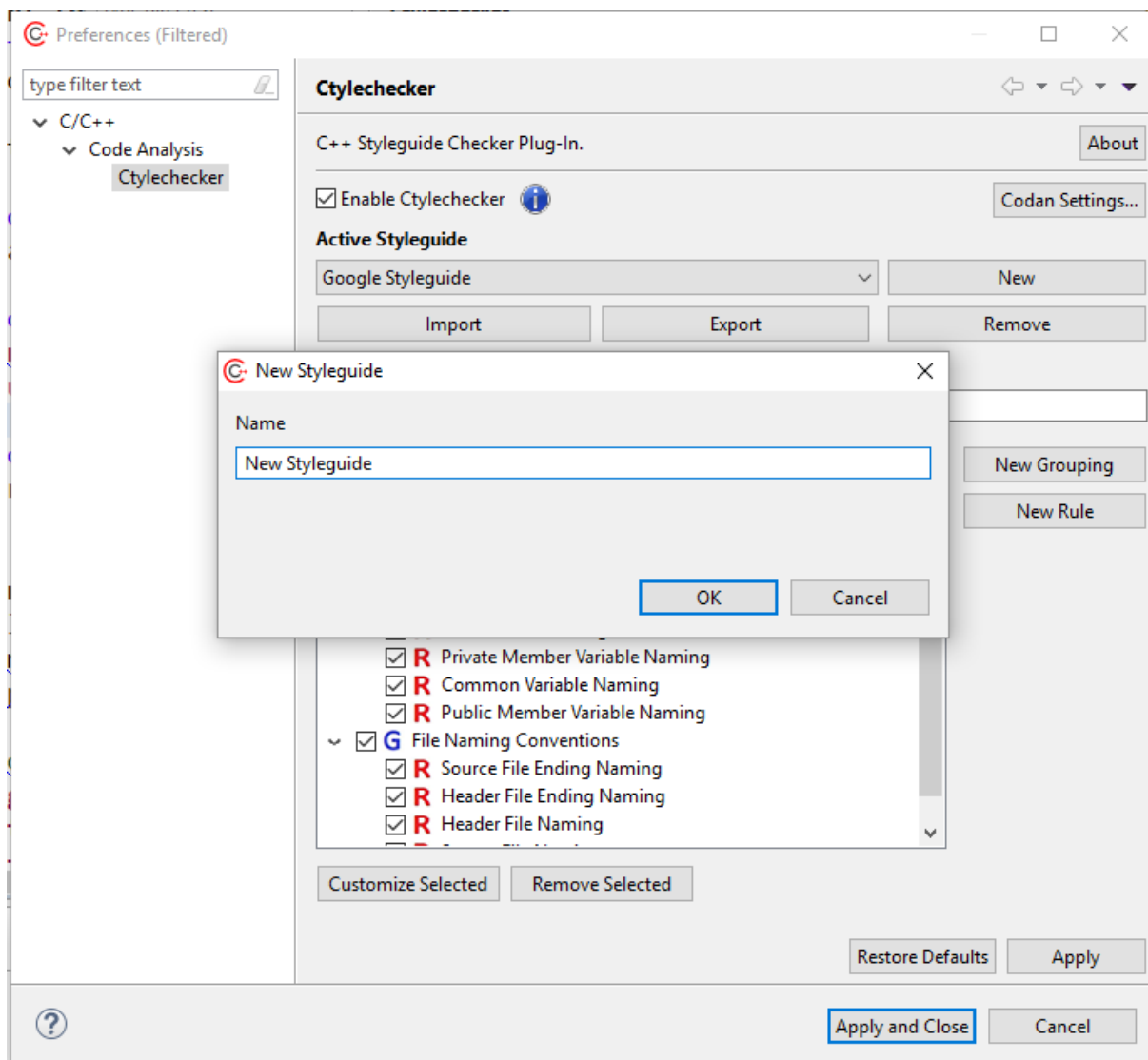


Figure 54: New styleguide creation

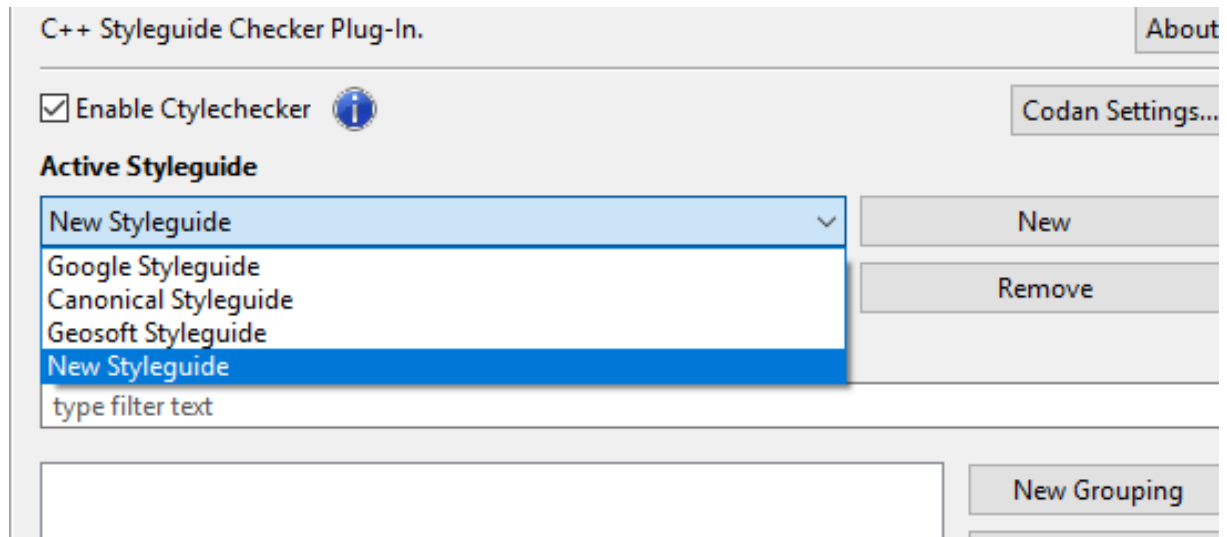


Figure 55: New styleguide available in list

When a new styleguide is created, it is empty, no groupings, no rules. An empty styleguide can be used like a normal styleguide, it just won't check anything and is basically equal to having the Stylechecker plug-in disabled.

### Creating a Grouping

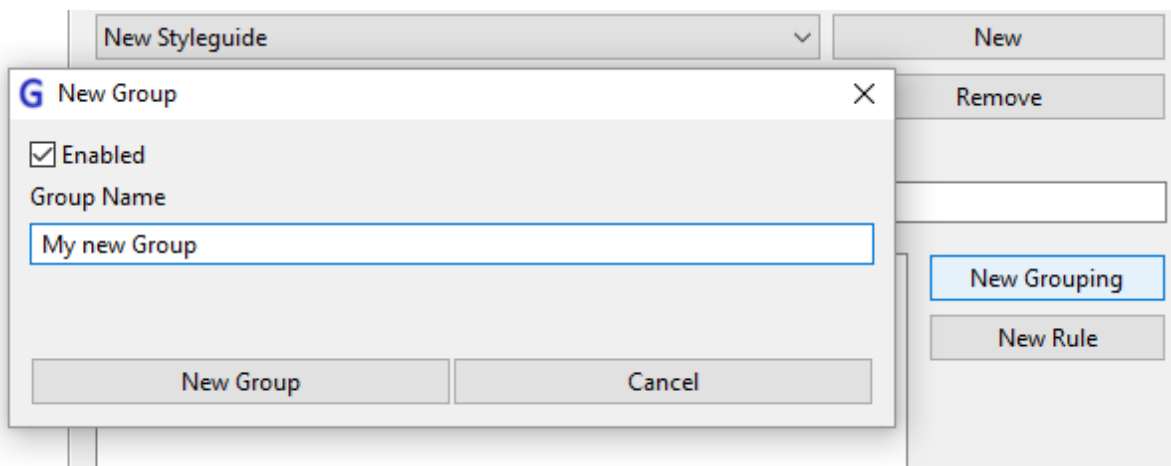


Figure 56: New Grouping dialog

A new Grouping can be created via the "New Grouping" button, a simple dialog for the Grouping name and enabled state is presented. To confirm the Grouping, press the "New Group" button in the dialog. Once a new grouping is created, it will appear as an item in the tree view of the preference page. Groups can be identified by their "G" icon. To add a rule to a group directly, keep a selection on the Grouping while clicking on "New Rule"; this way it will automatically set the root of the Rule to that Grouping. See the "Root" combo box in the rule dialog Figure 57.

## Creating a Rule

To create a new Rule, click on the "New Rule" button in the preference page. A new dialog will be opened, like in Figure 57. The New Rule dialog has five significant sections. The top row controls the enabled state of the Rule and also contains info about which Grouping the Rule belongs to. Next comes the Rule Name, with which a Rule can be identified by in the tree presentation of the styleguide. After the Rule Name, a message can be entered for the Rule. This message is used when a Rule violation is reported in the IDE. Following that comes the section for Expressions. Expressions make up the guts of a Rule and define Expression and ExpressionGroups, which will be checked on the selected concepts. The Expressions section has two sets of Expression lists, predefined and custom. Predefined expressions are special types of Expressions, prepared up front to be used. Custom expressions allow for the definition of own regular expressions to check a language element with. And finally, there is the Checked concepts section, where the language elements to check the Rule with can be selected.

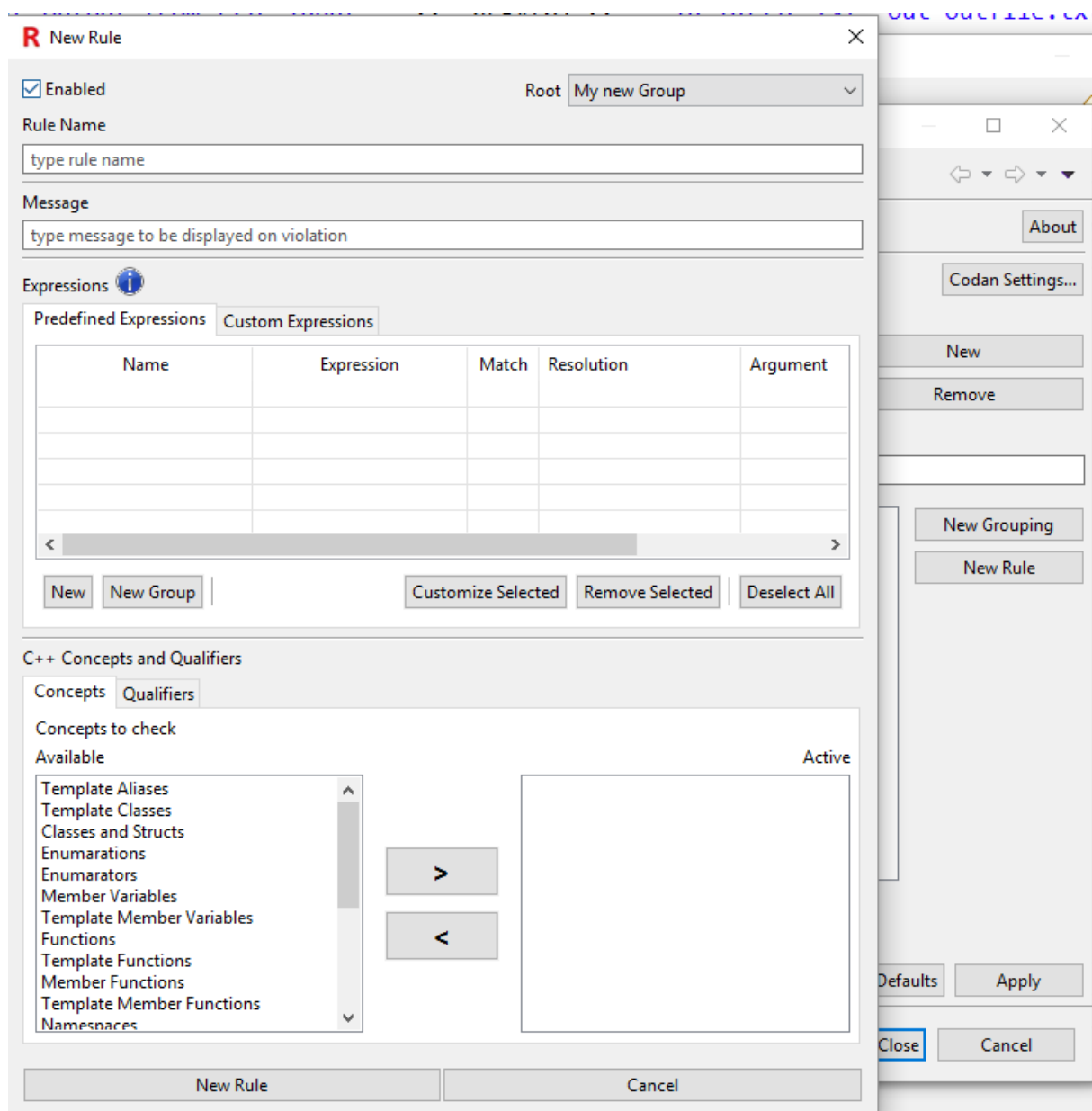


Figure 57: New Rule dialog

## Creating a New Predefined Expression

A new predefined expression can be created via the “New” button. A new dialog will be opened, which allows to select from a predefined list of expressions. For predefined expressions, only a certain degree of adjustment is possible. This is to ensure the proper use of predefined expressions.

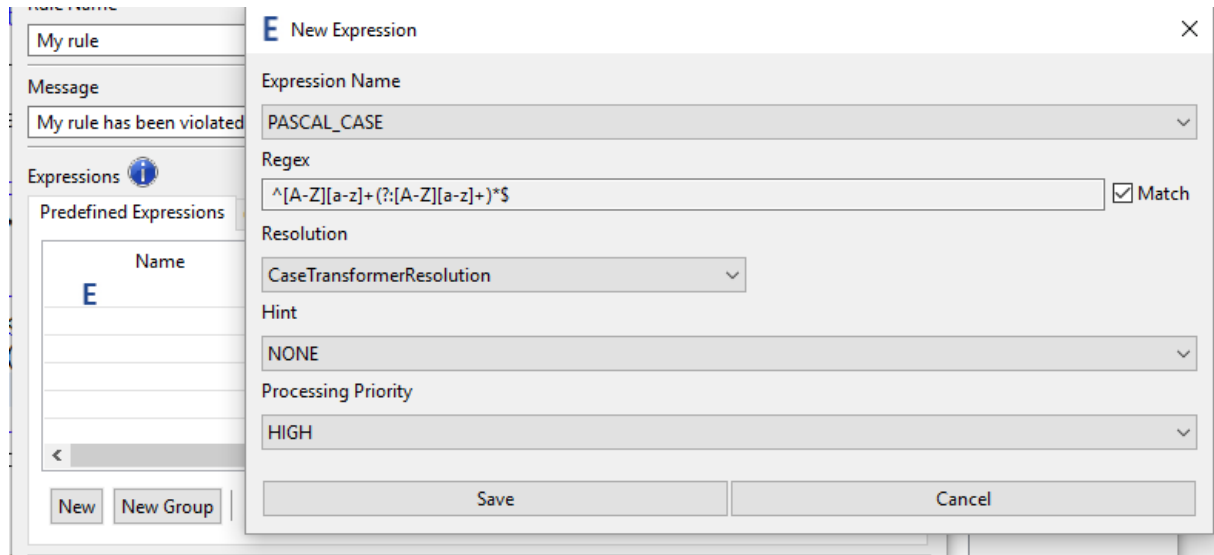


Figure 58: New Predefined Expression dialog

A new expression has several fields that can be set. In the case of a predefined expression, the name and the regex can't be set. The predefined regex's match policy can be manipulated via the Match checkbox. A checked Match checkbox means the regex is to validate to true. The resolution can be changed as well. The resolution field determines how an expression is to be resolved in case it is violated. The Hint field is a helper field for the expression and is tied to the resolution field. With it the resolution logic can be manipulated. Next to casing hints and the lack of a hint (NONE), there is a special hint called PREFERRED. The PREFERRED hint is an indicator for the resolution logic which expressions and expression groups to consider when resolving a violation. The last field, processing priority, controls in what order resolutions are applied. HIGH means they're applied in the first turn, LOW means they're applied in the last run. This way checking and resolution logic can be explicitly controlled.

## Creating a New Expression Group

Next to singular Expressions, Expression groups can be created via the “New Group” button. This will open another simple dialog to enter group relevant information, like group name, Match policy and hint. The Match policy serves to determine how the contained expressions in the expression groups are to be checked with, i.e. do all expressions have to be fulfilled or does only one expression need to be fulfilled (alternatives for language elements can be defined this way, i.e. if a const variable can be `CONST_CASE` or `SNAKE_CASE`). Expression groups, similar to singular expression, can have hints as well.

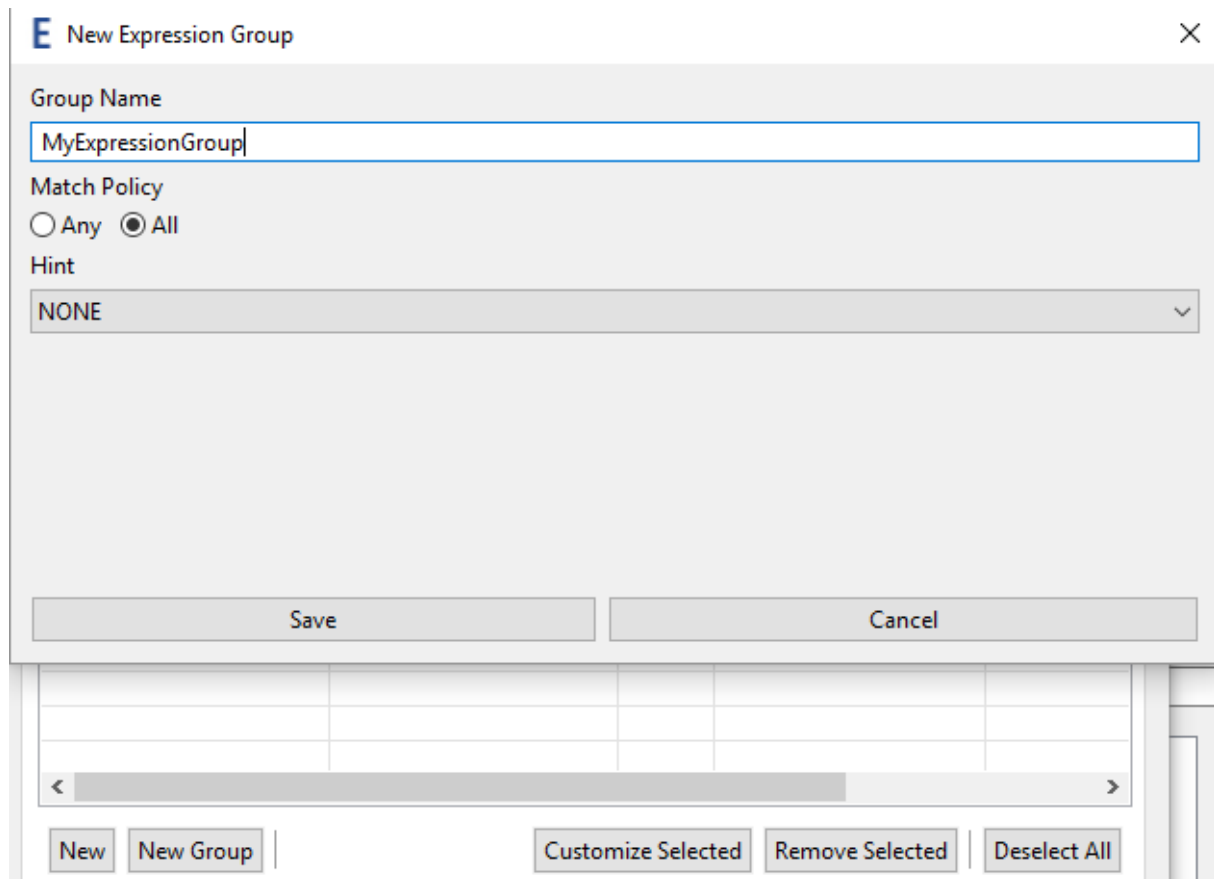


Figure 59: New grouping dialog

Hints for Expression Groups are limited to two types, NONE and PREFERRED. When the PREFERRED hint is set on an Expression Group, it and the contained Expressions and Expression Groups will be considered when applying a resolution. In combination with the Hints for singular expressions, a specific set for a resolution logic can be set via the hint field. The selective application of resolutions of Expressions and Expressions contained in Expression Groups gets only activated if the PREFERRED Hint is applied at least once, otherwise all resolutions of all Expressions and Expression Groups will be applied. This approach is valid for all nested levels as well (i.e. if an expression group contains other expression groups).

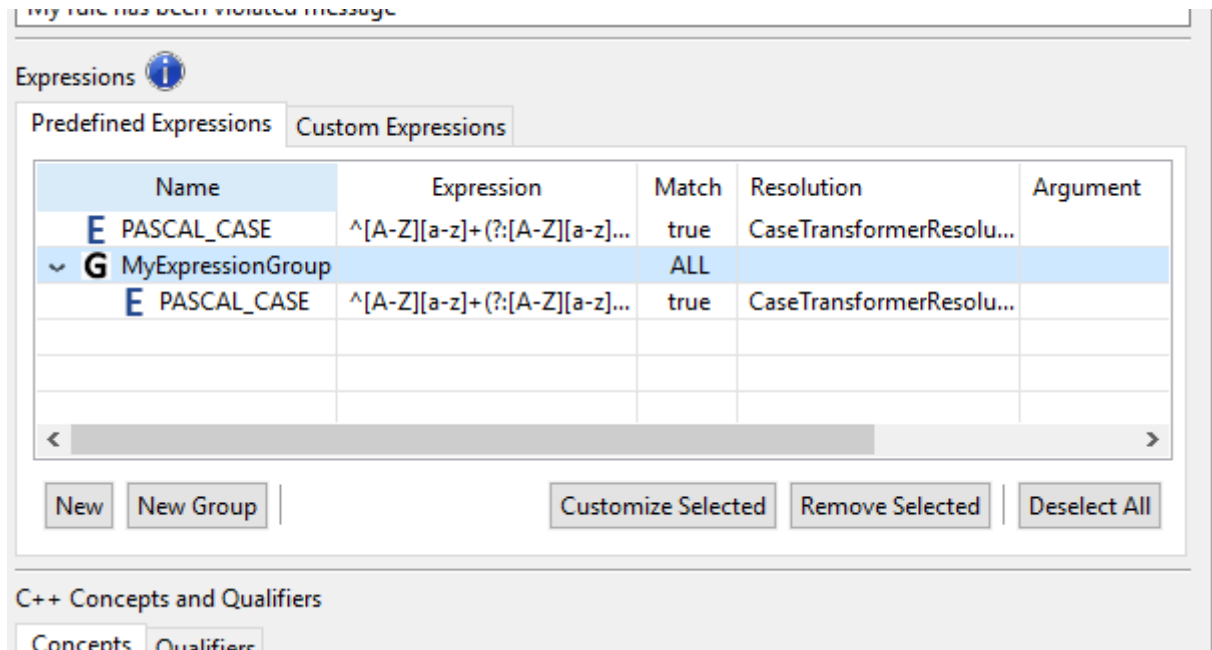


Figure 60: Predefined Expressions tree view with Expression and Expression Group

Created Expressions and Expression Group will be shown in the tree view of the corresponding tab in the Rule dialog. They give an overview of the relationships and display all the relevant fields in the tree views for the Expressions section as can be seen in Figure 60. The same compositions can be achieved for custom expressions as well.



### Creating a New Custom Expression

The only way custom expressions differ from the predefined expressions is that in the New Expression dialog, the user can additionally set a custom name and regex in the dialog. Here as well the Hints are tied to the resolutions, as can be seen in Figures 62 and 63 .

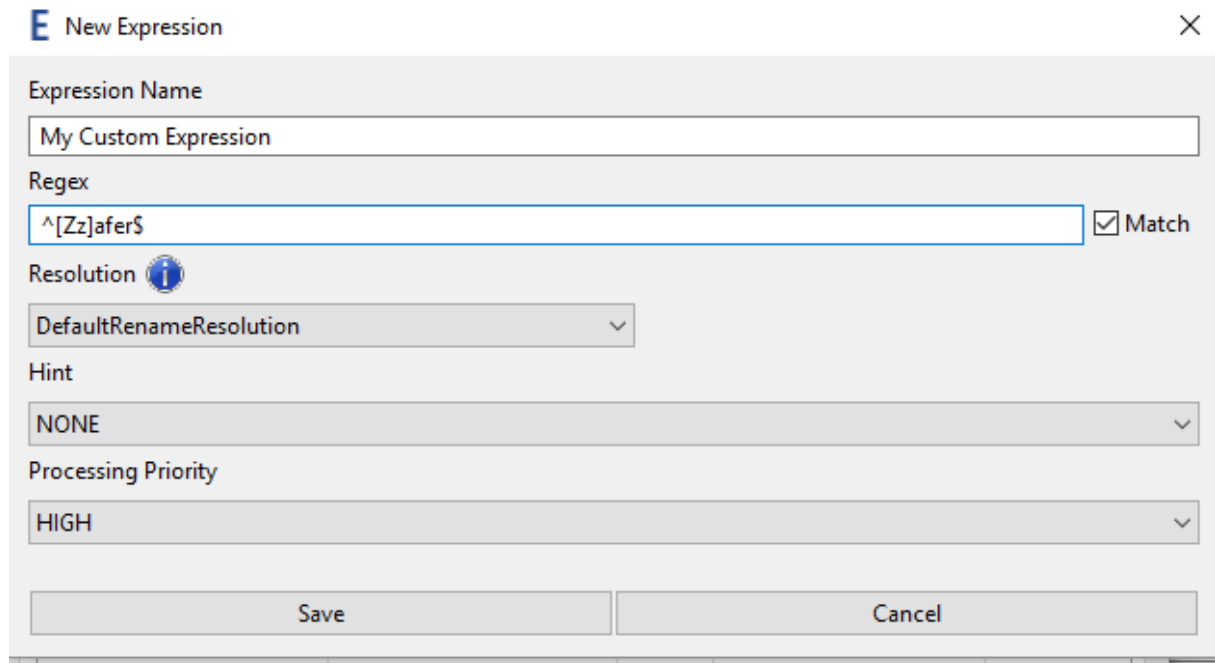


Figure 61: New grouping dialog

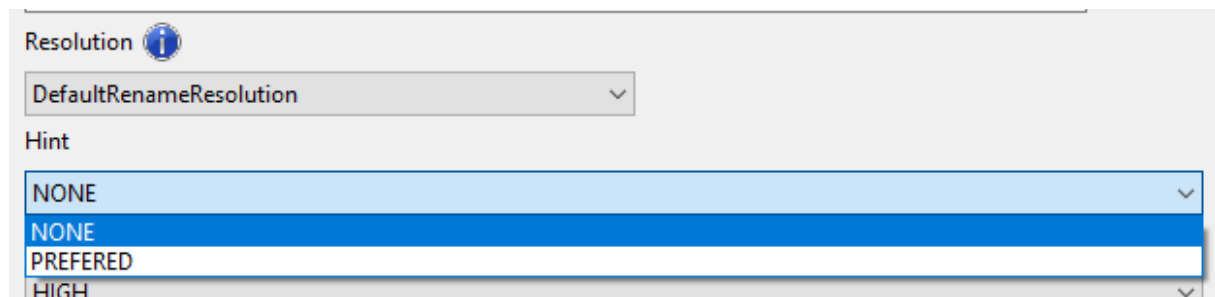


Figure 62: Hints for DefaultRenameResolution

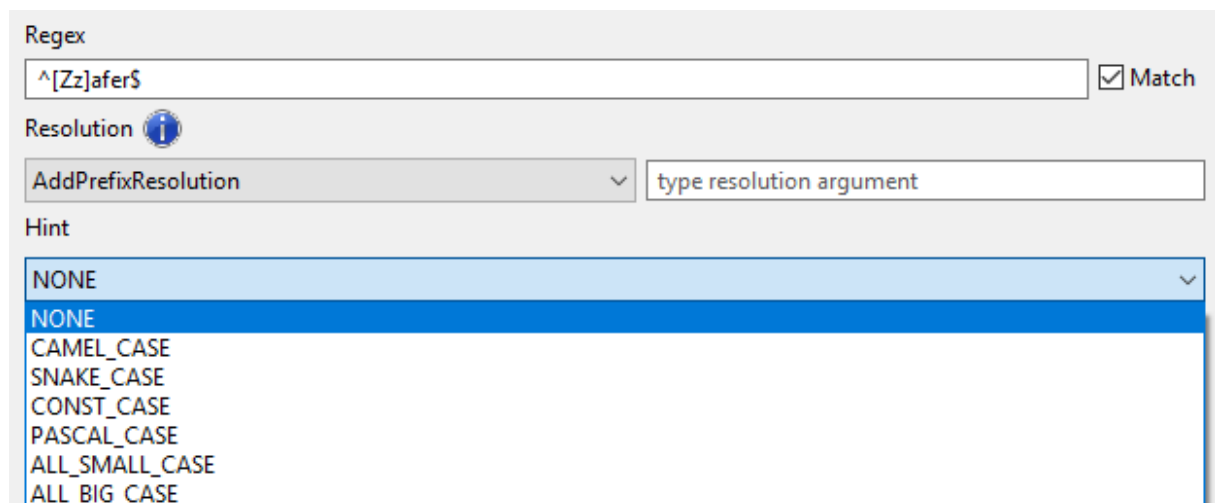


Figure 63: Hints for AddPrefixResolution

As a user can enter custom regexes in the custom expression dialog, a simple regex checking is performed. If a regex is invalid, it is reported with a warning yellow background. If a regex is invalid, an expression can't be saved.

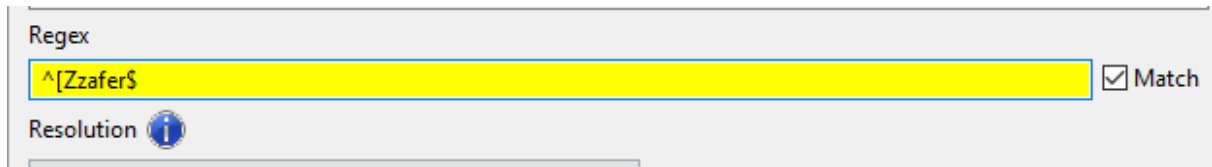


Figure 64: Regex checking in Custom Expression dialog

When defining regexps, it's important to do so with the active resolution in mind. Certain expressions need to be defined in a special way in order to function properly, an example of this being the ReplaceResolution. The ReplaceResolution depends on capturing groups, with which it knows where to replace a target string. The info icon next to the Resolution label shows in a tool tip what needs to be considered when working with the currently active resolution type.

### Setting the language elements to check in a Rule

Finally, within a Rule, a user can decide which language elements the current rule applies to. In the qualifiers tab, a language element can be limited even further to qualifiers like public, const and the like as shown in Figures 65 and 66.

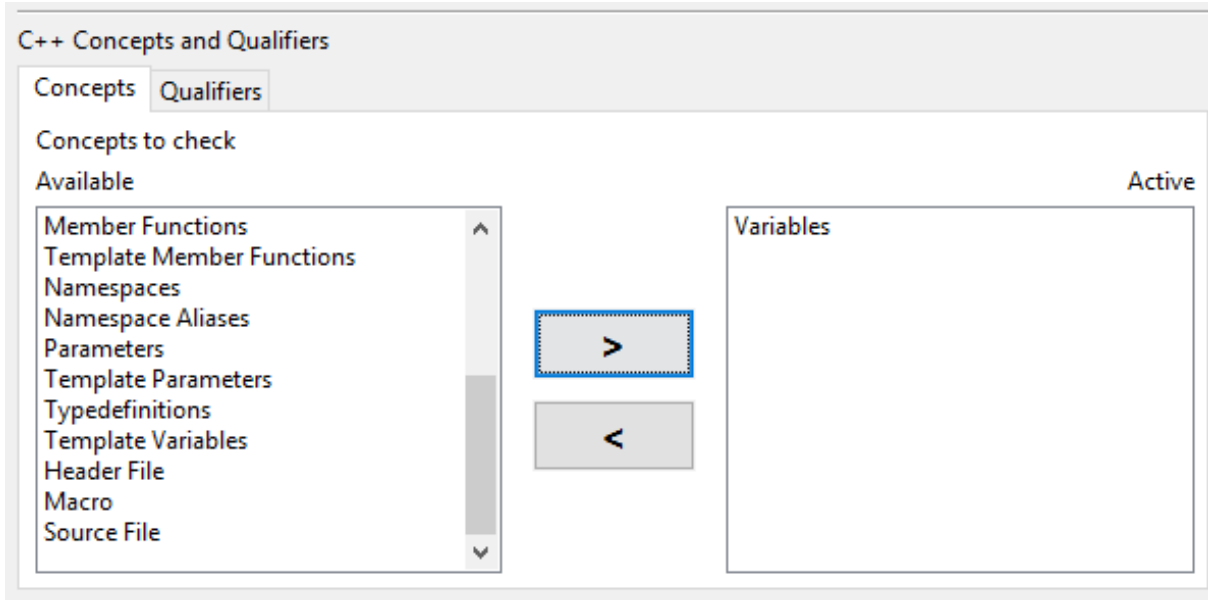


Figure 65: New grouping dialog

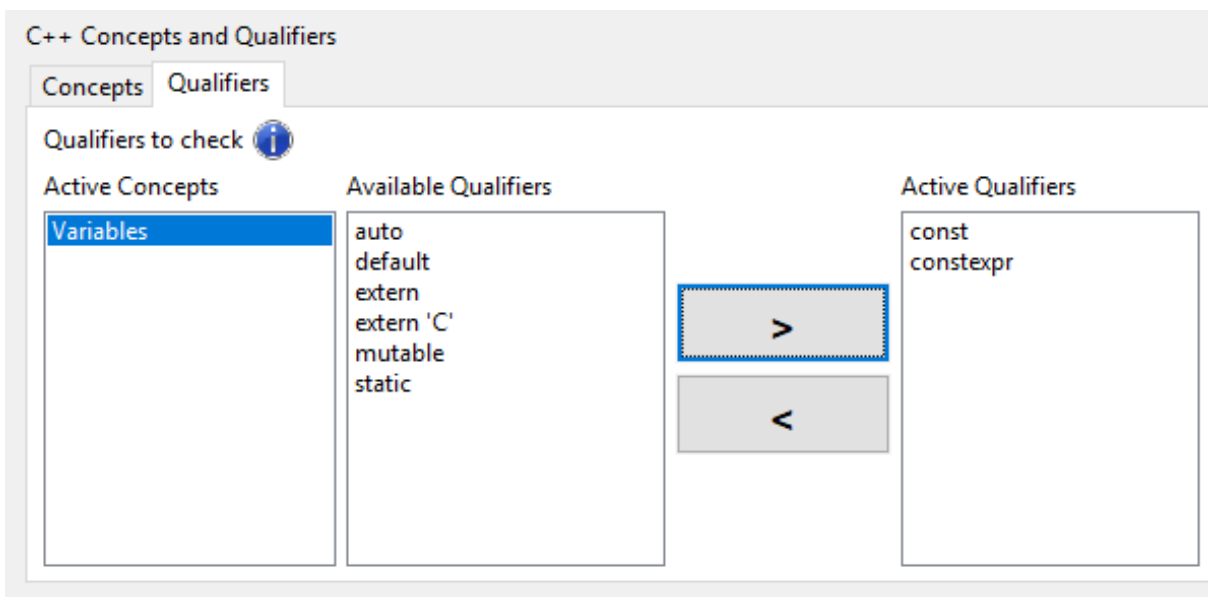


Figure 66: New grouping dialog

### Changing the Grouping of a Rule

A Rule can be adjusted via the "Customize Selected" button in the preference page. From here, you can change also the belonging to a Grouping of a Rule via the Root combo at the top. The value ROOT stands for root level, i.e. rules that don't belong to any group.

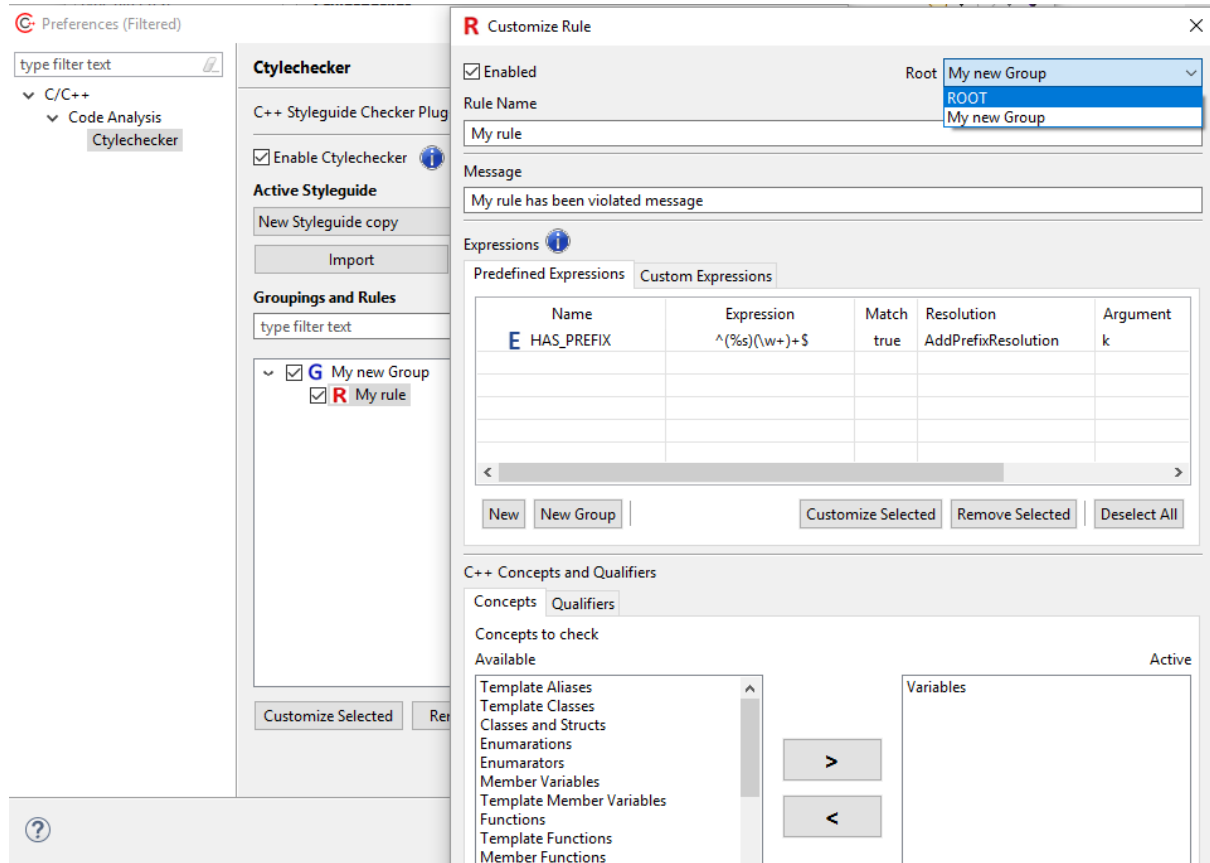


Figure 67: Customizing a Rule's grouping

Via the "Customize Selected" button Groupings and Rules can be adjusted in general. Figures 67 and 68 show a before and after of a Grouping change of a Rule.



Figure 68: Updated tree view after Grouping of a Rule is changed

## Setting project specific settings

Property pages offer additional settings for the Stylechecker plug-in. Here project specific settings can be applied. There are three variations. Workspace, Reference Workspace and Project settings. When using workspace settings, the project will reference the workspace itself for the currently active styleguide. When referencing a workspace, it can be chosen on the project level which styleguides from the workspace one wants to use for this project. And finally, the last is the option to use project settings, which is defining your own styleguides entirely on project level.

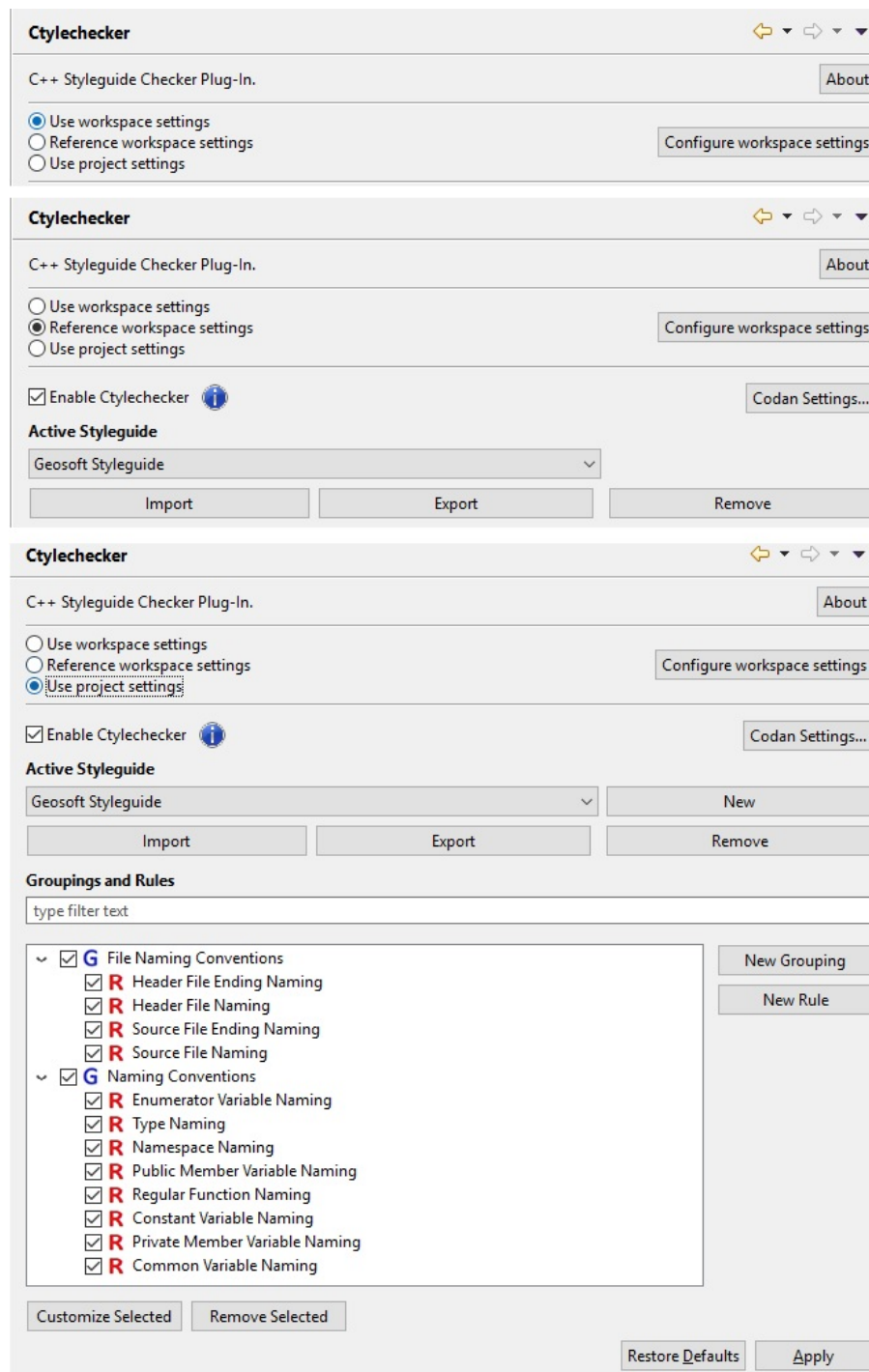


Figure 69: Project specific setting variations

### Import and Export of Styleguides

Via the "Import" and "Export" buttons, styleguides can be exported and imported to be shared among team members. When exporting a styleguide, it is saved with .ctyleguide file ending. It contains the JSON representation of the styleguide. Figures 70 and 71 show the corresponding dialogs.

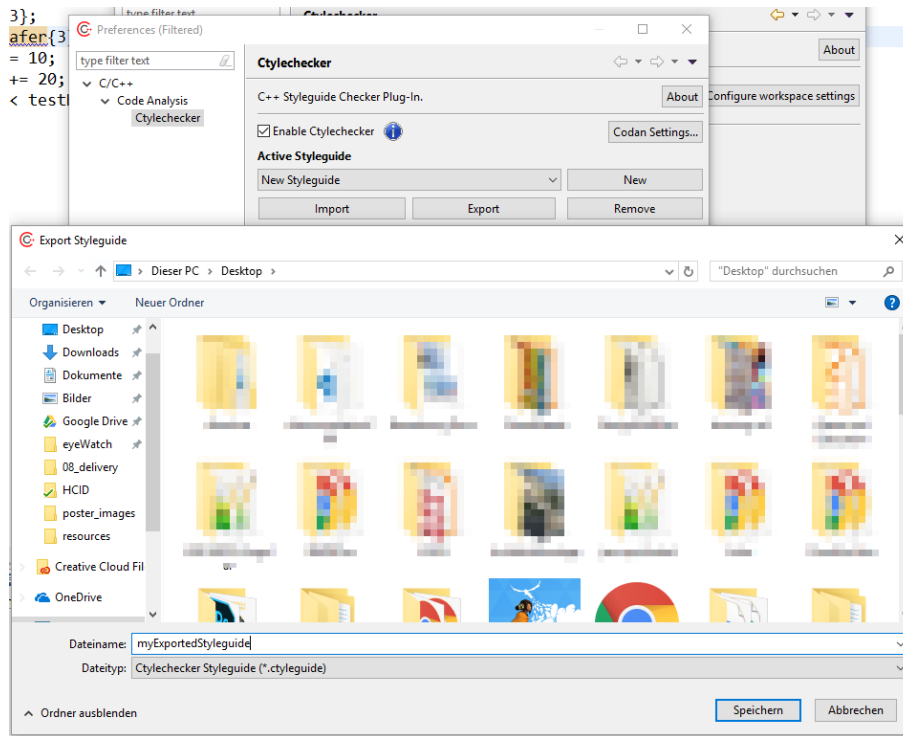


Figure 70: Export Styleguide dialog

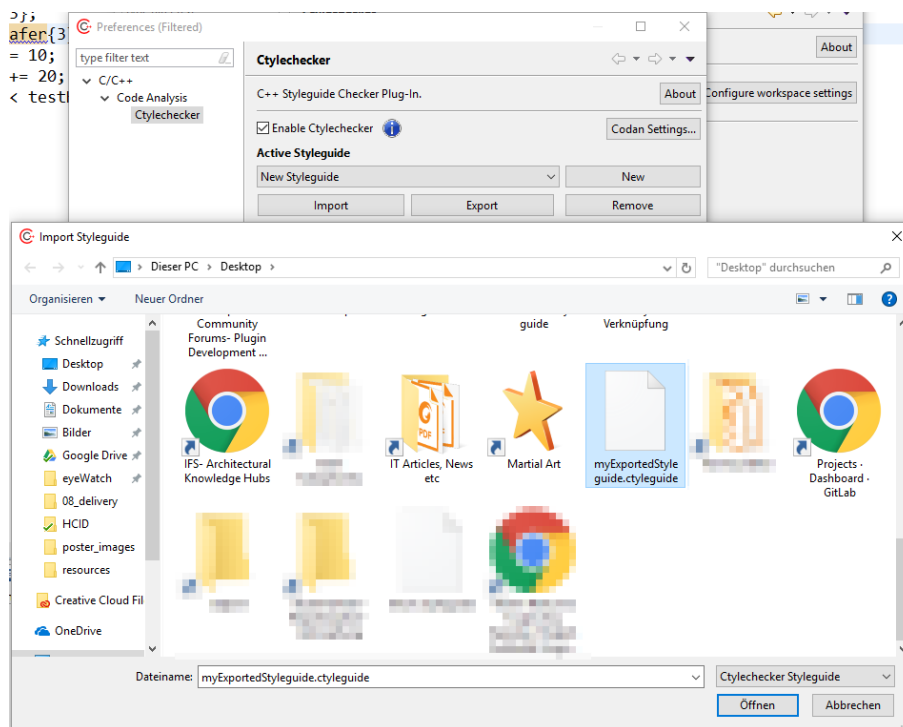


Figure 71: Import Styleguide dialog

When importing a styleguide, if the name already exists, it's name will have a "copy" appended to it automatically to distinguish it from the existing one. This can be seen in Figure 72. An imported styleguide is directly available in the combo of available styleguides and can be directly activated by selecting it in the preference page.

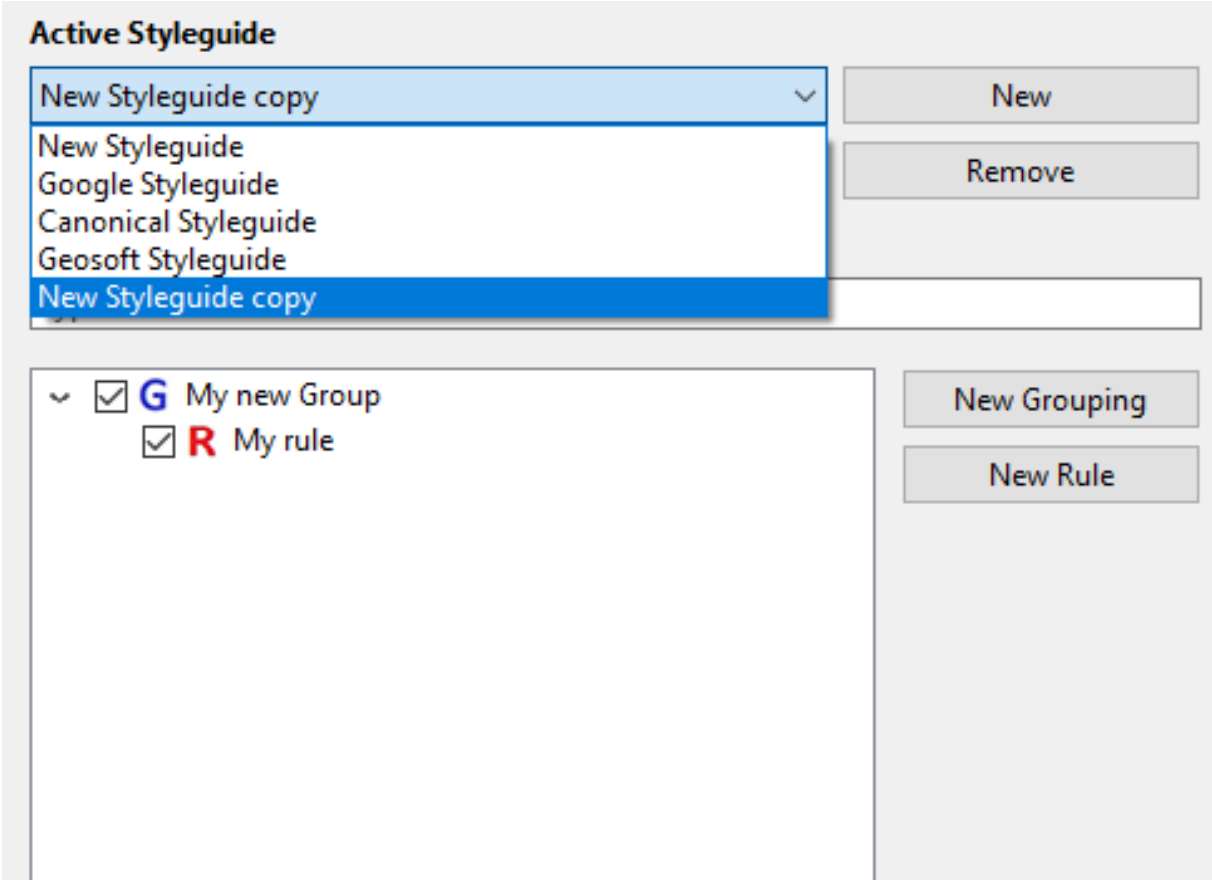


Figure 72: New grouping dialog

## Filtering for Rules and Groupings

Especially in more complex styleguides, it can be difficult to find certain rules. To find Rules and Groupings quickly, a user can use the filter input field. It filters the Rules and Groupings while typing.

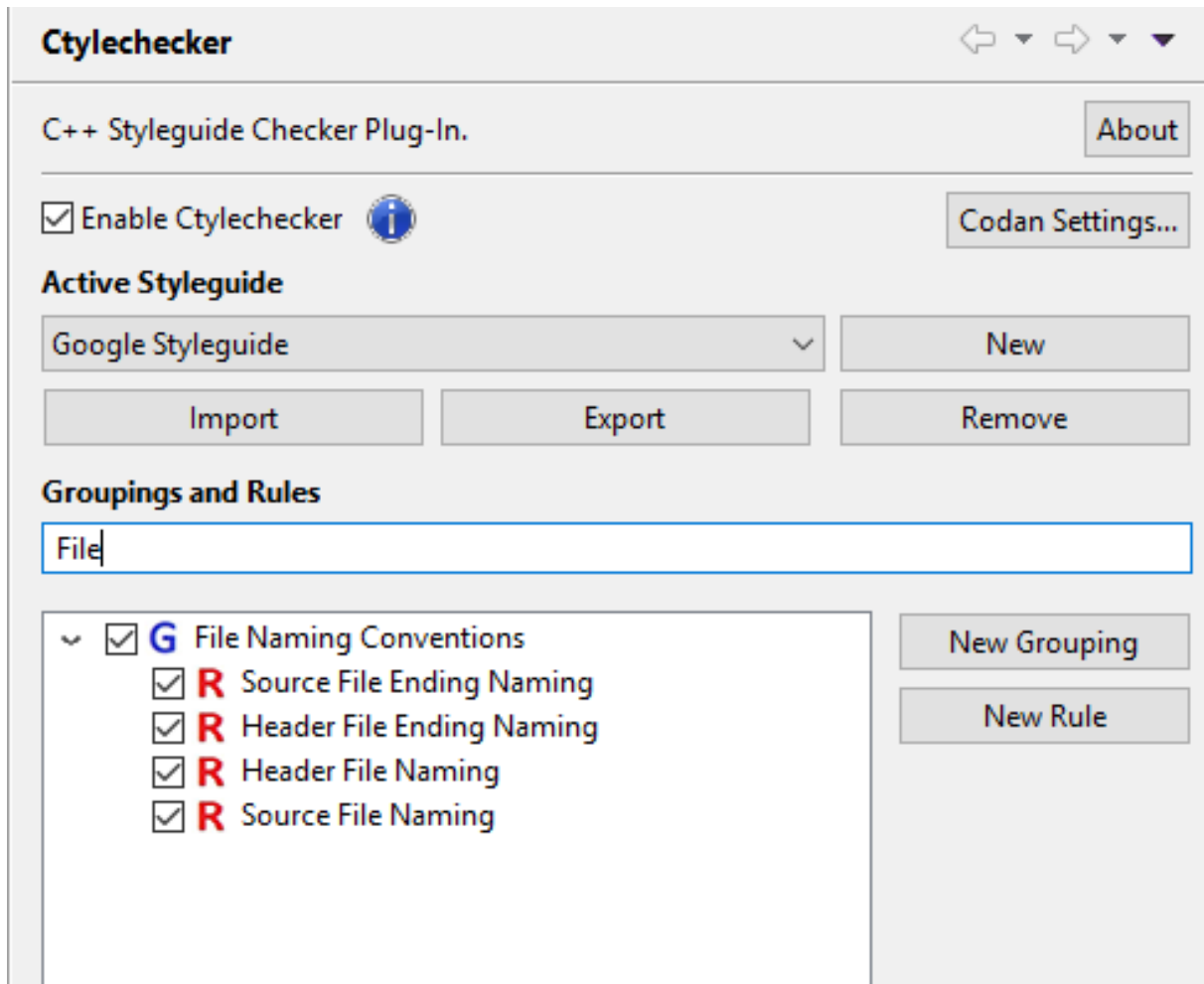


Figure 73: Filter being applied



## Stylechecker vs Codan settings

As the Stylechecker plug-in is connected to the Codan plug-in, so are some of its settings. It is important to keep the Codan settings in mind as well. For this, a user needs to ensure that the Codan settings page has the necessary Checkers activated on the correct level (Codan also has workspace and project specific settings).

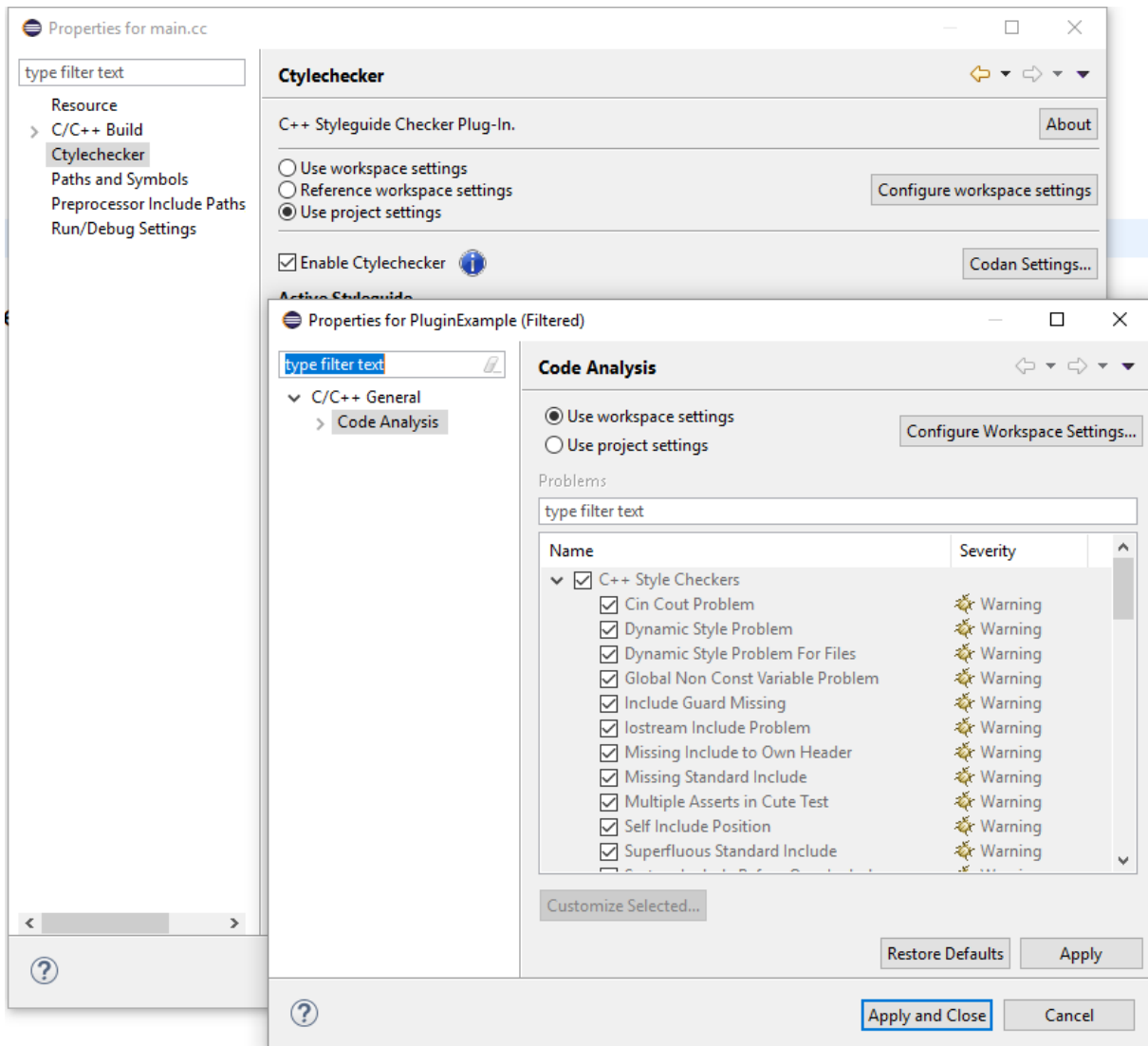


Figure 74: Codan settings

The relevant Checkers for the Stylechecker plug-in are Dynamic Style Problem and Dynamic Style Problem for Files.

## File inclusion and exclusion

Via the Codan Checker settings, the Stylechecker plug-in can additionally be limited to certain files and folder via the Scope tab in the Checker settings.

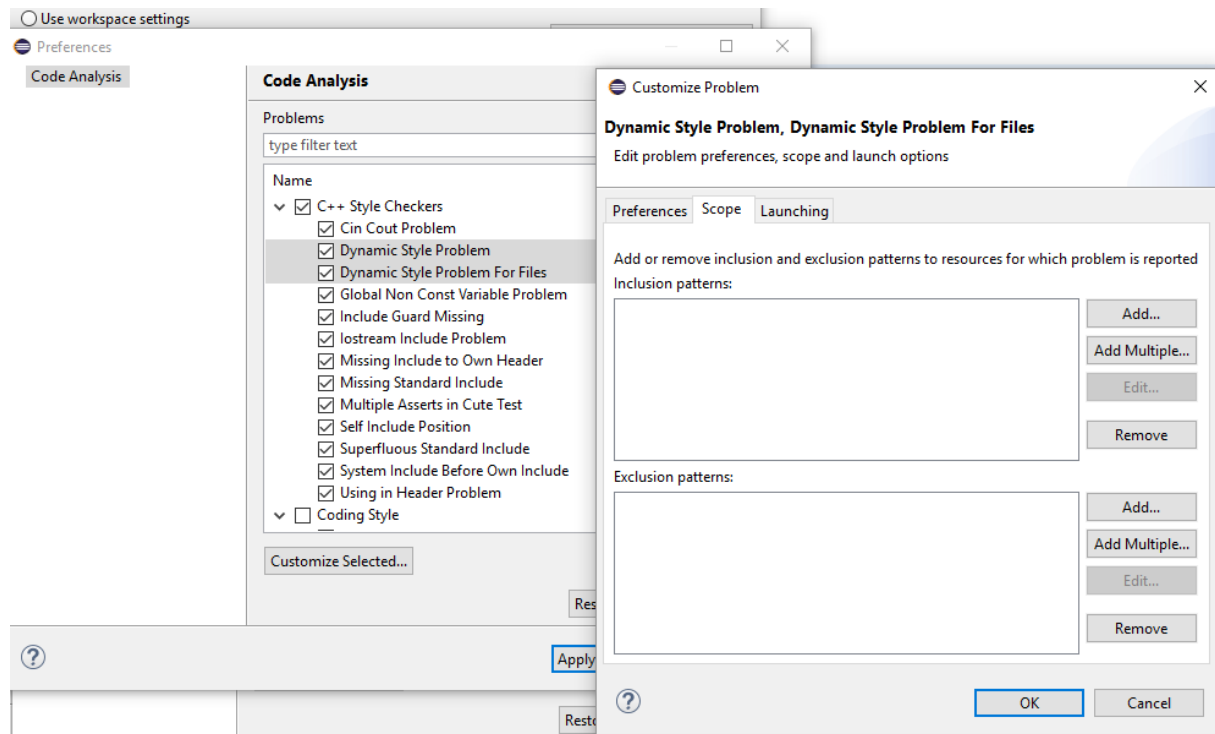


Figure 75: Codan settings for Stylechecker Checkers