HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

INSTITUTE FOR
SOFTWARE

Bachelor Thesis, Departement of Computer Science

CCGLator

C++ Core Guidelines Constructor Rules Checker and Quick-fixes

University of Applied Science Rapperswil

Spring semester 2016
17.06.2016

Autoren:	Özhan Kaya & Schmidiger Kevin
Advisor:	Prof. Sommerlad
Co Advisor:	Toni Suter
Partner:	IFS Institute for Software
Duration:	22.02.2016 - 17.06.2016
Workload:	360 hours, 12 ECTS / Student
Link:	`sinv-56012.edu.hsr.ch`

I Abstract

C++ is a complex language evolving over the years. Especially with the release of C++11 many recommended practices became obsolete because the language and standard library provide safer and more efficient mechanisms to create type-safe and performant code. Higher level issues like resource management, memory management and concurrency are common mistakes in C++. To help the developer to create error free and safe code, the C++ Core Guidelines[BS15a] were introduced at the CppCon15[Cpp15] by Bjarne Stroustrup and Herb Sutter. Following those guidelines leads to code that is statically type safe, should have no resource leaks, and catches many more programming logic errors beeing quite common in code today. The guidelines are an kind of "an early release" meaning that the rules are not yet complete, if they will ever be. Its is split in sections depending on which problem needs to be concerned.
In this thesis we took the section C: Classes and Class Hierarchies[BS15b] and there the subsection C.ctor: Constructors, assignments, and destructors[BS15c]. Reduced to those rules we then analyzed them in terms of what problems could appear when they will be ignored. Some of the problems are not easy to detect in later production code. After the analysis we created an Eclipse plug-in to help the developer concern the problems occurring when creating a class in C++.
The plug-in provides highlighting the part of code violating a rule from this section. It is then possible to read more about the rule and use its suggestion to quick-fix the code in a way the guideline suggests to do. Because the plug-in covers the rules concerning creating data-types, it can handle some problems of resource and memory management, as well as creating data-types in a more modern way.

II Management summary

Our bachelor thesis consists of two separate parts. The first part (first three weeks) included implementing additional features to our term project called AliExtor. The main part though, was implementing a separate Eclipse plug-in called CCGLator, which we are going to focus on in this management summary.

II.1 Initial situation

C++ is a complex language evolving over the years. Especially with the release of C++11 many recommended practices became obsolete because the language and standard library provide safer and more efficient mechanisms to create type-safe and performant code. And since C++ is a backwards compatible language, the language itself is becoming more and more a superset of possibilities. Of course the idea of every new standard, is also to get rid of legacy code and making things safer and easier.
At the CppCon15[Cpp15] in September in Washington, Bjarne Stroustrup, the inventor of C++ and Herb Sutter announced a set of rules generating a subset of the entire C++ language, giving suggestions which legacy codes can be replaced with modern C++ methodics. Currently, there is a project under development called GSL[Mac15] (Guideline Support Library) from Microsoft, providing helper types and functions to make the implementation of the C++ Core Guideline easier. This project has been under development even before the announcement of the C++ Core Guidelines due to Herb Sutter also working for Microsoft as a Software Engineer. Currently there are no final working implementations of the C++ Core Guidelines for any IDE.
The goal of our bachelor thesis was to analyze the rules in section "C.ctor: Constructors, assignments, and destructors", as well as implementing the analyzed rule by our own prioritization.
We decided to take on this challenge, because we are very aware of the problem which comes with C++ growing as a language and the Core Guidelines appealed to us as a good approach to creating a subset with the idea of replacing legacy code with features from C++11/14/17.
Our goal was pretty much defined as open scoped. Depending on our analysis we were free to decide which we are going to implement and to which extend we are going to implement them.

II.2 Approach

The first part of our bachelor thesis involved a lot of analysis. Each rule had to be broken down into several parts and we had to elaborate how and if a checker and quick-fix can be implemented or how the rule itself can be interpreted. Parallely, we had to get into the Codan[Ecl15] framework (light-weight static analysis framework in the Eclipse CDT) and develop a working prototype of a checker, a quick-fix and the according test cases.
After most of the analysis was done, we were quite confident that most of the Core Guideline rules should be implementable. But along the way, we found out that some rule implementations required us to handle 10-15 cases of which each required different implementations as well as working with different features of the Codan framework. In week nine we discussed with our supervisors that focus on less rules but making those work in any case makes more sense than implementing as many rules as possible in the given timeframe. From that point on we focussed on perfecting our existing implementation.
During implementation we received weekly feedbacks. They were quite helpful, since most rule implementations could have been interpreted in many different ways.

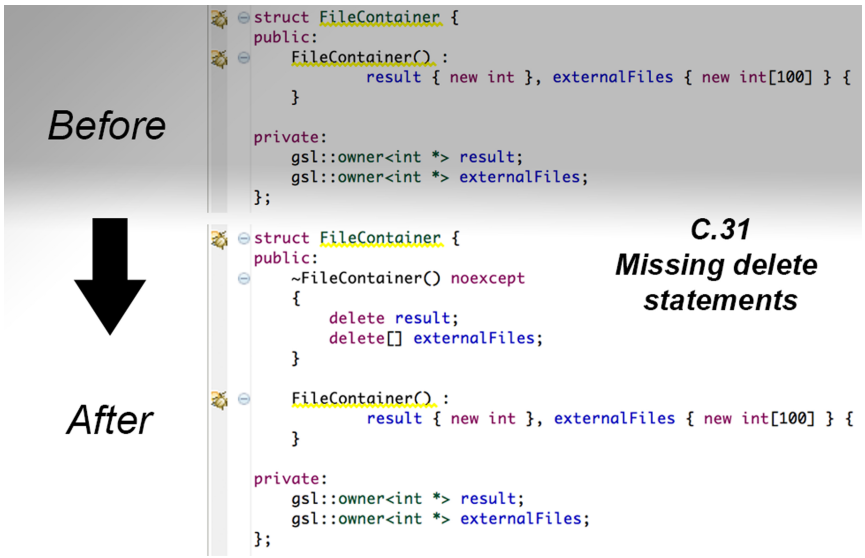

Figure 1: Before/After: Delete statements of owning pointer variables

II.3 Results

We ended up with a set of working rules. Sadly, the final amount of implemented rules were lower than we anticipated, but the core of our work also lies in our helper methods. We had to cover many cases, such as C++ visibility concerns, cross file analysis and refactoring and so on, for which we implemented various helper methods, making it easier to implement additional rules. Therefore, even though we did not manage to implement all the rules we desired to, we have laid a good ground basis, making building up on our bachelor thesis simple and fast.
We ended up with 14 implemented rules. All offer a checker as well as a quick-fix, as well as all of these rules have separate test-cases, each handling a unique case. We ended up with 350+ test cases resulting in more than 25 test cases per rule implementation. Also we have a easily expandable project which can be improved with additional rule implementations.
While working on our bachelor thesis, we also managed to improve our know-how concerning the Eclipse CDT[ecl] environment, Codan framework and of course a lot of C++ knowledge while analyzing each rule.

II.4 Forecast

The C++ Core Guidelines as a whole are very large-scaled. Although, we only focused on a sub selection of the entire rule set, we were still only able to implement parts of it. This, of course also has to do with us having to analyze each rule and getting into the Codan framework at first. But we are confident, that with the pre-work we have been able to achieve with our bachelor thesis, that implementing the remaining rules under "Constructors, assignments, and destructors" should be doable.

III Declaration of Authorship

We declare that this bachelor thesis and the work presented in it was done by ourselves and without any assistance, except what was agreed with the supervisor. All consulted sources are clearly mentioned and cited correctly. No copyright-protected materials are unauthorizedly used in this work.

Place and date	Özhan Kaya
Place and date	Kevin Schmidiger

Contents

1 Introduction

This bachelor thesis is about two projects. That is why we decided to write this section, where we are going to explain how this documentation is built up.
At first, we will work on some open points of our term project, the AliExtor [KO15b]. In that project we also built an Eclipse plug-in to extract alias declarations.
The second part is about the CCGLator, where we analyzed the problems discussed by the C++ Core Guidelines [BS15a] and created an Eclipse plug-in to improve coding of classes like suggested by the guidelines in the paragraph C.ctor: Constructors, assignments, and destructors[BS15c].

Note: The main part of the bachelor thesis is about the CCGLator, so we decided to write first about the CCGLator and then about the AliExtor.

1.1 CCGLator

The CCGLator is an Eclipse plug-in, supporting to adapt classes to the C++ Core Guidelines[BS15a]. Those guidelines were first released at the CppCon15[Cpp15]. They provide a huge set of rules leading to type-safe and less error-prone C++ code. The document is split up in different sections with each a different problem and its rules to avoid the problem. It is also to mention that the rules are far from complete. It is still being worked on as an open-source project hosted on GitHub[Git16].
For the bachelor thesis we selected the section called C.ctor: Constructors, assignments, and destructors[BS15c] from the super section C: Classes and Class Hierarchies[BS15b]. The rules are listed from C.20 up to C.89. We analyze all of the rules to see if they are possible for us to implement in the given time and how difficult they may be to implement. Each rule handles a problem which we will analyze and create a plug-in to highlight code violating a certain rule. It relates to the violated rule and provides a quick-fix for the rule like the guideline suggests.
A better description will be found in the section of the task description at section 2.1 on page 10, where it will be fully described what will be done in this thesis.

1.2 AliExtor

This section is about the term project AliExtor[KO15b]. At first we describe shortly and easily what we did in the term project and what the plug-in is capable of. After that we will give an overview of what additional work we are going to do.

1.2.1 Main goal of the term project

The term project was mostly gaining knowledge about the plug-in development and then apply the knowledge on our plug-in. It is capable of extracting type aliases. The result is shown in the next section.

1.2.2 The plug-in

We managed to extract type aliases from normal declarations up to parts of a declaration (see listing 1 on page 8).

```
int main() {
  // Normal extraction
  // Before:
  const int a { 42 };
  // After:
  using CINT1 = const int;
  CINT1 a { 42 };

  // Partial extraction
  int const * const b { &a };
  // After:
  using CINT2 = const int;
  CINT2 * const b { &a };
}
```

Listing 1: Full and partial alias extraction

We also managed to extract template alias, where you can choose which types should be template parameters (see listing 2 on page 8).

```
int main() {
  // Before:
  std::vector<std::pair<const int, std::string>> mymap { };

  // Possible transformations:
  // #1
  template<typename T>
```

```
8   using mycoolmap1 = std::vector<std::pair<const int, T>>;
9   mycoolmap1<std::string> mymap { };
10
11  // #2
12  template<typename T>
13  using mycoolmap2 = std::vector<std::pair<T, std::string>>;
14  mycoolmap2<const int> mymap { };
15
16  // #3
17  template<typename T1, T2>
18  using mycoolmap3 = std::vector<std::pair<T1, T2>>;
19  mycoolmap3<const int, std::string> mymap { };
20
21  // etc. ...
22 }
```

Listing 2: Alias Template extraction

In the project are some open points, which we closed at the beginning of this bachelor thesis.

1.2.3 Open points

Following points were to be fixed in our term project AliExtor:

- Simplification of the UI
- Add shortcut to invoke the plug-in
- Eliminate known bugs

This is only a short overview and will be fully described in the section of the task description at section 2.2.1 on page 12. They will be fixed in the first three weeks of the bachelor thesis and then we will go to the CCGLator.

2 Task description

In this section we will describe our task description for the bachelor thesis. The part is split into two sections, because we were working on two projects. The first section is about our term project AliExtor[KO15b], consisting mainly of bug-fixes and general improvements. The second part is about the CCGLator, where we spent the rest of our time.

2.1 CCGLator

In this section we will explain the task of the main part of our bachelor thesis, CCGLator.

2.1.1 Problem

C++ is a complex language evolving over the years. Especially with the release of C++11 many recommended practices became obsolete because the language and standard library provide safer and more efficient mechanisms to create type-safe and performant code. Not rarely it happens that a developer, even an experienced one, has to deal with some higher level issue such as resource management, memory management and concurrency concerns. Especially for people writing legacy code, these issues can become severe.

2.1.2 Solution

At the CppCon15[Cpp15] the C++ Core Guidelines[BS15a] were released. Those guidelines provide a large set of rules for clean coding in C++. Microsoft already released a support library for the guidelines called GSL[Mac15] on GitHub[Git16]. These guidelines focus on the higher level issues that are mention before such as resource management, memory management and concurrency and in general enforce usage of modern C++ (C++11/14/17). Although those guidelines are not complete yet, most of them can already be implemented as an IDE plug-in.

2.1.3 Our goals

The C++ Core Guidelines are very comprehensive with tons of rules structured into several categories. We took the section C: Classes and Class Hierarchies[BS15b] and there the subsection C.ctor: Constructors, assignments, and destructors[BS15c]. These rules are listed in rule C20 up to C.89. Reduced to those rules we then analyze them in terms of what problems could appear when they will be ignored and what problems we have to face to provide a refactoring for the rule. After the analysis we create a plug-in which helps the developer concern the problems which occur during creating a datatype in C++. The goal is to analyze all of the rules and implement as many as possible in the given time.

2.1.3.1 Features

The plug-in will cover the following rules with a corresponding quick-fix.

- **Set of default operations rules**
  - C.20: If you can avoid defining default operations, do
  - C.21: If you define or =delete any default operation, define or =delete them all
- **Destructor rules**
  - C.31: All resources acquired by a class must be released by the class's destructor
  - C.35: A base class with a virtual function needs a virtual destructor
  - C.37: Make destructors noexcept
- **Constructor rules**
  - C.44: Prefer default constructors to be simple and non-throwing
  - C.45: Don't define a default constructor that only initializes data members; use member initializers instead
  - C.46: By default, declare single-argument constructors explicit
  - C.47: Define and initialize member variables in the order of member declaration

  - C.48: Prefer in-class initializers to member initializers in constructors for constant initializers
  - C.49: Prefer initialization to assignment in constructors

- **Copy and move rules**
  - C.60: Make copy assignment non-virtual, take the parameter by const&, and return by non-const&
  - C.63: Make move assignment non-virtual, take the parameter by &&, and return by non-const&
  - C.66: Make move operations noexcept

2.2 AliExtor

This section is about the open points of our term project, the AliExtor[KO15b]. The points will be listed and also fully described.

2.2.1 Additional work

In our term project AliExtor we had some open points, which we finished in the first three weeks before beginning with the CCGLator.

These are the open points:

- Simplification of the UI
- Add shortcut to invoke the plug-in
- Eliminate known bugs

The open points will be fully described in the next sections (2.2.1.1 on page 12, 2.2.1.2 on page 13, 2.2.1.3 on page 14).

2.2.1.1 Simplification of the UI

The wizard for the refactoring of an alias template is a bit too complicated and will be simplified. At the beginning one had to select the type which should be extracted to an alias. We then analyzed this type and if it contained some template arguments

we added a possibility for the user to choose if this type should be an alias template. If the user wants to extract an alias template we then opened a new wizard page to let the user choose which types should be template parameters. This process will be simplified.
Now if a user extracts a type with template arguments we automatically show the possible arguments which can be chosen to be template parameter. If the user selects none of the arguments the extracted type will automatically be a simple alias.

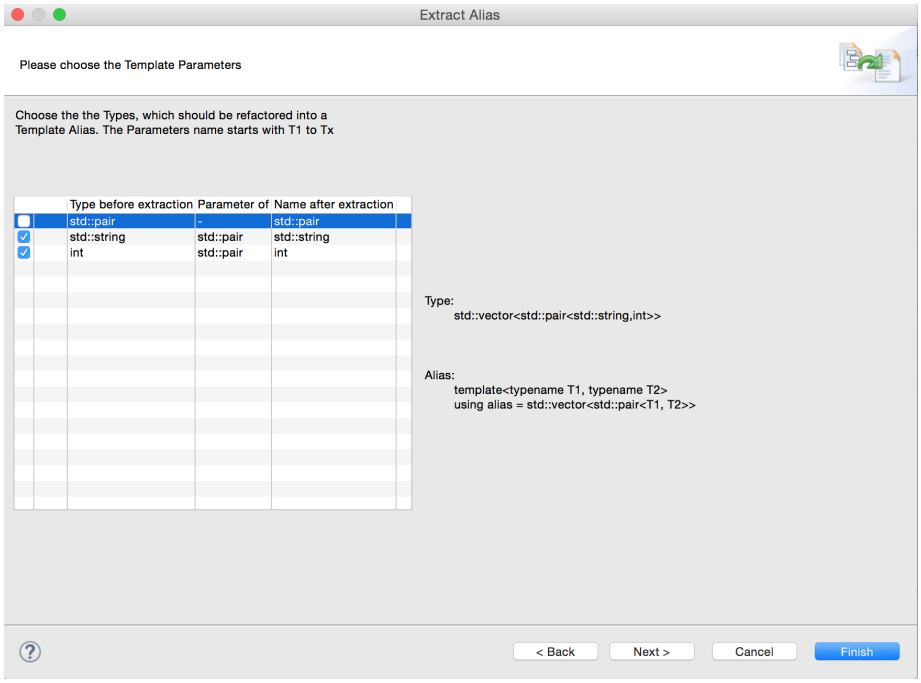

Figure 2: AliExtor UI with template parameters

2.2.1.2 Add a shortcut

Our plug-in has to be invoked implicitly by clicking on the selected type or via the menu in the menubar. We did not provide a shortcut for it (like the search function with ctrl + F). This will be changed to ctrl + cmd/win + A on the selected type.

2.2.1.3 Eliminate known bugs

After our term project we found a bug we are going to eliminate. The bug was if a user selects a type like in listing 3 on page 14.

```
1 int main() {
2   std::vector<std::pair<int, std::pair<int, std::string>>> keyKeyValue { };
3 }
```

Listing 3: Type which could be extracted to an alias template

Selecting the first std::pair caused a NullPointerException. We are going to analyze this and try to fix it.

2.2.2 Our goals

We want to make AliExtor better and easier to use. This will be done by fixing our open points listed in the previous section.

2.3 Time Management

Our project started on the 22nd of February, 2016. It will end on June the 17th 2016 on the final release day. This bachelor thesis is rewarded with 12 European Credit Transfer System (ECTS) points. For each point, 30 hours of work are estimated. This gives a total of 12 * 30 = 360 hours for the project per student.

2.4 Final Release

The following items will be included in the final release of the project:

- 2 color-printed exemplars of the documentation
- Poster for presentation
- Link to a demo video of our final product
- Management summary and abstract

- 2 CD/DVD with update site that contains the plug-in, project resources, documentation, video, poster and so on

3 CCGLator Analysis

In this section we are going to analyze each point of the C++ Core Guidelines[BS15a] section C.ctor: Constructors, assignments, and destructors[BS15c] from the super section C: Classes and class hierarchies[BS15b]. It starts with a few words about the Guidelines and the GSL[Mac15] and what its difference is.
To be able to fully understand the entries we will have to take a look at a subject called ownership and ensure a basic C++ knowledge so anyone can fully understand the problems discussed in the C++ Core Guidelines. Next we will provide a short tabular overview of all the rules. And finally there will be our analysis of the rules.

3.1 The C++ Core Guidelines

The C++ Core Guidelines were released at the CppCon15[Cpp15]. The guidelines contain a huge set of rules to create error free and safe code as well as modern C++ code. The release is far from complete, but already contains a lot of problems and possible solutions. The guidelines are available on GitHub[Git16] meaning they are open source and everybody can improve them.
The document is split up in different sections which are all about a special problem and then further divided into more concrete problems and possible solutions.
As already mentioned this project is about the rules C: Classes and Class Hierarchies[BS15a] with the concrete section C.ctor: Constructors, assignments, and destructors[BS15c].

3.2 Guideline Support Library from Microsoft

The Guideline Support Library GSL[Mac15] was released almost the same time as the C++ Core Guideline. It is a C++ Library containing some helper types or functions for safer code. Those types and functions can also be used for static analysis tools to indicate the user he possibly did something wrong. We can also benefit from those types especially gsl::owner<T*>. It is better described in the section 3.3.1 on page 17.

3.3 Ownership

Ownership is a very important subject when analyzing each rule. The GSL introduces a type called gsl::owner<T*> which does not do much. It is more like an annotation that this pointer is the owner of a resource it is pointing to. This can be very useful when it comes to developing an IDE plug-in which this project sometimes needs the information. The reason why ownership is so important is that a lot of the Guideline rules depend on knowing the owner of a variable beforehand. For example rule C.33 (see section 3.9.4 on page 36) is called "If a class has an owning pointer member, define a destructor".

3.3.1 What is an owner?

In principal it is very easy to explain what an owner is. See listing 4 on page 17.

```
1  struct mydata {
2    mydata(int size) : length { size }, owner { new int[length] { } }  { }
3  private:
4    int length;
5    int *owner;
6  };
7
8  int main(){
9    mydata data { 42 };
10 }
```

Listing 4: Example of an owner

It is very clear to see that the pointer owner (on line 5) is actually an owner of its resource. In this case it is pointing to an array on the heap with 42 possible entries. But what does that mean? Lets have a look on figure 3 on page 18.

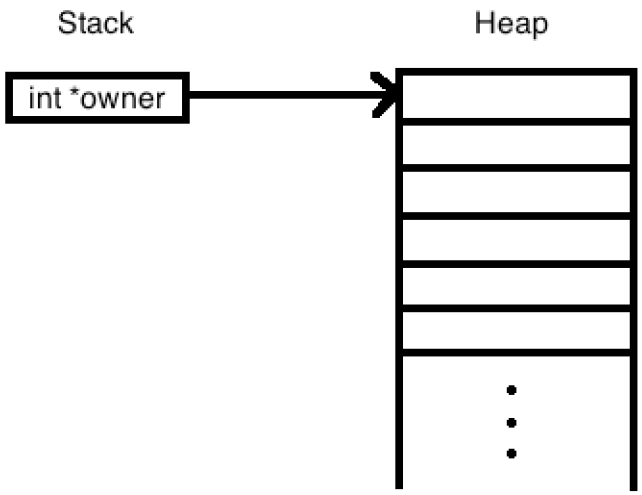

Figure 3: Illustration pointer pointing on heap allocation

The pointer owner (on line 5), is literally pointing to a memory address, where in this case the array is stored. To mention is, that the pointer itself is on the stack and the array is on the heap. Now if the array is not used anymore the memory on the heap needs to be released. This can be done with the call of delete[] owner. What happens then is the memory is released so that the operating system can use the space for other programs.

Note: There are two delete operators (see listing 5 on page 18).

```
1  struct example {
2    example() :
3        pointer { new int { 42 } }, array { new int [ &pointer ] } {
4    }
5
6    ~example(){
7      delete pointer; // for normal pointers
8      delete[] array; // for array allocations
9    }
10 private:
11   int *pointer;
12   int *array;
13 };
```

Listing 5: The two kinds of delete operators[Sta16b]

Sometimes even if the wrong delete operator is called it seems to do the right thing. Especially on arrays on primitive datatypes, because they do not have a destructor. But if there would be an array of objects which its destructor must be called, this could lead to serious problems better known as undefined behavior.

Note: Using the wrong delete operator and its behavior depends on the compiler. Some of them even detect the wrong use of it and generate warnings.

It is possible that the pointer is copied and placed all over the code because everybody needs access to this array. But what happens if, caused of some reason, one calls delete[] on one of the copies of the pointer? This is a serious problem because the pointers do not know (also there is no chance that they could even know) that the array is deallocated. And what happens if one tries to dereference such a pointer? Hello undefined behavior[ISO14a][1.3.24 undefined behavior].
Another problem that could occur is if the pointer to the array is deleted because it is removed from the stack frame. No nobody has access to the elements in the array and there is no chance on getting access back on it. The operating system thinks the program still uses the memory and in its descriptor table is has an entry that the space is used and so it does not use it for something other. It can not know that the pointer to this space is lost. This is called a memory leak[Wik16d].
The space will only be freed after the process is killed. These are two problems of pointers. Because of those problems it makes sense that only the creator of the array is allowed to free the space and all the holders of the copies are not. They are only allowed to read or change the values but nothing more. The creator now is the owner of the array. That is where the annotation gsl::owner<T*> comes in handy. It shows the programer that this pointer is an owner and is allowed (and at the end forced) to free the allocated memory.

3.3.2 How to find an owner?

When working with pointers we can never really know who is owning a piece of resource. That is why we appreciate when we see a gsl::owner<T*> annotation which we can work with. But of course the plugin has to work with any type of code and that is why an ability to manually find the owner of a resource is needed. But this kind of analysis could get very complex. The plug-in just looks if the pointer is matched on an owner and if so it forces to deallocate the memory properly and if not make sure the pointer is handled well.

Of course the problem would be a lot easier if smart pointer would be used so the problem with dangling pointer and memory leaks would not appear. We then would not have to care about pointers, since the smart pointer handles the deallocation of the resource by itself. But the guideline just talks about raw pointer so we had to stick with it. Otherwise the guidelines have to be rewritten to include smart pointers.

3.4 Codan

It is also necessary to explain what the Codan framework does and how it can be used. Codan is a light-weight static analysis framework in CDT that allows to easily plug-in "checkers" which perform real time analysis on the code to find common defects, violation of policies, etc. Following is an image showing some checker examples 4 on page 20.

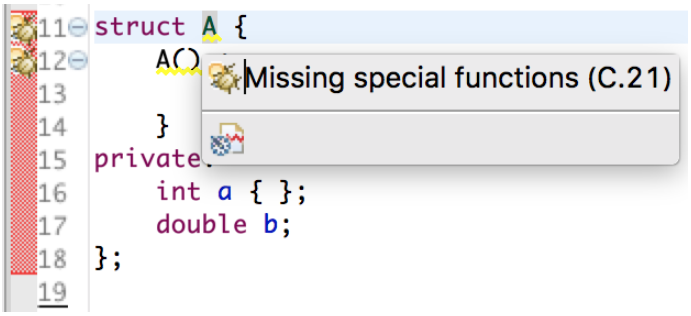

Figure 4: Some checkers

The checker highlights the code, in this case yellow and on the left side appears a bug icon. That means we can now extend the AbstractIndexAstChecker from the Codan due to build our own checkers to analyze the code due to some guideline violations. Quick-fixes are another feature from Codan. If a checker is set a quick-fix can be invoked to fix those defect or violation. This looks like the following figure 5 on page 21.

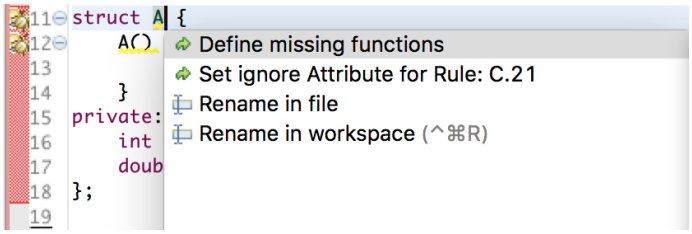

Figure 5: Quick-fix possibilities

The AbstractAstRewriteQuickFix can also be extended to provide our own quick-fixes.

3.5 C++ Core Guidelines structure

In this section we are going to analyze each entry of the C++ Core Guidelines. They are listed under the section "C: Classes and class hierarchies"[BS15a] with the concrete section "C.ctor: Constructors, assignments, and destructors"[BS15c]. Each entry is divided into several subcategories which are going to be described in the following sections.

Reason
Explains why and under which circumstances a rule makes sense. In general Bjarne Stroustrup and Herb Sutter are in the mindset that whenever you want to point out something bad, you should give a reason to why that is and that is what this section resembles.

Discussion
Link to a different section of the Core Guidelines paper giving additional information about a certain entry.

Example
Gives a code example explaining a point. Some examples show bad practice to show possible improvements.

Note
Gives additional information on a certain entry. Usually certain special rules or additions to the rule.

Alternative
Gives an alternative which conflicts with the rule but can make sense in some special cases.

Exception
Some rules have exceptions. Those will be listed in this area.

Enforcement
This is probably the most important section of the C++ Core Guidelines. This section only focuses on how the points discussed in the guidelines can be implemented. Some of the entries can not be implemented at all and some have various ways of implementation. The enforcements are often also marked as simple, hard or complex. Though sometimes there are no real ways to enforce a rule because a rule strongly depends on the semantics of a code which can not easily be determined. In cases like those the guidelines often offer a simplification of the rule as an enforcement (example: "C.30: Define a destructor if a class needs an explicit action at object destruction" (section 3.9.1 on page 34)).

3.6 Table overview of all the rules and its goals

This section gives a tabular overview of the rules we are going to analyze. The table consists of the rule and its reason (see section 3.5 on page 21). In the next section there will be the analysis of the rules. This table is to give an overview an a small understanding on what the rules are about (mostly the titles are very descriptive). The table shows the rules on the left and its description to the right. It contains the text from the guidelines but for some rules, we shortened the title to make it fit to the table.

Set of default operations rules (C.20 to C.22)

Rule	Short description
C.20: If you can avoid defining default operations, do	It's the simplest and gives the cleanest semantics.
C.21: If you define or =delete any default operation, define or =delete them all	The semantics of the special functions are closely related, so if one needs to be non-default, the odds are that others need modification too.
C.22: Make default operations consistent	Users will be surprised if copy/move construction and copy/move assignment do logically different things.

Table 1: Overview of the rules Set of default operations rules C.20 to C.22

Destructor rules (C.30 to C.37)

Rule	Short description
C.30: Define a destructor if a class needs an explicit action at object destruction	Only define a non-default destructor if a class needs to execute code that is not already part of its members' destructors.
C.31: All resources acquired by a class must be released by the class's destructor	Prevention of resource leaks, especially in error cases.
C.32: If a class has a raw pointer (T*) or reference (T&), consider whether it might be owning	There is a lot of code that is non-specific about ownership.
C.33: If a class has an owning pointer member, define or =delete a destructor	An owned object must be deleted upon destruction of the object that owns it.
C.34: If a class has an owning reference member, define or =delete a destructor	A reference member may represent a resource.
C.35: A base class with a virtual function needs a virtual destructor	To prevent undefined behavior.

Continuation of table 2 on the next page

Table 2 – *Continuation of table 2*

Rule	**Short description**
C.36: A destructor may not fail	In general we do not know how to write error-free code if a destructor should fail.
C.37: Make destructors noexcept	A destructor may not fail. If a destructor tries to exit with an exception, it's a bad design error and the program had better terminate.

Table 2: Overview of the rules Set of default operations rules C.30 to C.37

Constructor rules (C.40 to C.52)

Rule	**Short description**
C.40: Define a constructor if a class has an invariant	That's what constructors are for.
C.41: A constructor should create a fully initialized object	A user of a class should be able to assume that a constructed object is usable.
C.42: If a constructor cannot construct a valid object, throw an exception	Leaving behind an invalid object is asking for trouble.
C.43: Ensure that a class has a default constructor	Many language and library facilities rely on default constructors to initialize their elements, e.g. T a[10] and std::vector<T> v(10).
C.44: Prefer default constructors to be simple and non-throwing	Being able to set a value to "the default" without operations that might fail simplifies error handling and reasoning about move operations.
C.45: Don't define a default constructor that only initializes data members; use member initializers instead	The compiler-generated function can be more efficient.
C.46: By default, declare single-argument constructors explicit	To avoid unintended conversions.

Continuation of table 3 on the next page

Table 3 – *Continuation of table 3*

Rule	**Short description**
C.47: Define and initialize member variables in the order of member declaration	To minimize confusion and errors. That is the order in which the initialization happens (independent of the order of member initializers).
C.48: Prefer in-class initializers to member initializers in constructors for constant initializers	Makes it explicit that the same value is expected to be used in all constructors. It leads to the shortest and most efficient code.
C.49: Prefer initialization to assignment in constructors	An initialization explicitly states that initialization, rather than assignment, is done and can be more elegant and efficient.
C.50: Use a factory function if you need "virtual behavior" during initialization	If the state of a base class object must depend on the state of a derived part of the object, we need to use a virtual function (or equivalent) while minimizing the window of opportunity to misuse an imperfectly constructed object.
C.51: Use delegating constructors to represent common actions for all constructors of a class	To avoid repetition and accidental differences.
C.52: Use inheriting constructors to import constructors into a derived class that does not need further explicit initialization	If you need those constructors for a derived class, re-implementing them is tedious and error prone.

Table 3: Overview of the constructor rules: C.40 to C.52

Copy and move rules C.60 to C.67

Rule	**Short description**

Continuation of table 4 on the next page

Table 4 – *Continuation of table 4*

Rule	**Short description**
C.60: Make copy assignment non-virtual, take the parameter by const&, and return by non-const&	It is simple and efficient. If you want to optimize for rvalues, provide an overload that takes a && (see F.24[BS15a, Rule F.24]).
C.61: A copy operation should copy	After a copy x and y can be independent objects (value semantics, the way non-pointer built-in types and the standard-library types work) or refer to a shared object (pointer semantics, the way pointers work).
C.62: Make copy assignment safe for self-assignment	If x=x changes the value of x, people will be surprised and bad errors will occur (often including leaks).
C.63: Make move assignment non-virtual, take the parameter by &&, and return by non-const&	It is simple and efficient.
C.64: A move operation should move and leave its source in a valid state	That is the generally assumed semantics. After x=std::move(y) the value of x should be the value y had and y should be in a valid state.
C.65: Make move assignment safe for self-assignment	If x = x changes the value of x, people will be surprised and bad errors may occur.
C.66: Make move operations noexcept	A throwing move violates most people's reasonably assumptions. A non-throwing move will be used more efficiently by standard-library and language facilities.
C.67: A base class should suppress copying, and provide a virtual clone instead if "copying" is desired	To prevent slicing, because the normal copy operations will copy only the base portion of a derived object.

Table 4: Overview of the copy and move rules (C.60 to C.67)

Other default operations rules (C.80 to C.89)

Rule	**Short description**
C.80: Use =default if you have to be explicit about using the default semantics	The compiler is more likely to get the default semantics right and you cannot implement these function better than the compiler.
C.81: Use =delete when you want to disable default behavior (without wanting an alternative)	In a few cases, a default operation is not desirable.
C.82: Don't call virtual functions in constructors and destructors	A direct or indirect call to an unimplemented pure virtual function from a constructor or destructor results in undefined behavior.
C.83: For value-like types, consider providing a noexcept swap function	A swap can be handy for implementing a number of idioms, from smoothly moving objects around to implementing assignment easily to providing a guaranteed commit function that enables strongly error-safe calling code.
C.84: A swap may not fail	swap is widely used in ways that are assumed never to fail and programs cannot easily be written to work correctly in the presence of a failing swap. The standard-library containers and algorithms will not work correctly if a swap of an element type fails.
C.85: Make swap noexcept	If a swap tries to exit with an exception, it's a bad design error and the program had better terminate.
C.86: Make == symmetric with respect of operand types and noexcept	Asymmetric treatment of operands is surprising and a source of errors where conversions are possible.
C.87: Beware of == on base classes	It is really hard to write a foolproof and useful == for a hierarchy.
C.89: Make a hash noexcept	Users of hashed containers use hash indirectly and don't expect simple access to throw. It's a standard-library requirement.

Table 5: Overview of other default operations rules C.80 to C.89

3.7 C.ctor: Constructors, assignments, and destructors

Because the C++ Core Guidelines structured the guidelines in four categories the same decision was made in this structure. So the rules are a bit more structured then just a big set of rules.

Each rule begins with the reason copied from the guidelines and the the problem which is meant to be avoided will be analyzed. We will show what consequences it could happen if the rule is ignored (problem paragraph). At the end of the section it is also analyzed what problems we have to face when it comes to implement the plug-in (Problems concerning the implentation paragraph). Some times the last paragraph is missing because the rule is not implementable or there are none detected or already mentioned in the problem paragraph.

Before the rules are analyzed we explain what the default operations are and some idioms that are good to know.

3.7.1 Default operations

Good to know is that these are the default operations

- Default constructor: X()
- Default copy constructor: X(const X&)
- Default copy assignment: operator=(const X&)
- Default move constructor: X(X&&)
- Default move assignment: operator=(X&&)
- Default destructor: ∼X()

The compiler has well defined rules on generating default operations depending on what the developer already declared. See figure 6 on page 29.

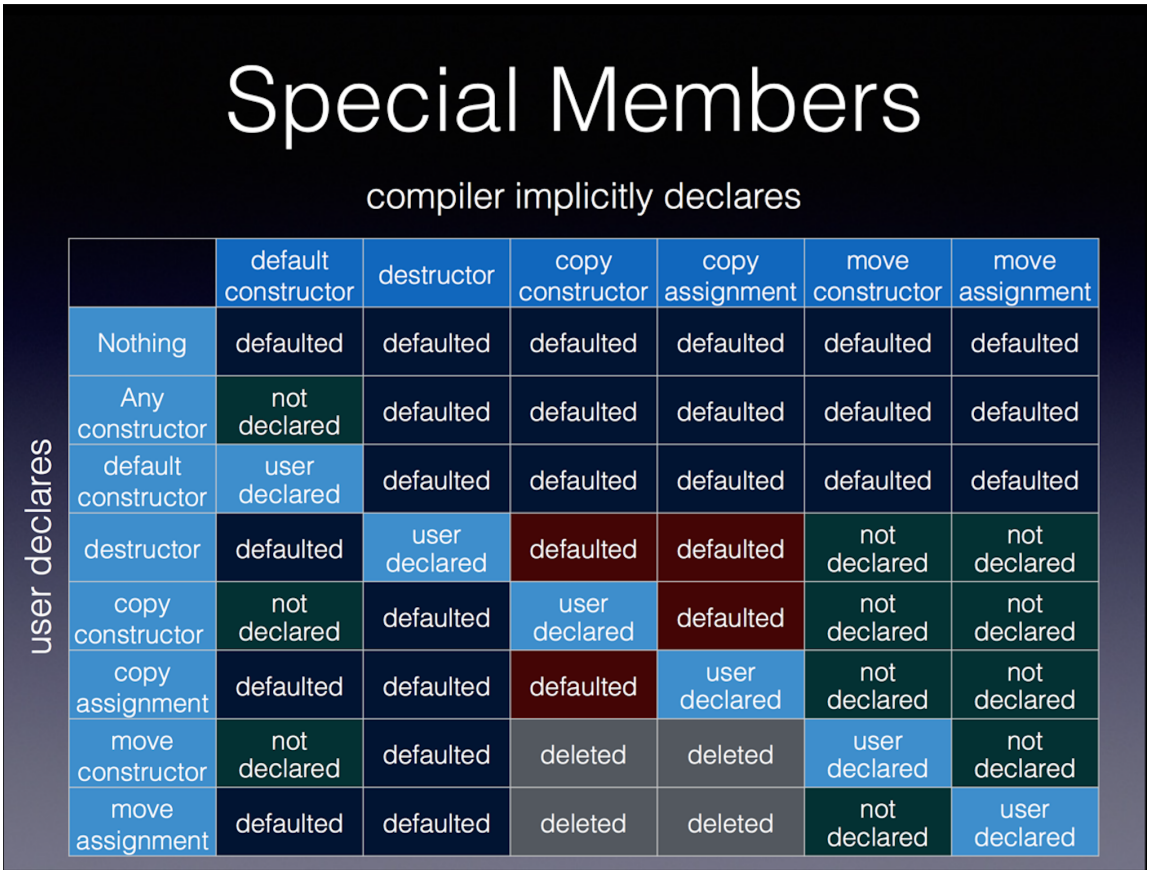

Figure 6: Compiler generating default operation rules[Hin14]

3.7.2 Rule of three/five/zero

There are some programming idioms like the rule of three/five/zero[Wik16e]. Those idioms deal with which default operations should be explicitly declared. The first of them was the rule of three. It came up in the early 90's. Then it was improved by the rule of five with C++11. The rule of zero is kind of the same like the rule of five and is not mentioned any further.

Rule of three
This idiom handles the following default operations.

- destructor

- copy constructor
- copy assignment operator

Rule of five
This idiom handles the following default operations.

- destructor
- copy constructor
- copy assignment operator
- copy assignment operator
- move assignment operator

For each of the idioms, if one or more of them is defined the developer should define all of them.

3.7.3 RAII (Resource Acquisition Is Initialization)

> *"In RAII, holding a resource is a class invariant, and is tied to object lifetime: resource allocation (acquisition) is done during object creation (specifically initialization), by the constructor, while resource deallocation (release) is done during object destruction (specifically finalization), by the destructor. Thus the resource is guaranteed to be held between when initialization finishes and finalization starts (holding the resources is a class invariant), and to be held only when the object is alive. Thus if there are no object leaks, there are no resource leaks."*
>
> - Wikipedia, RAII

3.8 Set of default operations rules C.20 to C.22

These are the rules about default operations (section 3.7.1 on page 28).

3.8.1 C.20: If you can avoid defining default operations, do

Reason

> *"It's the simplest and gives the cleanest semantics."*
>
> - C.20 from the C++ Core Guidelines

Problem
While the C++ Core Guidelines paper refers to this rule as "not enforceable". Though there are some patterns which we could look for indicating that default operations may not be necessary in a certain case. Lets see listing 6 on page 31.

```
1 struct Named_map {
2 public:
3     Named_map(){}
4 private:
5     std::string name;
6     std::map<int, int> rep;
7 };
```

Listing 6: Default constructor[BS15c](Rule C.20)

Here the default constructor makes no sense because there is a default constructor for std::string and std::map.

Problems concerning the implementation
The default constructor is what the plug-in looks for. If there are the member variables default initialized, then no default constructor is needed. If one of the member variables is a pointer its an indicator that a some default operations are needed. This leads to the next rule.

3.8.2 C.21: If you define or =delete any default operation, define or =delete them all

Reason

> *"The semantics of the special functions are closely related, so if one needs to be non-default, the odds are that others need modification too."*

- C.21 from the C++ Core Guidelines

Problem
The default operations come in pairs. If one needs to define a copy constructor the chance is high that a destructor is needed too. The same for the copy assignment and copy constructor as well as the move constructor and the move assignment.
In the guidelines there is a code example. So lets analyze this (see listing 7 on page 32).

```
1  struct M2 {   // bad: incomplete set of default operations
2  public:
3      // ...
4      // ... no copy or move operations ...
5      ~M2() { delete[] rep; }
6  private:
7      pair<int, int>* rep;  // zero-terminated set of pairs
8  };
9
10 void use()
11 {
12     M2 x;
13     M2 y;
14     // ...
15     x = y;   // the default assignment
16     // ...
17 }
```

Listing 7: Default operators[BS15c](Rule C.21)

There are no explicit default operations except for the destructor. That means that the compiler generates the default operations except for the move operations. Now, what happens if the function use() on line 10 is called? At first the two M2 objects are created. With the assignment of x = y; on line 15 the object x calls the from the compiler generated copy assignment operator.
Now there are two of the same type and the default assignment operator copied the pointer, meaning two pointers are now pointing to the same location. And at the end of the function the destructor of the objects is called. Because of the copied pointer this works with the first object and with the second object this leads to undefined behavior. This is also called double delete.

Problems concerning the implementation
Since the enforcement says if one of the default operation is declared all of them

should be declared. This is basically the rule of five 3.7.2 on page 29. So what we will do is analyze if there are default operations and if so generate the rest of them.

3.8.3 C.22: Make default operations consistent

Reason

> *"The default operations are conceptually a matched set. Their semantics are interrelated. Users will be surprised if copy/move construction and copy/move assignment do logically different things. Users will be surprised if constructors and destructors do not provide a consistent view of resource management. Users will be surprised if copy and move do not reflect the way constructors and destructors work."*
>
> - C.22 from the CPP Core Guidelines

Problem

This rule may be a little harder to implement since there are quite a few things that can be checked for. The main goal of this rule is that default operations have to be consistent as one would expect from them. There are various checks we can do to achieve consistency as also listed in the enforcements of the paper. Checking for the level of dereference in the corresponding default operations makes sense and is a great first step to guarantee consistency and in general all the 4 enforcement points make sense. But when it comes to delivering a quick-fix things start to become more complicated. Let's say we perform a deep copy in the copy constructor and a shallow copy in the copy assignment (see figure 8 on page 33).

```
1  class Silly {   // BAD: Inconsistent copy operations
2      class Impl {
3          // ...
4      };
5      shared_ptr<Impl> p;
6  public:
7      Silly(const Silly& a) : p{a.p} { *p = *a.p; }   // deep copy
8      Silly& operator=(const Silly& a) { p = a.p; }   // shallow copy
9      // ...
10 };
```

Listing 8: Deep and shallow copy[BS15c](Rule C.22)

Problems concerning the implementation
A quick-fix could either be setting both assignments to a deep or shallow copy. There can also be cases where an initialization of a member variable is missing. In this case we would have to initialize it with a quick-fix. This is also a case where there could be various ways to initialize said variable.

3.9 Destructor rules C.30 to C.37

These are the rules about the destructor.

3.9.1 C.30: Define a destructor if a class needs an explicit action at object destruction

Reason

> *"A destructor is implicitly invoked at the end of an object's lifetime. If the default destructor is sufficient, use it. Only define a non-default destructor if a class needs to execute code that is not already part of its members' destructors."*
>
> - C.30 from the C++ Core Guidelines

Problem
This rule can be looked at in various ways. We can not semantically determine if the purpose of a class main purpose is to execute a final action before destruction so it is not possible for us to tell if a destructor is missing given the semantic context of the class. But what we can do is look for resources which have to be cleared by the destructor and check for their existence inside the destructor code. What we can also do is detect if there is a piece of code in a user defined destructor being already present in one of the member variables destructor (redundant destruction). What the paper requests as an enforcement is also to look likely "implicit resources" like pointers or references and check for their destruction in the user defined destructor which makes sense.

3.9.2 C.31: All resources acquired by a class must be released by the classes destructor

Reason

> *"Prevention of resource leaks, especially in error cases."*
>
> - C.31 from the C++ Core Guidelines

Problem

Starting with this rule, things start to become a bit more complex. Member variables of a class that are not a pointer or reference are not problematic at all since they already have implemented destructors being automatically called upon destruction of our own class if those classes follow the guidelines. But when we are working with pointers or references there is no clear way for us to tell who the actual owner of a resource is. What the GSL header offers as a help is the gsl::owner<T*> annotation. We already discussed it in section 3.3.1 on page 17. Basically by having a gsl::owner<T*> annotation we can be sure that we are working with an owner of any type and therefore know that we have to clear a piece of resource ourselves to prevent memory leak. But usually we work on code which does not have owner annotations. That makes the implementation of this rule even harder (see listing 9 on page 35). To determine of a pointer or reference is an owner the team GSLatorPtr[PM16] is working on that.

```
1 class X2 {     // bad
2     FILE* f;   // may own a file
3     // ... no default operations defined or =deleted ...
4 };
```

Listing 9: Possible class needing a destructor[BS15c](Rule C.31)

But after having a look on the other rules we decided to split the detection of an owner and working with an owner to enforce a quick-fix in 2 parts. For more information on this, see the following rule.

3.9.3 C.32: If a class has a raw pointer (T*) or reference (T&), consider whether it might be owning

Reason

> *"There is a lot of code that is non-specific about ownership."*
>
> - C.32 from the C++ Core Guidelines

This rule mainly focuses on detecting owners inside classes or structs. A lot of rules depend on knowing which variable owns a piece of resource. But it is very hard to detect if an pointer or reference is an owner. Here is where the GSL annotation gsl::owner<T*> comes in handy. If a pointer is annotated with that we can make sure that the class needs a destructor. Otherwise the user needs to be warned that this pointer could be an owner.
The guidelines requests that an owning pointer should be marked as owning and the a non owning pointer should be marked as ptr.

This rule will be ignored because another student project tries to provide owner<T*> refactoring. It will only make sense after such annotations exist to enforce this rule.

3.9.4 C.33: If a class has an owning pointer member, define or =delete a destructor

Reason

> *"An owned object must be deleted upon destruction of the object that owns it."*
>
> - C.33 from the C++ Core Guidelines

Problem

This rule is easy to implement if there is a given gsl::owner<T*> annotation. But if there is a pointer as a member variable, guidelines suggest this case should be marked as "suspect". Sadly doing that is not helpful at all. But once rules C.32 (section 3.9.3 on page 35) is implemented this case can easily be solved by only having gsl::owner<T*> for owners and no annotation for non-owners.

```
1 template<typename T>
2 class Smart_ptr {
3     T* p;   // BAD: vague about ownership of *p
4     // ...
5 public:
```

```
6     // ... no user-defined default operations ...
7 };
8
9 void use(Smart_ptr<int> p1)
10 {
11     auto p2 = p1;   // error: p2.p leaked (if not nullptr and not owned by some
       other code)
12 }
```

Listing 10: No clear ownership[BS15c](Rule C.33)

In the listing 10 on page 36 the member variable on line 3 is not clearly defined as owner or non-owner. So there could occur some memory leaks.

3.9.5 C.34: If a class has an owning reference member, define or =delete a destructor

Reason

> *"A reference member may represent a resource. It should not do so, but in older code, that's common. See pointer members and destructors. Also, copying may lead to slicing."*
>
> - C.34 from the C++ Core Guidelines

Problem
Although this rule should only apply to legacy code it can easily be solved by implementing rule C.32 (section 3.9.3 on page 35). In general it has a lot of similarities with rule C.33 (section 3.9.4 on page 36).

3.9.6 C.35: A base class with a virtual function needs a virtual destructor

Reason

> *"To prevent undefined behavior. If the destructor is public, then calling code can attempt to destroy a derived class object through a base class pointer, and the result is undefined if the base class's destructor is non-virtual. If the destructor is protected, then calling code cannot destroy*

> *through a base class pointer and the destructor does not need to be virtual; it does need to be protected, not private, so that derived destructors can invoke it. In general, the writer of a base class does not know the appropriate action to be done upon destruction."*
>
> - C.35 from the C++ Core Guidelines

Problem

This rule makes sense, since it is something an inexperienced C++ programmer can easily mess up. Also it can be forgotten to implement. The guidelines suggest that a class with virtual functions must have a virtual destructor. Otherwise this could lead to memory leaks and undefined behavior (see listing 11 on page 38).

```
1  struct Base {  // BAD: no virtual destructor
2      virtual f();
3  };
4
5  struct D : Base {
6      string s {"a resource needing cleanup"};
7      ~D() { /* ... do some cleanup ... */ }
8      // ...
9  };
10
11 void use()
12 {
13     unique_ptr<Base> p = make_unique<D>();
14     // ...
15 } // p's destruction calls ~Base(), not ~D(), which leaks D::s and possibly more
```

Listing 11: Undefined behavior caused by a non-virtual destructor[BS15c](Rule C.35)

Problems concerning the implementation

What the we can do is to lookup if there is a virtual function declared. If so look the super type up and check if there is an virtual destructor. What has to be considered is that the super type most likely is not in the same file meaning we would have to look up the file. This leads to the problem that the file may not be parsed yet. That means the file must be found which can be done over the binding. Next the file needs to be parsed to work on the AST[Wik16a].

3.9.7 C.36: A destructor may not fail

Reason

> *"In general we do not know how to write error-free code if a destructor should fail. The standard library requires that all classes it deals with have destructors that do not exit by throwing."*
>
> - C.36 from the C++ Core Guidelines

Problem
A big problem occurs if a destructor fails. There could be resources being not released or a socket which will not close (see listing 12 on page 39).

```
1  class X {
2  public:
3      ~X() noexcept;
4      // ...
5  };
6
7  X::~X() noexcept
8  {
9      // ...
10     if (cannot_release_a_resource) terminate();
11     // ...
12 }
```

Listing 12: A failing destructor[BS15c](Rule C.36)

A better understanding of the noexcept[IEC14c] specifier is needed. If a function is marked with noexcept (or noexcept(true)) and the functions throws an exception, std::terminate() will be called. In case of a possible memory leak (like in the listing 12 on page 39) the termination of the program will most often release its memory from the heap. This is done by the OS. The mainstream OS will free the heap but this is not necessarily true from an embedded system.

Note The guidelines also mentions that there are no fool-proof scheme for writing a destructor which does not fail. Also it is not possible to retry "close/release" operations. It also says, if at all possible, consider failure to close/cleanup a fundamental design error and terminate.

Note It is also mentioned, if a resource cannot be released and the program may not fail, try to signal the failure to the rest of the system somehow. The destructor could send a message (somehow) to the responsible part of the system and return normally.

Problems concerning the implementation
This rule focusses on how a destructor should handle the case of an exception. But considering all this things it could get quite problematic. The easy way to get around this problem is to declare the destructor noexcept. The hard way would be to check all function calls and mark them as noexcept.
The plug-in will take the easy way. That is basically rule C.37 (section 3.9.8 on page 41). It could check if there is an throw statement in the body and surround it with a try catch(...).
There are two ways to catch the exception. See listing 13 on page 40.

```
1  // First example
2  struct X {
3    ~X() noexcept {
4      try {
5        throw 42;
6      } catch (...) { // catch all
7        // handle ...
8      }
9    }
10 };
11
12 // Second example
13 struct Y {
14   ~Y() noexcept  try {
15     throw 42;
16   } catch (...) { // catch all
17     // handle ...
18   } // implicit throw; here
19 };
```

Listing 13: Different kind of catches

The first example is the most common. A normal try catch in the destructor scope. The exception is caught and handled in the scope itself. Other in the second example, being the better one. It now is a Function-try-block[IEC14a]. The difference now is, if used in on a con- or destructor, that the member variables will be destructed before the catch scope will be entered. Also after the catch clause there is an implicit throw of the current exception. Because it makes no sense to continue the program, if the throw is caused by a fail of deallocation and this would lead to a memory leak, it is

better to terminate the program. And that is where the second example comes in handy. With its implicit rethrow of the exception and the behavior of the noexcept specifier, it leads to a std::terminate().

3.9.8 C.37: Make destructors noexcept

Reason

> *"A destructor may not fail. If a destructor tries to exit with an exception, it's a bad design error and the program had better terminate."*
>
> - C.37 from the C++ Core Guidelines

Problem
An easy to implement rule. Like described in rule C.36 (section 3.9.7 on page 39) the plug-in will not differ between this two rules. And the rule is to declare destructors noexcept.

3.10 Constructor rules C.40 to C.52

These are the rules for the constructor.

3.10.1 C.40: Define a constructor if a class has an invariant

Reason

> *"That's what constructors are for."*
>
> - C.40 from the C++ Core Guidelines

Problem
In C++ the constructor takes the role of a validator that the class is in a valid state when created. This is important if the class contains at least one pointer or other data-types that should be specially initialized. Sometimes a class does not have an invariant meaning the constructor is just a default constructor. But then rule C.20

(section 3.8.1 on page 31) says, If you can avoid defining default operations, do. So the constructor is just unnecessary. The problem now is, if the the member data-types need to be in a special state (see listing 14 on page 42).

```
class Date {  // a Date represents a valid date
              // in the January 1, 1900 to December 31, 2100 range
    Date(int dd, int mm, int yy)
        :d{dd}, m{mm}, y{yy}
    {
        if (!is_valid(d, m, y)) throw Bad_date{};  // enforce invariant
    }
    // ...
private:
    int d, m, y;
};
```

Listing 14: A possible invariant[BS15c](Rule C.40)

Problems concerning the implementation
The plug-in can not know if members need to be initialized in a way for the class to be invariant. The guidelines suggests that if a class has a user defined copy constructor it is very likely that this class has an invariant as well. But then rule C.21 (section 3.8.2 on page 31) If you define or =delete any default operation, define or =delete them all comes in (basically rule of five[**?**]). Which automatically means the default constructor should be defined as well as maybe the destructor. So if the constructor is needed, rule C.20 will check if it is just a default constructor, but now there should be something like a is_valid(/*...*/) function to check its invariant. If so the rule C.20 will be ignored at this constructor. This rule basically introduces the next rule.

3.10.2 C.41: A constructor should create a fully initialized object

Reason

> *"A constructor establishes the invariant for a class. A user of a class should be able to assume that a constructed object is usable."*
>
> - C.41 from the C++ Core Guidelines

Problem
RAII3.7.3 on page 30 stands for resource allocation is initialization. so if a class

handles a resource it should be initialized at construction. if it is not possible is not fully initialized.
If the class needs to do more to be fully initialized, just initialize member data types.

```
1  class X1 {
2      FILE* f;   // call init() before any other function
3      // ...
4  public:
5      X1() {}
6      void init();   // initialize f
7      void read();   // read from f
8      // ...
9  };
10
11 void f()
12 {
13     X1 file;
14     file.read();   // crash or bad read!
15     // ...
16     file.init();   // too late
17     // ...
18 }
```

Listing 15: Should call init() at construction[BS15c](Rule C.41)

Because of the RAII principle if the init() is not called at construction the class is not fully initialized. Other a user of this class could use it wrong like shown. The plug-in can not know if there are any data types which should be init()ed at construction. There is also no chance to get any information about such a data-type. To avoid this problem see the following rule.

3.10.3 C.42: If a constructor cannot construct a valid object, throw an exception

Reason

> *"Leaving behind an invalid object is asking for trouble."*
>
> - C.42 from the C++ Core Guidelines

Problem
A constructor checks if an object is initialized correctly. But what to do if invariant of the object could not be established? An object should not be constructed if it

would be invariant. That can be done by throwing an exception in the constructor. See listing 16 on page 44.

```
class X2 {
    FILE* f;   // call init() before any other function
    // ...
public:
    X2(const string& name)
        :f{fopen(name.c_str(), "r")}
    {
        if (f == nullptr) throw runtime_error{"could not open" + name};
        // ...
    }

    void read();      // read from f
    // ...
};

void f()
{
    X2 file {"Zeno"}; // throws if file isn't open
    file.read();      // fine
    // ...
}
```

Listing 16: Throw exception if a valid object can not be constructed[BS15c](Rule C.42)

There is a problem if there are any resources allocated in the constructor. If there would be a raw pointer owning some memory on the heap this memory needs to be released if an exception would occur. The exception needs to be caught and and the memory released. It would be something different if the pointer would be a smart pointer. In that case the memory would be released by the destructor of the smart pointer. Like in described in rule C40 section 3.10.1 on page 41 we can not determine if a class has an invariant neither what that invariant would look like.

3.10.4 C.43: Ensure that a class has a default constructor

Reason

> *"Many language and library facilities rely on default constructors to initialize their elements, e.g. T a[10] and std::vector<T> v(10)."*
>
> - C.43 from the C++ Core Guidelines

Problem
This rule is better explained with a code example. Lets see listing 17 on page 45.

```
1 class Date { // BAD: no default constructor
2 public:
3     Date(int dd, int mm, int yyyy);
4     // ...
5 };
6
7 vector<Date> vd1(1000);   // default Date needed here
8 vector<Date> vd2(1000, Date{Month::october, 7, 1885});   // alternative
```

Listing 17: Default constructor needed[BS15c](Rule C.43)

In line 7 there is a vector constructed. In this case the constructor explicit vector(size_type count)[cpp13] is called. It constructs a vector with 1000 default initialized elements. The problem now is, if there is no default constructor it is not possible to do so. The compiler only generates the default constructor if there is no other constructor. It makes sense because if there is a user defined constructor this means the object needs to be initialized in a way the compiler can not know.
Now this would mean the vector needs to be initialized like in line 8 meaning you have to implicitly construct the object in a valid way. This not very nice and could make some problems in other cases. So if there would be a default constructor initializing the object like in line 8 it would be possible to initialize the vector in like in line 7.
It is also to mention that there is no natural start date like the guidelines says: "There is no "natural" default date (the big bang is too far back in time to be useful for most people)"[1]. Also default initializing would not work, because a date 0.0.0 is not a valid date. A good default date could be 1.1.1970 (January 1, 1970 is the date where the Unix time[Wik16c] started).

Problems concerning the implementation
The guideline says classes without default constructor should be flagged. But the plug-in will be able to detect, that there is no default constructor and provides to implement it, wether if it sets it to default or needs to delegate to another constructor with valid default parameters. But it would be better if the default constructor is set to default and with rule C.45 (section 3.10.6 on page 46) Do not define a default constructor that only initializes data members; use in-class member initializers instead, it would be ensured that the object is in a valid state after construction.

[1]Note how its says "for most people" :D

3.10.5 C.44: Prefer default constructors to be simple and non-throwing

Reason

> *"Being able to set a value to "the default" without operations that might fail simplifies error handling and reasoning about move operations."*
>
> - C.44 from the C++ Core Guidelines

This rule is kind of a preparation for the next rule. It is also a rule where some or the rules work together. Lets analyze some code (listing 18 on page 46).

```
template<typename T>
class Vector0 {   // elem points to space-elem element allocated using new
public:
    Vector0() :Vector0{0} {}
    Vector0(int n) :elem{new T[n]}, space{elem + n}, last{elem} {}
    // ...
private:
    own<T*> elem;
    T* space;
    T* last;
};
```

Listing 18: Default constructor noexcept[BS15c](Rule C.44)

The default constructor delegates to the constructor Vector0(int n). This constructor allocates memory which could possibly fail or in other words could possibly throw an exception. If a constructor is defined as noexcept the compiler can generate more efficient code and it can better be handled by the STL[Wik16f] (standard template library.

Problems concerning the implementation
It is better if the default constructor is set to noexcept and the default initialize values are then handled by rule C.45. So what the plug-in does, is set the default constructor to noexcept and then the rule C.45 takes its magic (explained in the next section).

3.10.6 C.45: Don't define a default constructor that only initializes data members; use member initializers instead

Reason

> *"Using in-class member initializers lets the compiler generate the function for you. The compiler-generated function can be more efficient."*
>
> - C.45 from the C++ Core Guidelines

Problem

Why write a default constructor, initializing just the member variables? They are better in-class initialized and if there is a non default constructor initializing the variable its in-class initialize will be ignored. It is less code and the user can directly see what the default initialization is. Also it is possible that the compiler can generate efficient code because if they are not default initialized the compiler could evaluate the expression if possible. And that means he would generate much more efficient code than the user would have done.

Problems concerning the implementation

So the plug-in has to check if the default constructor just initializes the data members. If so, take the values, if not default initialized, and in-class initialize the data members. This is somehow the same what rule C.20 (section 3.8.1 on page 31) does: If you can avoid defining any default operations, do. So the quick-fix for the constructor in rule C.20 will be this rule.

```
1  class X1 { // before
2      string s;
3      int i;
4  public:
5      X1() :s{"default"}, i{1} { }
6      // ...
7  };
8
9  class X1 { // after
10     string s = "default";
11     int i = 1;
12 public:
13     // use compiler-generated default constructor
14     // ...
15 };
```

Listing 19: Example of set to noexcept[BS15c](Rule C.45)

3.10.7 C.46: By default, declare single-argument constructors explicit

Reason

> *"To avoid unintended conversions."*
>
> - C.46 from the C++ Core Guidelines

This rule is also better explained with code an example. So lets have a look at listing 20 on page 48.

```
class String {
    // ...
public:
    String(int);   // BAD
    // ...
};

String s = 10;   // surprise: string of size 10
```

Listing 20: Surprising conversion[BS15c](Rule C.46)

Problem

Maybe it is necessary to talk about the different types of object initialization. They are listed in listing 21 on page 48.

```
// #1
T a;

// #2
T a();

// #3
T a(1, 2, 3... args);

// #4
T a = 1;

// #5
T a = {1, 2, 3, 4, 5, 6};

// #6
T a{1, 2, 3, 4, 5, 6};

// #7
T* a = new T(1, 2, 3);

// #8
```

```
23 T* a = new T;
```

Listing 21: Different types of object initialization

If the object is created like in example 1 it would create an object, but its trivial data member would be uninitialized. It is kind of the same if an int would be created like int a;. The variable is uninitialized and trying to work with it would lead to undefined behavior if not initialized.
The second example is basically declaring a function a() and not creating anything at all.
The third example calls the constructor with the exact amounts of parameter past to.
Number four would call the a single argument constructor with the matching type. On this kind of initialisation we will have a closer look later. Lets explain the other initializations first.
Number five is the same as number 4. Since the right hand side is an initializer list it to the constructor with the initializer list. So number five and six are the same. Seven and eight are allocations on the heap with the corresponding constructor calls. In listing 20 on page 48 on line 8 the user would expect s to be "10". But instead the constructor which takes an int argument will be invoked. If the constructor would have been declared explicit that would not happen. Ant thats what this rule is about

3.10.8 C.47: Define and initialize member variables in the order of member declaration

Reason

> *"To minimize confusion and errors. That is the order in which the initialization happens (independent of the order of member initializers)."*
>
> - C.47 from the C++ Core Guidelines

Problem
Most compilers warn if the order of the constructor is not the same as the data member order. The problem is that the compiler ignores the order of the user defined initialization list in the constructor and initializes it in the order of the data

members. And that could be a problem. See listing 22 on page 50.

```
1 class Foo {
2     int m1;
3     int m2;
4 public:
5     Foo(int x) :m2{x}, m1{++x} { }   // BAD: misleading initializer order
6     // ...
7 };
8
9 Foo x(1); // surprise: x.m1 == x.m2 == 2
```

Listing 22: Surprise initialisation[BS15c](Rule C.47)

Problems concerning the implementation
So the plug-in needs to check if the order of the initializing list is the same as the data member. If not it signals the user that this is not the right order and that the user defined initializer list will not be the order in which the compiler initializes them.. And if the user wants to it could arrange them in the right order.

3.10.9 C.48: Prefer in-class initializers to member initializers in constructors for constant initializers

Reason

> *"Makes it explicit that the same value is expected to be used in all constructors. Avoids repetition. Avoids maintenance problems. It leads to the shortest and most efficient code."*
>
> - C.48 from the C++ Core Guidelines

Problem
Like described in rule C.45 (section 3.10.6 on page 46), don't define a default constructor that only initializes data members; use member initializers instead, it is better to initialize data members in-class. The compiler can generate efficient code to initialize them and it is also for the user better to see what the default values are. This rule also helps to avoid repetition and also to avoid that data members may could be uninitialized. See listing 23 on page 51.

```
class X {   // BAD
    int i;
    string s;
    int j;
public:
    X() :i{666}, s{"qqq"} { }   // j is uninitialized
    X(int ii) :i{ii} {}         // s is "" and j is uninitialized
    // ...
};
```

Listing 23: Problem with data members initialized by constructor[BS15c](Rule C.48)

A class could have multiple constructors and each would need to initialize the data members, leading to lots of typing. Of course there would be a constructor initializing hopefully all data members and all other would somehow call at the end of the delegating chain this constructor. But this would mean all this delegating constructors would use default values.
Now if the data members are in-class initialized it is not needed to initialize them in every constructor. Also it is not needed to default all values if one of its data member should have an other default value. Like described in rule C.45 if there is an in-class initializer of a data member it will be ignored if it would be initialized in a constructor.

Problems concerning the implementation
The guidelines suggest to either check that every constructor initializes every member variable or to check if there are the same default values and then prefer to use in-class initializers. If every constructor initializes every variable it leads to lots of code and it may not be clear what the default values are unless one uses constructor delegating. The second suggest seems more reasonable because it leads to less code and better overview of the default values.
The plug-in could do both but maybe there is some user input needed for the default values. It could detect which values are used most, but it can not know for sure that these are the values needed to be the default values.

3.10.10 C.49: Prefer initialization to assignment in constructors

Reason

> *"An initialization explicitly states that initialization, rather than assignment, is done and can be more elegant and efficient. Prevents "use before*

set" errors."

- C.49 from the C++ Core Guidelines

Problem
It is just bad code practice. The member variable is default initialized and then reassigned. That is unnecessary and unproductive code. Lets have a look at listing 24 on page 52.

```
1 class B {   // BAD
2     string s1;
3 public:
4     B() { s1 = "Hello, "; }   // BAD: default constructor followed by assignment
5     // ...
6 };
```

Listing 24: Surprising conversion[BS15c](Rule C.49)

Problems concerning the implementation
We just have to look out for assignments in constructor bodies and then remove them and initialize the variables in the constructor member initializer list.

3.10.11 C.50: Use a factory function if you need "virtual behavior" during initialization

Reason

"If the state of a base class object must depend on the state of a derived part of the object, we need to use a virtual function (or equivalent) while minimizing the window of opportunity to misuse an imperfectly constructed object."

- C.50 from the C++ Core Guidelines

Problem
This rule is for safety reason. And it is also harder to explain. It is kind of a special case of rule C.42 (section 3.10.3 on page 43), if a constructor cannot construct a valid

object, throw an exception. Because the object with this solution is at the beginning not valid, but will be after a call of a virtual function (listing 26 on page 53). For better understanding have a look at listing 25 on page 53.

```
class B {
public:
    B()
    {
        // ...
        f();   // BAD: virtual call in constructor
        // ...
    }

    virtual void f() = 0;

    // ...
};
```

Listing 25: Bad example of pure virtual function call[BS15c](Rule C.50)

The problem here is if a derived class implements f() it would mean that the Base class B may not be fully initialized if called. That would violates rule C.41 (section 3.10.2 on page 42 a constructor should create a fully initialized object). To avoid this problem a factory function should be used. See listng 26 on page 53.

```
class B {
protected:
    B() { /* ... */ }                   // create an imperfectly initialized object

    virtual void PostInitialize()       // to be called right after construction
    {
        // ...
        f();    // GOOD: virtual dispatch is safe
        // ...
    }

public:
    virtual void f() = 0;

    template<class T>
    static shared_ptr<T> Create()    // interface for creating objects
    {
        auto p = make_shared<T>();
        p->PostInitialize();
        return p;
    }
};

class D : public B { /* "? */ };            // some derived class

shared_ptr<D> p = D::Create<D>();        // creating a D object
```

Listing 26: Factory function to call virtual function[BS15c](Rule C.50)

Note how the default constructor and the pure virtual function which should be called after creation are protected. Only the object itself (and of course its subtypes) are allowed to create it. Even better with the factory function Create() a fully and valid object is provided. With this example there is no chance that the object B could be imperfectly.

Problems concerning the implementation
What the plug-in needs to do is to check if there is a call of a virtual function in the constructor. If so warn the user that this could be a problem. The provision of a quick-fix seems a bit of a problem because we would have to determine the semantics of the program.

3.10.12 C.51: Use delegating constructors to represent common actions for all constructors of a class

Reason

> *"To avoid repetition and accidental differences."*
>
> - C.51 from the C++ Core Guidelines

Problem
This rule handles the case where the default values of the in-class initializers should be ignored. In the rule C.49 (section 3.10.10 on page 51) prefer initialization to assignment in constructors, is explained that before C++11 there was no initializer list in the constructor and that the in-class initializers are automatically ignored, if a constructor initializes this data member. With the initializer list it is also allowed to delegate to other constructors, which was not possible before C++11. Have a look at listing 27 on page 54.

```
1 class Date2 {
2     int d;
3     Month m;
4     int y;
5 public:
6     Date2(int ii, Month mm, year yy)
7         :i{ii}, m{mm} y{yy}
```

```
8         { if (!valid(i, m, y)) throw Bad_date{}; }
9
10    Date2(int ii, Month mm)
11        :Date2{ii, mm, current_year()} {}
12    // ...
13 };
```

Listing 27: Constructor delegation[BS15c](Rule C.51)

The constructor in line 6 initializes all data members and other can delegate to this constructor with special values. In the listing above it was current_year().

Problems concerning the implementation
What the plug-in does it scans the constructors and if there are DRY[Wik16b] violations with data member initializations, it marks it. The quick-fix is the harder problem. The plug-in needs to check what kind of special values are used and the delegate it to the right constructor. It is not sure that this will work. The pattern matching algorithm could possibly fail sometimes.

3.10.13 C.52: Use inheriting constructors to import constructors into a derived class that does not need further explicit initialization

Reason

> *"If you need those constructors for a derived class, re-implementing them is tedious and error prone."*
>
> - C.52 from the C++ Core Guidelines

Problem
In all this new features in C++11 for constructors there is one to inherit constructor from the super-class. By default the constructors of a super class are not inherited. But when explicitly inherited the super class constructor the user has to make sure that the data members of the derived class are initialized after construction. See listing 28 on page 56.

```
class Rec {
    // ... data and lots of nice constructors ...
};

struct Rec2 : public Rec {
    int x;
    using Rec::Rec;
};

Rec2 r {"foo", 7};
int val = r.x;   // uninitialized
```

Listing 28: Problem with implementing constructors of super class[BS15c](Rule C.52)

The base class is in a valid state after construction. But if a Rec2 is constructed first the base classes constructor will be called and then finish. There is no default constructor because there are others defined and even if there would be one the variable x would not be initialized. There is rule C.45 (section 3.10.10 on page 51) don't define a default constructor that only initializes data members; use member initializers instead. But this rule is for default values respectively default initialization. If x should have a special value after construction with this method it is not possible. The user would need to define constructors in the derived class and then call the super class constructor in the initializer list.

Problems concerning the implementation
What the plug-in needs to do is to check if super constructors are inherited. If so check if there are data members. If there are data members then warn that it would be better to define constructors calling the super constructors. If there are no data members everything seems okay. But there is no chance to provide a quick-fix because the plug-in can not know which of the derived class constructor should call which super class constructor.

3.11 Copy and move rules C.60 to C.67

These are the rules about copy and move assignments.

3.11.1 C.60: Make copy assignment non-virtual, take the parameter by const&, and return by non-const&

Reason

> *"It is simple and efficient. If you want to optimize for rvalues, provide an overload that takes a && (see F.24[BS15a][F.24])."*
>
> - C.60 from the C++ Core Guidelines

Problem

This rule is about value and reference semantics[ISO14b]. It also mentions the slicing problem like also described in rule C.50 (section 3.10.11 on page 52) use a factory function if you need "virtual behavior" during initialization.
Value and reference semantics can easily be explained on a function. The parameters of the function can be passed by reference or by value. If they are passed by reference this simply means they are passed by pointer or (as the name says) by reference. Other if they are passed by value. In that case they would be copied and then passed in. This can be a problem if the object is huge like a vector with 1000 of arrays. Then the whole vector with its values has to be copied and that could take a while. Passed in by reference would simply copy the pointer or reference and that is it. So it is better if the parameters are passed by reference. But in some cases it makes sense to pass it as value.
Now lets have a look at listing 29 on page 57.

```
1  class Foo {
2  public:
3      Foo& operator=(const Foo& x)
4      {
5          auto tmp = x;   // GOOD: no need to check for self-assignment (other than
       performance)
6          std::swap(*this, tmp);
7          return *this;
8      }
9      // ...
10 };
11
12 Foo a;
13 Foo b;
14 Foo f();
15
16 a = b;    // assign lvalue: copy
```

```
17 a = f();  // assign rvalue: potentially move
```

Listing 29: Copy assignment[BS15c](Rule C.60)

The parameter should be passed as const T& and the return should be non-const T&. Const T& means that this is a pointer to a const type. Which is good because an assignment operator should not change an object other than this. The return is of course a non-const T& because the type should most often be changeable.
In the code listing swap is used on a temporary copied object of the passed argument and this. This is now a perfect copy of the passed in object.

Problems concerning the implementation
What the plug-in needs to check is simply check if the type are passed in are correctly and that the function is not virtual.

3.11.2 C.61: A copy operation should copy

Reason

> *"That is the generally assumed semantics. After x=y, we should have x == y. After a copy x and y can be independent objects (value semantics, the way non-pointer built-in types and the standard-library types work) or refer to a shared object (pointer semantics, the way pointers work)."*
>
> - C.61 from the C++ Core Guidelines

Problem
Rule C.61 is kind of common sense. The basic idea is that when an object is copied both should be equal. Also should the two objects not depend on a common resource because with C++ usually value semantics instead of reference semantics is used. Like the value and reference semantics, described in rule C.60 (section 3.11.1 on page 57), meaning that the object not should be copied with reference semantics. Instead the value semantics should be used because they are the simplest to reason about.
For the average C++ programmer this is common sense and they would not implement their copy assignments / copy constructors wrong, but for an inexperienced programmer this can happen quite quickly (See in listing 30 on page 59 how copy

should work).

```
class X {   // OK: value semantics
public:
    X();
    X(const X&);      // copy X
    void modify();    // change the value of X
    // ...
    ~X() { delete[] p; }
private:
    T* p;
    int sz;
};

bool operator==(const X& a, const X& b)
{
    return a.sz == b.sz && equal(a.p, a.p + a.sz, b.p, b.p + b.sz);
}

X::X(const X& a)
    :p{new T[a.sz]}, sz{a.sz}
{
    copy(a.p, a.p + sz, a.p);
}

X x;
X y = x;
if (x != y) throw Bad{};
x.modify();
if (x == y) throw Bad{};   // assume value semantics
```

Listing 30: Copy operation[BS15c](Rule C.61)

Problems concerning the implementation
Of course in production code this case can easily be checked with the code shown above. But the plug-in does not execute C++ code. It has to analyze the given code and the only way to check if the copy operations are correct, is by analyzing the semantics inside the copy operations. The problem is that the gain for the time invested in this is not reasonable. Also the guideline itself says that this rule is "not enforceable" which is true to some extend. This is why this rule will be ignored by the plug-in.

3.11.3 C.62: Make copy assignment safe for self-assignment

Reason

> *"If you need those constructors for a derived class, re-implementing them is tedious and error prone."If x=x changes the value of x, people will be surprised and bad errors will occur (often including leaks).*
>
> - C.62 from the C++ Core Guidelines

Problem

Self-assignment basically means when you set x=x. This is a very bad code practice. But mostly this will not happen. And if it happens it is not that obvious. It will happen over multiple dereferencing where the user can not easily see that it is actually the same object. Static analysis tools like CODAN[Ecl15] will warn if you ever try to perform a self-assignment but for some reason it is accepted by the compiler. Still some programmer may implement their copy operations to check if a self-assignment is being performed with the idea to speed up the copy. This is okay but since self-assignment happens so rarely it is better to skip this check and it will result in better performance (see listing 31 on page 60).

```
1 Foo& Foo::operator=(const Foo& a)   // simpler, and probably much better
2 {
3     s = a.s;
4     i = a.i;
5     return *this;
6 }
```

Listing 31: No self-assignment check[BS15c](Rule C.62)

What also could be done is using swap to assign the object (see listing 30 on page 59).
It would be much better if the copy swap idiom was used because it would also be exception safe. Even better if the swap function is declared as noexcept the STL[Wik16f] could do some optimizations. What is the copy and swap idiom[Wik15]?. The basic idea is to make the assignment operator elegant and avoids duplicate code as well as exceptions. Here is an example of coding it the bad way (listing 32 on page 60).

```
1 // the hard part
2 dumb_array& operator=(const dumb_array& other)
3 {
4     if (this != &other)
5     {
6         // get rid of the old data...
```

```
7         delete [] mArray;
8         mArray = 0;
9
10        // ...and put in the new
11        mSize = other.mSize;
12        mArray = mSize ? new int[mSize] : 0;
13        std::copy(other.mArray, other.mArray + mSize, mArray);
14    }
15
16    return *this;
17}
```

Listing 32: Bad assign operator[Sta16a]

At line 4 there is a check for self assignment. This is to provide a basic exception guarantee but if the array would have been modified and the allocation, the new int[mSize] at line 12 would throw the data is gone. It would be easier if swap had been used (listing 33 on page 61).

```
1 void std::swap(T& lhs,T& rhs)
2 {
3     T tmp(lhs);
4     lhs = rhs;
5     rhs = tmp;
6 }
```

Listing 33: STL swap implementation[**?**]

Note There is also an example in rule C.60 (listing 29 on page 57).

This is much less code and does exactly the same. Now why should the programmer provide a swap function? The answer is because of efficiency. As seen in the listing above the STL makes a temporary copy. This is a cost and that is the reason why the programer should provide an own swap function. It mostly can be done better by doing so (see listing 34 on page 61).

```
1  class Foo {
2      // ...
3  public:
4      void swap(Foo& rhs) noexcept
5      {
6          m1.swap(rhs.m1);
7          std::swap(m2, rhs.m2);
8      }
9  private:
10     Bar m1;
11     int m2;
12 };
```

```
13
14 void swap(Foo& a, Foo& b)
15 {
16     a.swap(b);
17 }
```

Listing 34: Virtual functions calls[BS15c](Rule C.83)

A non member swap function should be declared for callers' convenience.
The guidelines say that a class without virtual functions should provide a swap member function and it should also be noexcept. So the plug-in has to mark once again those violations and provide a useful swap function.

Problems concerning the implementation
The plug-in needs to check if there is a if (this == &a) return *this; check. If so mark the statement and if the user wants to remove it. Because it will not be efficient if there are lots of checks of self-assignment if most of the time this will not be the case. The plug-in could also replace all the member assignments and replace it with the swap copy like in listing 30 on page 59.

3.11.4 C.63: Make move assignment non-virtual, take the parameter by &&, and return by non-const&

Reason

> *"It is simple and efficient."*
>
> - C.63 from the C++ Core Guidelines

Problem
This is the same rule as C.60 (section 3.11.1 on page 57) only that it takes the parameter by rvalue[IEC14b]. Take by parameter T&& and return by non-const T&. This is reference semantics (described in rule C.60). Non-virtual because of the slicing problem (also described in rule C.60).

3.11.5 C.64: A move operation should move and leave its source in a valid state

Reason

> *"That is the generally assumed semantics. After x=std::move(y) the value of x should be the value y had and y should be in a valid state."*
>
> - C.64 from the C++ Core Guidelines

Problem

This rule is about the correctness of move operations. The source of a std::move should be in a valid state after the operation. A valid state could be interpreted as the default state (variable with default values set), but also some state that can not be interpreted by a simple plug-in due to its semantical context. An enforcement of the guidelines is to compare the assignments to the source inside the move operation to the ones in the default constructor. But according to the guidelines there should always be a default constructor. Wether it is deleted or defaulted. And they also say that data members should be in class initialized so if a class has a defined default constructor it does not need to only initialize the data members. Lets see how this could look like in listing 35 on page 63.

```
1  template<typename T>
2  class X {   // OK: value semantics
3  public:
4      X();
5      X(X&& a);           // move X
6      void modify();      // change the value of X
7      // ...
8      ~X() { delete[] p; }
9  private:
10     T* p;
11     int sz;
12 };
13
14
15 X::X(X&& a)
16     :p{a.p}, sz{a.sz}  // steal representation
17 {
18     a.p = nullptr;      // set to "empty"
19     a.sz = 0;
20 }
21
22 void use()
23 {
```

```
24      X x{};
25      // ...
26      X y = std::move(x);
27      x = X{};    // OK
28  } // OK: x can be destroyed
```

Listing 35: Move constructor[BS15c](Rule C.64)

Here the data members are not in class default initialized, but it would make sense that if such an X class would be default created that this are its default values. After the move operation the source object where the data was moved from is then in a valid state. There it could be assigned with a new X object.

Problems concerning the implementation
So what the plug-in needs to do is to compare the assignments in the move operators to the default in class initializers. First the assignments or initializations of the data members and then the default value assignments from the moved from object. if that is done correctly it seems to be a good move operation. Pointers will always be set to nullptr. But if handled in a move assignment first the this->ptr needs to be deleted and then reset to the other->ptr. What we could also consider doing is that if the source contains a pointer to dynamic memory, we could check if it is set to nullptr after a successful move. This would help ensuring that there are not two pointers from different objects pointing to a shared resource. We may also consider a non-strict implementation. It only checks if for example 50% of the assignments to the source match with the default constructor and warn if it does not. A quick-fix does not make much sense here.

3.11.6 C.65: Make move assignment safe for self-assignment

Reason

> *"If x = x changes the value of x, people will be surprised and bad errors may occur. However, people do not usually directly write a self-assignment that turn into a move, but it can occur. However, std::swap is implemented using move operations so if you accidentally do swap(a, b) where a and b refer to the same object, failing to handle self-move could be a serious and subtle error."*

- C.65 from the C++ Core Guidelines

Problem
Like with rule C.62 (section 3.11.3 on page 59) there is always the change of a self-assignment happening. And since we are analyzing the code inside a class we have to determine how our class would react in case of a self-assignment. A check if the source variable is the same as our current object is also redundant since one in a million checks do not fit the ideology of C++ laying its focus on performance. It would better be done by the copy swap idiom described in rule C.62 section.

Problems concerning the implementation
What we can implement to ensure correct dealing with self-assignments though is to look for problematic patterns inside the move assignment and warn if one was detected.

3.11.7 C.66: Make move operations noexcept

Reason

> *"A throwing move violates most people's reasonably assumptions. A non-throwing move will be used more efficiently by standard-library and language facilities."*

- C.66 from the C++ Core Guidelines

Problem
In a normal move operation there should not be a new memory allocation (see listing 36 on page 65).

```
1 template<typename T>
2 class Vector {
3     // ...
4     Vector(Vector&& a) noexcept :elem{a.elem}, sz{a.sz} { a.sz = 0; a.elem = nullptr
      ; }
5     Vector& operator=(Vector&& a) noexcept { elem = a.elem; sz = a.sz; a.sz = 0; a.
      elem = nullptr; }
6     // ...
```

```
7 public:
8     T* elem;
9     int sz;
10 };
```

Listing 36: Move operations[BS15c](Rule C.66)

The move operations just steal the values from an object and uses them for itself. But the other object needs to be set to a valid state.
Since the rule C.64 (section 3.11.5 on page 63) takes care of using the move operator the right way the plug-in now needs to make sure it is marked as noexcept.

3.11.8 C.67: A base class should suppress copying, and provide a virtual clone instead if "copying" is desired

Reason

> *"To prevent slicing, because the normal copy operations will copy only the base portion of a derived object."*
>
> - C.67 from the C++ Core Guidelines

Problem

This is again a slicing problem (described in rule C.50 section 3.10.11 on page 52). See listing 37 on page 66.

```
1  class B { // GOOD: base class suppresses copying
2      B(const B&) =delete;
3      B& operator=(const B&) =delete;
4      virtual unique_ptr<B> clone() { return /* B object */; }
5      // ...
6  };
7
8  class D : public B {
9      string moredata; // add a data member
10     unique_ptr<B> clone() override { return /* D object */; }
11     // ...
12 };
13
14 auto d = make_unique<D>();
15 auto b = d.clone(); // ok, deep clone
```

Listing 37: Delete copy operations; provide virtual clone()[BS15c](Rule C.67)

If the object in the last line would be created with make_unique<B>(d) more data would be lost. But with the virtual function of clone() it now is possible to create class B in a valid way.

Problems concerning the implementation
The guidelines suggest "A class with any virtual function should not have a copy constructor or copy assignment operator (compiler-generated or handwritten)."

3.12 Other default operations rules C.80 to C.87

These are the rules about other default operations.

3.12.1 C.80: Use =default if you have to be explicit about using the default semantics

Reason

> *"The compiler is more likely to get the default semantics right and you cannot implement these function better than the compiler."*
>
> - C.80 from the C++ Core Guidelines

Problem
The focus of this rule is to not rewrite your own default operations when then the compiler can do it the same or even better and more efficient. Doing so is quite redundant and also error-prone. See listing 38 on page 67.

```
1  class Tracer2 {
2      string message;
3  public:
4      Tracer2(const string& m) : message{m} { cerr << "entering " << message << '\n';
       }
5      ~Tracer2() { cerr << "exiting " << message << '\n'; }
6
7      Tracer2(const Tracer2& a) : message{a.message} {}
8      Tracer2& operator=(const Tracer2& a) { message = a.message; }
9      Tracer2(Tracer2&& a) :message{a.message} {}
10     Tracer2& operator=(Tracer2&& a) { message = a.message; }
```

```
11 };
```

Listing 38: Redundant code[BS15c](Rule C.80)

In this case the other default operations from line 7 upwards are redundant. The compiler would generate them exactly the same or even better and more efficient. Less code less error can be done and if it is set =default it will be more efficient. See listing 39 on page 68

```
1  class Tracer {
2      string message;
3  public:
4      Tracer(const string& m) : message{m} { cerr << "entering " << message << '\n'; }
5      ~Tracer() { cerr << "exiting " << message << '\n'; }
6
7      Tracer(const Tracer&) = default;
8      Tracer& operator=(const Tracer&) = default;
9      Tracer(Tracer&&) = default;
10     Tracer& operator=(Tracer&&) = default;
11 };
```

Listing 39: Redundant code[BS15c](Rule C.80)

Problems concerning the implementation
The plug-in now can mark those functions and could quick-fix them to =default and remove the body. But if there are other statements in the body than just the assignments it will ignore the violation.

3.12.2 C.81: Use =delete when you want to disable default behavior (without wanting an alternative)

Reason

> *"In a few cases, a default operation is not desirable."*
>
> - C.81 from the C++ Core Guidelines

Problem
This rule will be ignored by the plug-in. There is no chance that it can analyze whether one or more default operations should be deleted. There is always a good reason to do so. So if it is needed delete and there are no other guideline violations the user is free to delete the (or these) default operation(s).

3.12.3 C.82: Don't call virtual functions in constructors and destructors

Reason

> *"The function called will be that of the object constructed so far, rather than a possibly overriding function in a derived class. This can be most confusing. Worse, a direct or indirect call to an unimplemented pure virtual function from a constructor or destructor results in undefined behavior."*
>
> - C.82 from the C++ Core Guidelines

Problem

This is another rule where the plug-in can not help. See listing 40 on page 69

```
1  class base {
2  public:
3      virtual void f() = 0;   // not implemented
4      virtual void g();       // implemented with base version
5      virtual void h();       // implemented with base version
6  };
7
8  class derived : public base {
9  public:
10     void g() override;      // provide derived implementation
11     void h() final;         // provide derived implementation
12
13     derived()
14     {
15         f();                // BAD: attempt to call an unimplemented virtual
       function
16
17         g();                // BAD: will call derived::g, not dispatch further
       virtually
18         derived::g();       // GOOD: explicitly state intent to call only the
       visible version
19
20         h();                // ok, no qualification needed, h is final
21     }
22 };
```

Listing 40: Virtual functions calls[BS15c](Rule C.82)

This is kind of the same problem like C.50 (section 3.10.11 on page 52). If virtual functions are needed the user may consider to use a factory function like in rule C.50. Since the rule is the same as rule C.50 the plug-in will handle it like so.

3.12.4 C.83: For value-like types, consider providing a noexcept swap function

Reason

> *"A swap can be handy for implementing a number of idioms, from smoothly moving objects around to implementing assignment easily to providing a guaranteed commit function that enables strongly error-safe calling code. Consider using swap to implement copy assignment in terms of copy construction. See also destructors, deallocation, and swap must never fail[BS15a][E.16]."*
>
> - C.83 from the C++ Core Guidelines

Problem
It is not really a problem it just helps to provide exception safety and should not be that big of a problem if the STL is used.

Problems concerning the implementation
The guidelines suggest "A class without virtual functions should have a swap member function declared. When a class has a swap member function, it should be declared noexcept."

3.12.5 C.84: A swap may not fail

Reason

> *"swap is widely used in ways that are assumed never to fail and programs cannot easily be written to work correctly in the presence of a failing swap. The standard-library containers and algorithms will not work correctly if a swap of an element type fails."*
>
> - C.84 from the C++ Core Guidelines

Problem
Rule C.83 (section 3.12.4 on page 70) takes care of writing a good swap function which does not throw. With using the normal swap and copy idiom this should be the case (described in rule C.83).

3.12.6 C.85: Make swap noexcept

Reason

> *"A swap may not fail. If a swap tries to exit with an exception, it's a bad design error and the program had better terminate."*
>
> - C.85 from the C++ Core Guidelines

Problem
In rule C.36 there is described how the noexcept declaration is handled. So that is the reason why a swap function should be noexcept.

3.12.7 C.86: Make == symmetric with respect of operand types and noexcept

Reason

> *"Asymmetric treatment of operands is surprising and a source of errors where conversions are possible. == is a fundamental operations and programmers should be able to use it without fear of failure."*
>
> - C.86 from the C++ Core Guidelines

Problem
Here we have to take care on how to implement the comparison operators . Lets look at some code examples (listing 41 on page 71).

```
1 class B {
2     string name;
3     int number;
```

```
4     bool operator==(const B& a) const {
5         return name == a.name && number == a.number;
6     }
7     // ...
8 };
```

Listing 41: Bad Example[BS15c](Rule C.86)

The operator==() as member function can lead to some conversion problems. For example if there is a statement like b1 + b2 the compiler would translate it to b1.operator+(b2);. That means, the overloaded operator+ member function gets called on the first operand. And this is totally okay. Unless one of the types is not of type B. Lets say b1 is an int and b2 of type B. Now the compiler tries to call operator+() on the first operand which is int. But this would fail because ints can add only ints. This also means that the operator overload as member functions are asymmetric. So how should it be done? Lets have a look at a good example (listing 42 on page 72).

```
1 class X {
2     string name;
3     int number;
4 };
5
6 bool operator==(const X& a, const X& b) noexcept {
7     return a.name == b.name && a.number == b.number;
8 }
```

Listing 42: Good non-member operator function[BS15c](Rule C.86)

Here the compiler does not call a member function. Instead the ::operator==(x1, x2); is called. And this is type safe and also symmetrical.

Problems concerning the implementation
The guideline says to flag member comparators as well as comparators where the parameter types differ. What the plug-in can do is to check these requirements and if they are violated it marks it as violation and offers a quick-fix to create non member comparators with the same types and also declare them noexcept. It make really no sense that such an call could throw.

3.12.8 C.87: Beware of == on base classes

Reason

"It is really hard to write a foolproof and useful == for a hierarchy."

- C.87 from the C++ Core Guidelines

Problem

To explain this rule it is better to have a look at some code (listing 43 on page 73).

```
1  class B {
2      string name;
3      int number;
4      virtual bool operator==(const B& a) const
5      {
6           return name == a.name && number == a.number;
7      }
8      // ...
9  };
10
11 class D :B {
12     char character;
13     virtual bool operator==(const D& a) const
14     {
15         return name == a.name && number == a.number && character == a.character;
16     }
17     // ...
18 };
19
20 B b = ...
21 D d = ...
22 b == d;    // compares name and number, ignores d's character
23 d == b;    // error: no == defined
24 D d2;
25 d == d2;   // compares name, number, and character
26 B& b2 = d2;
27 b2 == d;   // compares name and number, ignores d2's and d's character
```

Listing 43: Virtual member comparator operators[BS15c](Rule C.87)

Lets ignore that comparator operators should not be member functions. If the comparators are implemented like shown above this is a problem. In line 22 the operator on b is called because its the best match for the compiler. D is type of B (polymorphism) and B is type of B and so this function is called.

Problems concerning the implementation

With rule C.86 (section 3.12.7 on page 71) it is made sure that the comparators are non member functions and so this is kind of the same rule. It is better that the

comparators are always overloaded for each type to make sure that it compares the right way.

3.12.9 C.89: Make a hash noexcept

Reason

> *"Users of hashed containers use hash indirectly and don't expect simple access to throw. It's a standard-library requirement."*
>
> - C.89 from the C++ Core Guidelines

Problem
Lets have a look at listing 44 on page 74.

```
1  template<>
2  struct hash<My_type> {  // thoroughly bad hash specialization
3     using result_type = size_t;
4     using argument_type = My_type;
5
6     size_t operator() (const My_type & x) const
7     {
8        size_t xs = x.s.size();
9        if (xs < 4) throw Bad_My_type{};    // "Nobody expects the Spanish inquisition
      !"
10       return hash<size_t>()(x.s.size()) ^ trim(x.s);
11    }
12 };
13
14 int main()
15 {
16    unordered_map<My_type,int> m;
17    My_type mt{ "asdfg" };
18    m[mt] = 7;
19    cout << m[My_type{ "asdfg" }] << '\n';
20 }
```

Listing 44: Specialized hash functor[BS15c](Rule C.89)

The code on line nine is very bad. A developer does not expect an exception if a hash function is called. But here it is also hard to detect a user defined hash function.

Problems concerning the implementation
The guidelines suggest "Flag throwing hashes".

3.12.10 Attributes

There are cases where a checker can be ignored. And for that our supervisor prof. Sommerlad suggested to "define" attributes[Str14][7.6 Attributes] in order to ignore a specific checker. Attributes are specified in the C++ standard but they are mostly syntax free. In listing 45 on page 75 is an example of an attribute.

```
1 struct A {
2   [[ccglator::ignore("C.XX")]]
3   A();
4 };
```

Listing 45: Example of an attribute for a constructor

The compiler ignores most of the attributes as long they are not defined in one of the standard compilers. But it does warn that an unknown attribute was ignored. We decided to use this syntax for the attribute so it looks like a function call with the parameter as a string containing the rule number that can be ignored.

3.12.11 Better overview of the Rules and its difficulty

This section gives an overview of all the rules and and our thought about its difficulty. We differ between the checker (the analysis of the code to find the violated rule) and if possible the provision of a quick-fix.

1 Easy
10 Hard
- Not possible
() Implementation partially possible

Rules	**Checker difficulty**	**Quick-fix difficulty**
C.20: If you can avoid defining default operations, do	9	1
C.21: If you define or =delete any default operation, define or =delete them all	4	6
C.22: Make default operations consistent	9	9
C.30: Define a destructor if a class needs an explicit action at object destruction	8	4
C.31: All resources acquired by a class must be released by the class's destructor	5	7
C.32: If a class has a raw pointer (T*) or reference (T&), consider whether it might be owning	-	-
C.33: If a class has an owning pointer member, define or =delete a destructor	2	6
C.34: If a class has an owning reference member, define or =delete a destructor	2	6
C.35: A base class with a virtual function needs a virtual destructor	3	7
C.36: A destructor may not fail	3	1
C.37: Make destructors noexcept	3	1
C.40: Define a constructor if a class has an invariant	4	3
C.41: A constructor should create a fully initialized object	6	-
C.42: If a constructor cannot construct a valid object, throw an exception	7	5
C.43: Ensure that a class has a default constructor	3	2
C.44: Prefer default constructors to be simple and non-throwing	6	-
C.45: Don't define a default constructor that only initializes data members; use member initializers instead	6	5

Continuation of table 6 on the next page

Table 6 – *Continuation of table 6*

Rules	**Checker difficulty**	**Quick-fix difficulty**
C.46: By default, declare single-argument constructors explicit	4	2
C.47: Define and initialize member variables in the order of member declaration	4	7
C.48: Prefer in-class initializers to member initializers in constructors for constant initializers	4	6
C.49: Prefer initialization to assignment in constructors	4	7
C.50: Use a factory function if you need "virtual behavior" during initialization	8	7
C.51: Use delegating constructors to represent common actions for all constructors of a class	8	6
C.52: Use inheriting constructors to import constructors into a derived class that does not need further explicit initialization	7	-
C.60: Make copy assignment non-virtual, take the parameter by const&, and return by non-const&	5	4
C.61: A copy operation should copy	9	9
C.62: Make copy assignment safe for self-assignment	8	-
C.63: Make move assignment non-virtual, take the parameter by &&, and return by non-const&	5	4
C.64: A move operation should move and leave its source in a valid state	-	-
C.65: Make move assignment safe for self-assignment	8	-
C.66: Make move operations noexcept	4	1
C.67: A base class should suppress copying, and provide a virtual clone instead if "copying" is desired	7	9

Continuation of table 6 on the next page

Table 6 – *Continuation of table 6*

Rules	**Checker difficulty**	**Quick-fix difficulty**
C.80: Use =default if you have to be explicit about using the default semantics	8	1
C.81: Use =delete when you want to disable default behavior (without wanting an alternative)	-	1
C.82: Don't call virtual functions in constructors and destructors	3	-
C.83: For value-like types, consider providing a noexcept swap function	5	4
C.84: A swap may not fail	2	1
C.85: Make swap noexcept	2	1
C.86: Make == symmetric with respect of operand types and noexcept	4	(2)
C.87: Beware of == on base classes	3	-
C.89: Make a hash noexcept	5	1

Table 6: Overview of the rules C.20 to C.89

4 CCGLator Implementation

In this section we are not going to explain how an Eclipse plug-in is implemented and which extension points we are working with. Most of this was already discussed in our term project AliExtor [KO15a]. Of course with this bachelor thesis, we operated on different extension points as with our term thesis. Those will be discussed as an addition when we talk about the plugin.xml. Also we are not going to mention the refactoring lifecycle when working with the light-weight static-analysis framework Codan since this has also been covered very well in the bachelor thesis of Toni Suter and Fabian Gonzalez [GF14].
Since our bachelor thesis is mainly about the C++ Core Guidelines, in this section we are going to discuss how we decided to implement each rule of the C++ Core Guidelines. Because some rules have strong correlation with others, we are not going to list each rule entry, but rather list our implementation parts and explain to which

rules they apply.

4.1 Declaration vs. Definition

Many rule implementations happened to be tricky because we always had to keep in mind that special member functions can be declared or both declared and defined. It is also possible for the declaration to be inside a class/struct scope or outside (usually in a separate .cpp file).

```
1  //Named_map.h (Declaration)
2  #include "gsl.h"
3  struct Named_map {
4  public:
5    ~Named_map();
6
7  private:
8    gsl::owner<int *> owy;
9  };
10
11 //Named_map.cpp (Definition)
12 #include "Named_map.h"
13 Named_map::~Named_map() {
14   delete owy;
15 }
```

Listing 46: Destructor needed when class has owners

Many of the C++ Core Guideline entries require analysis on both declaration as well as definition. Sometimes though we only have to do the analysis in our definition and it may even be the case that the there is no separate declaration existent. The same applies for quick-fixes. Changes to the source code can be done in a definition or on both definition and declaration. Also it is worth mentioning, that definition and declaration are usually spread across different files (header and cpp) in most cases. So our quick-fixes have to account that as well.
All in all we end up with a situation where each rule has to be able to handle all of the explained cases. Therefore we chose the approach to preferably set our checkers on top of declarations to have a common initial state in our quick-fixes.

4.2 CPPASTFunctionDefinition vs. CPPASTSimpleDeclaration

When working with the corresponding AST nodes of a declaration and a definition we quickly realized that the implementation will become tricky.
The declaration of a function is represented as a CPPASTSimpleDeclaration (listing 47 on page 80).

```
1 struct Auto {
2 public:
3   ~Auto() noexcept;
4 };
```

Listing 47: Destructor declaration

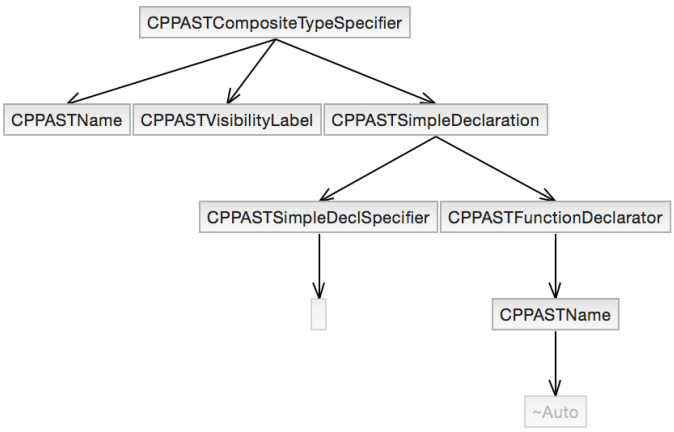

Figure 7: AST representation of a destructor declaration

On the other hand the definition looks as follows.

```
1 #include "auto.h"
2 Auto::~Auto() noexcept = default;
```

Listing 48: Destructor definition

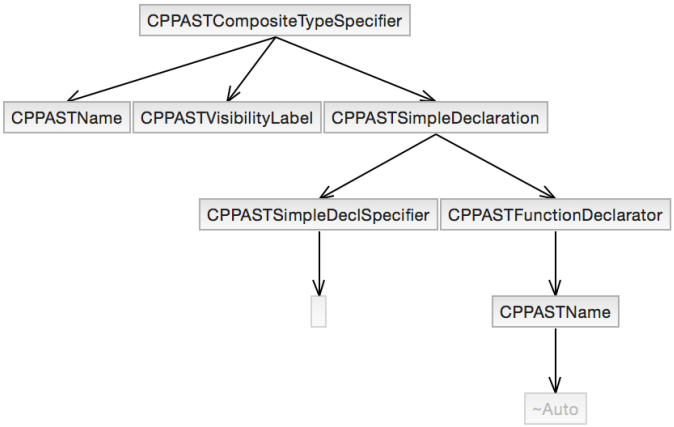

Figure 8: AST representation of a destructor definition

Our goal was to develop a common logic which can be used by both definitions and declarations. Those common logic include methods like getSpecialMemberFunction-Type() which should determine wether a function is a special member function and if so, even determine which type of special member function it is without depending on if we're working with a definition or a declaration. Having a set of common logic methods is extremely helpful the more rules we implement. Also it allows for a better maintainability in case our plugin is further developed afterwards.
Sadly it turned out that developing a common logic for both definition as well as declarations is harder than expected. Reason for that is the root of both CPPAST-FunctionDefinition as well as CPPASTSimpleDeclaration. Let's have a closer look at the root of each node type:

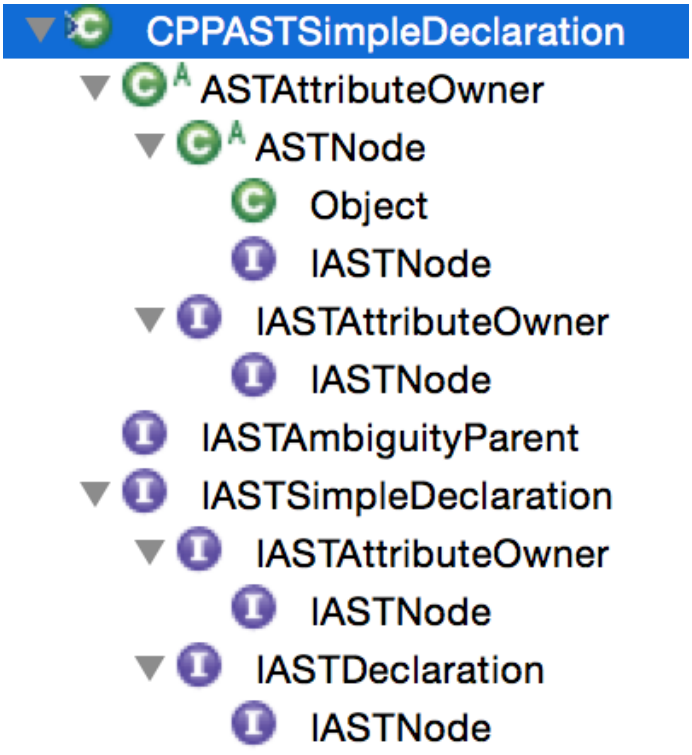

Figure 9: Type hierarchy of CPPASTSimpleDeclaration

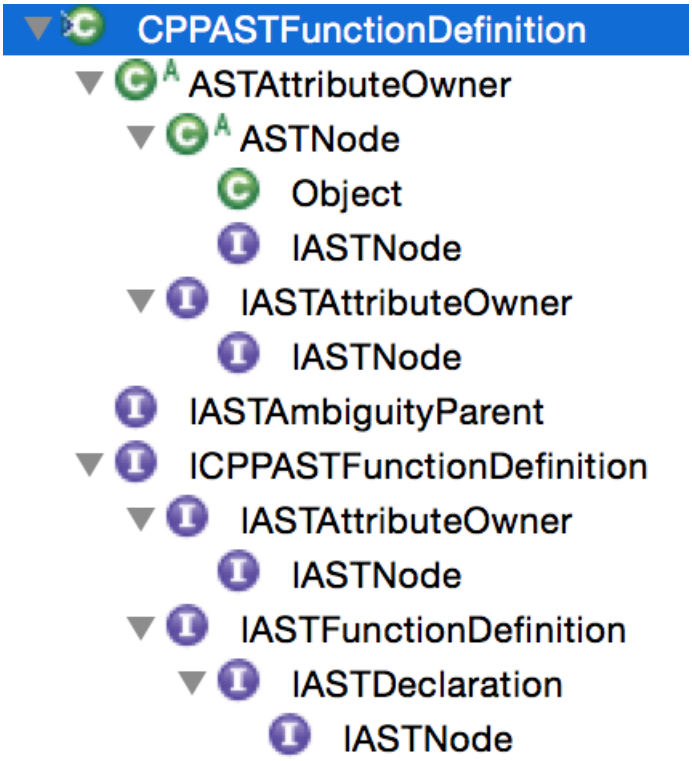

Figure 10: Type hierarchy of CPPASTFunctionDefinition

So the only common class we have from the type hierarchy is ASTNode and the common interfaces are IASTNode, IASTDeclaration and IASTAmbiguityParent. Sadly there is no common class or interface that can be used to implement a common logic for our use cases. For example, is there no way to get the declarator from each node using a common interface. We decided to build our helper methods to work with IASTDeclarations, forceing us to do lots of instanceof checks inside our core methods.
A good illustration of this problem is the ICPPASTFunctionDeclarator containing a lot of useful information, which have to be processed in many checkers as well as quick-fixes. Let's have a look at a sample code of a move assignment and the corresponding AST.

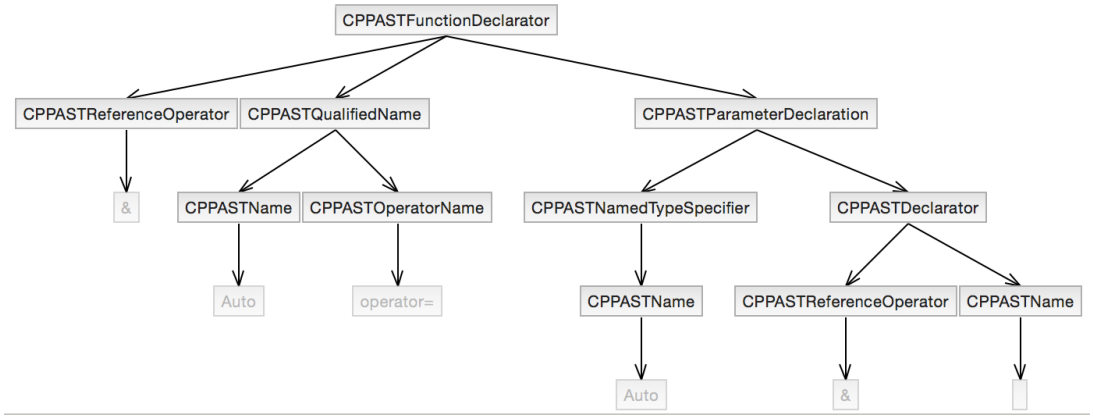

Figure 11: AST of a move assignment

```
1 Auto& Auto::operator=(Auto &&) = default;
```

Listing 49: Move assignment

To check if this is actually a move assignment, we have to verify various areas of the AST. But one thing is for sure. We need the ICPPASTFunctionDeclarator. But the IASTDeclaration interface does not offer any method to get the ICPPASTFunction-Declarator. Here is where our getFunctionDeclaratorFromDeclarationOrDefinition() method came in handy.

This of course does not fully solve the issue of having to deal with both CPPAST-FunctionDefinitions as well as CPPASTSimpleDeclarations. Of course many of our quick-fixes also have to do different things depending on if we have a declaration or definition marked. Here is a small example for better illustration:

```
1  ICPPASTFunctionDefinition newdestructor = null;
2
3  if (destructor instanceof IASTSimpleDeclaration) {
4    newdestructor = ASTHelper.getImplFromDeclarationFromQuickFix((
       IASTSimpleDeclaration) destructor, rewriteStore).copy();
5  }
6
7  if (destructor instanceof ICPPASTFunctionDefinition) {
8    newdestructor = (ICPPASTFunctionDefinition) destructor.copy();
9  }
10
11 newdestructor.setIsDefaulted(false);
12 newdestructor.setIsDeleted(false);
13 newdestructor.setBody(factory.newCompoundStatement());
14 newdestructor = addMissingDeleteStatements(newdestructor, ownerInformation);
15
```

```
16 if (destructor instanceof IASTSimpleDeclaration) {
17   final ICPPASTFunctionDefinition funcDef = ASTHelper.
       getImplFromDeclarationFromQuickFix((IASTSimpleDeclaration) destructor,
       rewriteStore);
18   final ASTRewrite implRewrite = rewriteStore.getASTRewrite(getOriginatingTUnit(
       funcDef));
19   implRewrite.replace(funcDef, newdestructor, null);
20 } else if (destructor instanceof ICPPASTFunctionDefinition) {
21   rewrite.replace(destructor, newdestructor, null);
22 }
```

Listing 50: Destructor needs a body quick-fix (C.3102)

With this code sample we basically want to show that the separation of both definition and declaration are not able to be centrally handled with a common shared logic. It is something that has to be accounted for in various parts of each rule implementation. Though we tried to create various helper methods like getFunctionDeclaratorFromDeclarationOrDefinition() allowing for an easier implementation.

4.3 Scoping

Declarations are usually stored in a header file whereas definitions usually are in a .cpp file. But this is not a must. For many core methods like determining if a function declaration or a simple definition is, for example, a move constructor, we need to determine the name of our class. This can be done quite easily when we are inside a class scope since we'd only have to look for a so called ICPPASTCompoundStatement and get the name of it.

```
1 final ICPPASTCompositeTypeSpecifier struct = ASTHelper.getCompositeTypeSpecifier(
      decl);
2 if (struct != null) {
3   return struct.getName();
4 }
```

Listing 51: Determine class name

When outside a class scope though, we have to find the class name inside the function declarator. This is just another place where separation of working with CPPASTSimpleDeclarations and CPPASTFunctionDefinitions has to be done.

4.4 Naming

We have decided to preface each file with it corresponding rule name.

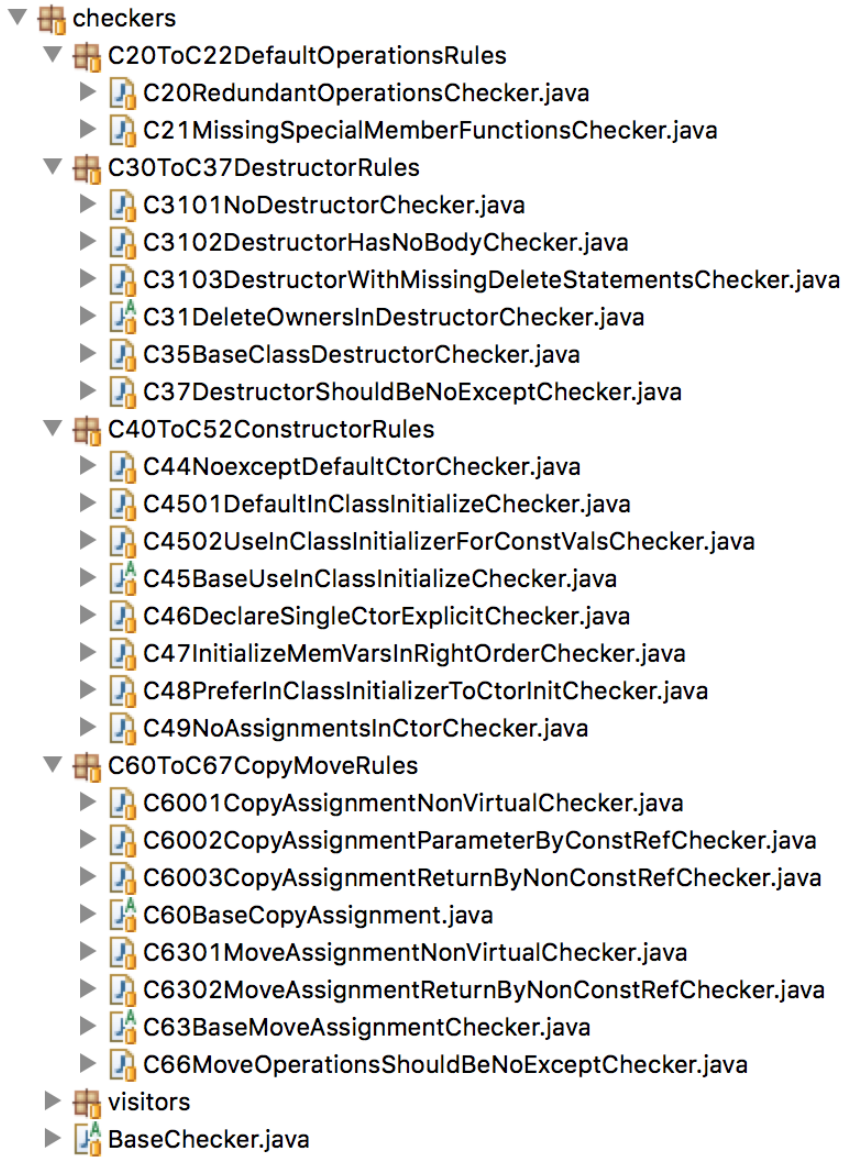

Figure 12: Naming example

This approach is especially benefiting since we have our code spread across several

packages, which are listed in the following section. It also is separated in the different types of checker rules so it is easier to find a rule.

4.5 Checker placement approach

Checkers are preferably placed inside a class/struct type. We chose to follow this approach to have a common process of quick-fixing. Quick-fixes that perform changes on both declaration as well as definition will all have the same approach of analyzing and modifying the declaration at first. This leads to consistency and makes synergizing quick-fixes that only perform changes on 1 node and quick-fixes that modify multiple nodes easier.

4.6 Node forwarding to Quick-Fix

Due to the limitations of the Eclipse CDT, a checker can not extend a line of code. For most rules this limitation was not a problem. But in some cases, we had to pass on a CPPASTFunctionDefinition to our quick-fix. This made especially sense in many cases, due to our quick-fix needing access to various parts of the function definition (ex. declarator, decl specifier, body,...). We solved this limitation with a bit of a poorly performing approach.
For quick-fixes where we needed the CPPASTFunctionDefinition, we got the CPPASTFunctionDeclarator and in our quick-fix we navigated back to the CPPASTFunctionDefinition.

```
1 //Checker code
2 checker.reportProblem(C3102DestructorHasNoBodyChecker.PROBLEM_ID, ASTHelper.
      getFunctionDeclaratorFromDeclarationOrDefinition(destructor));
3
4 //Quick-Fix code
5 final ICPPASTCompositeTypeSpecifier struct = ASTHelper.getCompositeTypeSpecifier(
      markedNode);
```

Listing 52: Node forwarding

4.7 getSpecialMemberFunction()

This method needs special mention. Reason for that is because it builds to core of many rule implementations. Within this method, we can request to get a specific

function declaration or function definition from a struct/class. This method is also especially useful when we have a function declaration in a header file and want to find the corresponding function definition in another file.

```
1  public static IASTDeclaration getSpecialMemberFunction(final
       ICPPASTCompositeTypeSpecifier struct, final SpecialFunction desiredFunction) {
2    for (final IASTNode node : struct.getChildren()) {
3      if (node instanceof ICPPASTFunctionDefinition || node instanceof
       IASTSimpleDeclaration) {
4        if (isDesiredFunction(desiredFunction, (IASTDeclaration) node)) {
5          return (IASTDeclaration) node;
6        }
7      }
8    }
9    return null;
10 }
11
12 private static boolean isDesiredFunction(final SpecialFunction desiredFunction,
       final IASTDeclaration decl) {
13   switch (desiredFunction) {
14   case DefaultConstructor:
15     return isDefaultConstructor(decl);
16   case DefaultCopyAssignment:
17     return isDefaultCopyAssignment(decl);
18   case DefaultCopyConstructor:
19     return isDefaultCopyConstructor(decl);
20   case MoveAssignment:
21     return isMoveAssignment(decl);
22   case MoveConstructor:
23     return isMoveConstructor(decl);
24   case DefaultDestructor:
25     return isDefaultDestructor(decl);
26   default:
27     return false;
28   }
29 }
```

Listing 53: getSpecialMemberFunction()

4.8 Packages

In this section we are going to explain each package and the general implementation of the classes. It can also be seen as a quick overview to our software architecture.

4.8.1 ch.hsr.ifs.cute.ccglator

Here a overview of our package dependency management captured with STAN[sta16c]. See figure13 on page 89.

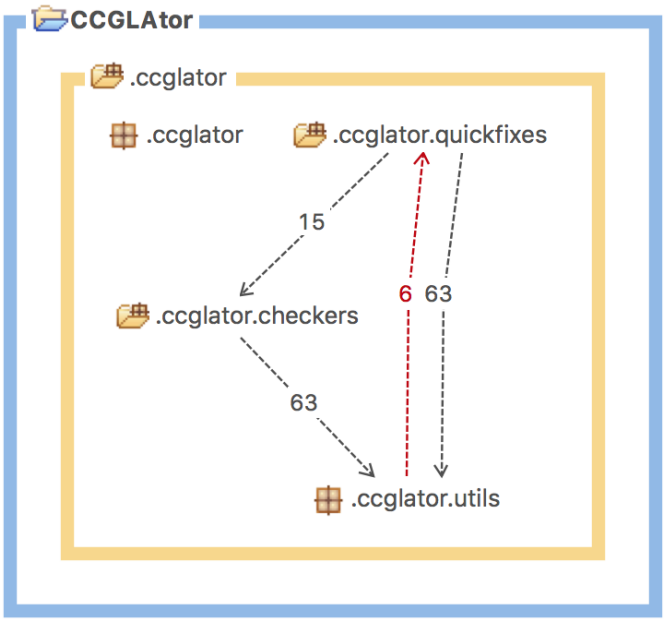

Figure 13: package ccglator

There are some cyclic dependencies because the utils package needs the access to the ASTRewriteStore in the quick-fix package. The ASTRewriteStore will be described section

This package only serves to contain our activator class. An activators purpose is to handle the lifecycle of an Eclipse Plug-in.

4.8.2 ch.hsr.ifs.cute.ccglator.checkers

Here a nearer overview of the checker package (see figure 14 on page 90).

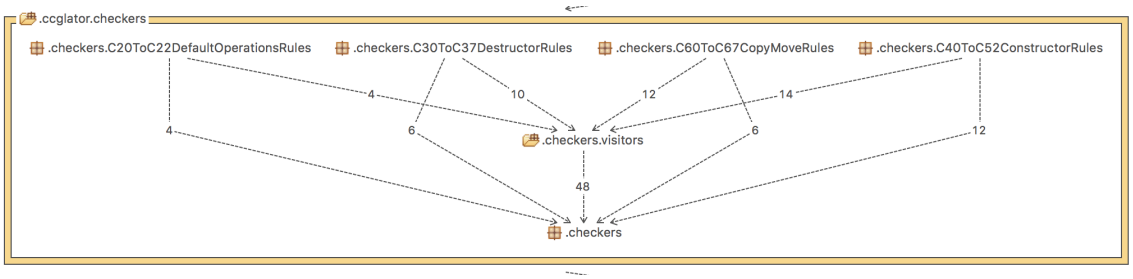

Figure 14: package checkers

This package only serves to hold our checkers (derived from AbstractIndexAstChecker) which will be referenced by our plugin.xml. Every checker needs a problem id which will be used when reporting a problem in the form of a checker. Also every checker has to implement a processAst() method which we are using to traverse the AST using various visitors.

The checkers are implemented as shown in listing 15 on page 90

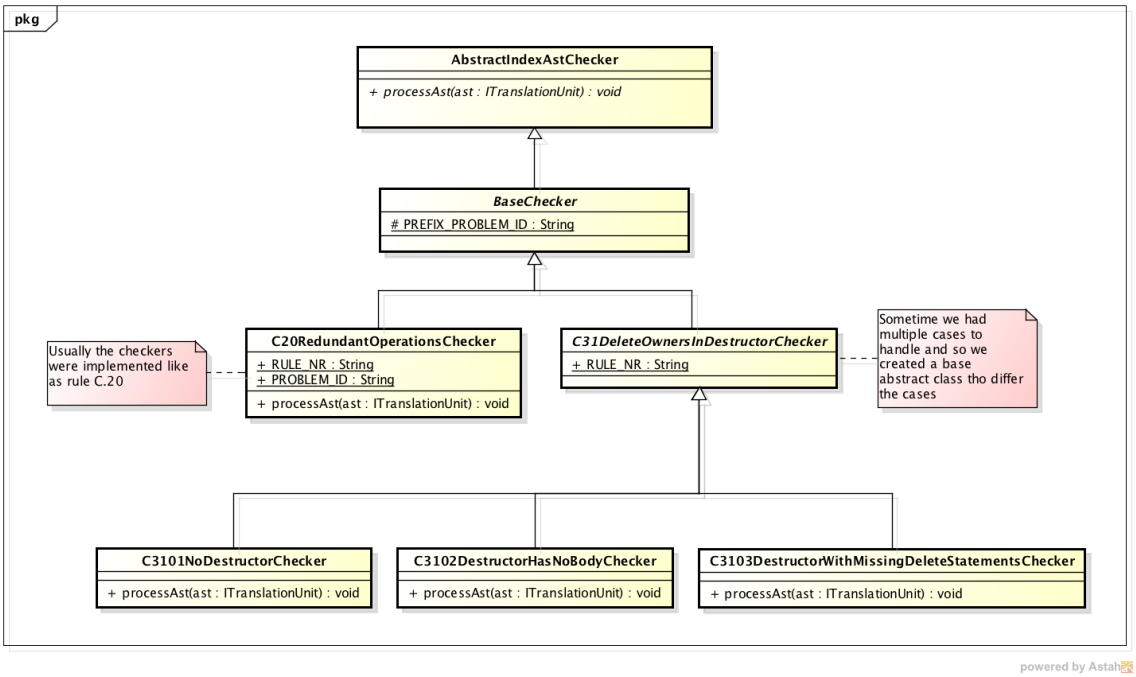

Figure 15: Checker UML

The example only contains two examples but we did not change the implementation scheme with the other checkers.

4.8.3 ch.hsr.ifs.cute.ccglator.checkers.visitors

Here a nearer overview of the checker package (see figure 16 on page 91)

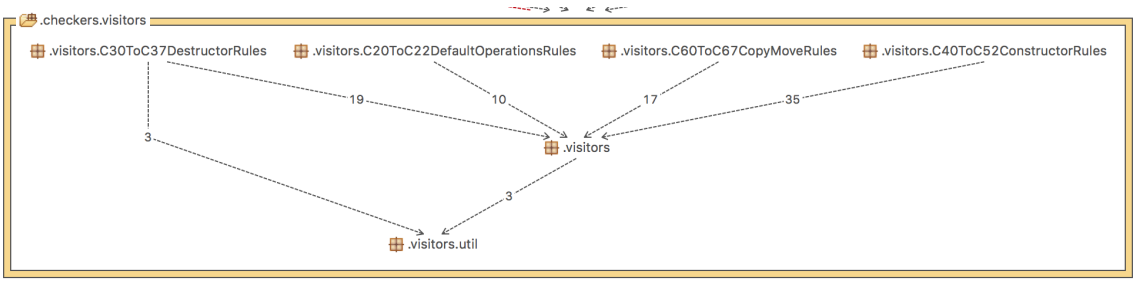

Figure 16: package visitors in checkers

This is where our visitors (derived from ASTVisitor) belong, or in other words, where the core analysis logic lies to determine wether a checker should be placed or not. Since we are mostly working on classes or structs which are represented as a ICPPASTCompositeTypeSpecifier, we are working with the approach of finding a class or struct first and working our way down from there. Of course we could also look for function declarations using our visitor and implement a logic to analyze the given scope of our function declaration. But we figured out that analyzing a class or struct is essential to many of our implementation parts and when working with a class or struct, rather than with the actual node we have to operate on, we can have the same initial situation for every implementation. A big benefit from having the same initial situation for every implementation is, that we can reuse a lot of code since we often have the same goals to achieve (like finding the default move assignment or getting the visibility for a function declaration). Here is an example implementation for rule C.31 where we have to ensure that a destructor exists for a class or struct with owners marked with the gsl::owner<T*> annotation to show how easy this approach made our implementation:

```
1 @Override
2 public int visit(final IASTDeclSpecifier declSpec) {
3   if (declSpec instanceof ICPPASTCompositeTypeSpecifier) {
4     if (hasGslOwners(declSpec) && hasNoDestructor(declSpec)) {
```

```
5         checker.reportProblem(C3101NoDestructorChecker.PROBLEM_ID, struct.getName());
6       }
7     }
8     return super.visit(declSpec);
9   }
10
11  protected boolean hasNoDestructor(final IASTDeclSpecifier declSpec) {
12    return ASTHelper.getSpecialFunction(struct, ASTHelper.SpecialFunction.
        DefaultDestructor) == null;
13  }
14
15  protected boolean hasGslOwners(final IASTDeclSpecifier declSpec) {
16    struct = (ICPPASTCompositeTypeSpecifier) declSpec;
17    final List<IASTSimpleDeclaration> gslOwners = ASTHelper.collectGslOwners(ASTHelper
        .collectMemberVariables(struct));
18
19    return gslOwners.size() > 0;
20  }
```

Listing 54: Destructor needed when class has owners

Simple as that. Of course some rule implementation ended up being more complex. But the idea of this approach is that we end up with a big set of methods which we can reuse for every rule. More on this will be discussed in section 4.8.6

4.8.4 ch.hsr.ifs.cute.ccglator.quickfixes

Here a nearer overview of the quick-fix package (see figure 17 on page 92)

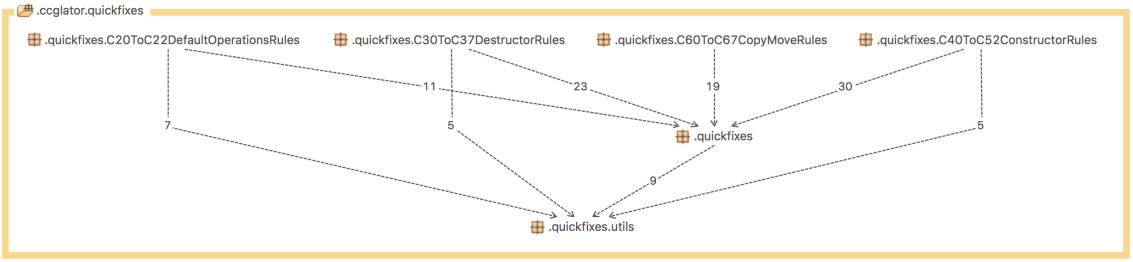

Figure 17: package quick-fixes

This package stores the quick-fixes for each rule including the rule ignore quick-fix using C++ attributes.

The checkers are implemented as shown in listing 18 on page 93

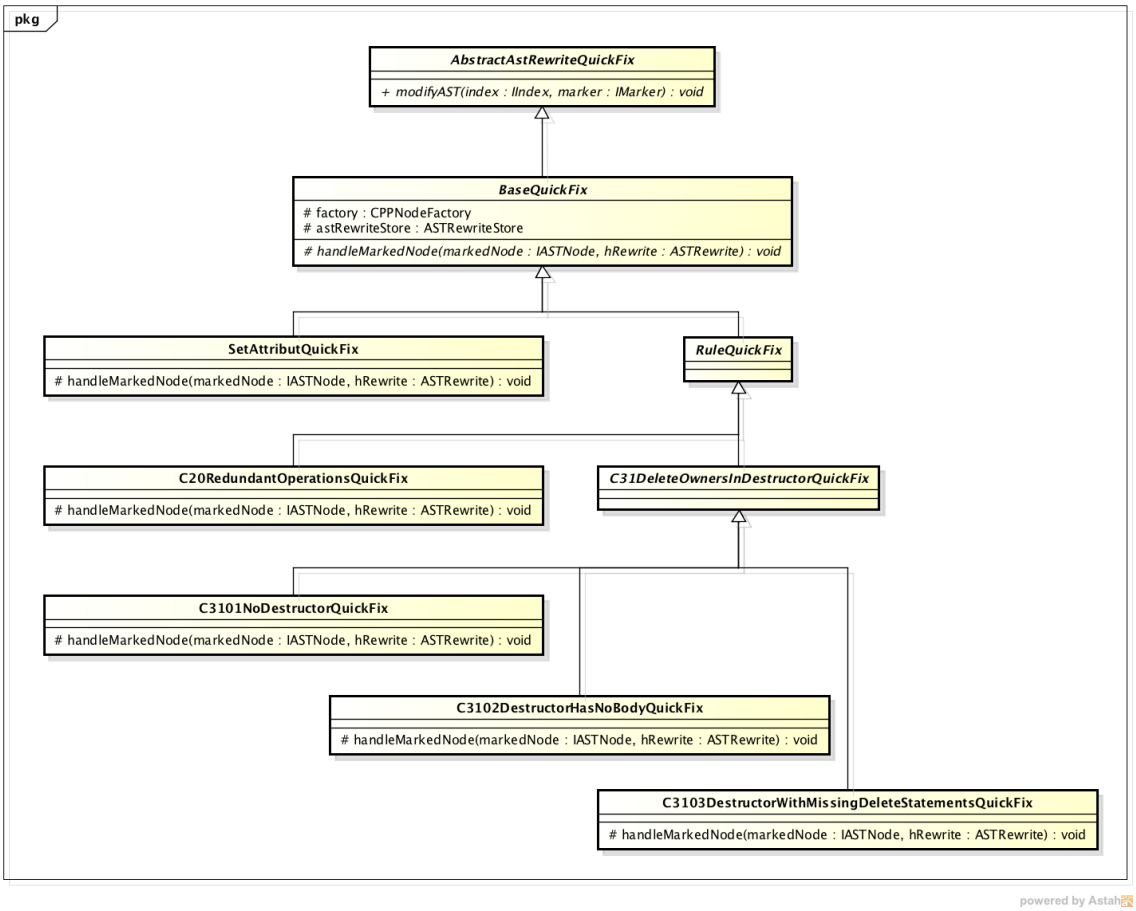

Figure 18: Quick-fix UML

4.8.5 ch.hsr.ifs.cute.ccglator.quickfixes.utils

Here is where we store our factory methods which are only used by the quick-fixes. Let's say for instance we have no default move assignment defined, but there has to be one with a certain body. Our ASTFactory class provides several methods to create such functions.

It also contains a very important class. The ASTRewriteStore. The class handles the different ASTRewrites and ASTTranslationUnits we had to deal with. This class is copied from the CharWars[GF] plug-in from Toni Suter and Fabian Gonzalez. At the beginning we thought that we would not need it because we did not have

any problems with the ASTTranslationUnit. But as we began implementing the possibility to refactor in the header as well as in the cpp file we had some very confusing results after a quick-fix. Sometimes the some code was missing like in a very bad git merge where half of the code was gone. We then integrated the ASTCache from the CharWars and then it was fine.

4.8.6 ch.hsr.ifs.cute.ccglator.utils

This package stores the core part when it comes to the logic behind our implemented visitors being our ASTHelper. The idea of the ASTHelper is to store public static methods which can be useful for any rule implementation. For example are there many rules in which we have to work with the destructor of a class. This of course involves finding the destructor at first hand. Methods like those and a lot more are implement in our ASTHelper.
Another class which is stored in this package is our CCGlatorCPPNodeFactory which only stores a static node factory which is used in various quick-fixes.

4.9 plugin.xml

Our plugin.xml file only consists of checkers and quick-fixes. The entire file quickly grew in size. So here is a small outtake of a checker and a quick-fix implementation:

```
<!-- Checker declaration -->
<extension point="org.eclipse.cdt.codan.core.checkers">
      <checker
          class="ch.hsr.ifs.cute.ccglator.checkers.C20RedundantOperationsChecker"
      id="ch.hsr.ifs.cute.ccglator.checkers.C20RedundantOperationsChecker"
      name="Redundant operation">
       <problem
          category="ch.hsr.ifs.cute.ccglator.categories.ccglator"
            defaultSeverity="Warning"
            defaultEnabled="true"
            description="Avoid redundant default operations (C.20)"
            id="ch.hsr.ifs.cute.ccglator.problems.C20RedundantOperations"
            messagePattern="Avoid redundant default operations (C.20)"
            name="Avoid redundant default operations">
         </problem>
      </checker>
</extension>

<!-- Link checker with quick-fix -->
<extension point="org.eclipse.cdt.codan.ui.codanMarkerResolution">
      <resolution
```

```
22              class="ch.hsr.ifs.cute.ccglator.quickfixes.
    C20RedundantOperationsQuickFix"
23              problemId="ch.hsr.ifs.cute.ccglator.problems.C20RedundantOperations">
24        </resolution>
25 </extension>
```

Listing 55: Checker and quick-fix in plugin.xml

There is more that can be done inside the plugin.xml file, like defining a category for all the checkers, set a color in which the checker will be marked, an icon for the checker and a lot more.

4.10 Redundant operations (C.20)

This rule focuses on detecting wether a default operation is redundant or not 3.8.1.

4.10.1 Checker

Default operations (also known as rule of three where the default constructor isn't included) consist of these 4:

C::C(); Default constructor

C::C (const C&); Copy constructor

C& operator= (const C&); Copy assignment

C::~C(); Destructor

Default operations usually don't have to be defined explicitly in C++. Though it is very hard to determine wether the definition of a default operation is needed or not from a static-analysis point of view. Therefore we decided to implement rule C.20 as follows:

1. If a class has member variables marked with gsl::owner<>, leave the default operations as they are.
2. If a default operation has a body with at least one statement, it is probably necessary.

When working with owners, default operations have to be implemented. Reason for that being the resource management that's behind owning a resource. Therefore we check for the existence of an owner.

The body of a default operation can be anything. Of course we could try to analyze the body and try to figure out if the statements inside are necessary. But since it's almost impossible to semantically say what is "necessary", we just say that an empty body makes the operation unnecessary.

4.10.2 Quick-Fix

A quick-fix basically just offers the deletion of an unnecessary default operation. But since the marker can be placed on a declaration or a definition we have to perform a check upfront. If we have a declaration marked, we first need to get the definition and set it to = default.

4.11 Missing special member functions (C.21)

If a special member function exists, it is very likely that all the others are needed as well 3.8.2.

4.11.1 Checker

Special functions (also known as rule of five where the default constructor isn't included) on the other hand consist of 6 different functions:

C::C(); Default constructor

C::C (const C&); Copy constructor

C& operator= (const C&); Copy assignment

C::C (C&&); Move constructor

C& operator= (C&&); Move assignment

C::∼C(); Destructor

The idea of this rule is to have either all or none of the special functions declared. This makes sense because if one special function is needed, chances are very high that the others are necessary as well. What we basically do when analyzing, is to count the amount of special functions. This of course is not a perfect way of analysis, since we don't explicitly know which of the special functions are missing. But this

is not very important at this stage of analyzing anyways, since we're only setting a checker at this stage.

```
if (ASTHelper.isSpecialFunction(function)) {
  numOfSpecialFunctionsFound++;
}
```

Listing 56: Counting special functions

4.11.2 Quick-Fix

Here's where we have to add the missing special functions. Since we don't know, which special functions are missing, we have to check this as well in our quick-fix:

```
if (hasNo(struct, ASTHelper.SpecialFunction.DefaultConstructor)) {
  insertNode(mainRewrite, struct, ASTFactory.newDefaultConstructor(struct.getName())
    );
}
if (hasNo(struct, ASTHelper.SpecialFunction.DefaultCopyAssignment)) {
  insertNode(mainRewrite, struct, ASTFactory.newDefaultCopyAssignment(struct.getName
    ()));
}
if (hasNo(struct, ASTHelper.SpecialFunction.DefaultCopyConstructor)) {
  insertNode(mainRewrite, struct, ASTFactory.newDefaultCopyConstructor(struct.
    getName()));
}
if (hasNo(struct, ASTHelper.SpecialFunction.MoveAssignment)) {
  insertNode(mainRewrite, struct, ASTFactory.newDefaultMoveAssignment(struct.getName
    ()));
}
if (hasNo(struct, ASTHelper.SpecialFunction.MoveConstructor)) {
  insertNode(mainRewrite, struct, ASTFactory.newDefaultMoveConstructor(struct.
    getName()));
}
if (hasNo(struct, ASTHelper.SpecialFunction.DefaultDestructor)) {
  insertNode(mainRewrite, struct, ASTFactory.newDefaultDestructor(struct.getName())
    ;
}
```

Listing 57: Quick-fix of missing special functions

4.12 Delete owners in destructor (C.31)

Our supervisor advised us quite a few times to consider refactoring owners into unique_ptr<> or shared_ptr<>. But we decided to follow the C++ Core Guidelines with all of our rule implementations, including this one. Reason for this decision

is that the Core Guidelines have never touched on unique_ptr<>'s and doing so, would require lots of rules to be rewritten. We completely agree that working with unique_ptr<> would make more sense since they are automatically deleted in the default destructor. That fact would make this rule quiet redundant. But following the Guidelines still seemed like a better approach to us. Otherwise many other rules could be altered as well.

4.12.1 Checker

This rule only applies to classes with owners marked as gsl::owner<>. The main idea is to delete every owner in its destructor. There are 3 possible initial states from where we could progress with our quick-fix:

1. **No destructor existant (C.3101)**
   When we have at least 1 owner, we also need a destructor. If it's missing, we have to set a checker placed on the struct or class itself.
2. **Destructor existent. But destructor has no body (C.3102)**
   A destructor can be set as defaulted or deleted, of which both are incorrect when working with owning pointers or references. A default destructor does not delete owners.
3. **Destructor with body existent. But delete statements in body missing (C.3103)**
   In this case our destructor has a body. This is also where the core of the analysis is done. Delete statements can be written in 2 different ways:

```
1 Auto::~Auto() noexcept
2 {
3         delete anhaenger, delete[] &raeder;
4         delete chasis;
5 }
```

Listing 58: Delete statements

   Missing delete statements have to be detected and if at least one exists, we are setting a checker. Also we decided to modify the error text with the member variables that have not been deleted yet.

Therefore we decided to implement 3 different checkers, as well as 3 visitors and 3 quick-fixes. We decided to take this approach, since it makes the implementation of our quick-fix easier having parts of the problem already analyzed and separated in

our checker using visitors.
Like explained in section 4.5, the checkers are preferably set in the header file. But if no declaration is existent, the checker is set on the member function definition. These checks have to be done in the quick-fix though.

4.12.2 Quick-Fix

Our quick-fixes are also separated in the same 3 cases as explained above. In this section we will briefly look at what each quick-fix implementation does and why we decided to implement them in such ways.

1. **No destructor existent (C.3101)**
   This is the most simple case for our quick-fix. If no destructor exists, we can simply create one in our class. We had the option to declare the destructor in the header file and define it in our .cpp file. But since a member function defined in a class scope is implicitly inlined, we decided to only define it.
   Is is also possible, that a declaration to a destructor exists, but no destructor is defined. In this case, we decided to remove the destructor declaration and define it in the header file.
2. **Destructor existent. But destructor has no body (C.3102)**
   If a destructor is existent, we have to determine first if we already have the destructor definition as a node or if we are working with a declaration. From there on it's basically just removing =delete or =default statements and adding a body with the missing delete statements to it.
3. **Destructor with body existent. But delete statements in body missing (C.3103)**
   This is the case where we have to determine which of the delete statements are missing. Depending on which are missing. we basically add them at the bottom of the destructor body. But of course it is not as easy as it sounds. Each owner also has to be deleted in the right way. This will be discussed in the following section 4.12.3

4.12.3 How to delete owners

Delete statements in C++ can be written in various ways. We can delete a simple pointer, a reference, a pointer array or an owning reference array. Depending on how

an owner was initialized, we have to delete them differently. This forced us to scan our entire class for initializations of any kind and for every owner using the index. The results were stored in a list for further processing in our quick-fixes.

```
final List<OwnerInformation> ownerInformation = ASTHelper.collectInformationOfOwners
    (struct, gslOwners, rewriteStore);
```

Listing 59: Collecting owner information

Of course, all of this would not be necessary if we would refactor owners to unique_ptr<>. But we decided to follow the C++ Core Guidelines and leave the owning pointer as they are and only focus on the correct deletion.

4.13 Base class destructor (C.35)

The destructor of a base class has to be either public and virtual or protected and non-virtual 3.9.6.

4.13.1 Checker

The main requirement for this rule to apply, is that our class contains at least one virtual member function and that a destructor exists. Knowing this, we only have to check if our destructor is either public and virtual or protected and non-virtual. Finding out if a statements is virtual or not is easy. It's also quite simple to determine if a statement lies in a protected scope since a scope can only be set as protected using a so called VisibilityLabel. But a statement can be in a public scope when the surrounding type is a struct instead of a class. Therefore we decided to implement a method to get the visibility of a statement so that we can compare it with what the visibility should be like. This of course, can also be used in various other rules.

```
public static int getVisibilityForStatement(final IASTNode statement) {
  final ICPPASTCompositeTypeSpecifier function = getCompositeTypeSpecifier(statement
    );
  int visibility = getDefaultVisibilityForStruct(function);

  for (final IASTNode node : function.getChildren()) {
    if (node instanceof ICPPASTVisibilityLabel) {
      visibility = ((ICPPASTVisibilityLabel) node).getVisibility();
    }
    if (statement == node) {
      return visibility;
    }
  }
```

```
13   return 0;
14 }
```

Listing 60: Finding out visibility of a statement

4.13.2 Quick-Fix

The checker has already done the biggest part of our analysis and we know that the destructor signature is not as it should be. Therefore we check for the visibility of our destructor at first. There's 3 possible cases:

- **Public Destructor**
  In this case, we set the destructor to virtual. We can assume that it wasn't virtual before, since this was already analyzed by our checker.
- **Protected Destructor**
  In the case of a protected destructor, we remove the virtual specifier. This disallows derived classes to destruct the base class via a pointer.
- **Private Destructor**
  Although in many cases a private destructor makes no sense, it can still exist. Cases where a private destructor would make sense are:
  - **Allowing instantiation only from another member function**
  - **Limit inheriting (Questionable approach)**
  - **Class could have a "disposed method" as a static member (Manager Pattern)**

  Since there are not many usages of having a private destructor, we decided to remove it out of the private scope and set it to protected and non-virtual. Reason for that is because per default we did not want to allow polymorphic deletion through a base pointer which is a safer approach. In case of a user wanting any of the above mentioned behavior, one could still set an ignore attribute.

4.13.3 Placing a statement into a visibility scope

In the case of a private destructor, we place the destructor into a protected scope like explained. There's 2 possible cases doing so. There could be a protected scope already existent (in this case we just place our destructor into that existing scope) and also there could be no protected scope in our class. In this case, we create a protected visibility label at the end of our class and place our non-virtual destructor inside.
Another problem that arises when wanting to place a statement under a visibility label, is that we want to place node B after node A. And since this node A may or may not already exist, we could also end up with a situation where we have to create both. But if it already exists, we could simply place our node B after node A. No we can't. The Eclipse CDT offers 3 methods to modify an AST:

```
rewrite.insertBefore(parent, insertionPoint, newNode, editGroup);
rewrite.remove(node, editGroup);
rewrite.replace(node, replacement, editGroup);
```

Listing 61: ASTRewrite methods

So there's no method to place a node B after node A. We implemented our own getNextNode() method to provide remedy for our use case.

```
public static IASTNode getNextNode(final IASTNode node) {
  boolean readyToPull = false;
  for (final IASTNode nextNode : node.getParent().getChildren()) {
    if (readyToPull) {
      return nextNode;
    }
    if (nextNode == node) {
      readyToPull = true;
    }
  }
  return null;
}
```

Listing 62: getNextNode()

What's so special about this method, is that we are only looking for the next node in the same or higher hierarchy. So this way we also eliminate the case, where we want to insert a node using insertBefore() when there is no next node existent.

4.14 Destructor should be noexcept (C.37)

Destructors should be set to noexcept. Easy and simple.

4.14.1 Checker

The only thing our checker has to do is to find the destructor and if it exists, check if it's set to noexcept.

4.14.2 Quick-Fix

At this point we already know about the existence of a destructor that is not set to noexcept. We simply add the noexcept statement to the found destructor.

4.15 noexcept default constructor (C.44)

Like the name already says, this rule is makes sure that the default constructor is set to noexcept. This rule was not difficult to implement as analyzed in section 3.10.5.

4.15.1 Checker

The Checker visited all IASTDeclarators and checked if it is a declarator from a default constructor. If so it checked if it was set to noexcept. Then if this check fails a marker is set on the declarator.

4.15.2 Quick-fix

The quick-fix for this rule was also not that hard to implement. We just had to check if the declarator was in a IASTSimpleDeclaration or in a ICPPASTFunction-Definition. The easy case is the IASTSimpleDeclaration. In that case we just hat to replace the IASTDeclarator with a new one which was set to noexcept. With the "harder" case we needed to get the implementation as well and modify its declarator too.

4.16 In-class initializer (C.45)

This rule makes sure, that member variables are properly initialized when the object is created. This rule was harder to implement, because we had to do more analyzing of the AST in our checker.

4.16.1 Checker

This checker scans all IASTDeclSpecifier and checks if it is a ICPPASTCompositeTypeSpecifier. Then we separate two cases. One where the member variables are not all in-class initialized nor are they initialized by the default constructor. And the other one is if the member variables are also not all in-class initialized but the default constructor is const initializing some of them. For either of the cases are set different checkers to run different quick-fixes. Because of the checker C.20 (section 4.10 on page 95, which covers the rule C.20 3.8.1 on page 31) the case where all the member variables are in-class default initialized when the constructor only default initializes them, is covered.

4.16.2 Quick-fix

The first quick-fix fixes the case where neither the member variables are in-class initialized or initialized in the default constructor. This quick-fix simply collects all member variables and checks if they are in-class initialized and of course default initializes them if this is not the case. After that the default constructor can be set to default if it has not some kind of side effect. The second quick-fix initializes the member variables with the const initial values from the default constructor. Here we needed to differ from declaration and definition. If it is a declaration we need to get the implementation and if not we can directly get the all the member variables which are initialized in the constructor initializer chain. Those who are not initialized in the chain are default in-class initialized. The constructor now has to be changed (also if there is no side effect in it) to default.

4.17 Declare single argument constructor explicit (C.46)

This rule makes sure that single argument constructors are declared explicit so that no conversion problems occur. This rule was not that hard to implement as analyzed

in section 3.10.7.

4.17.1 Checker

The checker visits all IASTDeclarators and checks if its a function declarator. Then it checks if it is a single argument constructor. If it is a single argument constructor it marks the declarator. If not it checks if it is a two argument constructor with the second argument defaulted. This constructor can be handled as a single argument constructor.

4.17.2 Quick-fix

The quick-fix now has the easy job. It takes the declarator and get its function definition. From there he can get the simple decl specifier and check set it to explicit.

4.18 Initialize member variables in the right order (C.47)

This rule is to make sure that the member variables are initialized in the right order in the constructor initializer chain. If not this could lead to serious bugs because the compiler does it in the way they are declared. This rule was a bit harder to implement but not really a big problem.

4.18.1 Checker

The checker visits all IASTDeclarators and checks if it is a function declarator. Then it checks if it is a declaration or definition. If it is a declaration we need to get the implementation (as always) and then get the function definition to continue. We then need to check if the initializer list is in the same order as the declarations of the member variables. This was a bit tricky because it could be possible that not every member variable is initialized, which had to be considered as well. If the initialization in the chain was not in the right order we mark the declarator.

4.18.2 Quick-fix

The quick-fix needs to get the function definition. Also it is needed to know the order of the member variables, which we can get from the ASTHelper. Then the order of the initializer chain needs to be compared to the member variables. If the compare fails the initializer chain needs to be changed to the right order. This is basically done by iterating over the member variables and the initializer chain at the same time. So the two variables can be compared and if they do not match a new constructor chain initializer is created with the right name. Because the checker C.45 (section 4.16 on page 104 which covers the rule C.45 3.10.6 on page 46) we made sure that there are no const initial values and that all member variables are initialized in the constructor initializer list.

4.19 Prefer in-class initializer to constructors (C.48)

This rule was a bit tricky because it somehow had lots of similarities to the rules C.45 (section 4.16 on page 104 which covers the rule C.45 3.10.6 on page 46) and C.20 (section 4.10 on page 95 which covers the rule C.20 3.8.1 on page 31). So we decided to implement the part where the constructor with the most parameters has default values should be in-class initialized. This rule was kind of confusing because we got sense of it a the beginning.

4.19.1 Checker

Here we need to get the constructor with the most parameters and check if they have all default values. If so we mark the declarator of this constructor.

4.19.2 Quick-fix

In the quick-fix we simply collect the member variables and in-class initialize them with the default values from the constructor. Then the constructor needs to be changed. The default values are no longer needed and can be removed.

4.20 No assignments in constructor (C.49)

The problem here was to avoid assignments in the constructor. It is better to initialize them in the constructor initializer chain then re-assign them in the constructor body.

4.20.1 Checker

What the checker does is it goes through the body and looks for an member variable assignment. If there is one the constructor will be marked.

4.20.2 Quick-fix

The quick-fix then removes the member variable assignment in the body and initializes the member variable in the constructor chain initializer.

4.21 Copy Assignment Signature (C.60)

Copy assignment should be non-virtual. The parameter has to be taken by const& and return by non-const&.

4.21.1 Checker

As the intro already tells, this rule could be separated in 3 parts. For the checkers that's actually what we did. The reason for that is to make the implementation of the quick-fixes easier since most of the analysis is already done by the time we reach our quick-fix. Our 3 separations consist of:

- **Non-virtual (C.6001)**
  Simple rule where we just check if the copy assignment is set to virtual. If so, we set a checker with the type C.6001.
- **Parameter by const& (C.6002)**
  This rule includes several checks such as if there's only 1 input parameter, if it's taken by const& as well as if the input parameter is of the same type as our class.

- **Return by non-const& (C.6003)**
  Rule C.6003 brought an additional hurdle with it. The return type as well as the const do not belong to the function declarator. They are both stored in the ICPPASTDeclSpecifier. This forced us to do analysis on various nodes. The same problem occurred when having to quick-fix this sub rule. This will be discussed in the quick-fix section though.

4.21.2 Quick-fix

Rules C.6001 and C.6002 are straight forward. If there's a virtual keyword, we remove it. If the const or reference operator are missing in the input parameter, we add them. But C.6003 is a bit more tricky.

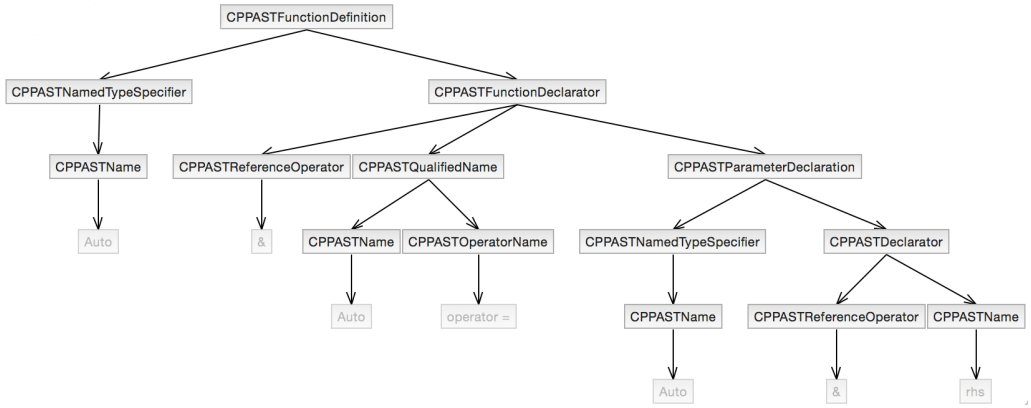

Figure 19: AST of a copy assignment member function

Handling rule C.6003 was especially tricky since we had to make sure that the return type is both of the same type as our class, as well as a non const&. Now to what made the implementation tricky. Like shown above in the AST, the return type and const declaration both belong to the so called CPPASTNamedTypeSpecifier whereas the reference operator belongs to the function declarator. And since we're only passing the DeclSpecifier (which in our case is the NamedTypeSpecifier), we also needed to implement helper methods to get our function declarator from a NamedTypeSpecifier of the same hierarchy.

4.22 Move Assignment Signature (C.63)

Pretty similar to C.60 4.21 and not worth mentioning any further.

4.23 Move operations should be noexcept (C.66)

Very similar to C.37 4.14 and C.44 4.15 and not worth mentioning any further.

4.24 Attributes

Sometimes the checker can be ignored because the developer knows exactly what he is doing. For that case the plug-in provides for every rule a quick-fix to ignore the checker.

4.24.1 Visitor

Every checker uses a visitor to do its analysis and if necessary set a marker. Now in order to prevent the checker to mark a node with an ignore attribute we came up with the following implementation. See figure 20 on page 109.

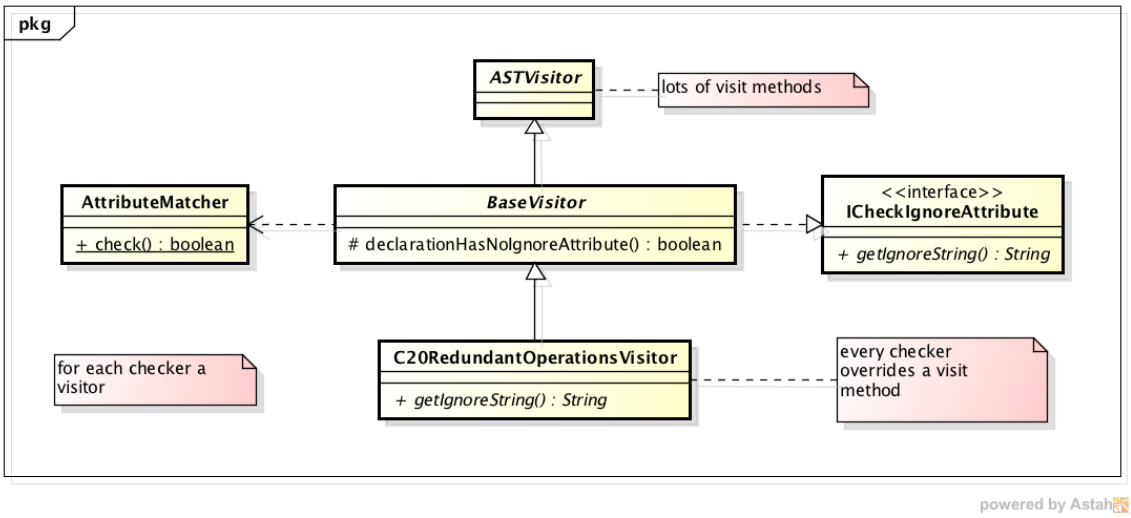

Figure 20: Implemenation of the visitor

The BaseVisitor "implements" a interface to force the sub classes to implement the method getIgnoreString(). This is basically the rule number. The method declarationHasNoIgnoreAttribute() then passes this information to the AttributeMatcher for checking if there is any attribute set and if so ignore the rule the attribute stands for.

4.24.2 Quick-fix

This quick-fix is designed to set an ignore attribute. It does not matter were the checker is set. It detects if it is a declaration or a definition and adds the attribute on the right place. Because the checker in our plug-in is only set on definitions or declarations the quick-fix had not to do much analysis on where the checker was set. However the marked node is mostly a declarator or name so the quick-fix need to get the declaration or definition depending on is it inline implemented or not. From the declaration or definition the method getAttributSpecifier() can be used to check if there are some attributes set. We used it to replace the specifier with a new one containing the ignore attribute for the checker invoked for this quick-fix.

4.25 CDT Bug

During our implementation we came across a AbortFormattingException. After some investigation on why this exception occurs we found out that it only occurs when we used a refactoring with the keyword noexcept. At first we were not sure if this is just us doing something wrong with the syntax in C++, but after consulting our co-advisor Toni Suter we could be sure that it is a bug in the CDT.

5 Testing on real-life code

We wanted to know how our plug-in works on a real C++ project. For that we choose the open broadcast software OBS[OBS16].

5.1 Statistics

Our plug-in found over 600 violations. What we can say, most of the violations are declarations of noexcept which are missing. Also on the next place not all special member functions are properly declared.

5.1.1 Checkers

The checkers are listed in table 7 on page 111.

% of Checkers	Rule violation
11%	C.20 Avoid redundant defaul operations
25%	C.21 Missing special member functions
3%	C.35 A base class destructor should either be public virtual, or protected non-virtual
18%	C. 37 Destructor should be declared noexcept
13%	C. 44 Default constructor should be declared noexcept
3%	C.45 Default constructor shouldn't only initialize data members
6%	C45 In-class member variables should be default initialized
9%	C.46 Single-argument constructors should be declared explicit
5%	C.47 Member variables should be initialized the same order as they are declared
1%	C.48 Prefer in-class initializers to member initializers in constructors for constant initializer
1%	C.49 Prefer initialization to assignment in constructors
2%	C.60 Return parameter should be non-const&
3%	C.66 Move operations should be declared noexcept

Table 7: Overview of the rules Set of default operations rules C.20 to C.22

5.1.2 Quick-fixes

We also tried to quick-fix those checkers. All checkers with noexcept worked properly (we have not all quick-fixed but we took many random samples). The set explicit worked fine as well. The quick-fix for return parameter should be non-cont& also worked. Sadly, some quick-fixes from C.47 do not work properly.

6 AliExtor

This section is all about the additional work on the term project AliExtor.

6.1 Implementation

This section is about the implementation of the open points. It will be described what is done and if there were some difficulties.

- Simplification of the UI
- Add shortcut to invoke the plug-in
- Eliminate known bugs

6.1.1 Simplification of the GUI

The wizard to create an alias is a bit simplified. The page where the user can select whether he wants to extract an alias template or a normal template is merged into one.

Where the user had to click through two pages (see Figure 21 on page 114 and 22 on page 114).

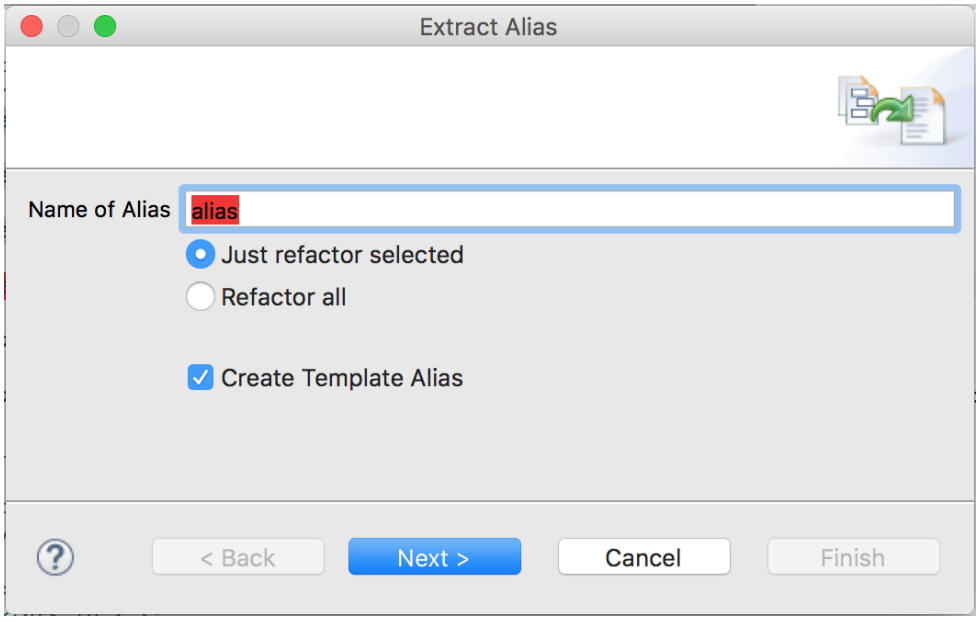

Figure 21: Page 1 of the wizard

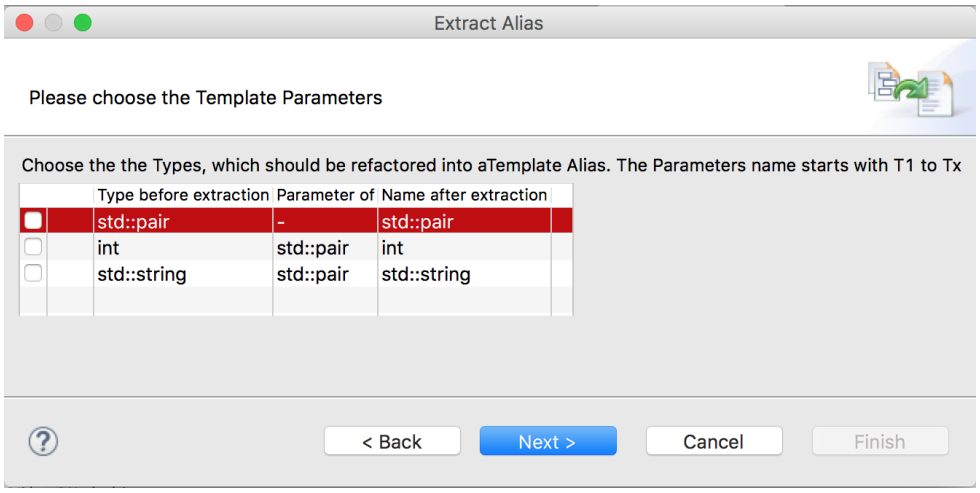

Figure 22: Page 2 of the wizard

Where the new page looks like Figure 23 on page 115.

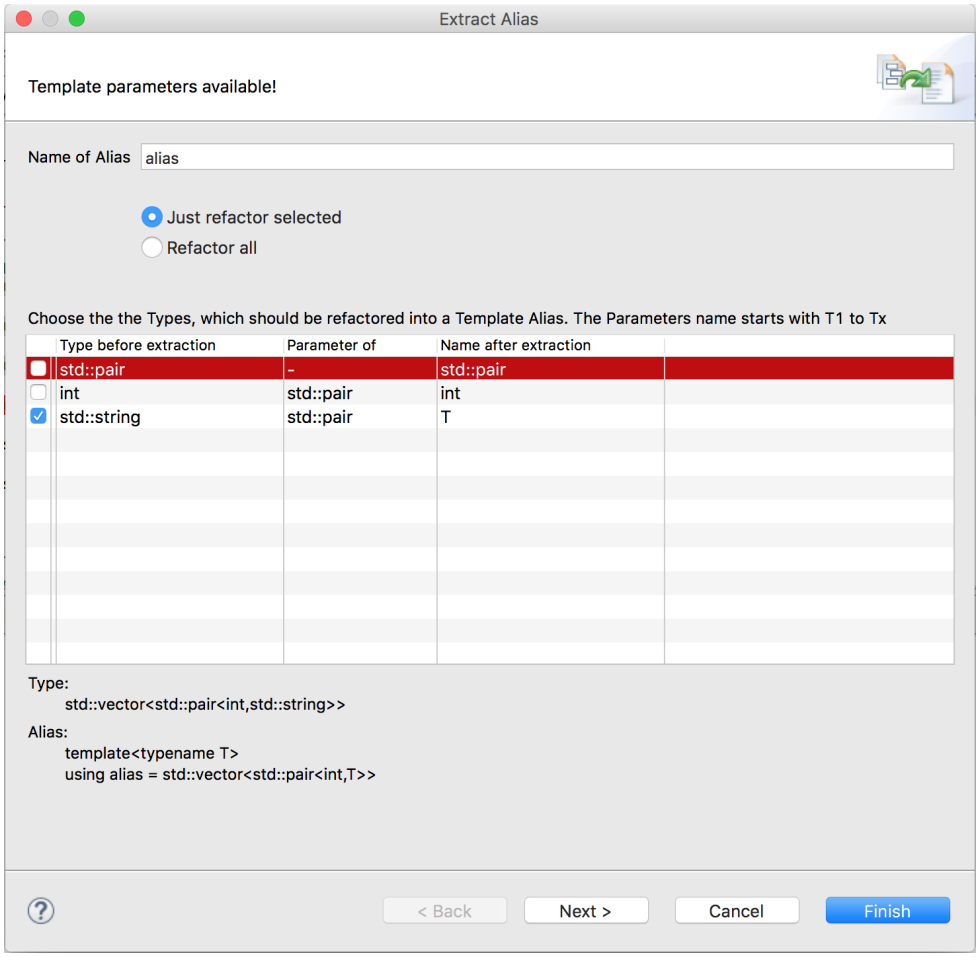

Figure 23: New wizard page

Note: The preview of the new type in the lower part of the page was also in the old page, but cut off during the shrinkage of the window.

The new wizard page analyzes if the selected type has template parameters and lists them in the list. If there are no parameter the list and the new type preview will not be shown.

6.1.2 Add a shortcut

For better usability we added a shortcut. The user now has to select the type where he wants to extract an alias. With the keys cmd/win + ctrl + A the refactoring can be called (see Figure 24 on page 116).

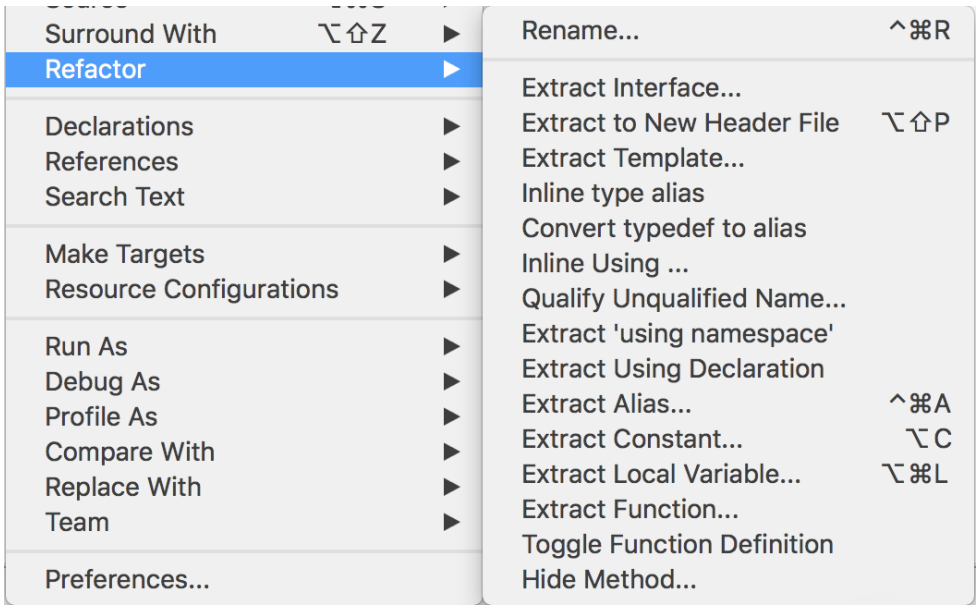

Figure 24: The shortcut cmd/win + ctrl + A

Implementation of this feature was solely done in the plugin.xml. When working with Eclipse there is an easy way to implement a shortcut for a command. But since we are working with the CDT, which doesn't support Eclipse commands and we have to work with the deprecated actions, implementing a shortcut was a bit more tricky. While investigating on how to implement such trivial addition we also encountered that there was no single plug-in existent in the CUTE repository which had a shortcut for a menu entry or refactoring entry implemented. We ended up with the following working solution:

```
1 <!-- Shortcut for Menu tab & Refactoring Entry -->
2 <extension
3   point="org.eclipse.ui.commands">
4   <activeKeyConfiguration
5     value="org.eclipse.ui.defaultAcceleratorConfiguration">
6   </activeKeyConfiguration>
7   <category
8     name="Refactor - C++"
```

```
9       id="ch.hsr.ifs.cute.aliextor.shortcutCategory">
10    </category>
11    <command
12      category="ch.hsr.ifs.cute.aliextor.shortcutCategory"
13      name="Extract Alias..."
14      id="ch.hsr.ifs.cute.aliextor.shortcut">
15    </command>
16    <keyBinding
17      command="ch.hsr.ifs.cute.aliextor.shortcut"
18      string="Ctrl+Command+A"
19      scope="org.eclipse.cdt.ui.cEditorScope"
20      configuration="org.eclipse.ui.defaultAcceleratorConfiguration">
21    </keyBinding>
22  </extension>
```

Listing 63: Shortcut implementation in plugin.xml

6.1.3 Debug known bugs

There was a bug which was detected after we were finished with the term project AliExtor. Somehow there was a stackoverflow if the type in listing 64 on page 117 is selected and if the pair in the pair is choosen to be a template parameter.

```
1 int main(){
2   std::vector<std::pair<int, std::pair<int, std::string>>> keyKeyValue { };
3 }
```

Listing 64: The problem type

During the debugging it came out that the helper class for creating the table was a bit too complex and was simplified. The recursion was not the problem, it works properly. The problem was the filling of the table and its data was not done right. For better explanation lets have a look in the type from listing 64 on page 117 .

	Type before extraction	Parameter of	Name after extraction
☐	std::pair	-	std::pair
☐	int	std::pair	int
☐	std::pair	std::pair	std::pair
☐	int	std::pair	int
☐	std::string	std::pair	std::string

Figure 25: Parameter list of the type from listing 64 on page 117

By the creation of the table each row needs to know its type. And if it is a parameter it needs to know of whom it is the parameter of. In order to do that a help datatype

was used to save the ordering of the types from left to right in an ArrayList<T>(). A second one was used to store the parameters of a type in a Hashmap<Pair<String, Integer>,ArrayList<Pair<String, Integer>>>. The key type pair is the normal type and the second is a list with its parameter. The type pair contains the type as string and the integer is kind of a counter, because a type could occur several times in a parameter list. Together with the two types it is possible to create the table with all the necessary informations. Now the bug was the filling was not done right. If a row gets created it scans the existing types in the table and looks if it is a parameter of this type. It happened that this process failed because the first pair added itself as a parameter which resulted in a stack overflow. This process was revised and works now properly.

7 Conclusion

This section explains what we were able to achieve during the timeframe of our bachelor thesis. Additionally we'll cover our personal reflections ranging from project management to implementation related concerns as well as general reflections about our bachelor thesis time.

7.1 Achievements

We ended up with 15 implemented rules of which all offer a checker as well as a quick-fix. All of these rules have separate test-cases, of which each handles a unique case. We ended up with 300+ test cases which results in more than 20 test cases per rule implementation. Our focus was to implement each rule correctly and cover as many use cases as possible rather than implementing as many rules as possible.
Also what we're especially proud of our helper methods. Because we were mainly operating on declarations and definitions of special member functions, we created a big set of helpful methods which can be used to easily implement further rules. The main struggle with analyzing certain rules is to gather the needed information about the given code using the AST. With our helper methods, we managed to simplify some tasks which are not very well supported by the Eclipse CDT. Some examples for these simplifications are:

- collectMemberVariables(ICPPASTCompositeTypeSpecifier struct)
- collectGslOwners(List¡IASTSimpleDeclaration¿ memberVars)
- getImplFromDeclaration(IASTSimpleDeclaration decl)
- getSpecialMemberFunction(ICPPASTCompositeTypeSpecifier struct, SpecialFunction desiredFunction)
- getVisibilityForStatement(IASTNode statement)
- and many more...

7.2 Future work

There's quite a lot of future work that can be done. The C++ Core Guidelines are still under development, but even with its current state, there are quite a lot

of rules that can be implemented. Especially the left out rules from the section "C.ctor: Constructors, assignments, and destructors" have a higher priority and can be implemented easier, since the analysis on each rule was already done in beforehand. But in general, the future work possibilities are almost endless. The C++ Core Guidelines are quite extensive and there's enough legacy code that can be replaced with C++1x alternatives.

7.3 Personal reflections

7.3.1 Özhan Kaya

Our bachelor thesis started a bit unconventional. Our supervisor P. Sommerlad was not present in the first 4 weeks and we also our first 3-4 weeks mainly consisted of improving our term project AliExtor. At a certain point where we were already 4 weeks in and haven't even really started with our actual project, I had some mixed feelings. But once we had a working prototype of a checker and a quick-fix working, I was motivated to start with the rule implementations. Thanks to our previous term project AliExtor, we already had experience when it comes to navigating through the AST and the nodes we had to analyze and modify. Therefore the implementation went quite fluently at first. Later on as we decided to add more and more features we encountered some issues with the Codan framework, which we managed to resolve by time. By week 9-10 we already had quite a few rules implemented and we were running on track. But at that point, we started implementing cross-file checkers as well as cross-file quick-fixes, making sure the implemented logic would work with both definitions and declarations, which turned out to be trickier than expected. Our implemented rules and helper methods had to be refactored. Also we had to work with the index and we kept on running into problems. At that point we decided to stop with new rule implementations for a while and focused on getting the existing logic to work with all known cases, which turned out to be a good decision.
Of course I was not always fully motivated during the bachelor thesis time. There have been times where I focused on our documentation for about 4-5 weeks straight, which got quite repetitive and demotivating after a while. But we had a good team organization and we tried to make sure that we brought some variety in the weekly tasks we did. Working with Kevin Schmidiger worked flawlessly like with our term project. We never had any big disagreements or conflicts which resulted in a good working atmosphere.

7.3.2 Kevin Schmidiger

At the start of our bachelor thesis our supervisor was absent. But he gave his co-advisor the information on what the project should be about. First of all we could improve our term project AliExtor and prepare ourselves for the main part. Because the thesis is based in a section in the C++ Core Guidelines this made it a very structured project.
Analyzing the rules was the hardest part for me. We had to understand the rule and what problems it want to avoid and how we possibly could implement it. It was hard because at the beginning the rules were hard to understand and because after improving the term project, getting back on documenting was a bit of a motivation killer. But after the analysis we could go back on implementing. Because we already had a done a term project on refactoring we could rapidly develop the first rule and get it to run with test cases and on the continuous integration server. We then slowly realized that we are not able to implement all of the rules in the given section. Instead we focused on making the rules we had as good as possible.
Somewhere in the middle of the project I was also moving in a new apartment. So I was a bit busy setting up my apartment. So I lost a bit time on the project. But this took only a week and then I was back on track but behind with time.
All in all while working on the project we never had any big problems. As already in the term project the team work was no problem with no disagreements. And problems could fast be solved with a short discussion.

A Usermanual

A.1 CCGlator

The checkers of the plug-in are default activated. So after the installation nothing further needs to be done. Once the developer is coding and a rule is violated the checker will be set. See figure 26 on page I.

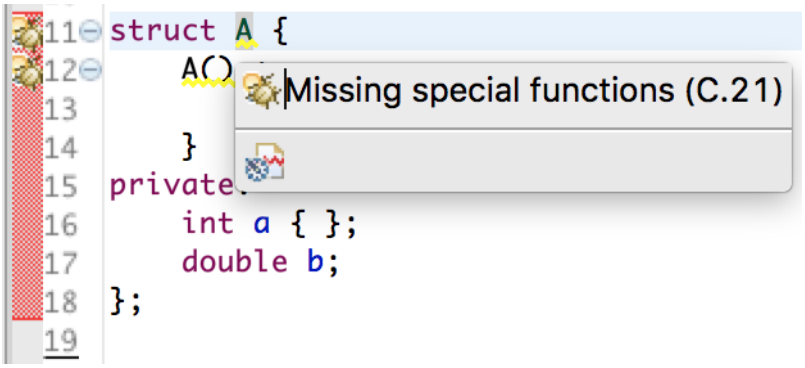

Figure 26: An example of a set checker

It can then be decided to quick-fix the violation. See figure 27 on page I

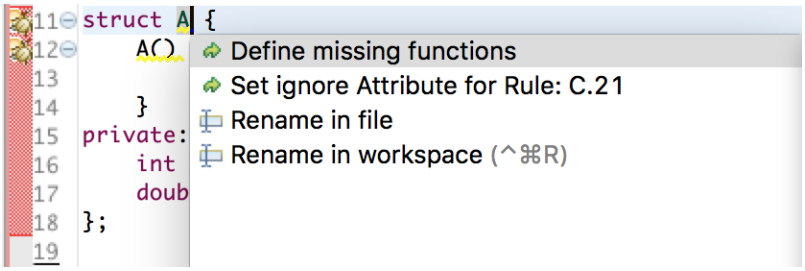

Figure 27: An example of possible quick-fixes

If the plug-in needs to be deactivated this can be done in the eclipse preferences. Go to C/C++ → Code Analysis. In the search bar look for Cpp Core Guideline Problems. See figure 28 on page II

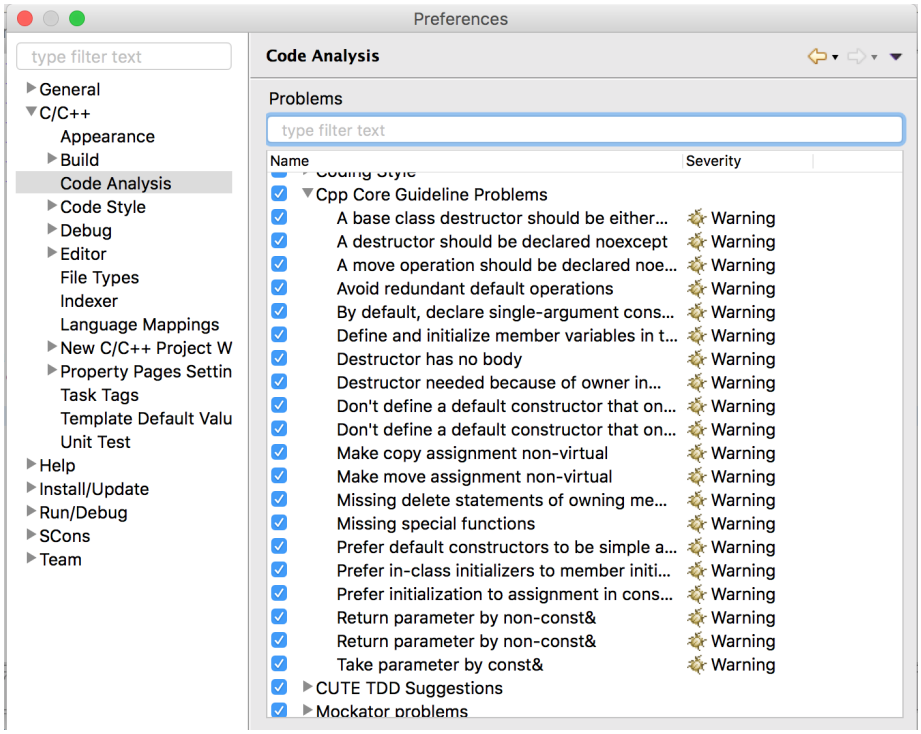

Figure 28: Configuration of the plug-in

The plug-in can here be partially or fully deactivated or even reactivated.

A.2 Demo Video

We've created a short video showing the main features of our plugin. This video is accessible on our final CD in the video folder as well as through a direct link on our server http://sinv-56012.edu.hsr.ch/.

B Project organization

In this section we introduce our project management. We shortly explain which technologies were used und what our time schedule looks like.

B.1 Local Development Environment

The plug-in was created on two Macbook Pros with OSX Yosemite 64bit. And for the development we used the software listed in table 8.

Software	Version
Eclipse Mars	4.5
TEXShop	3.56
OpenJDK	1.8.0_20
Apache Maven	3.3.3

Table 8: Table of used Software and Version

B.2 Continuous Integration Server

The continuous integration system used in this project was provided as a virtual server sinv-56012.edu.hsr.ch hosted at the University of Applied Sciences Rapperswil (HSR). The operating system of the virtual server is Ubuntu 14.04.3 LTS 64-Bit. To build the CCGLator plug-in and related artifacts the software listed in Table 10 was installed.

Software	Version
Apache	2.4.7
TEXLive	3.1415926
JRE	1.8.0_66-b17
Apache Maven	3.3.3
Redmine	2.4.2
Jenkins	1.641

Table 9: Table of used Software and Version

B.3 Project Plan

This section covers the time used for this bachelor thesis. The project is rewarded with 12 European Credit Transfer System (ECTS) points. For each point, 30 hours of work are estimated. This gives a total of 17 * 21.2 = 360 hours for the project per student.

B.3.1 Actual vs. Planned work hours

In Figure 29 the actual vs. the target hours/week are compared.

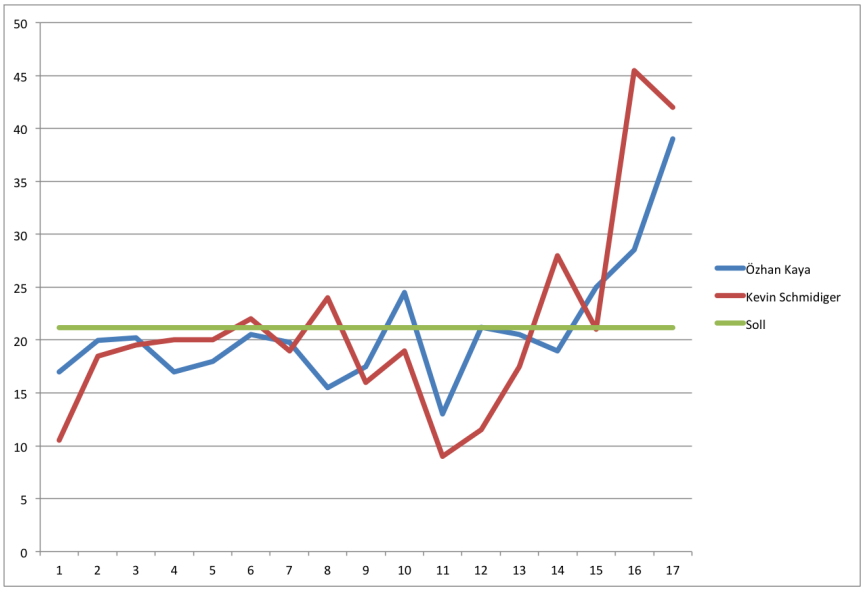

Figure 29: Actual vs. Target Hours/Week

During the project phase, we were always a bit below the expected working hours of 21.2h per week. This was caused by many events for example Kevin moving out of home into a new apartment, sickness and some other occasions. Though during the last 2 weeks, we managed to step up and more or less reach our final goal of 360h / student.

B.3.2 Hours spent per student

Together we spent 719 hours on our bachelor thesis according to our time tracking and project management tool Redmine. Here's how the distribution looked like:

Student	Hours
Kevin Schmidiger	363
Özhan Kaya	356

Table 10: Hours spent per student

References

[BS15a] Herb Sutter Bjarne Stroustrup. C++ Core Guidelines. https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md, 2015. [Online; accessed 17-June-2016].

[BS15b] Herb Sutter Bjarne Stroustrup. C++ Core Guidelines, section C: Classes and Class Hierarchies. https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#S-class, 2015. [Online; accessed 17-June-2016].

[BS15c] Herb Sutter Bjarne Stroustrup. C++ Core Guidelines, section C.ctor: Constructors, assignments, and destructors. https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#cctor-constructors-assignments-and-destructors, 2015. [Online; accessed 17-June-2016].

[cpp13] cppreference. Constuctors of std::vector. http://en.cppreference.com/w/cpp/container/vector/vector, 2013. [Online; accessed 17-June-2016].

[Cpp15] CppCon. CppCon 15. http://cppcon.org/2015program/, 2015. [Online; accessed 17-June-2016].

[ecl] eclipse. Eclipse CDT(C/C++ Development Tool). https://eclipse.org/cdt/. [Online; accessed 17-June-2016].

[Ecl15] Eclipse. CODAN Static Code Analysis. https://wiki.eclipse.org/CDT/designs/StaticAnalysis, 2015. [Online; accessed 17-June-2016].

[GF] Suter Toni Gonzalez Fabian. CharWars Documentation, year = 2014. https://eprints.hsr.ch/373/1/CharWars_eprint.pdf. [Online; accessed 17-June-2016].

[GF14] Suter Toni Gonzalez Fabian. CharWars Docu: Implementation. https://eprints.hsr.ch/373/1/CharWars_eprint.pdf#page=54, 2014. [Online; accessed 17-June-2016].

[Git16] Github. Github. https://github.com/, 2016. [Online; accessed 17-June-2016].

[Hin14] Howard Hinnant. Everything You Ever Wanted to Know About Move Semantics, Howard Hinnant, Accu 2014. http://de.slideshare.net/

ripplelabs/howard-hinnant-accu2014, 2014. [Online; accessed 17-June-2016].

[IEC14a] ISO / IEC. Function-try-block. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf#page=428, 2014. [Online; accessed 17-June-2016].

[IEC14b] ISO / IEC. Lvalues and rvalues. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf#page=92, 2014. [Online; accessed 10-June-2016].

[IEC14c] ISO / IEC. noexcept operator. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf#page=136, 2014. [Online; accessed 10-June-2016].

[ISO14a] ISO. Undefined behavior. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf, 2014. [Online; accessed 17-June-2016].

[ISO14b] ISOCPP. Value and Reference Semantics. https://isocpp.org/wiki/faq/value-vs-ref-semantics, 2014. [Online; accessed 17-June-2016].

[KO15a] Schmidiger Kevin Kaya Özhan. AliExtor Docu: Implementation. https://eprints.hsr.ch/479/1/aliextor.pdf#page=31, 2015. [Online; accessed 17-June-2016].

[KO15b] Schmidiger Kevin Kaya Özhan. AliExtor Student Research Project thesis. https://eprints.hsr.ch/479/1/aliextor.pdf, 2015. [Online; accessed 17-June-2016].

[Mac15] Neil Macintosh. GSL Microsoft. https://github.com/Microsoft/GSL, 2015. [Online; accessed 17-June-2016].

[OBS16] OBS. OBS Open Broadcast Software. https://github.com/jp9000/OBS, 2016. [Online; accessed 17-June-2016].

[PM16] Elias Geisseler Philipp Meier. GSLatorPtr, C++ Core Guidelines Pointer Checker and Support Library Refactorings. http://sinv-56033.edu.hsr.ch/jenkins/job/GslAtorPtr%20Documentation/ws/target/, 2016. [Online; accessed 17-June-2016].

[Sta16a] Stackoverflow. Bad design of operator=(). http://stackoverflow.com/questions/3279543/what-is-the-copy-and-swap-idiom, 2016. [Online; accessed 17-June-2016].

[Sta16b] Stackoverflow. Delete Operators. http://stackoverflow.com/questions/18046571/c-delete-array-memory-without-brackets-still-works, 2016. [Online; accessed 17-June-2016].

[sta16c] stan4j. STAN Structure Analysis. http://stan4j.com/, 2016. [Online; accessed 17-June-2016].

[Str14] Bjarne Stroustrup. Working Draft, Standard for Programming Language C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf, 2014. [Online; accessed 17-June-2016].

[Wik15] Wikibooks. Copy and Swap Idiom. https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Copy-and-swap, 2015. [Online; accessed 17-June-2016].

[Wik16a] Wikipedia. AST - Abstract Syntax Tree. https://en.wikipedia.org/wiki/Abstract_syntax_tree, 2016. [Online; accessed 17-June-2016].

[Wik16b] Wikipedia. Don't repeat yourself. https://en.wikipedia.org/wiki/Don%27t_repeat_yourself, 2016. [Online; accessed 17-June-2016].

[Wik16c] Wikipedia. January 1, 1970 special date. https://en.wikipedia.org/wiki/Unix_time, 2016. [Online; accessed 17-June-2016].

[Wik16d] Wikipedia. Memory leak. https://en.wikipedia.org/wiki/Memory_leak, 2016. [Online; accessed 17-June-2016].

[Wik16e] Wikipedia. Rule of three (C++ programming). https://en.wikipedia.org/wiki/Rule_of_three_(C%2B%2B_programming), 2016. [Online; accessed 17-June-2016].

[Wik16f] Wikipedia. STL Standard Template Library. https://en.wikipedia.org/wiki/Standard_Template_Library, 2016. [Online; accessed 17-June-2016].