

# Templator Reloaded

Modern C++ Template Code Analysis Tooling

University of Applied Sciences Rapperswil  
Master Thesis Fall Semester 2017  
Daniel Marty  
Supervised by Prof. Peter Sommerlad  
Technical Advisor: Thomas Corbat

# Abstract

C++ provides templates to build generic functions, classes, aliases and variables. Code resulting from template instantiations is not visible for the programmer. Seeing the code generated internally by the compiler helps to understand its behavior. To allow visualization of templates, a plug-in called Templator is used in the C++ IDE (integrated development environment) Cevelop. This plug-in was developed during a term project and a bachelor thesis in 2014 / 2015 at HSR.

However, not all template features were supported. Templates are very expressive which allows other usages than generic programming. Template meta-programming (TMP) is an example of its expressiveness and is used to perform computations at compile time. However, such code can be difficult to write and read. Various TMP libraries arose to offer help with writing it.

The main goal of this master thesis is to offer a tool for debugging template metaprograms on the basis of the Templator plug-in. This includes adding support for missing features and a new visualization concept that is able to handle template metaprograms. Additionally, the functionality of the plug-in is tested with examples written with the help of a modern library for template metaprogramming.

During the master thesis, support for non-type template parameters, alias and variable templates has been added. Furthermore, it is possible to trace overloaded operators, auto specifiers and normal classes. The user interface offers new features which ease the navigation and lead to a less overloaded view. In addition, examples using Boost.Hana, a modern TMP library, have been used to validate the implementations and to show its limitations.

# Management Summary

Cevelop is an IDE which extends Eclipse with many features. One of those features is the visualization of templates. This is provided by the plug-in Templator. It allows to browse through instantiation levels.

## Motivation

The Templator plug-in supports visualizing function templates which was developed during a student research project thesis [BJ14] and class templates that was later added during a bachelor thesis [BJ15]. However, features like alias templates and variable templates are not supported yet. Especially, the latter are widely used in C++ template metaprogramming, a technique to perform computations at compile-time. Therefore, the absence of such features makes it impossible to debug metaprograms with the Templator plug-in.

## Goals

The goals of this master thesis were to extend the functionality of the Templator plug-in to allow debugging template metaprograms. This includes following features:

- Allowing all names / overloaded operators to be traced.
- Adding support for alias templates, variable templates and class-template argument deduction.
- Reworking the Templator user interface to increase usability.
- Demonstrating that metaprograms with a recent library for metaprogramming can be analyzed with the extended Templator plug-in.
- Developing a view that is capable of showing all instantiations of a selected template (direct and indirect) inside the project.

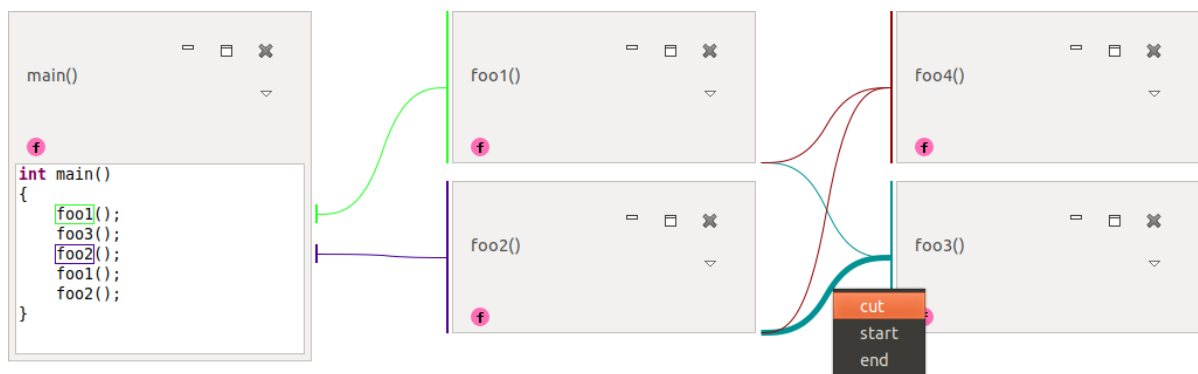
Optionally goals are:

- Extending the parser and indexer of Eclipse CDT to support C++17 class template argument deduction.
- Implementing a checker-quick fix to transform code without template argument deduction to code with it and vice versa.

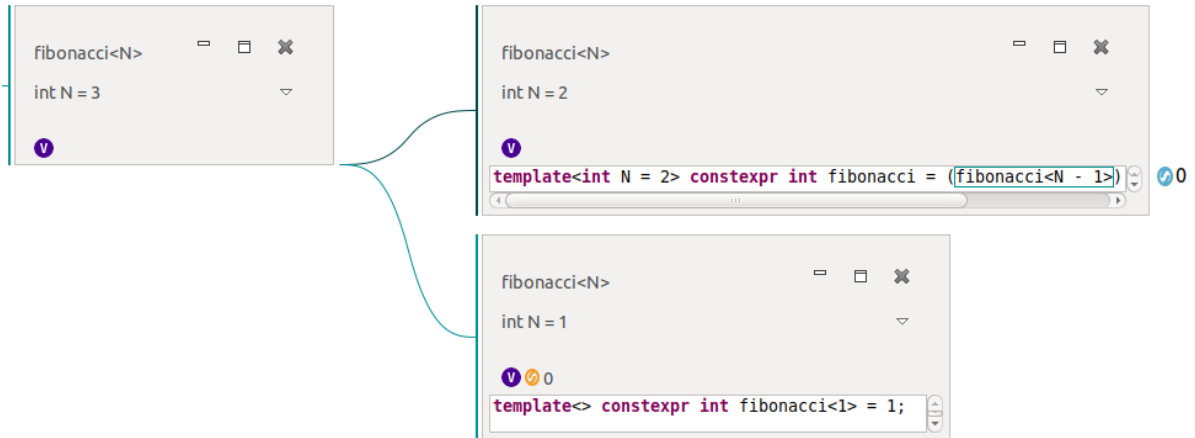
## Results

Almost all names and overloaded operators can be traced. The usage of auto specifiers does not prevent tracing anymore. Support for alias template and variable templates has been added. Non-type template parameters and template template parameters are replaced and are possible to be traced. Additionally, the Templator plug-in can also be used to trace code that is not templated now.

The user interface feels less overloaded and offers preferences to adjust to the users needs. Boxes around names are only shown if they are active and multiple entries can target the same destination. In addition, the navigation in the view was improved. A screenshot of the user interface with the new features is shown below.



Portals were added to travel to nodes that were opened in a previous or the same column. A portal is shown in the following screenshot.



# Declaration of Authorship

I declare that this report and the work presented in it was done by myself 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

---

Daniel Marty

# Contents

<b>1. Task Description</b>	<b>2</b>
1.1. Project Goals . . . . .	2
1.2. Expected Results . . . . .	3
1.3. Time Management . . . . .	3
1.4. Deliverables . . . . .	3
<b>2. Introduction</b>	<b>4</b>
2.1. Templates . . . . .	4
2.1.1. Function Templates . . . . .	4
2.1.2. Class Templates . . . . .	5
2.1.3. Alias Templates . . . . .	5
2.1.4. Variable Templates . . . . .	6
2.2. Specialization . . . . .	7
2.2.1. Primary Template . . . . .	7
2.2.2. Explicit Template Specialization . . . . .	7
2.2.3. Partial Template Specialization . . . . .	8
2.3. Template Parameters . . . . .	8
2.3.1. Type Template Parameter . . . . .	8
2.3.2. Non-Type Template Parameter . . . . .	9
2.3.3. Template Template Parameter . . . . .	9
2.4. Instantiation . . . . .	10
2.4.1. Template Arguments . . . . .	10
2.5. C++ Template Meta Programming . . . . .	12
<b>3. Templator Architecture</b>	<b>13</b>
3.1. Preparing the Data . . . . .	13
3.1.1. Opening the View . . . . .	14
3.1.2. Initializing the View . . . . .	16
3.1.3. Preparing Data for the View . . . . .	25
3.2. Displaying Data in the View . . . . .	30
3.2.1. User Interface . . . . .	30
3.2.2. Handling Click . . . . .	32
3.2.3. Opening Name . . . . .	33
3.2.4. Reflow . . . . .	36
3.3. Resolving . . . . .	37
3.3.1. Resolving Function Templates . . . . .	37
3.3.2. Resolving Class Templates . . . . .	39

<b>4. Analysis</b>	<b>41</b>
4.1. Tracing All Names and Operators . . . . .	41
4.1.1. Operators . . . . .	41
4.1.2. auto Specifier . . . . .	45
4.1.3. Classes . . . . .	51
4.1.4. Lambda Expressions . . . . .	51
4.1.5. Variable Templates . . . . .	52
4.1.6. Alias Templates . . . . .	53
4.1.7. Non-Type Template Parameter . . . . .	54
4.1.8. Template template parameter . . . . .	55
4.1.9. Incompatibility with CDT 9.4 . . . . .	56
4.2. User Interface . . . . .	57
4.2.1. Opening Already Opened TreeEntries . . . . .	57
4.2.2. Ordering of Entries . . . . .	60
4.2.3. Handling Events on Bézier Curves . . . . .	60
4.2.4. Default Width for TreeEntries . . . . .	61
<b>5. Implementation</b>	<b>62</b>
5.1. Tracing All Names and Operators . . . . .	62
5.1.1. Operators . . . . .	62
5.1.2. auto Specifier . . . . .	64
5.1.3. Classes . . . . .	73
5.1.4. Variable Templates . . . . .	74
5.1.5. Alias Templates . . . . .	75
5.1.6. Non-Type Template Parameter . . . . .	76
5.1.7. Template Template Parameter . . . . .	78
5.2. User Interface . . . . .	79
5.2.1. Multiple Entries . . . . .	79
5.2.2. Handling Events on Bézier Curves . . . . .	84
5.2.3. Ordering Entries . . . . .	85
5.2.4. Portals . . . . .	87
5.2.5. Hiding Rectangles . . . . .	89
5.2.6. InfoBar . . . . .	90
5.2.7. Templator Preferences . . . . .	91
<b>6. Testing</b>	<b>92</b>
6.1. TemplatorProjectTest . . . . .	93
6.1.1. Include Files . . . . .	93
6.1.2. Init Analyzer . . . . .	93
6.1.3. Init Top Level Definitions . . . . .	93
6.1.4. Properties . . . . .	94
6.2. TemplateResolutionTest . . . . .	95
6.3. TemplatorMultiLevelResolutionTest . . . . .	96
6.3.1. Properties . . . . .	97



6.4. TreeViewTest . . . . .	98
6.5. Sleak . . . . .	102
6.6. Analyzing Boost.Hana Examples . . . . .	103
<b>7. Conclusion</b>	<b>108</b>
7.1. Results . . . . .	108
7.2. Further Work . . . . .	108
<b>A. Package Structure</b>	<b>111</b>
<b>B. Templator Classes</b>	<b>114</b>
B.1. TemplatorPlugin . . . . .	114
B.2. ColorPalette . . . . .	114
B.3. RelevantNameCache . . . . .	115
B.4. ImageCache . . . . .	115
B.5. CursorCache . . . . .	115
B.6. ReflectionMethodHelper . . . . .	115
B.7. EclipseUtil . . . . .	115
B.8. ASTTools . . . . .	117
B.9. DefinitionFinder . . . . .	120
B.10. ASTAnalyzer . . . . .	120
B.11. NameTypeKind . . . . .	122
<b>C. Class Diagram TemplateArgumentMap</b>	<b>123</b>
<b>D. Class Diagram TemplateContextLookupData</b>	<b>124</b>
<b>E. Boost.Hana Switch Example</b>	<b>125</b>

# 1. Task Description

This chapter describes the task description for the project.

## 1.1. Project Goals

The C++ IDE Codeloop comes with the Templator plug-in that allows visualization of instantiated function and class templates and provides a drill-down clickable navigation to further investigate nested instances. However, the ultimate goal, to be a tool for debugging template meta programs, is not yet achieved. This Master thesis will further the Templator plug-in to allow visualization of newer template features (variable templates, alias templates, class template argument deduction) and more flexible navigation within the visualized template instantiation context. Furthermore, additional information on instantiated templates in a project is a good indicator for many projects that use templates intensively to see where potential problems are. In detail the following aspects should be resolved:

1. Improve Templator UI and navigation abilities
  - All names / overloaded operators should be possible to trace (at the moment it is only possible for class- and function templates).
  - Adding support for alias templates, variable templates, and class-template argument deduction that were added in C++11/14/17 respectively.
  - Enabling tracing all names / overloaded operators and adding other functionalities leads to an overloaded Templator view (Currently, all the clickable names are boxed. Allowing even more names to follow ends in a view full of boxes). Therefore, a new concept for the Templator view should be elaborated.
2. Plug-in (separate or as part of Templator) with a view to show template instantiations inside the project. All the instantiations of a selected template should be listed (direct and indirect). Following options to simplify navigation should be offered:
  - Show all instantiation positions
  - Show all specializations
  - Show all possible specializations prioritized with an information why they are not chosen (SFINAE).
  - This plug-in's ability should be cross checked against Templight's results <https://github.com/mikael-s-persson/templight>.

3. Extensions to the Eclipse CDT parsing infrastructure to support C++17 class template argument deduction (optional):
  - extension to the parser and indexer
  - visualize which constructor / template argument is chosen
  - Cevelp checker-quick fix to allow the transformation from code without template argument deduction to code with it and vice versa.
4. Demonstrate that at least one recent library for template meta programming (TMP) (Boost.hana, BRIGAND, Kvasir) can be analysed with the extended Templator and that Templator can be used to detect typical problems that users of the library face and that are hard to understand from just using a compiler. One possibility is to use Stackoverflow questions with respect to the selected TMP library and demonstrate how the problems could be solved through using the developed tooling.

## 1.2. Expected Results

Extended Templator Eclipse plug-in ready to be integrated into Cevelp (point 1). This plug-in or a separate plug-in allows the visualization of point 2 above.

Optional: A patch for CDT code base for supporting point 3 features and Cevelp plug-in (separate or integrated into Templator) for checker and quick-fix for point 3. All modified and extended code must be supported by automated tests that compile and run on a continuous integration server. The server demonstrates the project's progress by keeping significant build results during the semester. A technical report showing the results of the analysis of the above features of points 1,2, and 3; their design, implementation and testing as well as the result of the experiment position 4.

## 1.3. Time Management

The project started on 6.September 2017 and will end on 15.February 2018. The presentation will be held on 20.February 2018.

## 1.4. Deliverables

Following items will be delivered by the end of the project.

- 2 printed copies of the documentation
- 2 CDs that contain the code that was modified or added as well as the documentation with its sources.
- 1 CD with the documentation and abstract without personal information

## 2. Introduction

This chapter provides an introduction to templates and C++ template metaprogramming.

### 2.1. Templates

C++ is a typed language, that means variables, functions and most other entities use a specific type. Often, the code does look the same for different types. To not reinvent the wheel again and again, the language offers a solution to this problem called templates. Templates are written for one or more types that are not specified yet [Cppc]. The parametrization of a template is done by at least one template parameter. There are three different kinds of template parameters which are discussed in *section 2.3*. However, without instantiation which is explained in *section 2.4*, no code is generated from a source file that only contains template definitions.

#### 2.1.1. Function Templates

Parameterized functions that represent a family of functions are called function templates. In contrast to a normal function, they have types which are not known yet. In the example shown in *Listing 2.1*, the type parameter is T and both a and b must be of type T. What type T stands for is specified when using the function template and can be anything that supports the less than (<) operator [DV03, ch. 2.1].

```
1 template<typename T>
  T max(T const a, T const b) {
3     return a < b ? b : a;
  }
```

Listing 2.1: Function template that returns the maximum of two values

### 2.1.2. Class Templates

Class templates are similar to function templates but instead of a function a class is parameterized with one or more types. They are often used to implement container classes [DV03, ch. 3]. An example of a class template is shown in *Listing 2.2*.

```
1 template<typename A, typename B>
2 struct Pair {
3     A first;
4     B second;
5 };
```

Listing 2.2: Class template that stores two heterogeneous objects.

### 2.1.3. Alias Templates

In C++11, type aliases were introduced. Like typedefs, they refer to a previously defined type. They can be seen as a nickname for an already existing type and are often used to make code more readable by exchanging a long type name by a shorter name as shown in *Listing 2.3*.

```
1 #include <vector>
2
3 template<typename T>
4 struct MyAllocator {
5     // ...
6 };
7
8 typedef std::vector<int, MyAllocator<int>> VecInt;
9 using VecDouble = std::vector<double, MyAllocator<double>>;
```

Listing 2.3: Example of a typedef and a type alias

Besides the syntax, there is also another difference between type aliases and typedefs. Typedefs are not compatible with templates, whereas alias declarations are. An alias declaration that uses template parameters is called alias template (*Listing 2.4*). It represents a name that refers to a family of types [Cppd].

```
1 template<class T>
2 using Vec = std::vector<T, std::allocator<T>>;
```

Listing 2.4: Example of an alias template

### 2.1.4. Variable Templates

Variable templates were added to the language in C++14. The main purpose of variable templates is simplifying the definition and usage of parameterized constants as shown in *Listing 2.7*. Before they were introduced, workarounds like using constexpr static data members of class templates (*Listing 2.5*) or constexpr function templates that return the desired values (*Listing 2.6*) had to be used. However, those solutions required some additional syntactic salt.

```
1 #include <iostream>
2
3 template <typename T>
4 struct PI {
5     static constexpr T pi = T(3.141592653589793);
6 };
7
8 template <typename T>
9     constexpr T PI<T>::pi;
10
11 int main() {
12     std::cout << PI<int>::pi << '\n';
13 }
```

Listing 2.5: Workaround using a constexpr static data member

Using a constexpr static data member as shown in *Listing 2.5* requires a duplicated declaration. In addition to the one inside the class template (line 5), another is used (line 9) that provides the real definition if the constant is ODR (one definition rule) used [Zha].

```
1 #include <iostream>
2
3 template <typename T>
4 T PI() {
5     constexpr T pi = T(3.141592653589793);
6     return pi;
7 }
8
9 int main() {
10     std::cout << PI<int>() << '\n';
11 }
```

Listing 2.6: Workaround using a constexpr function template

The second workaround using a constexpr function template also adds some additional code. *Listing 2.7* shows the same example using a variable template which is much cleaner and easier to understand.

```

1 #include <iostream>
2
3 template <typename T>
4     constexpr T PI = T(3.141592653589793);
5
6 int main() {
7     std::cout << PI<int> << '\n';
8 }

```

Listing 2.7: Example using a variable template

## 2.2. Specialization

The purpose of a specialization is to provide another implementation for a special case. Such a specialization can be needed to optimize implementations or fix misbehaviors for certain types for an instantiation [DV03, ch. 3.3].

### 2.2.1. Primary Template

To be able to define a specialization, a primary template has to exist which A primary template is declared without the usage of template arguments inside the angle brackets which follow the template name. The example in *Listing 2.8* shows a primary template.

```

1 #include <iostream>
2 #include <string>
3
4 template <typename T, typename U>
5 struct A {
6     void foo() { std::cout << "generic \n" ; }
7 };

```

Listing 2.8: Primary template

### 2.2.2. Explicit Template Specialization

Because class templates cannot be overloaded a mechanism called explicit specialization was introduced (also called full specialization). Instead of template parameters which are unknown, it provides an implementation for an already implicitly declared template. This kind of specialization does not have to be related to the generic definition. It can contain member function or members that are not contained in the generic one. Only the name and the list of the specified template parameters have to match. Following example shows a specialization for T=int and U=int for the primary template figured in *Listing 2.8*.

```

1 ...
3 template <>
  struct A<int, int> {
5     void foo() { std::cout << "explicit specialization <int,int> \n"; }
  };

```

Listing 2.9: Explicit template specialization for  $T = \text{int}$  and  $U = \text{int}$

### 2.2.3. Partial Template Specialization

A partial template specialization is a way to specialize class templates. For this kind of specialization, only some template parameters are defined while others are still unknown. In the example shown in *Listing 2.10*, the primary template from above is partial specialized. Only the first template parameter is specialized. The template parameter  $U$  still stays unknown.

```

2 ...
  template <typename U>
4  struct A<int, U> {
      void foo() { std::cout << "partial specialization <int,U> \n"; }
6  };

```

Listing 2.10: Partial specialization for  $T = \text{int}$

## 2.3. Template Parameters

A template is parameterized by one or more template parameters. Each template parameter can be one of the three parameter kinds explained in this section.

### 2.3.1. Type Template Parameter

The keyword `typename` and `class` introduce a new type parameter as presented in *Listing 2.11*. For naming template parameters both keywords are equivalent [DV03, ch. 8.2].

```

  template <typename T>
2  struct A {
  };

```

Listing 2.11: Class template with a type template parameter  $T$



### 2.3.2. Non-Type Template Parameter

Template parameters are called non-type template parameters if they are not a type. The detail that is left open is a value instead. Non-type template parameters have the same syntax as a declaration of one of the types in the following list [IBMc]:

- integral or enumeration
- pointer to object or pointer to function
- reference to object or reference to function
- pointer to member

An example using a non-type template parameter is figured in *Listing 2.12*.

```
template <int N>
2 struct A {
    int a[N];
4 };
```

Listing 2.12: Class template with a non-type template parameter

### 2.3.3. Template Template Parameter

With template template parameters an additional level of abstraction can be introduced. A class template, for example, can be used as an argument itself instead of an instantiated class template.

```
template <typename K, typename V, template<typename> typename C>
2 class Map {
    C<K> key;
4    C<V> value;
};
```

Listing 2.13: Example using a template template parameter [Cppb]

## 2.4. Instantiation

Instantiation is the process of replacing template parameters by concrete types. The result of the instantiation is an instance of a template. If attempting to instantiate a template with types that do not support the operations in it, it results in a compile-time error [DV03, p. 12].

### 2.4.1. Template Arguments

To instantiate a template, every template parameter has to be replaced by a template argument. The arguments for function templates can be retrieved in following ways:

- explicitly provided:  
The function template `max` from *Listing 2.1* with explicitly provided template arguments looks like:  
**`max<int>(5,7);`**
- deduced from the context:  
**`max(5,7);`**
- defaulted: Defaulted template arguments are explained in 2.4.1.4.

The arguments for class templates can be explicitly provided, defaulted or deduced from the initializer which was added with C++17. However, Eclipse CDT does not support this feature yet [Rid17a]. Adding support for this feature is listed as an optional goal in item 3 in the task description specified in *section 1.1*.

#### 2.4.1.1. Template Type Argument

Only a type-id is a valid template argument for a type template parameter. *Listing 2.14* shows examples of valid template type arguments. The type-id also can be an incomplete type [Cppb].

```
1  template<typename T>
2  class A {
3  };
4
5  struct B;
6  using C = B;
7
8  int main() {
9      A<int> a1 { };
10     A<int()> a2 { };
11     A<B> a3 { };
12     A<B*> a4 { };
13     A<C> a5 { };
14 }
```

Listing 2.14: Template type arguments

### 2.4.1.2. Template Non-type Argument

The value of a non-type template argument can be retrieved at compile-time. That means that the arguments must be constant expressions, addresses of functions or objects with external linkage or addresses of static class members [IBMb].

### 2.4.1.3. Template Template Argument

Only id-expressions that names a class template or an alias template are allowed as template argument for a template template parameter.

### 2.4.1.4. Default Template Argument

For all kinds of template parameter, default template arguments can be specified in the parameter list after the = sign [Cppb]. The primary template in *Listing 2.15* has two defaulted template arguments. It is instantiated with int, double on line 8 with double, double on line 9 and with int, int on line 10.

```
template<typename T = int, typename U = double>
2 struct Pair {
    T first;
4    U second;
};
6
int main() {
8    auto pair1 = Pair<> { 4, 13 };
    auto pair2 = Pair<double> { 4, 13 };
10   auto pair3 = Pair<int, int> { 4, 13 };
}
```

Listing 2.15: Primary class template with two template parameters with default template arguments

## 2.5. C++ Template Meta Programming

The term metaprogramming names the generation or the manipulation of program code with the help of program code. There are different concepts do to that. C++ template metaprogramming deals with metaprogramming that is done by using C++ templates. [Ott09].

The discovery of C++ template metaprogramming almost happened by accident. One of the first metaprogram was an illegal code fragment whose error messages contained computed prime numbers.

The template system is Turing-complete. That means it can calculate anything that is computable. Template metaprogramming is like a functional language, it does not have side effects. The same patterns are used as in other functional languages [DA05]. However, because the template system was not designed for that, it can be quite confusing. To make template metaprogramming more user friendly different libraries arose.

## 3. Templator Architecture

In this chapter, an overview of the Templator plug-in is provided which helps to understand how the already offered functionalities are implemented. The Templator plug-in was developed during a term project [BJ14] and later improved during a bachelor thesis [BJ15]. Both were done by two students of the University of Applied Sciences Rapperswil, namely Jonas Biedermann and Marco Syfrig. The plug-in since then is part of Cevelop, an IDE that extends Eclipse CDT and is developed and maintained by the Institute for Software (IFS) in Rapperswil.

An overview of the package structure of the plug-in can be found in *Appendix A*. The purpose of this chapter is to show how the plug-in works. It is necessary to be able to perform the necessary code transformations to add additional functionalities. To achieve that three scenarios are analyzed. The first scenario is the preparation of the data for the view. The second scenario focuses on the user interface and helps to understand how the data which was prepared in the first scenario is visualized. Last, the third scenario deals with the instantiation of deferred templates.

### 3.1. Preparing the Data

Following brief description characterizes the scenario which is used to give an overview of the creation of the data. This data is later displayed in the user interface, which is explained in detail in *section 3.2*.

#### Preparing the Data

The user selects an IASTName in the editor and triggers the TreeTemplateView.

This scenario is divided into three parts. In the first part, the opening of the view is handled. The second part shows how the view gets initialized. Finally, the preparation of the data for the view is discussed in the third and last part.

### 3.1.1. Opening the View

The "Template Information" view can be opened in two different ways. It is possible to open the view over the context menu or by using the keyboard shortcut ALT + F8 (Option key + F8 on MacOS X). If the "Show In Template Information" entry (*Figure 3.2*) in the context menu is clicked, the class `TreeTemplateView`, which is an extension of the class `ViewPart` from the Eclipse CDT framework, is called. `ViewPart` is the base implementation for all workbench views and is subclassed to define new views. The keyboard shortcut is handled by the class `ShowTemplateInfoHandler` which implements the interface `IHandler`. A handler handles the execution of a command. Each command can have multiple handlers associated with it but only one can be active at the same time [C<sup>+</sup>17]. Both ways use the static method `showTemplateInfoUnderCursor` of the class `ViewOpener` to open and initialize the view. How the view is opened is also illustrated in the sequence diagram in *Figure 3.1*.

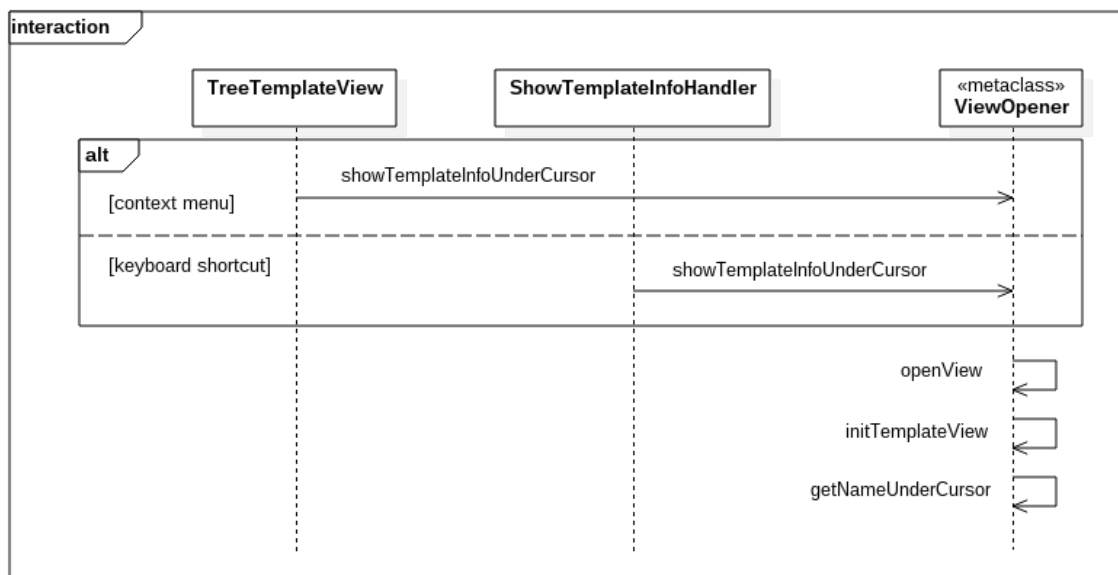


Figure 3.1.: Sequence diagram to show how the view is opened

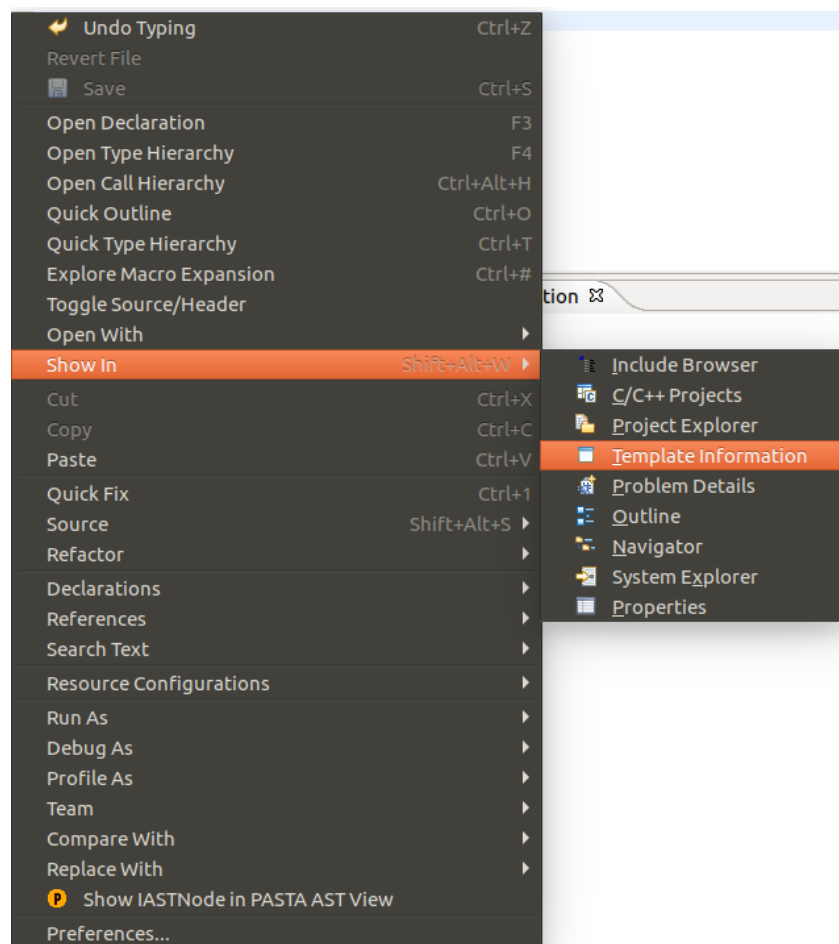


Figure 3.2.: Opening the view over the context menu

### 3.1.2. Initializing the View

First, the IASTName under the cursor is retrieved (this is done with the class EclipseUtils which is explained in *section B.7*). Not all the IASTNames can be used as a starting point for the visualization. The IASTName must not depend on a template argument and has to be fully resolvable by Eclipse CDT. If the selected IASTName satisfies this specification, the corresponding type identifier or function name is obtained. With the type identifier it is possible to retrieve the IASTName from the class template definition. At the end, a new ViewData instance is generated. This data is then set as the root data for the TreeTemplateView and a TreeEntry is created. However, this is not important for now and is later explained, in the next part, in 3.1.3. The diagram pictured in *Figure 3.3* shows the steps from the IASTName to the ResolvedName. Additionally, it shows where the classes and transitions are handled inside this report.

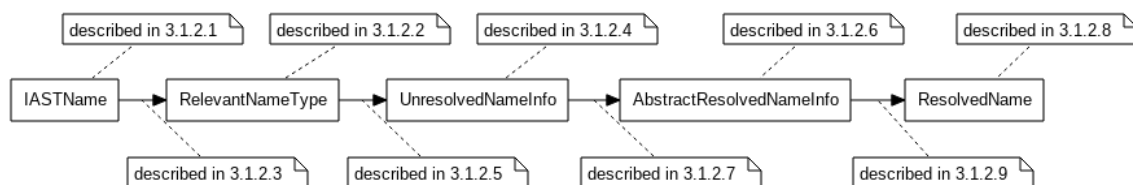


Figure 3.3.: Overview of the initialization with the additional information where each part is handled inside the report

#### 3.1.2.1. IASTName

The interface IASTName is part of the Eclipse CDT framework. An implementation of an IASTName represents a semantic object in the program. Each name holds a reference to an IBinding. All occurrences of a name (all IASTNames of the same identifier) references the same IBinding object. In our case, the IASTName is retrieved under the cursor in the editor. In the picture shown in *Figure 3.4* the retrieved name for the cursor would be main. The name does not have to be selected completely, the surrounding name can be retrieved.

```
1  #include "Point.h"
2
3  int main() {
4      Point<int> point { 1, 2 };
5  }
6
```

Figure 3.4.: Selecting an IASTName as starting point for the view in the editor



In 3.1.2, it was mentioned that some IASTNames cannot be used as a starting point. The explanation of what can serve as a starting point and what cannot as well as the example code (*Listing 3.1*) are taken from the bachelor thesis [BJ15]. The line numbers in the following lists refer to the lines of the example code.

```
1 #include <vector>
3 template<typename T>
4 class Stack {
5     std::vector<T> elems;
6 };
7
8 int main() {
9     Stack<int> mystack {};
10    mystack;
11 }
```

Listing 3.1: Example code to show what a valid starting point looks like

The following IASTNames cannot be used as starting points:

- line 1: The IASTName in `#include<vector>` marks an include which possibly contains type declarations and can thus not be selected. It does not identify a type or a function.
- line 3: `T` in `typename T` could potentially be a type that could be visualized but at this point the current context is unknown. `T` has not been substituted by anything at this point and just serves as a declaration for the template parameter.
- line 4: The name `Stack` is just the name of the class and the current context is not known. The name resolves to the template declaration itself and not to a template instance.
- line 5: `std::vector<T>` and `elems` both have the same type. The type specifier of `elems` is `std::vector<T>`, so they both resolve to an `ICPPDeferredClassInstance`. This means the IASTName depends on a yet unknown template argument for the parameter `T`.

The IASTNames in this lists are valid starting points:

- line 8: The function name `main` is a non-template-function and thus declares no template parameters. All further function calls and template instantiations in the body can be resolved by Eclipse CDT.
- line 9: `Stack<int>` and `mystack` have the same type. A template instance for `Stack<int>`. This time the current context is known as `T=int`.
- line 10: `mystack` has the same type as the two names from the previous line. It is a useless statement here but demonstrates a possible starting point.

### 3.1.2.2. RelevantNameType

RelevantNameType is an abstract class and is used to store the IASTName for the definition and the type as well as the corresponding bindings. A simplified class diagram is shown in *Figure 3.5*.

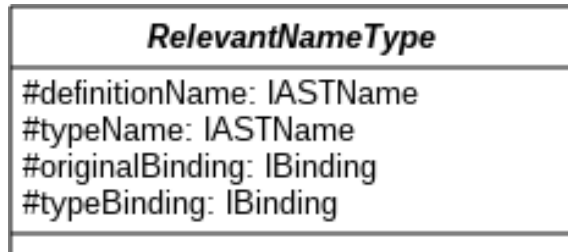


Figure 3.5.: Simplified class diagram of RelevantNameType

The definitionName represents the IAST-Name of the definition. If the member function `getFirst` is selected as a starting point in the example listed in *Listing 3.2*, the definition name for `intPair` on line 3 would be `intPair` on line 6. The other IASTName that is stored is the type name which would be `Pair<int,int>` from the same line. `Pair` is a simple class template that stores two heterogeneous objects. The class template is not listed here because the definition is not important, but it can be found in *Listing 3.3*.

```
1 struct A {
2     int getFirst() {
3         return intPair.first;
4     }
5 private:
6     Pair<int, int> intPair { 1, 2 };
7 };
```

Listing 3.2: Example code to show the difference of the original name and the definition name

How the type name is retrieved from the definition name is implemented in the subclasses of RelevantNameType which are figured in *Figure 3.6*. Depending on the binding of the definition name, the corresponding subclass is created. An `AlreadyRelevantType` is created if the definition name is also the type name. In this case, nothing has to be done, therefore it is already relevant as the name suggests.

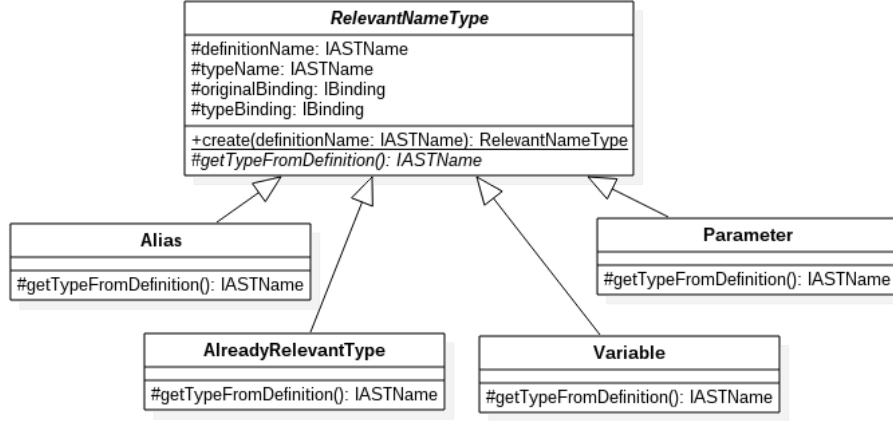


Figure 3.6.: Subclasses of RelevantNameType

### 3.1.2.3. IASTName → RelevantNameType

The transition from the IASTName to a RelevantNameType takes place in the static method `extractRelevantNameTypeFromName` of the class `NameDeduction` inside the package `plugin.asttools.resolving`.

During the initialization, unknown bindings are not accepted. A visitor is used to visit all the IASTNames. However, some implementations of IASTNames are a combination of multiple names. To avoid processing a name twice, some requirements have to be satisfied. In (3.1), the composition of a template id is shown. Only the template id as a whole should be processed. Therefore, the processing is skipped if the IASTName has an `ICPPASTTemplateId` as the parent. Another example is the `ICPPASTQualifiedName` which is shown in (3.2). In this case, the IASTNames are analyzed but the `ICPPASTQualifiedName` which contains those names is not.

$$\begin{array}{c}
 \text{IASTName} \\
 \underbrace{\text{SomeTemplate} < \text{int} >} \\
 \text{ICPPASTTemplateId}
 \end{array} \tag{3.1}$$

$$\begin{array}{c}
 \text{IASTName} \quad \text{IASTName} \\
 \underbrace{\text{Qualifier} :: \text{localName}} \\
 \text{ICPPASTQualifiedName}
 \end{array} \tag{3.2}$$

To speed up the whole deduction, a cache is used (*section B.3*). If the passed name is already available, the RelevantNameType is obtained from the cache. However, because it is the initialization, the passed cache is empty. To retrieve the RelevantNameType of an IASTName, the method extractResolvingName on the passed ASTAnalyzer (*section B.10*) is called. It first retrieves the definition name from the IASTName which is then passed to the factory method to create an instance of RelevantNameType. If the type name of the RelevantNameType could be retrieved and the target binding is relevant, the RelevantNameType is returned and added to the cache. A target binding is relevant if it is either a normal function or template dependent.

To get an idea what subclass of RelevantNameType is created, following code in *Listing 3.3* is provided and a mapping from IASTNames to the created RelevantNameTypes is shown in *Table 3.1*. The starting point to trigger the Templator view in this example is the function foo on line 14.

```

1  template<typename T, typename U> struct Pair {
    T first;
3   U second;
  };
5
  template<typename T>
7  Pair<T, T> makeSameTypePair(T value1, T value2) {
    return Pair<T, T> { value1, value2 };
9  }
11 using PairInt = Pair<int,int>;
13 void foo(int value1, int value2) {
    Pair<int,int> sameTypePair1 = makeSameTypePair(value1 , value2);
15   PairInt sameTypePair2 = sameTypePair1;
  }

```

Listing 3.3: Example code to show what subclass is created

#	IASTName	RelevantNameType
1	foo	AlreadyRelevantType
2	value1	Parameter
3	value2	Parameter
4	Pair<int,int>	AlreadyRelevantType
5	sameTypePair1	Variable
6	makeSameTypePair<int>	AlreadyRelevantType
7	value1	Parameter
8	value2	Parameter
9	PairInt	Variable
10	sameTypePair2	Variable

Table 3.1.: Found IASTNames and the corresponding resulting RelevantNameType

This example also shows two special cases. Even if there is a class that represents an alias, it is not used yet. Therefore #9 is a Variable and not an Alias. Further, the variable sameTypePair1 on line 15 is not listed at #11. This is because the RelevantNameType for it is already cached. The parameters value1 and value2 are not clickable because they are not relevant (not a function and not template dependent) which is also shown in *Figure 3.7*.

```

foo

void foo(int value1, int value2)
{
    Pair<int,int> sameTypePair1 = makeSameTypePair<int>(value1, value2);
    PairInt sameTypePair2 = sameTypePair1;
}

```

Figure 3.7.: Traceable IASTNames (boxed ones)

#### 3.1.2.4. UnresolvedNameInfo

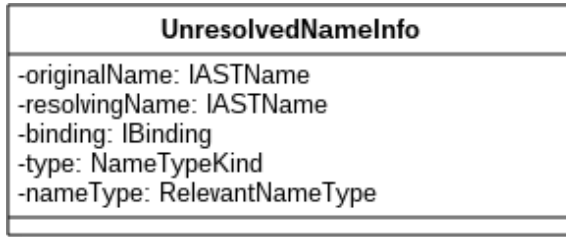


Figure 3.8.: Simplified class diagram of UnresolvedNameInfo

The UnresolvedNameInfo is a class used to store information about a name which was not resolved yet. This information is displayed in the simplified class diagram shown in *Figure 3.8*. The data contains two IASTNames, the original name which is the name that was found with the visitor and the resolving name which represents the name of the type. The binding represents the unwrapped type of the resolving name. Depending on the binding, an UnresolvedNameInfo holds an enum value (type) that shows information about the binding. The values for the enum NameTypeKind are listed in *Table B.2*.

#### 3.1.2.5. RelevantNameType → UnresolvedNameInfo

The transition from a RelevantNameType to an UnresolvedNameInfo is also done inside the class NameDeduction, but this time in the static method named createUnresolvedNameInfo.

First, a new instance of UnresolvedNameInfo is created with the IASTName found with the help of the visitor as an argument. If the RelevantNameType does not possess a type name, null is returned. If not, the binding on it is resolved and the ultimate type is retrieved with the help of the class SemanticUtils offered by Eclipse CDT. If the ultimate type is a type alias it is unwrapped. Only if the binding retrieved in this manner is relevant, the UnresolvedNameInfo is created. Relevant in the context of an UnresolvedNameInfo means that a NameTypeKind could be set for the binding (the binding corresponds to a type that is handled in the Templator plug-in).

### 3.1.2.6. AbstractResolvedNameInfo

The abstract class `AbstractResolvedNameInfo`, whose simplified structure is shown in the class diagram in *Figure 3.9*, resides in the package `plugin.asttools.data`.

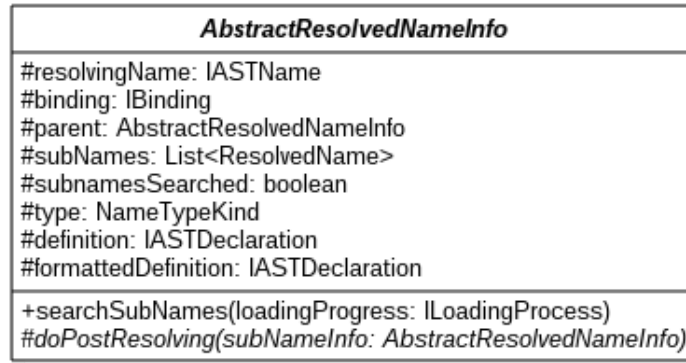


Figure 3.9.: Simplified class diagram for `AbstractResolvedNameInfo`

The resolving name, the binding and the type are taken from the passed `UnresolvedNameInfo`. The class offers a factory method that creates the concrete sub class depending on the `NameTypeKind` value. To be able to resolve template dependent parameters, the parent is also stored. Further, it contains a list of all the sub names. For a `FunctionCall` for example, that list contains all the relevant names in the body of the function. This step can be quite time consuming, therefore it is only done once. If the sub names were searched, the `subnamesSearched` flag indicates that it was already done and does not trigger a search again if requested. An overview of the class hierarchy is shown in *Figure 3.10*. The concrete classes offer the functionality to get the definition (`IASTDeclaration`) for its binding which is found and stored during creation. Later, another `IASTDeclaration` is stored which is used for the view and is modified as described in 3.1.3.2. The method `navigateTo` is used to jump to the location of the code in the editor.

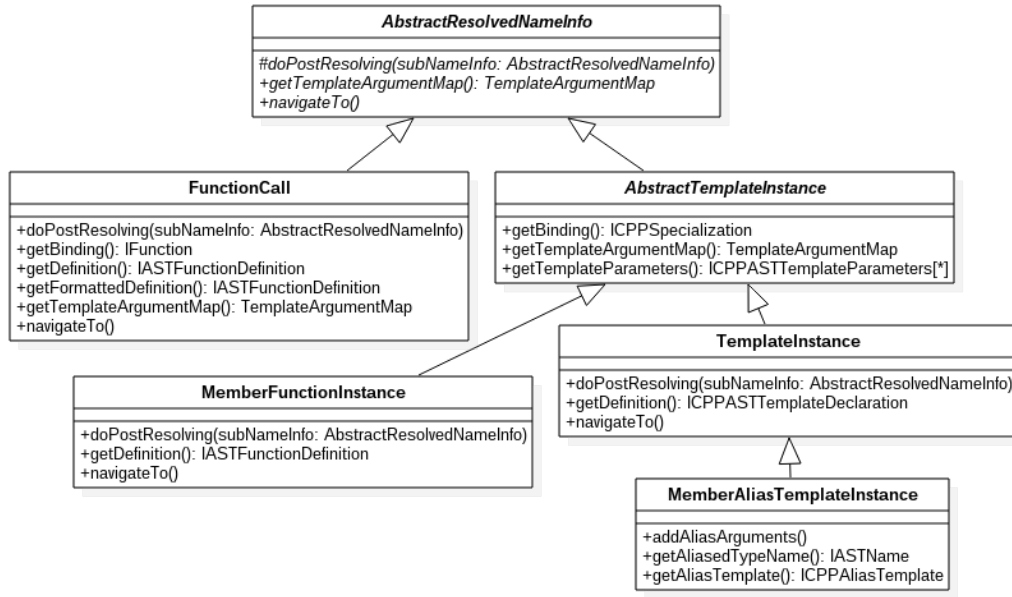


Figure 3.10.: Class hierarchy to model found statements

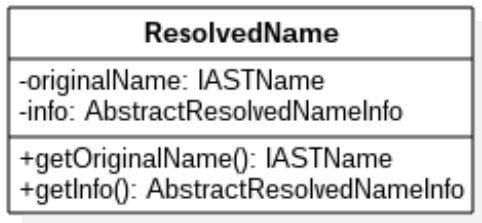
### 3.1.2.7. UnresolvedNameInfo → AbstractResolvedNameInfo

To create an `AbstractResolvedNameInfo`, the `UnresolvedNameInfo` is passed to the factory method `create` of the `AbstractResolvedNameInfo`. In a first step, the `PostResolver` resolves the binding of the `UnresolvedNameInfo` to its final binding. Only for deferred types the binding has to be resolved.

To resolve those bindings, the classes `FunctionCallResolver`(3.3.1) for deferred functions and `ClassTemplateResolver`(3.3.2) for deferred class templates are used. For all other `NameTypeKinds`, the `PostResolver` does not touch the binding stored inside the `UnresolvedNameInfo`. After the post resolving, the corresponding subclass of `AbstractNameInfo` is created depending on the binding which was set during the post resolving. If the `NameTypeKind` is a `FUNCTION`, a `FunctionCall` is created. In any other case, a subclass of `AbstractTemplateInstance` is created. This then again depends on the `NameTypeKind`.



### 3.1.2.8. ResolvedName



This class which is located in the `plugin.asttools.data` package is used to store the original `IASTName` and the corresponding `AbstractResolvedNameInfo` (*Figure 3.11*).

Figure 3.11.: Class diagram for `ResolvedName`

### 3.1.2.9. AbstractResolvedNameInfo → ResolvedName

The `ResolvedName` simply gets constructed with the `IASTName` that lies under the cursor and the prior created `AbstractResolvedNameInfo` as shown in *Figure 3.12*.

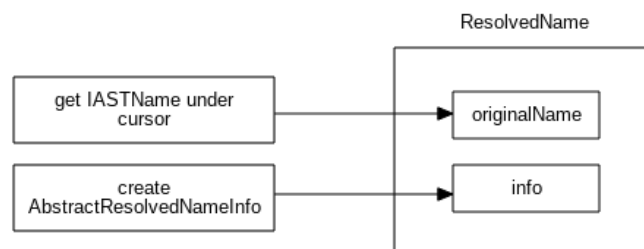


Figure 3.12.: Creation of a `ResolvedName`

## 3.1.3. Preparing Data for the View

During construction of a `TreeEntry`, an `AsyncEntryLoader` is created that is used for loading entries asynchronously. This allows preparing the information for the view without blocking Eclipse CDT. The `AsyncEntryLoader` holds an instance of `IAsyncLoadCallback` which is used to define what should be done asynchronously and what is done after the operation finished. An overview of this interface can be obtained from *Figure 3.13*. The `loadOperation` method inside the `TreeEntry` calls the `prepareForView` method on the `ViewData` instance. The `ViewData` is created after retrieving the `ResolvedName` which was explained in the previous step. The `ViewData` unites all the necessary information for the view (*Figure 3.14*).

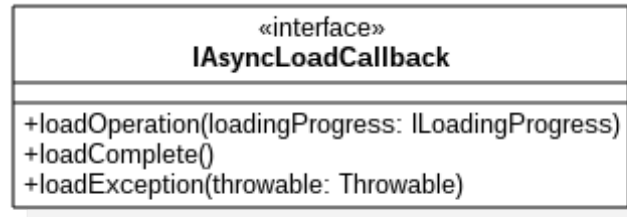


Figure 3.13.: Interface IAsyncLoadCallback

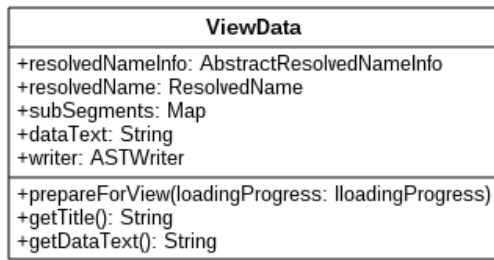


Figure 3.14.: Class ViewData

The sub names of the AbstractResolvedNameInfo in the ViewData have to be searched and resolved to know if they are traceable or not. After that, the code gets formatted and additional information is inserted. At last, the regions to handle the interactions of the user have to be retrieved. This process is also illustrated in *Figure 3.15*.

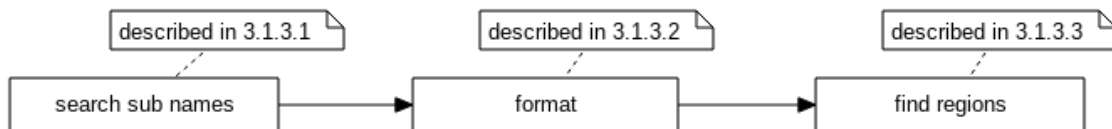


Figure 3.15.: Overview of the preparation for the view with the additional information where each part is handled inside the report

### 3.1.3.1. Search Sub Names

In 3.1.2, it is shown how the ResolvedName is created from the IASTName under the cursor of the editor. This ResolvedName deals as a starting point. The sub names of this starting point are searched. This is only done if the names are not already available (they could already be searched before).

A visitor is used to find all the IASTNames below the definition which was passed to construct the AbstractResolvedNameInfo. The visited IASTNames are then tried to be

used to create `UnresolvedNameInfos`. This is done in the same manner as described in 3.1.2, but there are two differences. First, the `RelevantNameTypeCache`(*section B.3*) is not empty anymore and second, unknown bindings are allowed now since the names do not represent starting points anymore. If an `UnresolvedNameInfo` could be obtained, it is checked if an entry for the `RelevantNameType` already exists in the cache. In case the entry exists, the corresponding `AbstractResolvedNameInfo` is obtained. If it is not available yet, the `UnresolvedNameInfo` is used to create an `AbstractResolvedNameInfo` with the help of the `PostResolver`.

Following example shows the post resolving. The member function `getFirst` returns the first element of the pair of type `T`.

```
template<typename T>
2 struct B {
    Pair<T, T> pair { 1, 2 };
4
    T getFirst() {
6         return pair.first;
    }
8 };

10 int main() {
    B<int> b { };
12    b.getFirst();
}
```

Listing 3.4: Code example to show the post resolving

In *Listing 3.4*, the function `main` on line 10 is used as the starting point for the Template plug-in. The sub name `getFirst` is opened and the name `pair` is clickable. Before the post resolving, the binding of the `UnresolvedNameInfo` of the name `pair` is a `CPPDeferredClassInstance` which is also represented by the `NameTypeKind DEFERRED_CLASS_TEMPLATE`. With the `TemplateArgumentMap` of the parent `AbstractResolvedNameInfo`, the dependent template arguments are replaced. After the replacement, the method is instantiated. How this is done is explained in 3.3.2. Therefore it is known that `Pair<T,T>` in this context means `Pair<int,int>` and the name is clickable which is also shown in *Figure 3.16*.

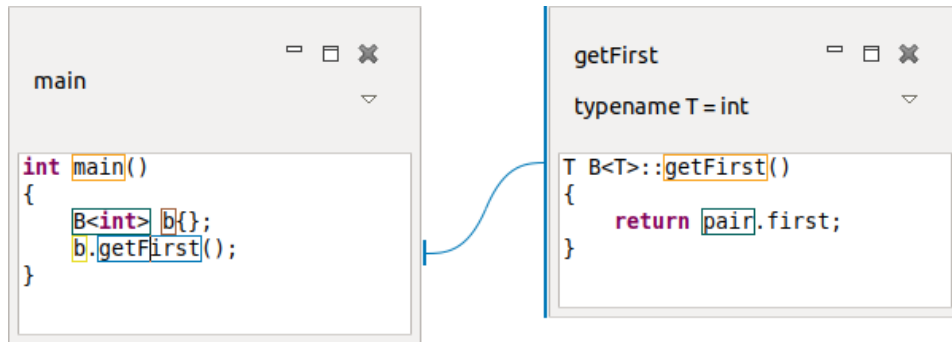


Figure 3.16.: `pair` is traceable because it could be resolved

Another situation arises if `getFirst` is taken as the starting point. `T` is unknown in this context and the deferred class template cannot be resolved. This is also visible in *Figure 3.17* where `pair` is not traceable anymore.

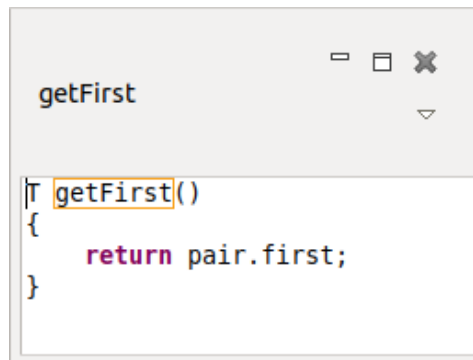


Figure 3.17.: `pair` is not traceable because it could not be resolved

### 3.1.3.2. Format

After the `AbstractResolvedNameInfo` instance is created and all the sub names are found, the stored definition is retrieved and copied. With the help of the `ASTTemplateFormatter`, the copy is modified. Basically, two things happen. The code is formatted to make it easier to read and the template parameters are filled in with default arguments as seen in *Figure 3.18*.

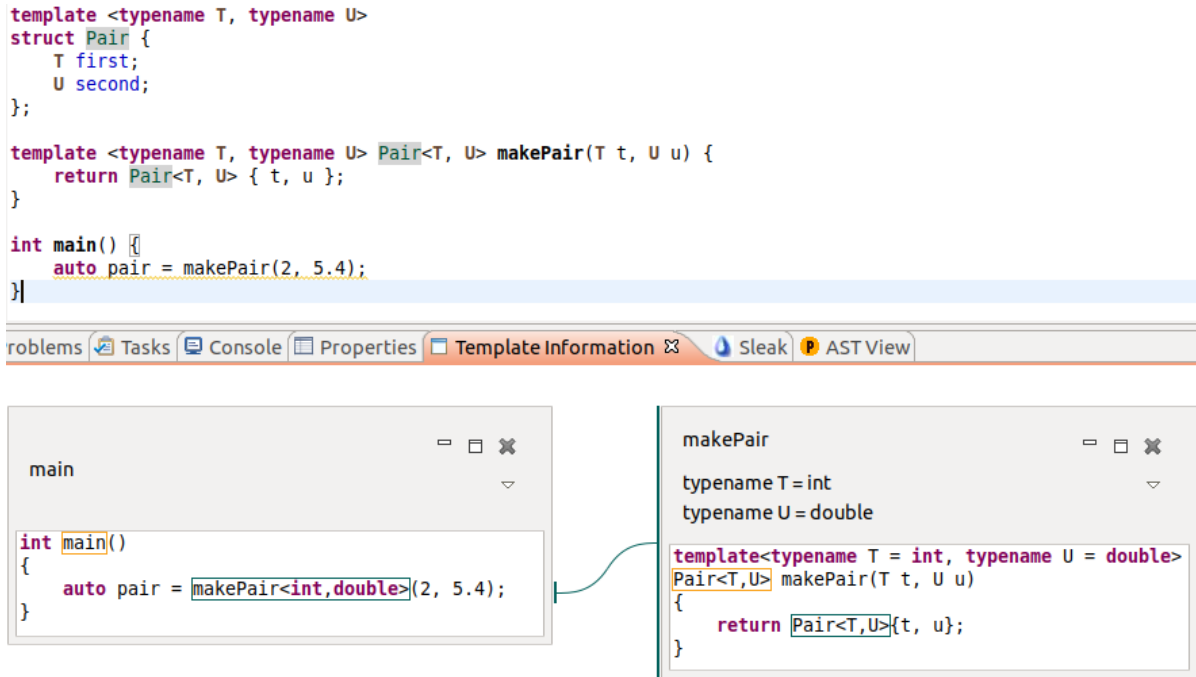


Figure 3.18.: Formatting of the code for the View

### 3.1.3.3. Find Regions

The problem is that the regions for drawing the rectangles are different after the formatting. Therefore, they have to be found in the formatted code again. An `ASTWriter` internally uses a `ChangeGeneratorWriterVisitor` for traversing the AST. To find the positions of the relevant `IASTNames` the `FindNodeRegionsVisitor` uses an `ASTWriter`. The offset can be retrieved from the scribe of the `ASTWriter`. Together with the length of the string which was written by the `ASTWriter`, it is enough to find the location in the formatted code. The structure to find the position of the names is also shown in the class diagram in *Figure 3.19*.

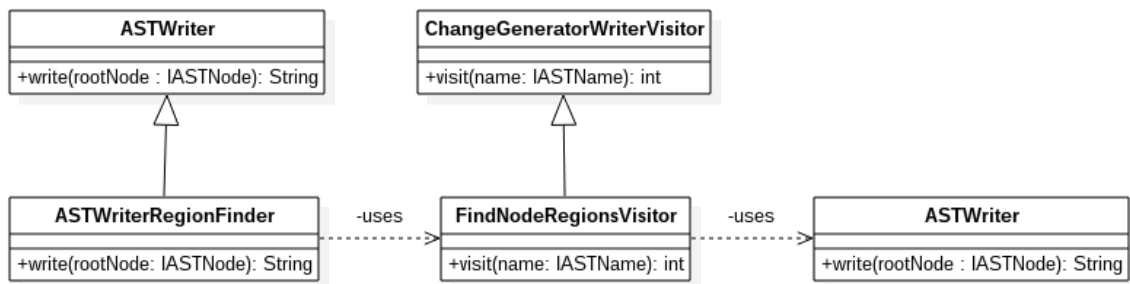


Figure 3.19.: Structure to find the names in the formatted definition

## 3.2. Displaying Data in the View

In *section 3.1*, the preparation of the data for the view is explained. This section discusses the process of displaying the data in the view.

Adding an entry to an already opened one.

An IASTName inside the SourceTextField (the text field of an entry) is clicked, which opens a new TreeEntry that is added to the TreeTemplateView

In 3.1.3, the ViewData was created and used to create the root entry inside the TreeTemplateView. The preparation of the data is done asynchronously with the AsyncEntryLoader which implements the IAsyncLoadCallback. After the data is prepared the method loadComplete is called where the components are created, the regions for the rectangles saved in a RectangleCollection and the layout calculated.

This scenario gives an overview of the user interface shows how the click on the rectangle is handled, how the TreeEntry is inserted into the TreeTemplateView and how it is rendered.

### 3.2.1. User Interface

Eclipse uses an open source widget toolkit for Java called SWT (Standard Widget Toolkit) [Fou17]. User interfaces often deal with similar tasks. The toolkit JFace is designed to work with SWT and handles those common UI-programming tasks [PH<sup>+</sup>10]. However, there is no widget that supports what is needed to build the functionality for the Templator plug-in. Therefore, the required functionalities were developed from scratch with the help of above mentioned toolkits.

To help finding the components in the code, following pictures show the classes responsible for certain user interface elements. The image shown in *Figure 3.20* displays the whole TreeTemplateView. Inside it, the TreeEntry elements are aligned like a tree. At the top, a GlobalToolBar is used to allow users to refresh, close, minimize and maximize the view.

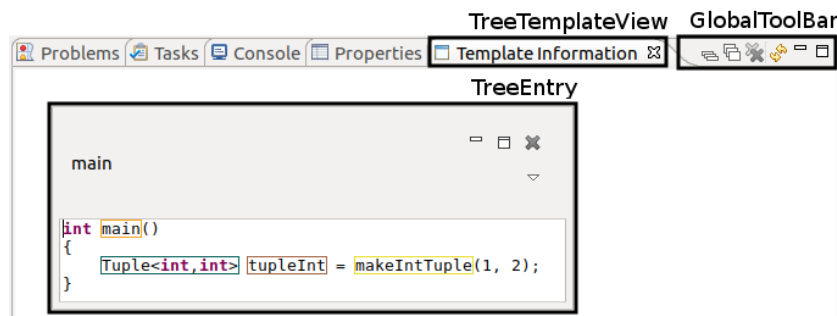


Figure 3.20.: Overview of the view

In *Figure 3.21*, a `TreeEntry` is displayed with another `TreeEntry` opened. A second `TreeEntry` is being loaded and a `LoadingBar` is shown during that time to indicate that something is done in the background.

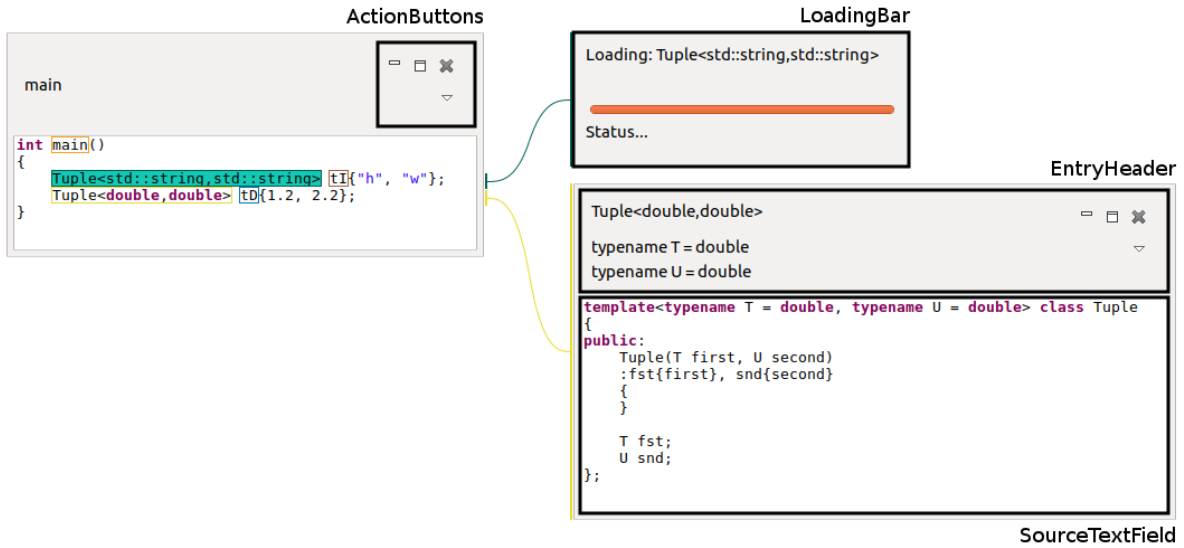


Figure 3.21.: Overview of the entries

The `EntryHeader` shows the name that was clicked as well as the template arguments. The corresponding definition is shown in a `SourceTextField` which is offering syntax highlighting. Sub entries are connected to their parents via Bézier curves. The names that are interactive (open a sub entry if clicked on it) are boxed, and the color of the Bézier curves indicate which box opened the sub entry.

At the top right of an entry, the `ActionButtons` can be used to minimize, maximize or close the entry. A pull-down menu gives access to even more features that can be applied, like opening a view to search or a view to show problems that occurred while generating the entry. However, they are not further explained here. A more detailed description of the view can be obtained in [BJ15] where all the classes and its base classes as well as the implementation of the resizing feature and other features which had to be developed from scratch, are explained in detail.

### 3.2.2. Handling Click

The RectangleCollection extends the MouseAdapter and implements the interface IRectangleHoverListener. If the mouse button is released, the hovered Rectangle is returned if the coordinates of the click are contained in any rectangle.

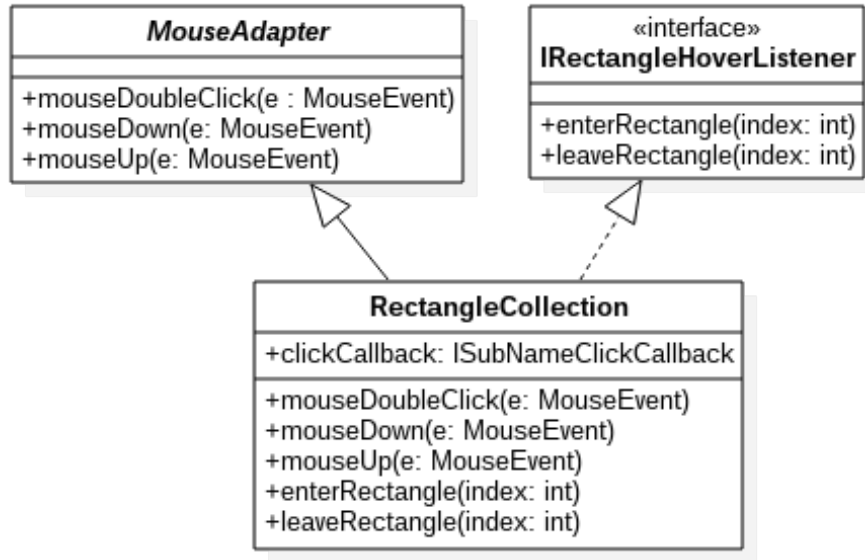


Figure 3.22.: Structure to handle interactions on a rectangle inside the RectangleCollection

The rectangles are saved as a stack to keep track of what color the rectangle must have. A click is handled the same way. It always sets the color of or handles the click on the innermost rectangle.

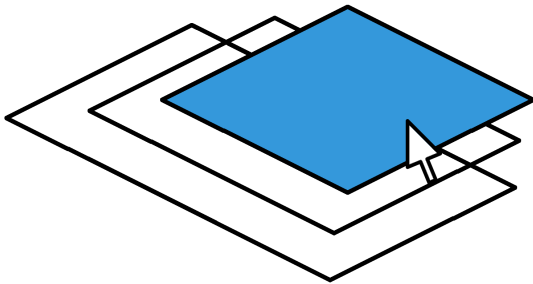


Figure 3.23.: Selecting innermost rectangle situation 1

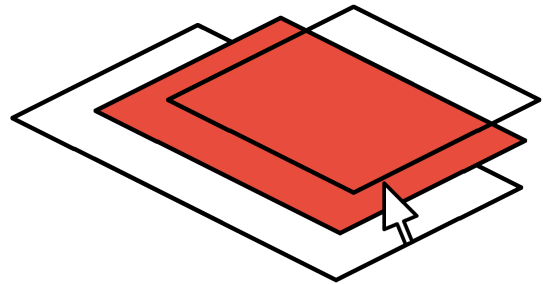


Figure 3.24.: Selecting innermost rectangle situation 2



If the click occurred on a Rectangle the method `nameClicked` on the `clickCallback` is called with the index of the Rectangle in the `RectangleCollection` and the information about which key was pressed as arguments. In this scenario, it is a left click which causes the name to be opened.

### 3.2.3. Opening Name

The class `TreeEntry` implements the interface `ISubNameClickCallback`. For a left click (`LEFT_CLICK`), the sub name data is retrieved from the `ViewData`. This is done by getting the `AbstractResolvedNameInfo` with the same index as the clicked `Rectangle`. After retrieving the `ViewData`, a sub entry is opened (or closed if it was already opened before). During the creation of the `TreeEntry`, the data is prepared with the `AsyncEntryLoader` as described in 3.1.3. The newly created `TreeEntry` is then added to the `TreeEntryCollection`. The `TreeEntryCollection` contains a `Map<TreeEntry, TreeEntryCollectionNode>` to retrieve the `TreeEntryCollectionNodes` and a `List<TreeSet<TreeEntry>>` that represents a list of entries per column.

A `TreeEntryCollectionNode` represents the entry that is shown in the view. It implements the interface `IConnection` that offers a method to navigate the tree and to find the origin (which rectangle was clicked and where it starts).

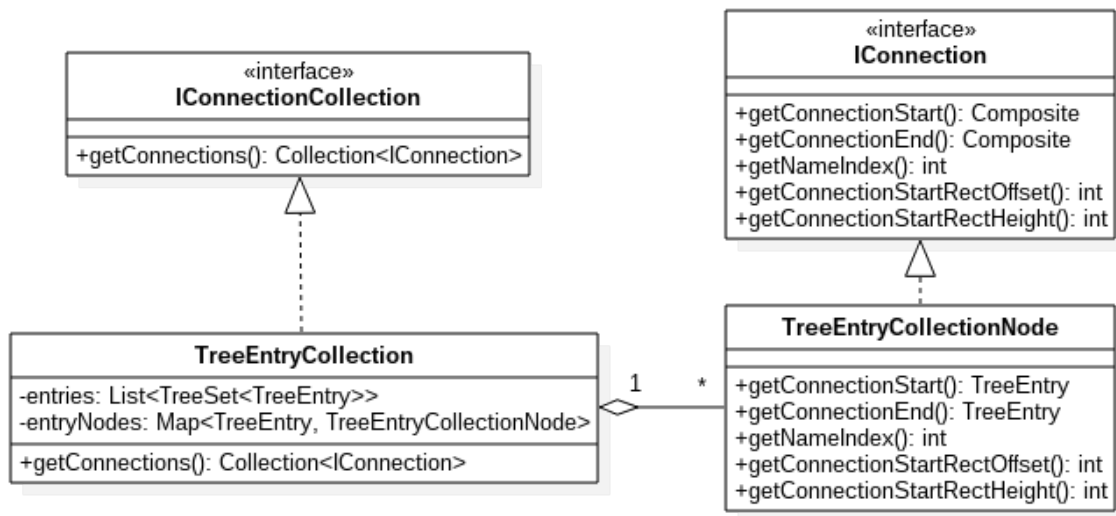


Figure 3.25.: Collection to save the nodes to present it as a tree

If a new entry gets added to the collection, the weight has to be calculated again. The weight is an array of integers that offers information about the ordering of the entries. With the new node, the calculated weight and the parent node, a `TreeEntryCollectionNode` is created.

The weights of the `TreeEntryCollectionNodes` are displayed in *Figure 3.26*. All the previous nodes are stored in the weight array, where the previous value represent the parent of the following value. The root node starts with an empty array. All the following nodes start with the array of the parent and add the position of their index. The length of the array represent the column number. For the weight array  $[2,1]$  for example the length is 2 which is identical to column 2 where the entry resides. If a column does not already exist, a new column is created by appending a new `TreeSet` to the list. A `TreeSet` is always ordered. For ordering the entries a simple comparison is used. The values in the array are compared from left to right. If the value is smaller the node is positioned above, if the value is bigger the position will be below the compared node. It is also possible that the entries have the same origin. In this case, the next value in the array is compared and the procedure is repeated.

This way, the connections will not cross and a unique order is guaranteed. All the weights are different. In the worst, case its decided by the last position, the index of the rectangle of an entry, which is unique.

$$[1] < [3] \tag{3.3}$$

$$[1, 2, 3] < [2, 1, 2] \tag{3.4}$$

$$[1, 3, 4] > [1, 3, 1] \tag{3.5}$$

In (3.3) and (3.4), the position of the entry in question can be retrieved by comparing only the first value in the weight array. For (3.5) this is not true. Only the last position in the weight array helps to sort the entries.

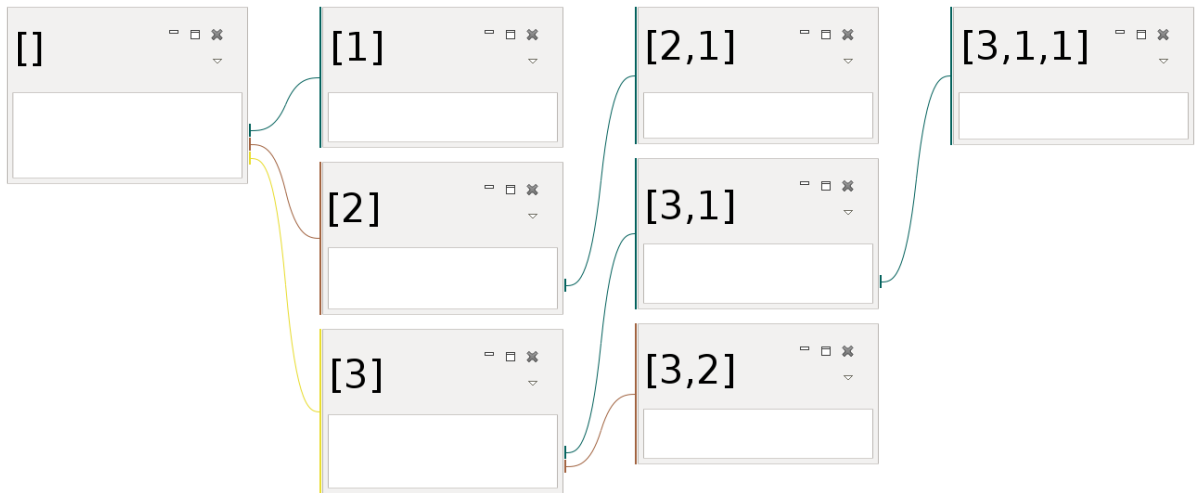


Figure 3.26.: Ordering of the entries

To handle a click on a sub name, the interface `ISubnameClickCallback` was introduced. The `TreeTemplateView` is implementing the interface `ITreeViewController` which offers methods to modify the view (*Figure 3.27*). The different actions are then handled with a switch statement inside `TreeEntry`.

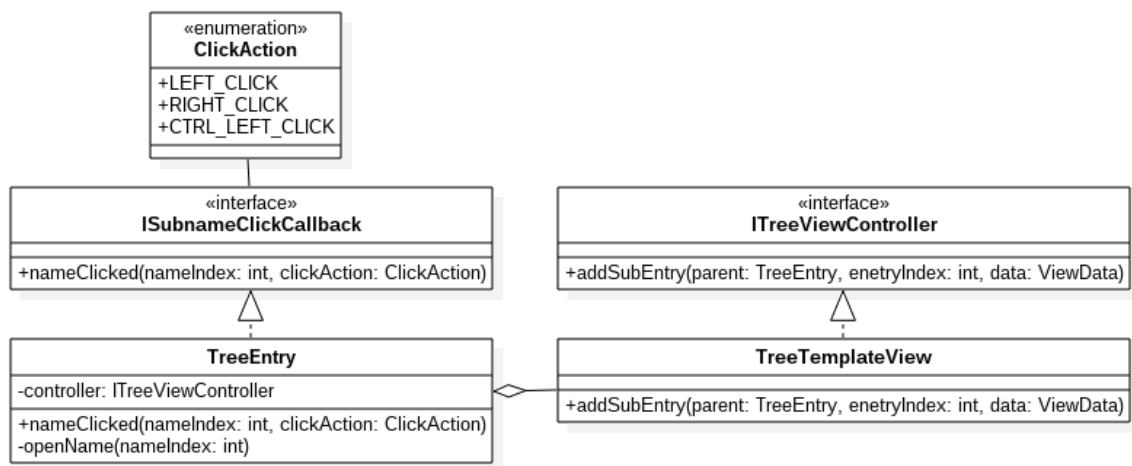


Figure 3.27.: Handling a click on a IASTName (not all methods of the interfaces are shown)

### 3.2.4. Reflow

The first step is to recalculate the layout. By iterating over the columns, the maximum size can be remembered to ensure that the nodes in the next column all start at a position which is not overlapping with nodes from a previous column. By implementing the interface `PaintListener`, a class provides methods that are generated when the control needs to be painted [dCR]. The `ConnectionRenderer` renders the Bézier curves and the lines. The rectangles around the traceable names are drawn by the `RectanglePaintListener` that also implements the `PaintListener` interface.

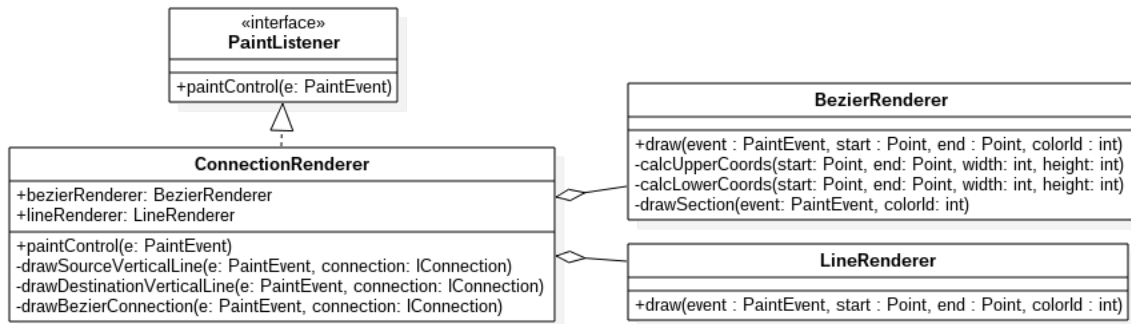


Figure 3.28.: Renderers to paint the lines between the nodes

## 3.3. Resolving

This section helps to understand how the resolving process works. It is important to understand how it was implemented to add support for newly added features inside the language.

### 3.3.1. Resolving Function Templates

Resolving function templates was implemented during the student research project thesis "Templator". Following parts will explain how function templates are resolved in the Templator plug-in.

#### 3.3.1.1. Getting the Function Call Expression

To resolve the function template, the `ICPPASTFunctionCallExpression` is needed. It is an ancestor of the `IASTName` and can be found with the help of `ASTTools` (*section B.8*).

#### 3.3.1.2. Resolving the Call

Resolving the `ICPPASTFunctionCallExpression` is done inside the **FunctionCallResolver**. The target binding of the `IASTName` of the `ICPPASTFunctionCallExpression` is retrieved. From there on three different cases can occur:

- Binding is an instance of `ICPPDeferredFunction`:  
The candidates for the deferred function are retrieved. If no candidates can be found, there is nothing that could be done and an exception is thrown. However, if there are candidates, the non-template function candidates are removed in case the function call has a template-id. In the following example the function on line 4 would be a better match but is not allowed because it has to be a function template call.

```
1 template<typename T>
  void foo(T value) { }
3
  void foo(bool b) {}
5
  int main() {
7      foo<>(true);
  }
```

It is already successful if only one candidate is left after the removal or it fails if no candidate is left anymore. The correct candidate has to be chosen if multiple candidates are still available. To do that, the `TemplateArgumentMap` from the parent `AbstractResolvedNameInfo` is used. A class diagram for the `TemplateArgumentMap` can be found in *Appendix C*.

To get the right binding, the `resolveFunction` method from `CPPSemantics` is used. The method uses an instance of `LookupData` (*Appendix D*). However, the `Templator` plug-in passes an extension of the class `LookupData`, that was developed for the plugin and is able to resolve calls that are dependent on template arguments.

- Binding is an instance of `IFunction`:  
The function is already fully resolved and can be returned.
- Anything else:  
The function call expression does not resolve to a function and an exception is thrown.

### 3.3.1.3. Instantiating the Call

If the resolved call is an instance of `ICPPFunctionTemplate`, it has to be instantiated. For instantiating, the class `InstantiateForFunctionCallHelper` is used. The template and function arguments as well as the function argument categories must be passed to instantiate it.

### Getting Template Arguments

A method to create an array of template arguments is offered by `CPPTemplates`. However, they are not resolved yet.

```
template<typename T, typename U>
2 void inner(T t, U u) {
3 }
4
5 template<typename T>
6 void outer(T t, int i) {
7     inner<T>(i, t);
8 }
9
10 int main() {
11     outer(5.5, 5);
12 }
```

Listing 3.5: Deferred function (inner)

In the example shown in *Listing 3.5*, the created template argument array for `inner` is `[T]`. It has to be replaced with the corresponding argument stored inside the `TemplateArgumentMap` of the parent `AbstractResolvedNameInfo`. In this example that means `T` is replaced by `double`.

### Getting Function Arguments

The function arguments can be obtained from the function call expression. For `inner` in *Listing 3.5*, the arguments are `[i,t]`. If a function argument is a template argument, it has to be replaced as well. Hence, `t` gets replaced by `double`.

## Getting Argument Categories

In C++, every expression possesses a value category. To select the correct function overload, this category is considered. To find the best matching overload the value categories have to be stored for each function argument as well [BJ15].

## Instantiating

Instantiating is done with the `instantiateForFunctionCall` from `CPPTemplates` which is used via reflection.

### 3.3.2. Resolving Class Templates

To resolve a class template, already in Eclipse CDT implemented functionality is used. The `instantiateClassTemplate` method in `ClassTemplateResolver` is a copy of the method `createBindings` in `CPPTemplates`. The method is modified to allow replacing dependent types. All the private methods and fields which are used inside the method are accessed via reflection. This situation is shown in *Figure 3.29*. The Templator plug-in uses a class `ReflectionMethodHelper` to ease the handling of reflection. However, it is important to keep in mind that this structure is very vulnerable to changes.

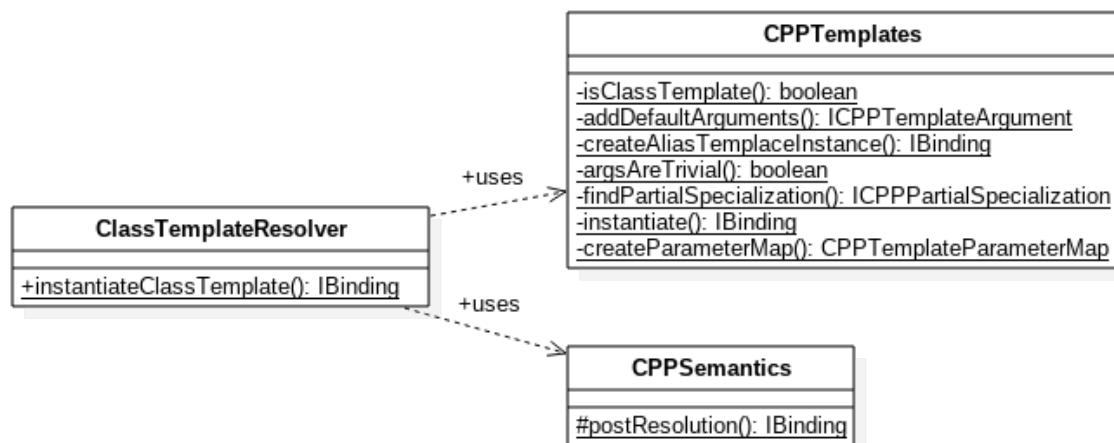


Figure 3.29.: `ClassTemplateResolver` uses methods from `CPPTemplates` and `CPPSemantics`. Private methods are used via reflection. The parameters were left out for layout reasons.

### 3.3.2.1. Replacing Dependent Template Arguments

While replacing dependent template arguments three different cases can occur [BJ15]:

- Template argument is a `ICPPTemplateParameter`:  
The corresponding argument is retrieved from `TemplateArgumentMap` of the parent `AbstractResolvedNameInfo`.
- Template argument is deferred and a template id:  
All the `ICPPASTTemplateId` get recursively instantiated to get the ultimate type for these nested template-ids.
- Template argument is deferred but not a template id:  
The type is recursively retrieved from parameters, variables, alias types etcetera.



## 4. Analysis

This chapter analyses the problems that occur with the current implementation in the Templator plug-in. Problems in this case describe something that is not functioning as well as it should be, either because the functionality was not yet available in Eclipse CDT while the plug-in was developed or because it simply was not implemented due to prioritization. Another problem area is the user interface where the usability still can be improved. All those problems are described in this chapter. The solutions are provided in *chapter 5*.

### 4.1. Tracing All Names and Operators

The Templator plug-in does not allow tracing all the names. In C++, it is possible to overload operators (ad-hoc polymorphism). While normal functions can be traced, overloaded operators cannot. Types can be traced if they are template dependent. Often types like that get big really fast and the auto specifier is used for convenience. In terms of the Templator plug-in this is not optimal because it is not possible to follow an auto specifier. Because of time issues, alias templates were not implemented during the previous works. Variable templates were not supported by Eclipse CDT at the time and therefore are not supported yet. This section deals with all the names (or declaration specifiers in case of auto) that are not supported.

#### 4.1.1. Operators

All the operators that can be overloaded are shown in *Table 4.1*. The table shows the operator as well as the subclass of the IASTNode which is used to represent it in the abstract syntax tree of a C++ program in Eclipse CDT. To see how the expressions are represented in the AST, the plug-in "pASTa" (Painless AST Analysis) can be used [IFS]. *Figure 4.1* shows that the operator / is not clickable and the user is therefore not able to follow it. The code for this example is listed in *Listing 4.1*. In addition to the operators shown in the table, there are other operators that can be overloaded which are handled later.

```

template<typename T, typename U>
2 struct Pair {
    T first;
4    U second;
};

6
template<>
8 struct Pair<int, int> {
    Pair<double, double> operator/((const Pair& rhs) {
10     double first { (double) (first) / rhs.first };
    double second { (double) (second) / rhs.second };
12     return Pair<double, double> { first, second };
    }

14
    int first;
16    int second;
};

18
int main() {
20     Pair<int, int> p1 { 1, 2 };
    Pair<int, int> p2 { 3, 4 };
22     Pair<double, double> p3 { p1 / p2 };
}

```

Listing 4.1: Example code

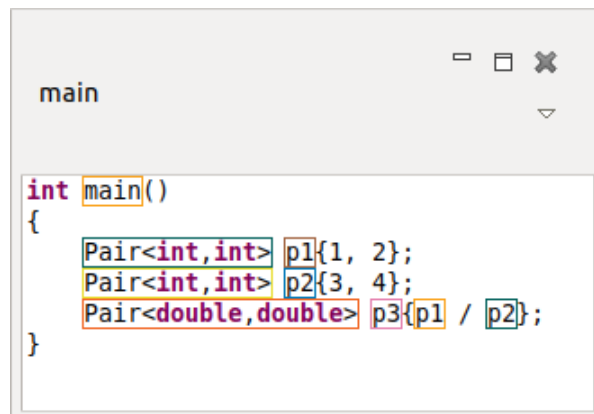


Figure 4.1.: operator/ is not clickable

operator	type
+	CPPASTBinaryExpression
-	CPPASTBinaryExpression
*	CPPASTBinaryExpression
/	CPPASTBinaryExpression
%	CPPASTBinaryExpression
^	CPPASTBinaryExpression
&	CPPASTBinaryExpression
	CPPASTBinaryExpression
!	CPPASTUnaryExpression
=	CPPASTBinaryExpression
<	CPPASTBinaryExpression
>	CPPASTBinaryExpression
+=	CPPASTBinaryExpression
-=	CPPASTBinaryExpression
*=	CPPASTBinaryExpression
/=	CPPASTBinaryExpression
%=	CPPASTBinaryExpression
^=	CPPASTBinaryExpression
&=	CPPASTBinaryExpression
=	CPPASTBinaryExpression
<<	CPPASTBinaryExpression
>>	CPPASTBinaryExpression
>>=	CPPASTBinaryExpression
<<=	CPPASTBinaryExpression
==	CPPASTBinaryExpression
!=	CPPASTBinaryExpression
<=	CPPASTBinaryExpression
>=	CPPASTBinaryExpression
&&	CPPASTBinaryExpression
	CPPASTBinaryExpression
++	CPPASTUnaryExpression
-	CPPASTUnaryExpression
,	CPPASTExpressionList
- > *	CPPASTBinaryExpression
- >	CPPASTFieldReference
( )	CPPASTFunctionCallExpression
[]	CPPASTArraySubscriptExpression

Table 4.1.: Overloadable operators in C++ and their corresponding node in the AST.

#### 4.1.1.1. Memory Management Operators

To customize allocation and deallocation, the memory management operators can be overloaded. The operators to do this are called **new** and **delete**.

```
1 #include <iostream>
3 struct X {
4     static void* operator new(std::size_t sz) {
5         std::cout << "custom new for size " << sz << '\n';
6         return ::operator new(sz);
7     }
8
9     static void* operator new[](std::size_t sz) {
10        std::cout << "custom new for size " << sz << '\n';
11        return ::operator new(sz);
12    }
13 };
15 int main() {
16     X* p1 = new X;
17     delete p1;
18 }
```

Listing 4.2: Example of overloaded memory management operators[]

#### 4.1.1.2. User-Defined Literals

User-defined literals are also possible to be overloaded. They were introduced in C++11 and are primarily used for convenience or for compile-time type deduction [Cppe]. An example where a user-defined literal is used as a conversion between degrees and radians is shown in *Listing 4.3*.

```
constexpr long double operator"" _deg ( long double deg ) {
2     return deg * 3.141592 / 180;
3 }
4
5 int main() {
6     double rad = 90.0_deg;
7 }
8 }
```

Listing 4.3: Example of a user-defined literal [Cppe]

### 4.1.1.3. User-Defined Conversions

To enable implicit or explicit conversions from a class type to another type, user-defined conversions are used. It is also possible to overload them.

```
1 struct X {  
    operator int() const { return 7; }  
3 }  
  
5 int main() {  
    X x { };  
7    int i = x;  
}
```

Listing 4.4: Example of a user-defined literal

### 4.1.2. auto Specifier

Since C++11 (in earlier versions the `auto` keyword served another purpose), it is possible to deduce the type of a variable from its initializer. A function can have a trailing return type as shown in *Listing 4.7* or since C++14 it is also possible to deduce the type from its return statements [Cppa]. Examples of variable and function definitions using auto deduction are shown in *Listing 4.5*.

```
template<typename T>  
2 auto makeSameTypePair(T value1, T value2) {  
    return Pair<T, T> { value1, value2 };  
4 }  
  
6 auto makeIntPair(int value1, int value2) {  
    return makeSameTypePair(value1, value2);  
8 }  
  
10 int main() {  
    auto p1 = Pair<int, int> { 1, 2 };  
12    auto p2 = makeSameTypePair(3, 4);  
    auto p3 { p1 / p2 };  
14    auto p4 = makeIntPair(4, 3);  
}
```

Listing 4.5: Example using auto deduction. (The definition of `Pair` was taken from *Listing 4.6*)

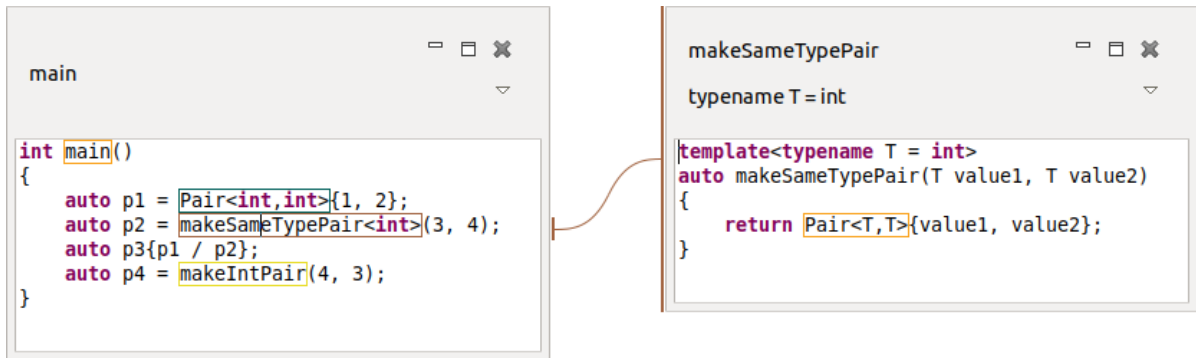


Figure 4.2.: auto specifiers are not clickable

In 3.1 the creation of an `AbstractResolvedNameInfo` is explained. For the whole process to work, an `IASTName` is needed for which its type name can be obtained. The type name is needed to get to the definition of the type. The definition then again is shown in the `TreeEntry` that gets opened by clicking on the rectangle. However, this is not possible for a variable whose type is retrieved via auto deduction. Also the declaration specifier itself is not found because it is not an `IASTName` (only `IASTNames` are visited). This means the functionality to get an `IASTName` for an auto specifier has to be added. First, it is important to know where the auto specifier can be used and how an `IASTName` can be retrieved.

#### 4.1.2.1. Definition of Variables

There are three different methods to initialize a variable in C++.

$$\textit{type identifier} = \textit{initial\_value} \quad (4.1)$$

$$\textit{type identifier} ( \textit{initial\_value} ) \quad (4.2)$$

$$\textit{type identifier} \{ \textit{initial\_value} \} \quad (4.3)$$

The method shown in (4.1) is inherited from the C language and therefore commonly known as c-like initialization. (4.2) represents the constructor initialization. The third method to initialize variables, known as uniform initialization, was introduced with C++11 and is shown in (4.3). Those three methods are equivalent [cpl]. In Eclipse CDT they are represented as CPPASTEqualsInitializer (4.1), CPPASTConstructorInitializer (4.2) and CPPASTInitializerList in case the method in (4.3) is used.

The initial value can be retrieved from following expressions (listed are the names of the C++ implementations in Eclipse CDT):

- CPPASTCastExpression
- CPPASTSimpleTypeConstructorExpression
- CPPASTFunctionCallExpression
- CPPASTConditionalExpression
- CPPASTBinaryExpression
- CPPASTUnaryExpression
- CPPASTIdExpression
- CPPASTFieldReference
- CPPASTLiteralExpression
- CPPASTArraySubscriptExpression
- CPPASTLambdaExpression

#### 4.1.2.2. Definition of Functions

There are two different concepts to declare functions using the auto decl specifier which are explained in the following.

##### Trailing Return Type

The example shown in *Listing 4.6* shows that the trailing return type has to be analyzed. If only the body of the function is analyzed, the return type will be `Pair<int,int>`. However, because the trailing return type is `Pair<double,double>` a cast is performed.

```
1  template<typename T1, typename T2>
   struct Pair {
3  };

5  template<>
   struct Pair<int, int> {
7      operator Pair<double,double>() const {
           return Pair<double,double> { };
9      }
   };

11 auto foo() -> Pair<double,double> {
13     return Pair<int,int> { };
   }

15 int main() {
17     auto p = foo();
   }
```

Listing 4.6: Trailing return type has to be analyzed else the wrong type name could be retrieved



```

1  template<typename T1, typename T2>
2  struct Pair {
3  };
4
5  template<>
6  struct Pair<int, int> {
7      Pair<double, double> operator/(Pair const& rhs) const {
8          return Pair<double, double> { };
9      }
10 };
11
12 template<typename T, typename U>
13 auto foo(T t, U u) -> decltype( t / u) {
14     //
15 }
16
17 int main() {
18     Pair<int, int> pI1 { };
19     Pair<int, int> pI2 { };
20     auto x = foo(pI1, pI2);
21 }

```

Listing 4.7: trailing return type for a generic function

Because the compiler is parsing code from left to right, the following is not possible.

```

1  template<typename T, typename U>
2  decltype (t/u) foo(T t, U u) {
3      //
4  }

```

The variables `t` and `u` are used before their declarations. However, this problem can be solved by using a trailing return type. The return type is postponed and the variables `t` and `u` are no longer used before their declaration [IBMa]. An example using this technique is pictured in *Listing 4.7*.

## No Trailing Return Type

This case is the more complex one because no trailing return type indicates that the return type will be deduced from the operands of its return statements according to the rules of template argument deduction. The return statements inside the function body have to be analyzed to be able to retrieve an IASTName that can be used to obtain an IASTName that represents the type.

The example in *Listing 4.8* shows that this can go over multiple levels.

```
template <typename T, typename S>
2 struct Pair {
3     };
4
5 template <>
6 struct Pair<int,int> {
7     Pair<double,double> operator/(const Pair<int,int>& rhs) {
8         return Pair<double,double> { };
9     }
10 };
11
12 template <typename T>
13 Pair<T,T> operator++(const Pair<T,T>& rhs) {
14     return Pair<T,T> { };
15 }
16
17 template <typename T>
18 auto makePair(T t) {
19     return Pair<T,T> { };
20 }
21
22 auto foo3() {
23     return makePair(1);
24 }
25
26 auto foo2() {
27     return makePair(4);
28 }
29
30 auto foo1() {
31     auto pair = foo2() / foo3();
32     return ++pair;
33 }
34
35 auto foo() {
36     return foo1();
37 }
38
39 int main() {
40     auto pair = foo();
41 }
```

Listing 4.8: Function using automatic type deduction

### 4.1.3. Classes

It is possible to trace template dependent code inside classes by selecting the corresponding starting point. However, classes itself cannot be traced. That means there is a disruption in the flow. It would be more convenient if classes could be traced as well since it is also possible to trace normal functions. The example displayed in *Listing 4.9* shows the problem.

```
1  template <typename T>
   struct A {
3    T t;
   };
5
   struct B {
7    A<int> a { };
   }
9
   int main() {
11    A<int> a { };
    B b { };
13 }
```

Listing 4.9: A<int> on line 7 cannot be analyzed if main is used as starting point

A<int> on line 11 is traceable if main is selected as the starting point. However, to be able to trace A<int> on line 7, the starting point has to be changed to A<int> directly.

### 4.1.4. Lambda Expressions

If functions and classes could be traced, it would only be logical to allow the tracing of lambda expressions as well. At a first glance, that might not be very useful. But a lambda could be defined outside the scope that is observed with the Templator plug-in (*Listing 4.10*). With the ability to trace function, classes and lambda expressions, the Templator also serves as a navigation tool.

```
1  #include <iostream>
3  auto lambda = [] (int a) { std::cout << a << '\n'; };
5  int main() {
    lambda(6);
7 }
```

Listing 4.10: Lambda that is outside the scope if main is used as starting point

### 4.1.5. Variable Templates

In *Listing 4.11*, an example for calculating the fibonacci numbers at compile-time with variable templates is shown. If this code is analyzed, with the Templator plug-in it is not possible to trace the variable templates.

```
1  template<int N>
   constexpr int fibonacci { fibonacci<N-1> + fibonacci<N-2> };
3
   template<>
5   constexpr int fibonacci<1> { 1 };
7
   template<>
   constexpr int fibonacci<0> { 0 };
9
11  int main() {
    int f0 = fibonacci<0>;
    int f15 = fibonacci<15>;
13 }
```

Listing 4.11: Example of the fibonacci numbers calculated at compile-time with variable templates

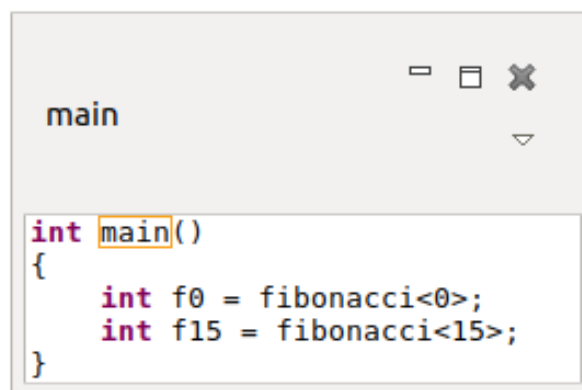


Figure 4.3.: No support for variable templates in the Templator plug-in

### 4.1.6. Alias Templates

The same applies to alias templates. They also cannot be examined with the Templator plug-in (*Figure 4.4*). A simple example of an alias template is shown in *Listing 4.12*.

```
1  template<typename T, typename U, typename V, typename W> struct
    Quadruple {
2      T fst;
3      U snd;
4      V trd;
5      W fth;
6  };
7
8  template<class T, class U>
9  using HalfHalfQuadruple = Quadruple<T,T,U,U>;
10
11 int main() {
12     HalfHalfQuadruple<int, double> { 1, 2, 4.2, 5.3 };
13 }
```

Listing 4.12: Example of an alias template



Figure 4.4.: No support for alias templates in the Templator plug-in

### 4.1.7. Non-Type Template Parameter

In 2.1.4, it is mentioned that variable templates only offer a better syntax for something that was already possible before. The fibonacci example from *Listing 4.11* can also be implemented without variable templates.

Because it only deals with ints, an enum can be used instead of a constexpr static data member as explained in 2.1.4. However, this version is also not fully examinable with the Templator plug-in. The sub entry opened by clicking on `Fibonacci<15>` does not allow further tracing.

To make this possible, non-type template parameters have to be replaced

```
1  template <int N>
2  struct Fibonacci {
3      enum { value = Fibonacci<N-1>::value + Fibonacci<N-2>::value };
4  };
5
6  template <>
7  struct Fibonacci<0> {
8      enum { value = 0 };
9  };
10
11 template <>
12 struct Fibonacci<1> {
13     enum { value = 1 };
14 };
15
16 int main() {
17     int f0 = Fibonacci<0>::value;
18     int f15 = Fibonacci<15>::value;
19 }
```

Listing 4.13: Example of the fibonacci numbers calculated during compile time without variable templates

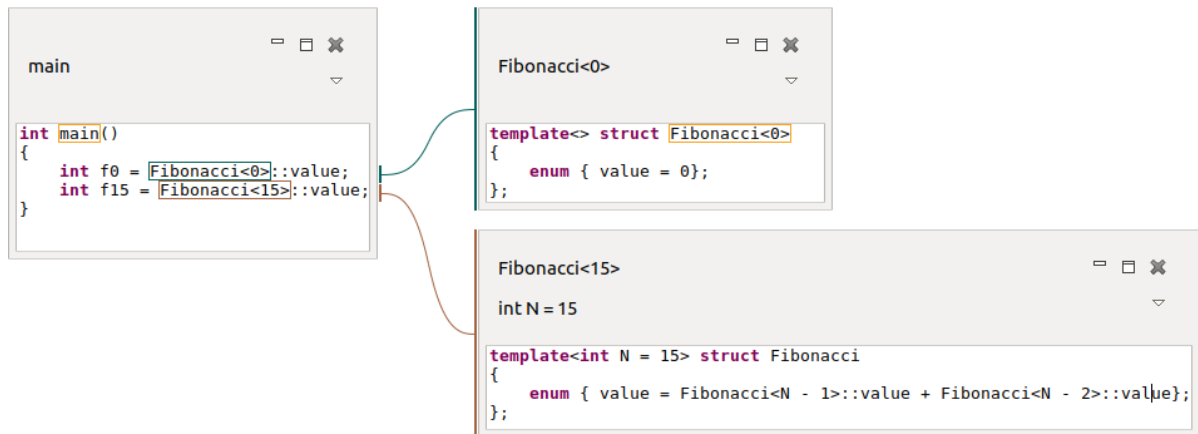


Figure 4.5.: It is not possible to trace  $\text{Fibonacci}\langle N - 1 \rangle$  and  $\text{Fibonacci}\langle N - 2 \rangle$  any further

#### 4.1.8. Template template parameter

If a template with a template template parameter as shown in *Listing 4.14* is analyzed, the same situation occurs. The template template parameter  $V$  has to be replaced to be able to examine it further.

```

1  template<typename T>
   class A {
3    int x;
   };
5
   template<typename T>
7   class A<T*> {
    long x;
9   };

11  template<template<typename> class V, class T>
   class C {
13    V<int> y;
    V<int*> z;
15   };

17  int main() {
    C<A, int> c;
19  }

```

Listing 4.14: Example using a template template parameter[Cppb]

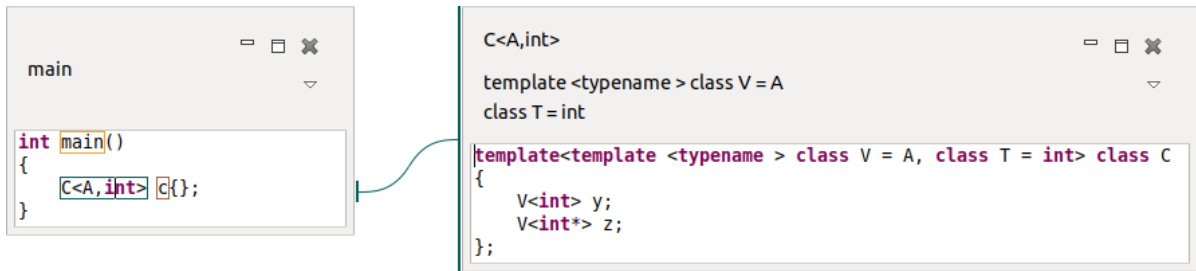


Figure 4.6.:  $V<\text{int}>$ ,  $y$ ,  $V<\text{int}*>$  and  $z$  are not traceable

#### 4.1.9. Incompatibility with CDT 9.4

In CDT 9.3 and below, the instantiation point was passed as an argument. However, in CDT 9.4 it was changed to propagate the instantiation to all the methods that need it. Following description for the new mechanism is taken from [Pri17]:

- Maintain a static thread-local stack of IASTNodes.
- Whenever a function "originates" a point of instantiation (passes an "IASTNode point" to another function that's not just the same "IASTNode point" that it took as input), it would instead push the node onto the stack for the duration of the call, and pop it afterwards.
- Whenever a function "consumes" the current point of instantiation ( uses its "IASTNode point" argument for a purpose other than propagating it to a function it calls), it would examine the top of the stack instead.
- The actual "IASTNode point" arguments are removed.

For the Templator plug-in that means that the point arguments in the calls used via reflection in the resolving classes have to be replaced by pushing the point onto the stack. After the point has to be popped again. To ensure that this always happen it is done with a finalize as shown in *Listing 4.15*.

```

1 CPPSemantics.pushLookupPoint(point);
  try {
3   // ...
  } finally {
5   CPPSemantics.popLookupPoint();
  }

```

Listing 4.15: Using the stack in CPPSemantics to propagate the instantiation point



## 4.2. User Interface

The user interface is not optimal yet. It often gets overloaded which decreases the user experience. If too many things are displayed at once, it is difficult to stay on top of things. This section discusses the potential to improve the user interface.

### 4.2.1. Opening Already Opened TreeEntries

By clicking on a name that possesses the same type with the same template arguments as another name, the entry is opened again. This can lead to an overloaded view and makes it hard to realize that two names actually possess the same type. Further, it is even possible to open itself again and again. The addressed situation is pictured in *Figure 4.7*.



Figure 4.7.: Starting point can be opened again and again and start and end open a new entry even if they have the same type

To get rid of this mechanism that leads to an overloaded view, an entry should be able to be referenced multiple times. However, this leads to other problems that first need to be solved. In the next segments those problems are discussed. How those problems are solved is described in *chapter 5*.

#### 4.2.1.1. Already Opened in Higher Column

The first problem arises if an entry is already opened but not in a directly following column as shown in *Figure 4.8*. Drawing a Bézier curve from the first to the third entry would be hard to trace because it would be hidden by the entry in the middle. Finding a way around the entry in the middle would be too complex regarding the calculation. If entries are resized, multiple curves could be affected and finding a path around for every curve would be too complex.

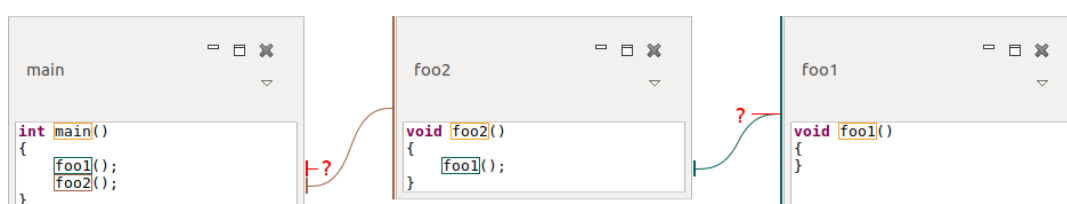


Figure 4.8.: Problem 1 : How to connect to an already opened entry in a higher column?

#### 4.2.1.2. Already Opened in Same Column

*Figure 4.9* shows another problem where the opening of an entry which is already opened in the same column is displayed. The current solution, where the curve starts at the right end of an entry and ends at the left side of its child is not possible without crossing other lines.

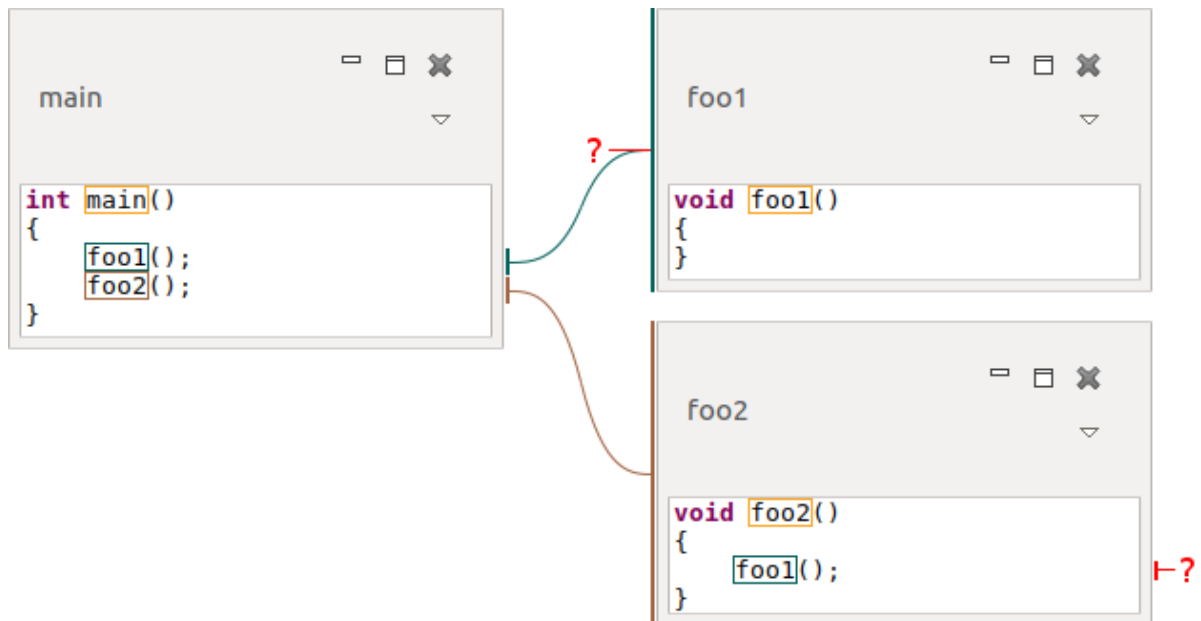


Figure 4.9.: Problem 2: How to connect to an already opened entry which is in the same column?

#### 4.2.1.3. Already Opened in a Previous Column

The same also applies if the already opened entry is in a previous column. The only difference is the distance which is even further as shown in *Figure 4.10*. It seems unlikely that those problems are solvable with Bézier curves only. A new concept has to be developed.

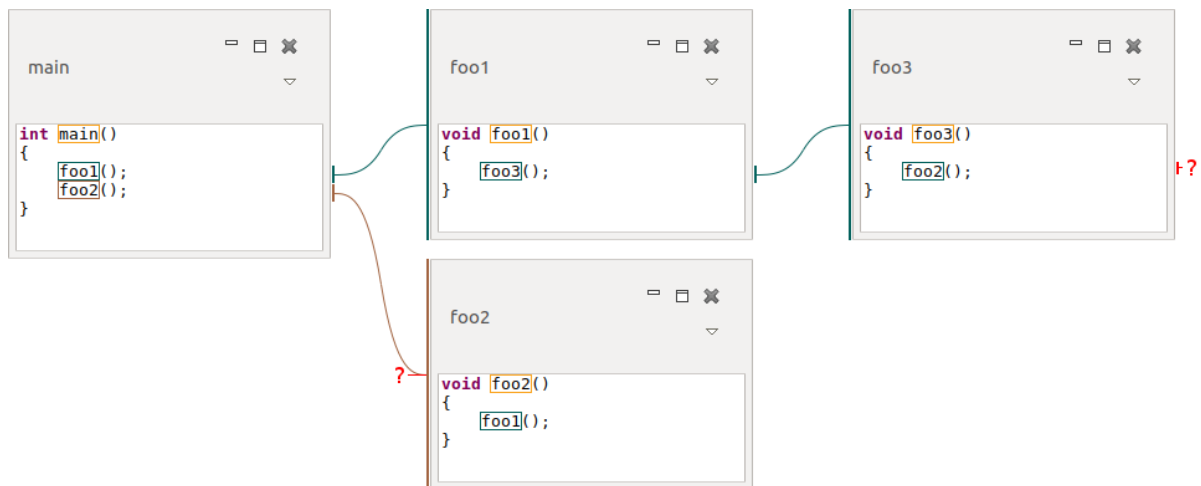


Figure 4.10.: Problem 3: How to connect to an already opened entry which is in a previous column?

#### 4.2.1.4. Entry Title and Description

In *Figure 4.7*, it is visible that the title for the opened entry has the name of the variable. However, if multiple sources could have a connection to this entry, it would be wrong because the other source entry possibly does not originate from a variable with the same name or not even from a variable. The title is divided into two parts. The first part shows the title of the name that was traced and the second one shows the description. In the description all the template parameters are listed together with the corresponding types.

#### 4.2.2. Ordering of Entries

The ordering of entries is done with the help of an array of ints (weights), which is explained in 3.2.3. However, with multiple possible connections to an entry, this is not as easy anymore and the current implementation does not work with it. How the ordering of the entries with multiple connections to it is realized is discussed in *chapter 5* because it depends on the implementation of the solution to the problems that arises with the removal of multiple entries with the same content *subsection 4.2.1*.

#### 4.2.3. Handling Events on Bézier Curves

At the moment it is only possible to remove an entry, either by clicking on the close icon or on the rectangle which represents the origin of the connection. However, if multiple connections are allowed, it would be handy to offer a way to close them by clicking on the connection. Additionally, there is no functionality available to jump from the start of a connection to the end and vice versa. So the most rational conclusion would be to allow interactions on the Bézier curves but in the current implementation that was not intended.

The calculation of the Bézier curves is made inside the `BezierRenderer` which is called by the methods that are called if a control gets painted. However, this is not optimal because of two reasons. First, it is calculated too often. The paint event is fired even if the curves do not need to be updated. The second and more important for tracing the curve is how they are drawn. After calculating the path, it is drawn and the path gets disposed. The information about where the path passes is lost. It is not possible to handle events on it if the points of the path are not known anymore.

#### 4.2.4. Default Width for TreeEntries

A TreeEntry has a default width that comes into play if the preferred width (depending on the length of the content) gets bigger than it. The width of the entry is then adjusted to the default width. In some cases this can be really annoying because the entries have to be resized manually to get to the rectangle if the rectangle is at a position which is not visible anymore because of the adjusted size. This problem was discovered in a later stage of the thesis. Therefore the image shown in *Figure 4.11* looks different. The problem is that `fibonacci<N-2>` cannot be clicked without resizing the TreeEntry or scrolling horizontally.

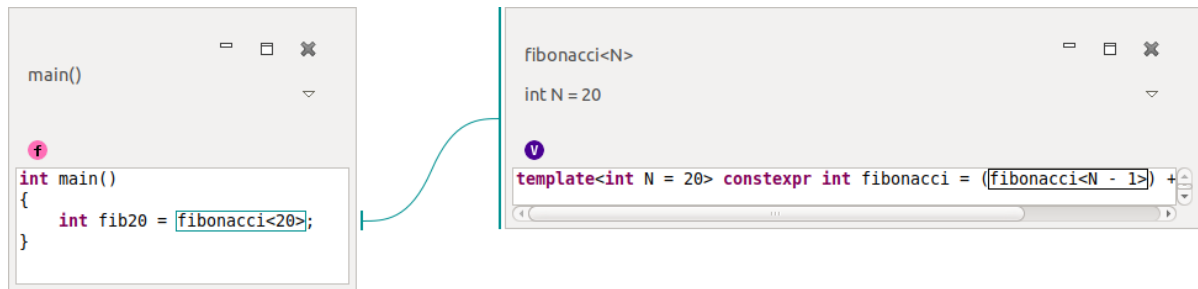


Figure 4.11.: It is not possible to trace `fibonacci<N-2>` without resizing the entry or scrolling

## 5. Implementation

The *chapter 4* showed listed the problems with the Templator plug-in. This chapter deals with the solutions and their implementations.

### 5.1. Tracing All Names and Operators

The Templator plug-in works by analyzing IASTNames. To get those IASTNames from the AST, a visitor is used.

#### 5.1.1. Operators

An operator is not visible in the AST (the expression is but not the operator) but nodes implementing the interface IASTImplicitNameOwner offer the method getImplicitNames to get implicit names. An implicit name is a name which is not represented in the AST. Those implicit names can also be visited. To do that, the flag shouldVisitImplicitNames of a class extending the ASTVisitor has to be set. The already available functionality to resolve an IASTName can be used for IASTImplicitNames as well. However, finding the implicit names again in the formatted definition is not possible yet.

##### 5.1.1.1. Find Region

Unluckily, the implicit names are not around anymore if the nodes get copied and the ASTWriter does not trigger the recreation of them. The formatted definition that is used for the view is a copy of the original ones, that means the implicit names are not available. To be able to find the implicit names that were visited in the formatted definition, the expressions are visited as shown in *Listing 5.1*. The example in this segment is focusing on an CPPASTBinaryExpression. All the other regions are retrieved in a similar manner and not explained here.

```
@Override
2 public int visit(IASTExpression expression) {
    if (expression instanceof IASTImplicitNameOwner) {
4         processExpression((IASTImplicitNameOwner) expression);
    }
6     return super.visit(expression);
}
```

Listing 5.1: Visiting all expressions

The implicit names are retrieved with the help of the original IASTImplicitNameOwner. For every IASTNode, it is possible to get the original node (the node which was copied). There the implicit name is available. If this implicit name is in the list of the visited names the analyzed expression is important. To find the position of the implicit name inside the expression, the position of the scribe is obtained. The scribe is responsible for concatenating strings and for managing indentations. This position however, is not the position needed. The scribe knows about the indentation level but it is not intended to be used. However, it is important to know the beginning of the expression, else it is impossible to draw a correct rectangle. The indentation level is retrieved via reflection.

```

1 private void processExpression(IASTImplicitNameOwner nameOwner) {
    IASTImplicitName implicitName = getImplicitName(nameOwner);
3   int namePosition = getNamePosition(implicitName);
    if (namePosition >= 0) {
5       int offset = scribe.toString().length() + getIdentation();
        int length = implicitName.getFileLocation().getNodeLength();
7       if (nameOwner instanceof CPPASTBinaryExpression) {
            offset += writePartBeforeImplicitName((CPPASTBinaryExpression)
nameOwner);
9       } else if () {
            ...
11      }
        IRegion region = new Region(offset, length);
13      foundRegions.put(namePosition, region);
    }
15 }

```

Listing 5.2: Processing the expressions

Since the operator is not at the beginning of the expression for a CPPASTBinaryExpression, the length of the first operand has to be added. *Equation 5.1* shows the different lengths. By adding the length of the "Indent" and the length of the "Operand1" to the scribe, the desired position can be retrieved. The length of the "Operand1" is retrieved by writing it and returning its length as shown in *Listing 5.3*. The length of the operator is equal to the length of the implicit name and is used for the length for the region which is created for the rectangles.

$$\underbrace{\hspace{1.5cm}}_{\text{Indent}} \quad \overbrace{\text{tupleInt1}}^{\text{Operand1}} \quad \underbrace{+}_{\text{Operator}} \quad \overbrace{\text{tupleInt2}}^{\text{Operand2}} \quad (5.1)$$

```

1 private int writePartBeforeImplicitName(CPPASTBinaryExpression
    binaryExpr) {
    return writer.write(binaryExpr.getOperand1()).length();
3 }

```

Listing 5.3: Getting offset to calculate starting position for the rectangle

### 5.1.1.2. Problems

Not all the operators could be implemented successfully.

#### IASTLiteralExpression

The suffix of an IASTLiteralExpression is not written by the ExpressionWriter. Therefore, the rectangle cannot be drawn at the correct position as shown in *Figure 5.1*. A bug report has been created [Mar17].

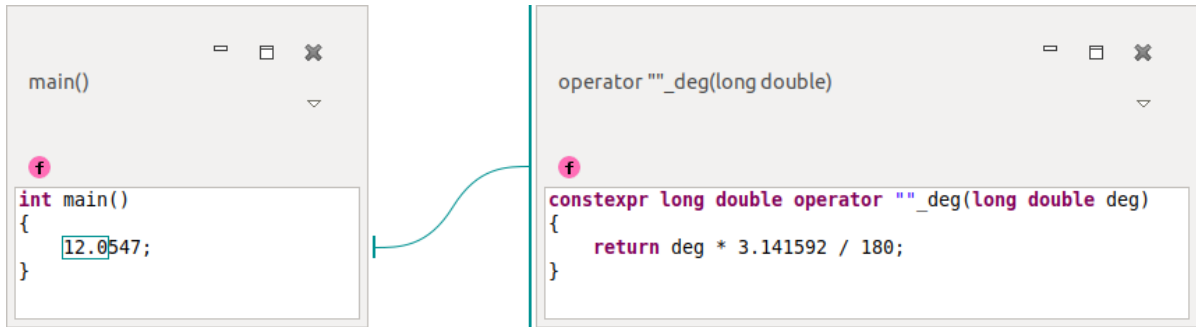


Figure 5.1.: Rectangle is not positioned correctly because suffix `_deg` is not written

#### ICPPASTEExpressionList

With the current approach it is not possible to handle ICPPASTEExpressionLists. The expression list can have multiple `,` operators. Therefore, multiple rectangles have to be drawn for one expression, which is not supported yet.

### 5.1.2. auto Specifier

To find the auto specifiers, the visitor also has to visit instances of IASTDeclSpecifier. To do that the flag `shouldVisitDeclSpecifiers` has to be set and the visit method for the IASTDeclSpecifier has to be overridden. The resolving of the auto specifier is done in the class `AutoResolver`. A resolved auto specifier is cached to increase performance. The diagram in *Figure 5.2* pictures the process. It is important to keep in mind that the type of the expression which is the last box can also be auto again. Therefore, this loop can be passed multiple times.



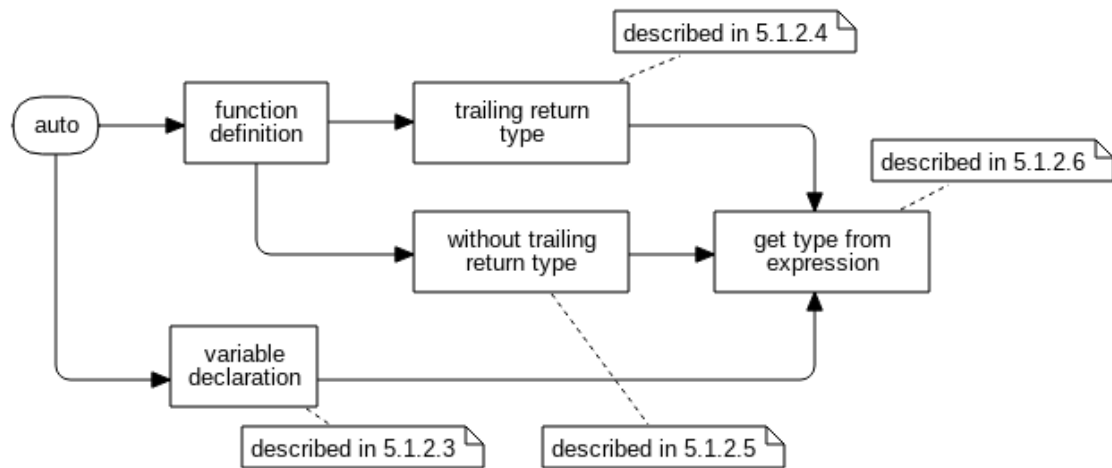


Figure 5.2.: Diagram to show how the auto specifier is handled with the information where it is explained in the report

#### 5.1.2.1. Storing Information about Nodes

Until now it was enough to only handle IASTNames because all the needed information could be retrieved from an IASTName. However, this is not possible anymore. At least to put the rectangle around the declaration specifier, it has to be stored. It is also not enough anymore to search for IASTNames only, declaration specifiers have to be visited as well. To do that the ResolvedName holds an IASTNode instead of an IASTName now.

#### 5.1.2.2. Checking if auto

First we have to know if the declaration specifier is auto. If the declaration specifier is not auto it is already handled or not important. The class CPPVisitor from Eclipse CDT which is an implementation of an ASTVisitor with the purpose to extract information from a C++ translation unit offers a method usesAuto that returns a PlaceholderKind which can be used to check if the passed declaration specifier is using auto or not (*Listing 5.4*).

```

2 public static boolean isAuto(IASTDeclSpecifier declSpecifier) {
    PlaceholderKind placeholder = CPPVisitor.usesAuto(declSpecifier);
    return placeholder == PlaceholderKind.Auto || placeholder ==
4     PlaceholderKind.DecltypeAuto;
}

```

Listing 5.4: checking if an IASTDeclSpecifier uses auto or not

### 5.1.2.3. Variable Declaration

In 4.1.2.1, three different ways to initialize a variable are presented. The abstract syntax tree is identical for those three methods except for one node which is representing the method that was chosen for the initialization. Eclipse CDT uses the classes `CPPASTEqualsInitializer`, `CPPASTConstructorInitializer` and `CPPASTInitializerList` to distinguish them.

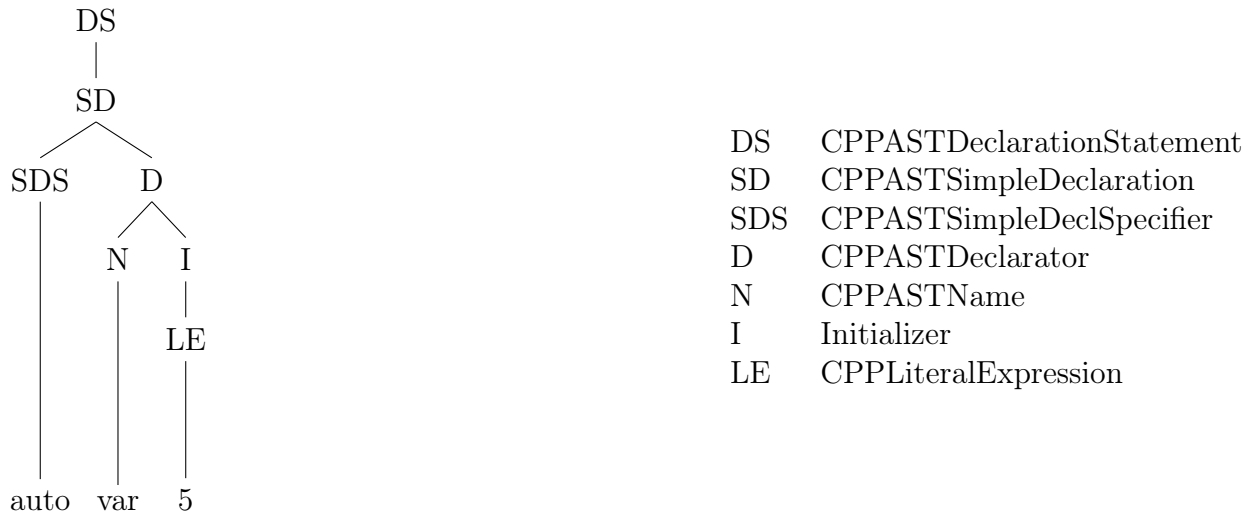


Figure 5.3.: Abstract syntax tree of a variable declaration

*Listing 5.5* figures how the `InitializerClause` can be extracted. There is no common interface to do that, therefore, each kind of initializer has to be handled separately.

```

1 private IASTInitializerClause getInitializerClause(
2     IASTSimpleDeclaration simpleDecl) {
3     IASTInitializer initializer = simpleDecl.getDeclarators()[0].
4     getInitializer();
5     if (initializer instanceof ICPPASTInitializerList) {
6         ICPPASTInitializerList iL = (ICPPASTInitializerList) initializer;
7         return iL.getClauses()[0];
8     } else if (initializer instanceof IASTEqualsInitializer) {
9         IASTEqualsInitializer eI = (IASTEqualsInitializer) initializer;
10        return eI.getInitializerClause();
11    } else if (initializer instanceof ICPPASTConstructorInitializer) {
12        ICPPASTConstructorInitializer cI = (ICPPASTConstructorInitializer)
13        initializer;
14        return cI.getArguments()[0];
15    }
16    return null;
17 }

```

Listing 5.5: getting the initializer from a simple declaration (checks for null were omitted in this example)

#### 5.1.2.4. Function Definition with Trailing Return Type

The abstract syntax tree in *Figure 5.4* shows that the `CPPASTFunctionDefinition` possesses a `CPPASTTypeId` which stands for the trailing return type. This trailing return type can be retrieved as shown in *Listing 5.6*.

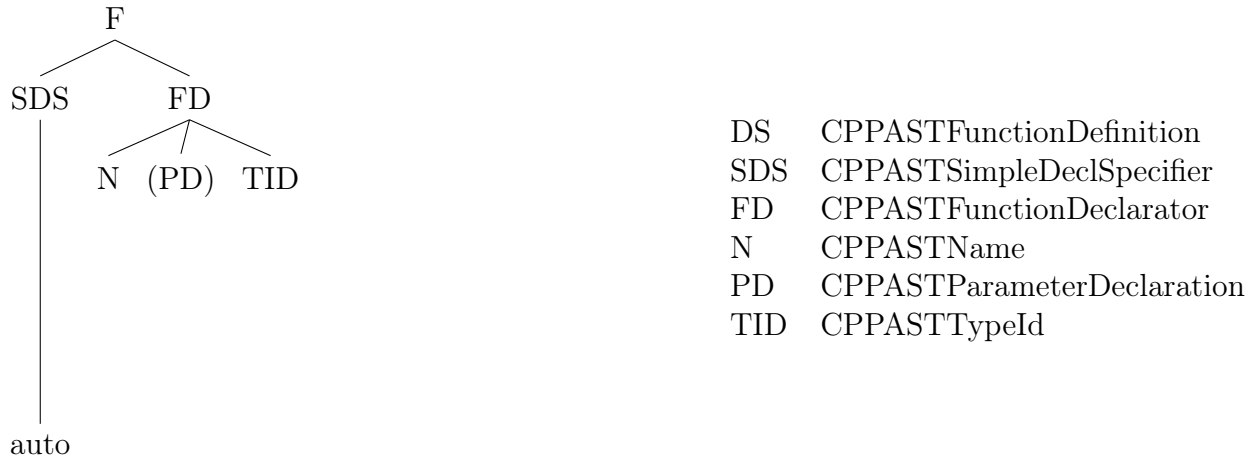


Figure 5.4.: Abstract syntax tree of a function definition with a trailing return type

```

1 private IASTTypeId getTrailingReturnType(ICPPASTFunctionDeclarator
   cppFunDecl) {
2     return cppFunDecl.getTrailingReturnType();
3 }

```

Listing 5.6: getting trailing return type of a function

If a trailing return type could be retrieved, the `ResolvedName` is created with it if possible. If it is already a name the creation works the same way as for other `IASTNames`. It can also be an expression for example `decltype(t / u)` as used in the example in *Listing 4.7*. Then the `ResolvedName` is created like described in 5.1.2.6.

```

1 private ResolvedName createResolvedNameFromTypeId(IASTTypeId typeId,
   AbstractResolvedNameInfo parent)
   throws TemplatorException {
3     IASTName name = ASTTools.getName(typeId);
   if (name != null)
5         return createResolvedNameFromName(name, parent);
   IASTDeclSpecifier declSpec = typeId.getDeclSpecifier();
7     if (declSpec.getChildren()[0] instanceof IASTExpression) {
   IASTExpression expr = (IASTExpression) declSpec.getChildren()[0];
9         return createResolvedNameFromExpression(expr, parent);
   }
11    return null;
}

```

Listing 5.7: creating a `ResolvedName` from the `IASTTypeId`

### 5.1.2.5. Function Definition without Trailing Return Type

A visitor called `FindingRelevantExpressionsInReturnStatementsVisitor` is used to visit all statements. Only return statements are important since we want to get the return type of the function. Some return statements are better suited to get the type name in the sense that they are less expensive to follow. For example, a function call expression could refer to a function which returns `auto` as well, where the whole process has to be repeated. A simple type construction expression is much easier because the type name is a part of it. Therefore the different expressions are prioritized to avoid additional overhead. *Table 5.1* shows the priority for each expression type. A lower value means it has a higher priority. The steps between were introduced to make it easier to add other expression types without having to change all the other values.

priority	expression
1	<code>ICPPASTSimpleTypeConstructorExpression</code>
3	<code>IASTCastExpression</code>
5	<code>IASTConditionalExpression</code>
7	<code>IASTBinaryExpression</code>
9	<code>IASTUnaryExpression</code>
11	<code>ICPPASTFunctionCallExpression</code>
13	<code>IASTIdExpression</code>
15	<code>ICPPASTFieldReference</code>
17	<code>ICPPASTArraySubscriptExpression</code>

Table 5.1.: Expressions and their priority

### 5.1.2.6. Get Type from Expression

How the `ResolvedName` is created out of different expression is explained in the following. Each part shows an example to show how an expression of this type could look like. It is important that the `AbstractResolvedNameInfo` are created during the process. In contrast to the resolving from an `IASTName`, it can go over multiple levels and all the levels have to be created as well. They are needed to pass as a parent which is needed to replace template dependent parameters with the help of its `TemplateArgumentMap`.

#### Simple Type Constructor Expression

The `IASTName` is retrieved from the declaration specifier of the `ICPPASTSimpleTypeConstructorExpression`. With the retrieved name the `ResolvedName` can be created like usual.

```
int main() {  
    auto x = Pair<int,int> { 4, 2 };  
}
```

## Cast Expression

The `IASTTypeId` is retrieved from the `IASTCastExpression`. How the `ResolvedName` can be created with the `IASTTypeId` is explained below.

```
1 int main() {  
    Tuple<int,int> ti { 1, 2 };  
3     auto x = (Tuple<double,double>) ti;  
}
```

## Conditional Expression

A conditional expression consists of a positive result expression and a negative one (the one in front of the `?` operator and the one after). To deal with a conditional expression, only the positive result expression is analyzed. It does not matter which one, because they need to have the same type. With this retrieved expression the process starts again.

```
int main() {  
2     Tuple<int,int> ti1 { 1, 2 };  
    Tuple<int,int> ti2 { 2, 3 };  
4     auto x = true ? ti1 : ti2;  
}
```

## Implicit Name Owner

With the `IASTImplicitNameOwner`, following type of expressions are handled:

- `ICPPASTBinaryExpression`
- `ICPPASTUnaryExpression`
- `ICPPASTLiteralExpression`
- `IASTArraySubscriptExpression`

If the name owner has an implicit name, an `AbstractResolvedNameInfo` is created with it. This is needed as a context (the parent for the next `AbstractResolvedNameInfo`). With the implicit name, it is possible to get the function definition. This is used to create the `ResolvedName` and is explained below. However, an implicit name is not always available, for example if the operands are not resolved yet. In that case, the evaluation has to be instantiated with the information of the parent `AbstractResolvedNameInfo` to get the overloaded function as shown in *Listing 5.8*. With the function definition (as described below), the process is continued.

```

1 ICPPEvaluation eval = cppExpr.getEvaluation();
  InstantiationContext context = new InstantiationContext(parent.
    getTemplateArgumentMap());
3 context.addToParameterMap(parent.getTemplateArgumentMap());
  ICPPEvaluation resEval = eval.instantiate(context, maxDepth);
5 CPPSemantics.pushLookupPoint(expr);
  try {
7     ICPPFunction overload = null;
    if (resEval instanceof EvalBinary) {
9         EvalBinary evalBinary = (EvalBinary) resEval;
        overload = evalBinary.getOverload();
11    } ...
    if (overload != null && overload instanceof CPPFunction) {
13        CPPFunction fun = (CPPFunction) overload;
        IASTFunctionDeclarator funDecl = fun.getDefinition();
15        if (functionDecl.getParent() instanceof ICPPASTFunctionDefinition)
        {
            ICPPASTFunctionDefinition funDef = (ICPPASTFunctionDefinition)
            funDecl.getParent();
17            return createResolvedNameFromFunctionDefinition(funDef, parent,
                true);
        }
19    }
  } finally {
21      CPPSemantics.popLookupPoint();
  }

```

Listing 5.8: Getting the overloaded function

## Function Call Expression

```

1 Tuple<int,int> fun() {
2     return Tuple<int,int> { 1, 2 };
3 }
4
5 int main() {
6     auto x = fun();
7 }

```

The name of the `ICPPASTFunctionCallExpression` is obtained. With this name an `AbstractResolvedNameInfo` is created, which is used as a parent. The definition of the info is used to get the function definition which is used to resolve the name (explained below).

## Id Expression

```
1 int main() {  
    Tuple<int,int> ti1 { 1, 2 };  
3     auto x = ti1;  
}
```

The definition name is obtained from the id expression with the help of the ASTAnalyzer (section B.10). The simple declaration of this name is then used to create the ResolvedName. How this is done from an IASTSimpleDeclaration is explained below.

## Field Reference

```
template<typename T>  
2 struct A {  
    Tuple<T,T> t;  
4 };  
  
6 int main() {  
    A<int> a {};  
8     auto x = a.t;  
}
```

The IASTName is retrieved from the field name. From there on it is like usual.

## Lambda Expression

The lambda expression is a special case. A class LambdaExpression was created to represent a lambda expression which is a subclass of AbstractResolvedNameInfo. It is similar to the classtype explained in 5.1.3 and only shows its definition if traced.

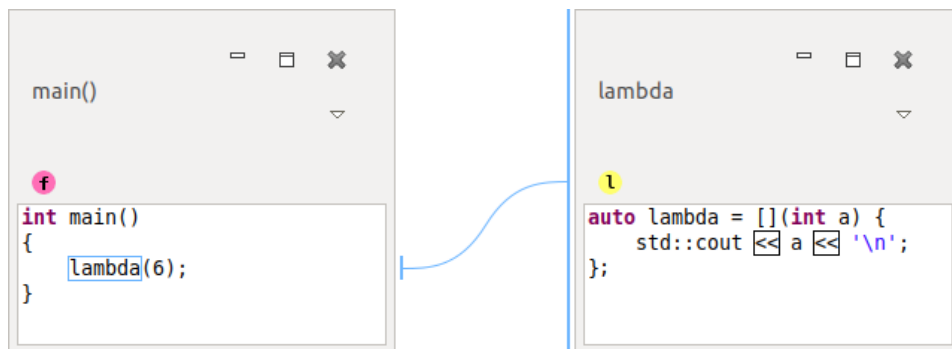


Figure 5.5.: Lambda expression analyzed with the Templator plug-in

## Type Id

It is tried to extract an IASTName from the IASTTypeId. If this is not possible, the expression from the declaration specifier of the type id is retrieved.

## Function Definition

First, the declaration specifier of the function is analyzed. If it is not auto, the type can easily be extracted from there. In a second check, it is tried to get the trailing return type. If a trailing return type can be retrieved, the IASTTypeId is used to create a ResolvedName as explained above. However, often this is not possible. In that case, the body of the function has to be analyzed. After inspecting all the expressions in return statements, the easiest one is taken (ordered by priority). With this expression the whole process starts again.

### 5.1.2.7. Simple Declaration

A simple declaration can either have a type name which can be used to create a ResolvedName as usual or can be auto. If it is auto, the initializer clause is obtained and the whole process is repeated.

### 5.1.2.8. Find Region

To find the region of the IASTNode, the original node of the visited node is searched in the sub names that are stored inside the AbstractResolvedNameInfo. This again is needed because the position of the node inside the AbstractResolvedNameInfo is the position of the original node. Since the nodes are copied and modified the position that can be retrieved on that name could differ (in most cases it does). Most of the important nodes are represented as IASTNames inside the AST, but for this specific case, we are interested in declaration specifiers.

However, the processing for IASTNames and IASTDeclSpecifiers is the same (*Listing 5.9*). The only important question is if the visited node is a node that was marked as important. To do that, the original node is obtained and compared to the stored one. If it is the same node, the region of the formatted node is added to the regions which are used to draw the rectangles and handle the click events.



```

1 private void processNode(IASTNode node) {
2     for (int i = 0; i < subNames.size(); i++) {
3         IASTNode originalNode = subNames.get(i).getOriginalNode();
4         if (node.getOriginalNode().equals(originalNode)) {
5             String nameString = writer.write(originalNode);
6             int offset = scribe.toString().length();
7             int length = nameString.length();
8             IRegion region = new Region(offset, length);
9             foundRegions.put(i, region);
10            break;
11        }
12    }
13 }

```

Listing 5.9: Processing names and decl specifiers

### 5.1.3. Classes

The class `ClassType` was introduced to represent classes in the Templator plug-in. It is a subclass of `AbstractResolvedNameInfo`. In a normal class, there are no arguments to replace or similar tasks. Only the definition is shown if clicked on it as pictured in *Figure 5.6*.

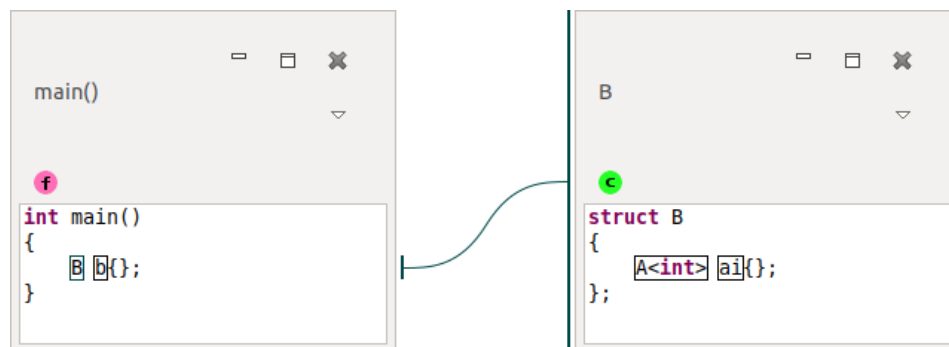


Figure 5.6.: Opening the definition of a class with the Templator plug-in

### 5.1.4. Variable Templates

The handling of variable templates was not implemented yet in the release of Eclipse CDT when the Templator plug-in was developed. Therefore, the changes in `createBindings` of `CPPTemplates` have to be added to the `ClassTemplateResolver` 3.3.2.

In Eclipse CDT, the variable template binding is created, but the dependent template arguments are not replaced. The already added method in `ClassTemplateResolver` is used to do that. In addition the replacing of non-type template arguments has to be added like described in 5.1.6.

#### 5.1.4.1. Parsing Problem

There is a problem with the parsing that prevents further tracing in variable templates if they are not surrounded with parenthesis.

Some assumptions that were made before variable templates were introduced no longer hold. For example if a `'>'` token is followed by a binary or unary operator, it is treated as a greater-than operator [Rid17b].

This means that

`fibonacci <N-1> + fibonacci<N-2>`

is interpreted as

`fibonacci < (N-1) > (+ fibonacci<N-2>)`

In this case, the template id is not available. And `fibonacci<N-1>` cannot be traced. At the moment this can only be avoided by setting parenthesis around the first template as shown in *Figure 5.7*.

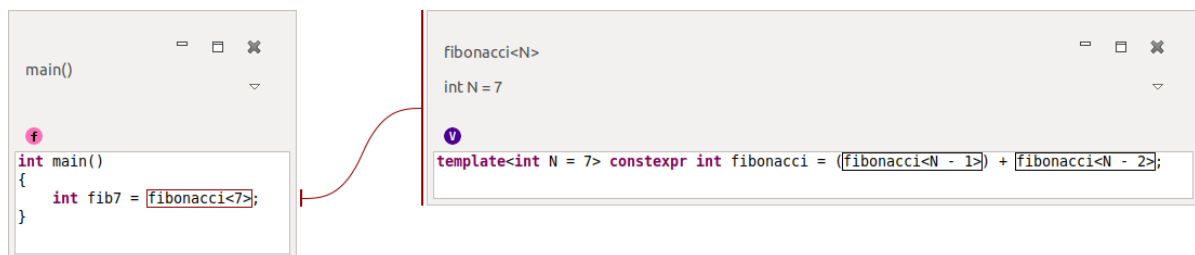


Figure 5.7.: Parentheses around template id

### 5.1.5. Alias Templates

Similar to variable templates, alias templates are also handled directly inside createBindings of the class CPPTemplates. If an alias template is analyzed, the first level shows the alias template declaration from where it is possible to trace the aliased type. This is pictured in *Figure 5.8*.

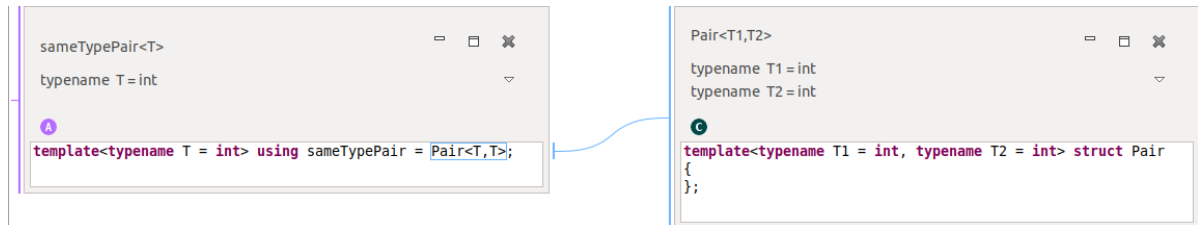


Figure 5.8.: Alias template analyzed with the Templator plug-in

### 5.1.6. Non-Type Template Parameter

The problem with non-type template parameters is that it is not possible to get to the correct template definition if the corresponding argument is not evaluated.

```
1 template <int N>
  constexpr int fibonacci = (fibonacci<N-1>) + fibonacci<N-2>;
3
4 template<>
5 constexpr int fibonacci<1> = 1;
6
7 template<>
8 constexpr int fibonacci<0> = 0;
9
10 int main() {
11     int fib7 = fibonacci<7>;
12 }
```

Listing 5.10: Template arguments N-1 and N-2 have to be replaced to allow tracing

For the example figured in *Listing 5.10*, the non-type template argument should be replaced by an evaluated non-type template argument. In this case that means N-1 should be replaced by 6 and N-2 by 5. The same applies for the next levels.

This replacement is done in the class `TemplateNonTypeArgumentResolver`. The evaluation of a non-type template parameter is retrieved. All the `EvalBindings` inside this evaluation could be bound to a non-template parameter. Hence, all the `EvalBindings` need to be extracted from the evaluation as shown in *Listing 5.11*.

```
private static List<EvalBinding> getEvalBindings(ICPPEvaluation
  cppEvaluation) {
2   List<EvalBinding> evalBindings = new ArrayList<>();
3   if (cppEvaluation instanceof EvalBinary) {
4       EvalBinary evalBinary = (EvalBinary) cppEvaluation;
5       evalBindings.addAll(getEvalBindings(evalBinary.getArg1()));
6       evalBindings.addAll(getEvalBindings(evalBinary.getArg2()));
7       // ...
8   } else if (cppEvaluation instanceof EvalBinding) {
9       EvalBinding evalBinding = (EvalBinding) cppEvaluation;
10      evalBindings.add(evalBinding);
11  }
12  return evalBindings;
13 }
```

Listing 5.11: Getting all `EvalBindings` in a `ICPPEvaluation`

For all the EvalBindings that are bound to a non-type template parameter, the corresponding argument is retrieved from the TemplateArgumentMap of the parent AbstractResolvedNameInfo. The evaluations of those arguments are then added to the ActivationRecord. This record is used to keep track of the values of parameters and local variables during the evaluation of function calls. After updating the ActivationRecord with all evaluations for the non-type template parameters, the resulting evaluation can be computed. If this resulting evaluation is a number, a new CPPTemplateNonTypeArgument is created and returned as pictured in *Listing 5.12*.

```

1 public static CPPTemplateNonTypeArgument getEvaluatedArgument(
    CPPTemplateNonTypeArgument nonTypeArg,
    ICPPTemplateParameterMap templateParamMap) {
3     ICPPEvaluation nonTypeEval = nonTypeArg.getNonTypeEvaluation();
    List<EvalBinding> bindings = getEvalBindings(nonTypeEvaluation);
5     ActivationRecord record = new ActivationRecord();
    ConstexprEvaluationContext context = new ConstexprEvaluationContext
    ();
7     for (EvalBinding evalBin : bindings) {
        IBinding bin = evalBin.getBinding();
9         if (bin instanceof CPPTemplateNonTypeParameter) {
            CPPTemplateNonTypeParameter parameter = (
CPPTemplateNonTypeParameter) bin;
11            ICPPTemplateArgument templateArg = templateParamMap.getArgument
            (parameter);
            ICPPEvaluation eval = templateArg.getNonTypeEvaluation();
13            record.update(binding, eval);
        }
15    }
    ICPPEvaluation resEval = nonTypeEvaluation.computeForFunctionCall(
record, context);
17    IValue value = resEval.getValue();
    IType type = resEval.getType();
19    if (value.numberValue() != null) {
        return new CPPTemplateNonTypeArgument(value, type);
21    }
    return null;
23 }

```

Listing 5.12: get the evaluated argument

### 5.1.7. Template Template Parameter

Just like the extension to replace dependent template arguments inside the copied method in `ClassTemplateResolver` 3.3.2, handling template template parameters is also added in a separate method. As stated in 2.4.1.3, a template template parameter can only be a class template or an alias template. Both are already handled inside the `ClassTemplateResolver`. This means, the alias template or class template needs only needs to be extracted before instantiating the template as shown in *Listing 5.13*.

```
1 private static IBinding extractTemplateTemplateParameter(  
    AbstractResolvedNameInfo parent, IBinding template {  
2     if (template instanceof ICPPTemplateTemplateParameter) {  
3         ICPPTemplateTemplateParameter param = (  
            ICPPTemplateTemplateParameter) template;  
4         ICPPTemplateArgument arg = parent.getArgument(param);  
5         IType type = arg.getTypeValue();  
6         if (type instanceof ICPPClassTemplate || type instanceof  
            ICPPAliasTemplate) {  
7             template = (IBinding) type;  
8         }  
9     }  
10    return template;  
11 }
```

Listing 5.13: Extracting the alias template or class template from a template template parameter

## 5.2. User Interface

The decisions made to solve the problems outlined in *section 4.2* are explained in this section. Additionally, it is shown how it was implemented, to serve as an entry point into the user interface for future developers.

### 5.2.1. Multiple Entries

In the analysis, it is shown that the current implementation can lead to an overloaded view. New classes are created to make it easier to understand the code while adding new functionality.

The end of a connection only contains a `TreeEntryCollectionNode`. To allow multiple connections to a node, the class has to be redesigned. More than one connection has to be stored and single connections must be removable. A class diagram is later displayed in *Figure 5.21*.

The start of a connection on the other hand needs information about the rectangle. How the `ConnectionStart` and `ConnectionEnd` is realized is presented in *Figure 5.9*.

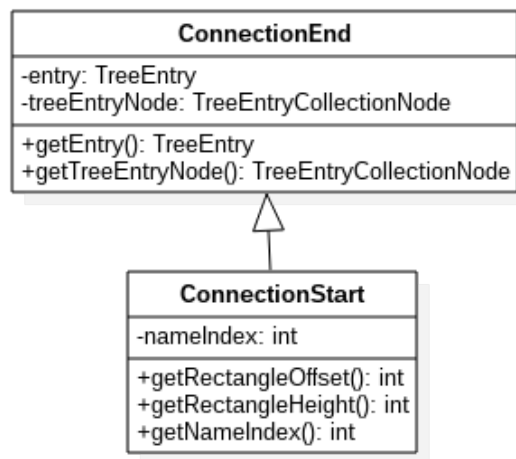


Figure 5.9.: `ConnectionStart` and `ConnectionEnd`

The `ConnectionStart` is always the `TreeEntry` with the active rectangle. In *Figure 5.10* this applies to the `TreeEntry` on the left side.

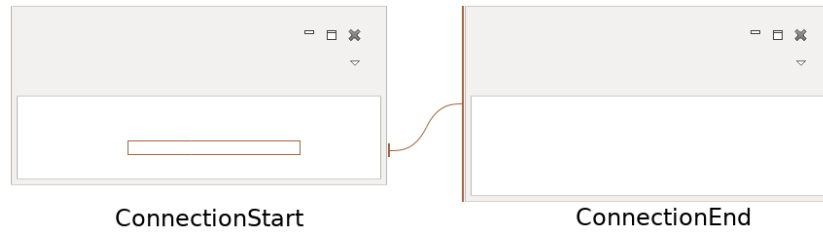


Figure 5.10.: ConnectionStart with the active rectangle and the ConnectionEnd

A Connection possesses a start and an end. The class diagram in *Figure 5.11* shows the classes that are used to model the different kind of connections. Those connections are explained in detail in the following parts.

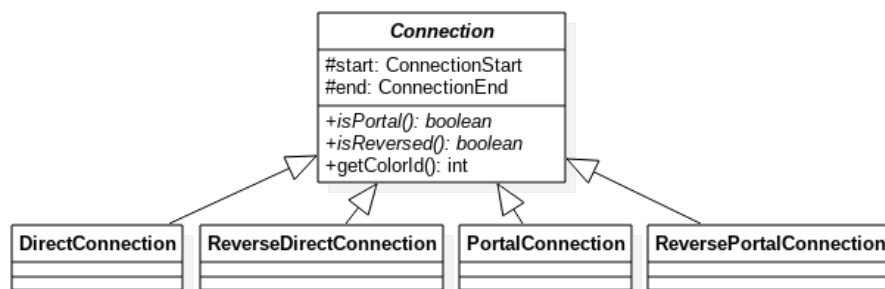


Figure 5.11.: Different kinds of connections

#### 5.2.1.1. Direct Connection

A DirectConnection (*Figure 5.12*) is the kind of connection which was already available. As the name suggests it models a direct connection between two entries. The start of a direct connection always is on the left side. Between the two entries a Bézier curve is drawn. The path of this curve is saved inside the DirectConnection to allow interactions on it.



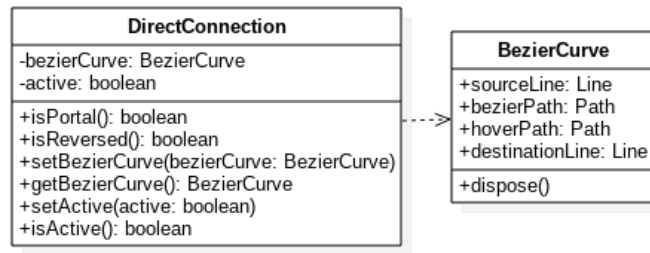


Figure 5.12.: Class diagram DirectConnection

As mentioned earlier, the Bézier curves have to be calculated outside of the Bezier-Renderer to be able to handle actions on it. Therefore, the calculation is moved to ConnectionCalculator, that calculates the positions for Bézier curves and portals. The ConnectionCalculator is always called if something happens that could possibly change the route of the curves. This includes resizing or minimizing entries as well as removing connections or entries.

The Bézier curves are quite thin which makes handling clicks really difficult. Making the lines thicker on the other hand would decrease the readability because it would be impossible to recognize the path of a line anymore, especially if multiple lines are very steep. Therefore, one path is used to draw but another path is stored to handle interactions. The clickable path results from the drawn path and copies of it which are shifted diagonally. Diagonally because the thickness should be increased for vertical and horizontal lines in the same manner.

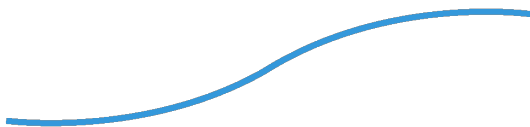


Figure 5.13.: Visible path

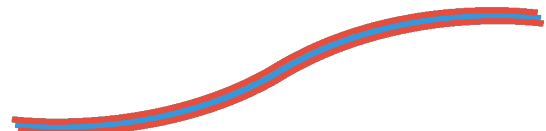


Figure 5.14.: Clickable path

#### 5.2.1.2. Portal Connection

The solution to the problems "Already Opened in Same Column" and "Already Opened in a Previous Column" discussed in the analysis is to introduce portals, which can be used to travel backwards. They are not drawn like DirectConnections. No line between the two entries is drawn. Drawing a line would overload the view even more. A PortalConnection consists of an entry portal (blue) which is drawn at the end of the line where the rectangle resides and a destination portal (orange) which is positioned in the InfoBar as shown in *Figure 5.16*. The InfoBar is explained in more detail later in *subsection 5.2.6*.

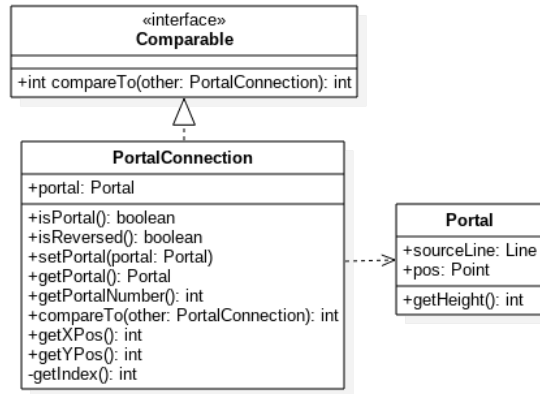


Figure 5.15.: class diagram of PortalConnection

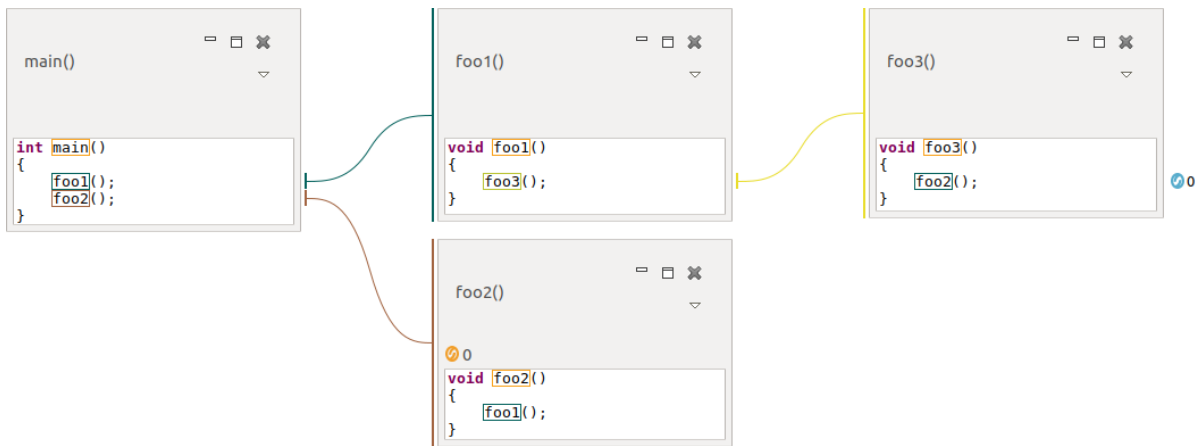


Figure 5.16.: portal from foo3() to foo2()

Portals only address two of the three presented problems from *section 4.2*. The problem "Already Opened in Higher Column" is solved by allowing to travel along Bézier curves and is described in 5.2.2.

### 5.2.1.3. Reverse Connection

The approach developed in the bachelor thesis uses connections that go from a child to its parent (from right to left) and by storing the child nodes inside the parents it is possible to navigate to the children. However, with allowing multiple connections to the same entry this is troublesome for multiple reasons.

Consider the connection in *Figure 5.17* should be removed. The click goes to the rectangle in entry 0. But there we do not know where it goes to because the connection is saved in the child node. Therefore, all the children (fortunately in this case there are only two)

have to be visited (in the worst case), to find the connection that should be closed.

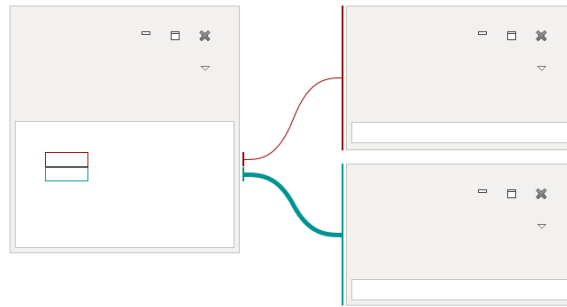


Figure 5.17.: Entry with two sub entries

The solution to this problem is to introduce reversed connections. In contrast to Direct- and PortalConnections, their reversed connections are not printed. They simply serve to navigate from the parent to the child. In *Figure 5.18*, the dashed line represents a reversed connection. It goes from the parent to the child.

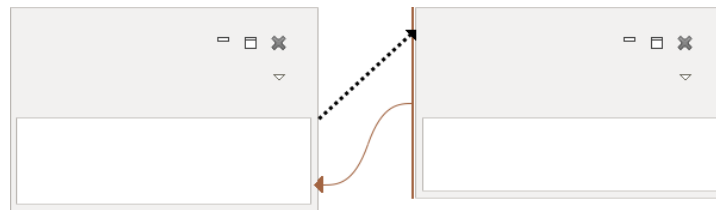


Figure 5.18.: The dashed line represents a reversed connection

### 5.2.2. Handling Events on Bézier Curves

All connections that need to be painted (DirectConnections or PortalConnections) are stored inside the ConnectionCollection. The collection is also responsible for disposing the curves. The curves are SWT paths which have to be cleaned up manually. How it can be tested, if the resources which were allocated are freed correctly, can be seen at 6.5. Another responsibility of the collection is to sort the portals.

To handle events, a listener is added to the ConnectionCollection. Mouse movements have to be tracked to check if a connection is hovered. A connection is hovered if the mouse is over the clickable path of a Bézier curve for DirectConnections (the thick path which is invisible) or a blue portal for PortalConnections. Additionally, it is important to know when the mouse button is pressed and released. Therefore, the interfaces MouseMoveListener and MouseListener need to be implemented by our listener (Figure 5.19).

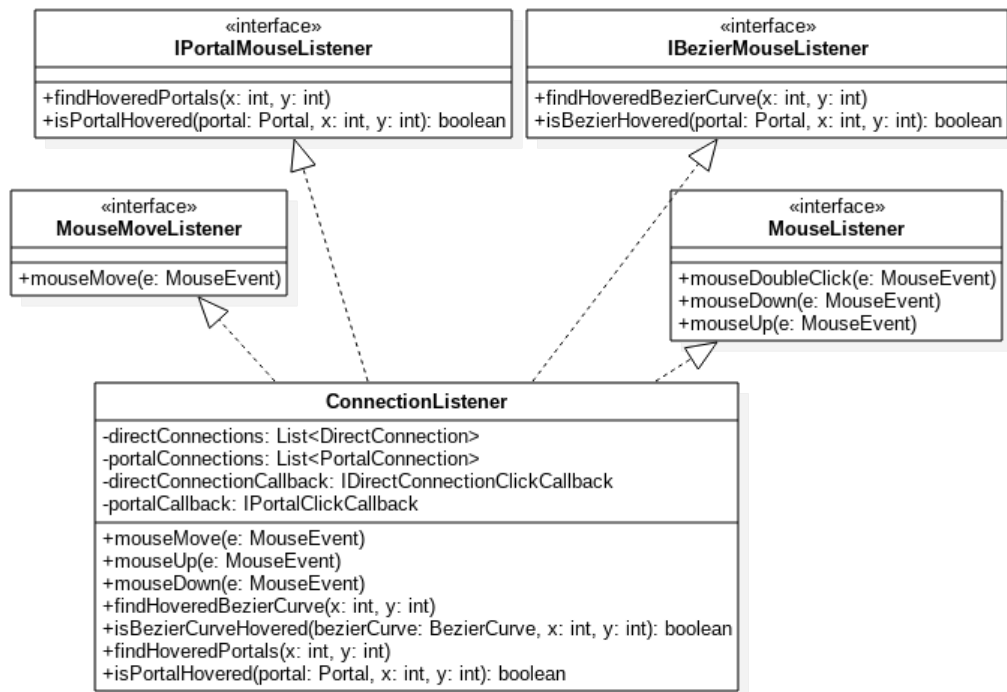


Figure 5.19.: ConnectionListener to allow interaction on Bézier curves and blue portals

There are three actions that can be triggered on a Bézier curve which are shown in a context menu after clicking on the curve. The context menu is implemented similar to the one shown in Figure 5.24. A Bézier curve can be cut directly by clicking on it and selecting the corresponding action.

The problem "Already Opened in Higher Column" shown in 4.2.1 is solved by connecting the entries with a direct connection. However, this can lead to confusion because the path could be hidden by another entry. Therefore, additional actions are offered to travel to the TreeEntry at the start or end of the curve. This helps to ease the navigation. By selecting a curve, the curve will be painted with a different width. This allows to identify the path of a curve even if parts are covered by other entries. All those mentioned features are displayed in *Figure 5.20*.

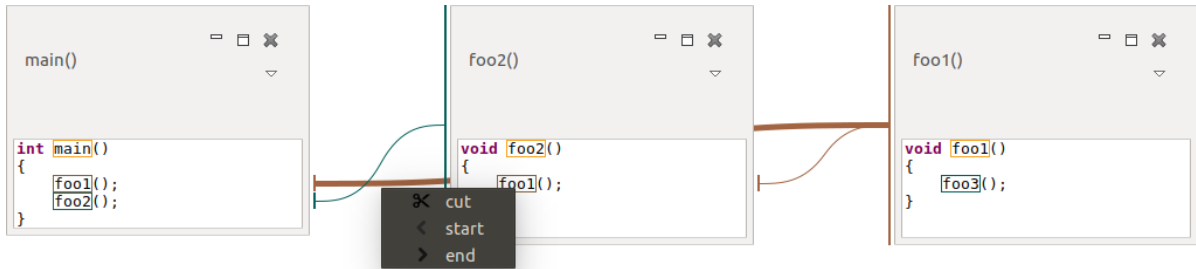


Figure 5.20.: Selected curve is marked and actions can be triggered over a context menu

### 5.2.3. Ordering Entries

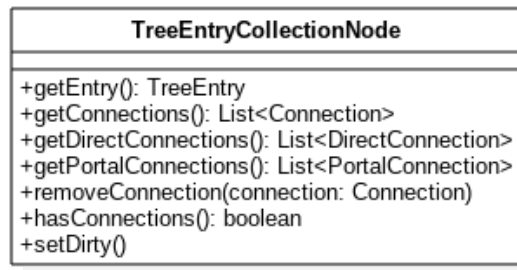


Figure 5.21.: Class diagram TreeEntryCollectionNode

The ordering of the entries is not as static as needed to follow the approach that uses weights for ordering (3.2.3) anymore. The column where an entry resides was fixed before but if multiple connections target the same entry, the column of the entry could change after closing a connection. The problem is shown in *Figure 5.22* where the entry in column 2 (the last column) goes back to column 1 if the connection between the 2nd and the 3rd entry (from the left) gets removed.

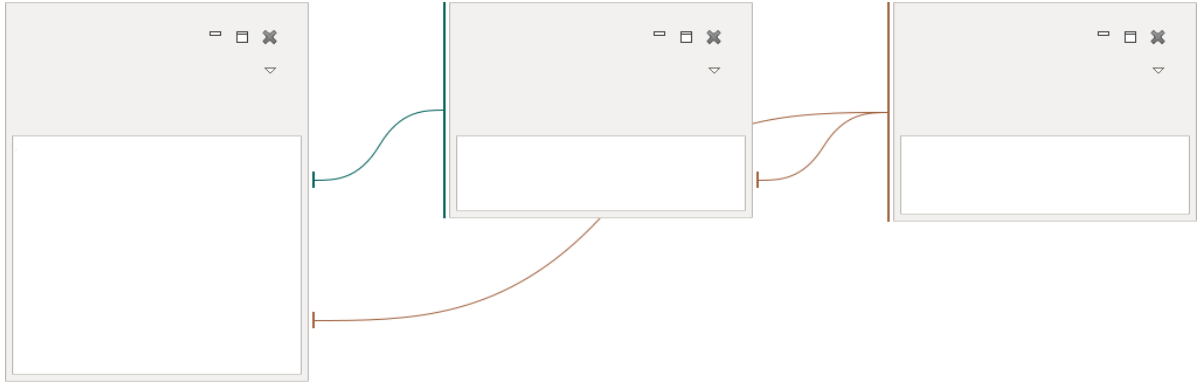


Figure 5.22.: Problem 1: Changing column.

In contrast to the sorting method with the weight array, an entry can have more than one parent. The position of the entry depends on the parent with the smaller y coordinate (in a coordinate system where top left is 0/0). If multiple connections originate from the same entry, the position depends on the index of the rectangle as it was before.

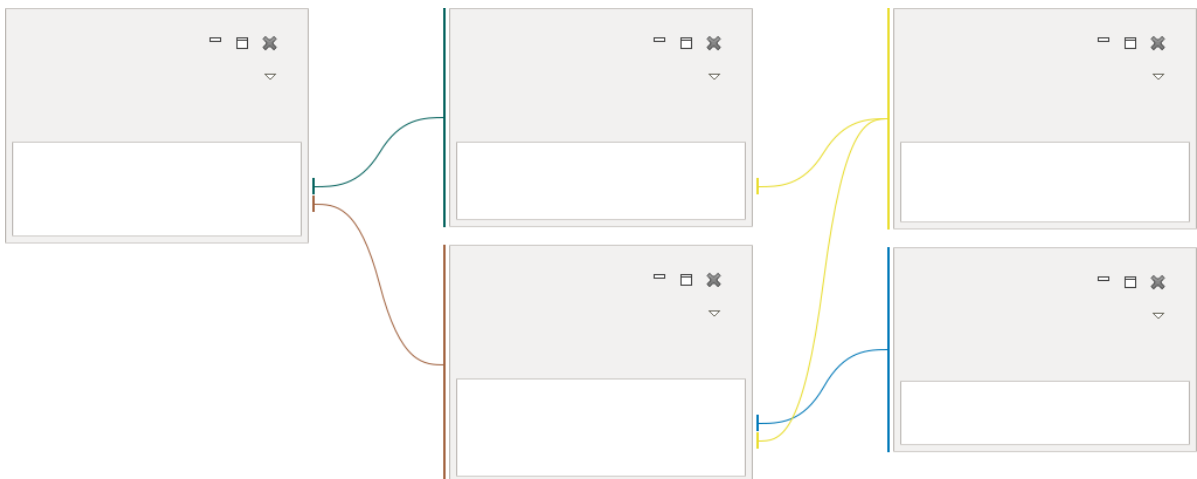


Figure 5.23.: Problem 2: Changing order inside column

The order the entries following method is used. To calculate the column (the x position) for an entry, all the direct connections are taken until the root node is reached. At the end the column id corresponds to the longest path. For the last TreeEntry in *Figure 5.22* there are two possible paths to reach the root node. One has the value 1 and directly travels to the root node. The other goes over another entry which corresponds to the value 2. This means the column number of the entry is 2. The maximum path is stored inside the node. If another node calculates the maximum path, it can use the already stored maximum paths of the other nodes. The setDirty method ?? is used to mark the maximum path as dirty. That means it cannot be taken by other nodes and it has

to be recalculated. The y position of the TreeEntries depend on the y position of their parents. The TreeEntry which has the parent with the smallest y value is positioned on top. Should more than one TreeEntry have the same parent, the smallest position of an active rectangle decides the position.

#### 5.2.4. Portals

Portals, just like Bézier curves, are also interactive. If only one blue portal is opened in a line, a click directly jumps to the TreeEntry with the corresponding orange portal. Sometimes more than one portal can be opened in the same line. If this happens, a context menu *Figure 5.24* helps to select the portal to jump to as shown in *Figure 5.26*.

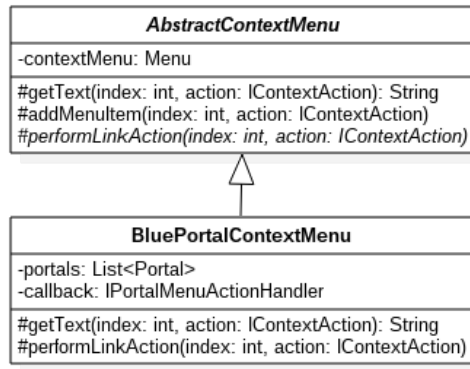


Figure 5.24.: Context menu abstraction and concrete implementation to show after clicks on the blue portals

At the moment, there is only one action available on portals. However, it was designed to easily add more. By adding values to the enumeration `PortalAction` (*Figure 5.25*), it is possible to extend the available actions.

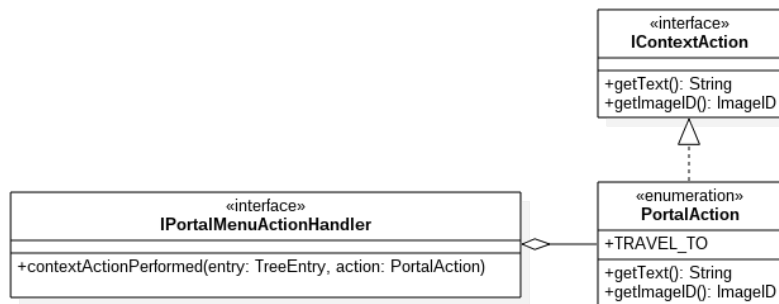


Figure 5.25.: Handling clicks on portals

Those actions are handled with a switch case in the TreeTemplateView as shown in *Listing 5.14*.

```
1 @Override
  public void contextActionPerformed(TreeEntry treeEntry, PortalAction
    action) {
3     switch (action) {
        default:
5         case TRAVEL_TO:
            scrollToEntry(treeEntry);
7         break;
    }
9 }
```

Listing 5.14: Handling clicks on a portal

#### 5.2.4.1. Multiple Portals on Same Line

As mentioned before, it is possible to open more than one portals per line.

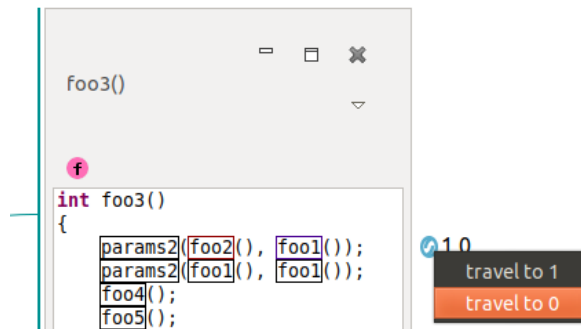


Figure 5.26.: Two portals on the same line

The layout of the TreeEntries could be destroyed if the text of the portals is longer than the gap between two entries. To avoid that scenario, not more than three portal numbers are shown. If more than three portals are shown, only three destinations are shown as pictured in *Figure 5.27*. However, by clicking on it, it is still possible to travel to all locations.



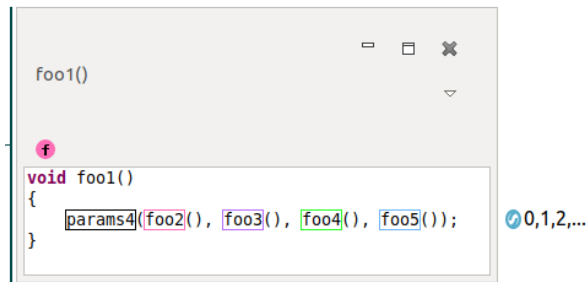


Figure 5.27.: ... indicates that more portals are opened than shown

The size of the portals changes with the font size. If the font size gets bigger, the portals do as well. To avoid drawing a portal over a TreeEntry, the gap between two entries depends on the font size.

### 5.2.5. Hiding Rectangles

Allowing to trace more nodes leads to an even more overloaded view if all the traceable nodes are boxed. The option to only show active rectangles (the rectangle serves as a start of a connection) was added. TemplatorRectangle stores the current state to indicate if it has to be printed or not.

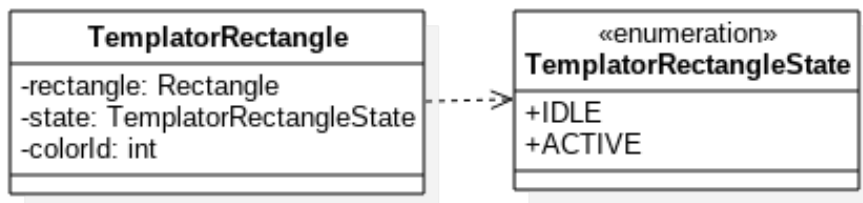


Figure 5.28.: Class to represent the rectangles

However, sometimes it is useful to directly see all the rectangles, especially for debugging purposes. For this reason, a preference allows to enable or disable the drawing of all rectangles. How the preferences are implemented is shown in 5.2.7.

### 5.2.6. InfoBar

The InfoBar shows the type of the TreeEntry and the destination portal (orange). Since only two features are placed there, it could also have been put somewhere else. But it would not allow anything other to be added afterwards. Therefore, it allows an easy way to add additional interactive icons or information to a TreeEntry. The two already added functionalities are described below.



Figure 5.29.: InfoBar inside a TreeEntry

#### 5.2.6.1. TreeEntry Type

An icon in the InfoBar helps to recognize the type of the TreeEntry. Following types are currently distinguished.

- class
- function
- member function
- class template
- function template
- alias template
- variable template
- lambda

#### 5.2.6.2. Destination Portal

A PortalConnection allows travelling from a blue to an orange portal or vice versa. While the blue portal is next to the rectangle, the orange portal is in the InfoBar. Next to the portal, there is a number which helps to see how they are connected (same numbers are connected). Like Bézier curves, it is also possible to follow portals by clicking on them.

### 5.2.7. Templator Preferences

The enum `TemplatorPreference` is used to hold the keys and descriptions for the preferences.

The preference page is provided by `TemplatorPreferencePage` which extends `FieldEditorPreferencePage` and implements the interface `IWorkbenchPreferencePage`. The `init` method is used to set the preference store. The preference store is retrieved from the plug-in. Preferences set inside the store are persisted along with the workbench [Cre01]. The page was added to C/C++ - Templator and offers preferences to disable tracing of normal functions, classes or of auto specifiers (*Figure 5.30*). This was added to be able to reduce loading time if the user is not interested in following functions for example. However, if tracing functions is disabled, it is still possible to select them as a starting point.

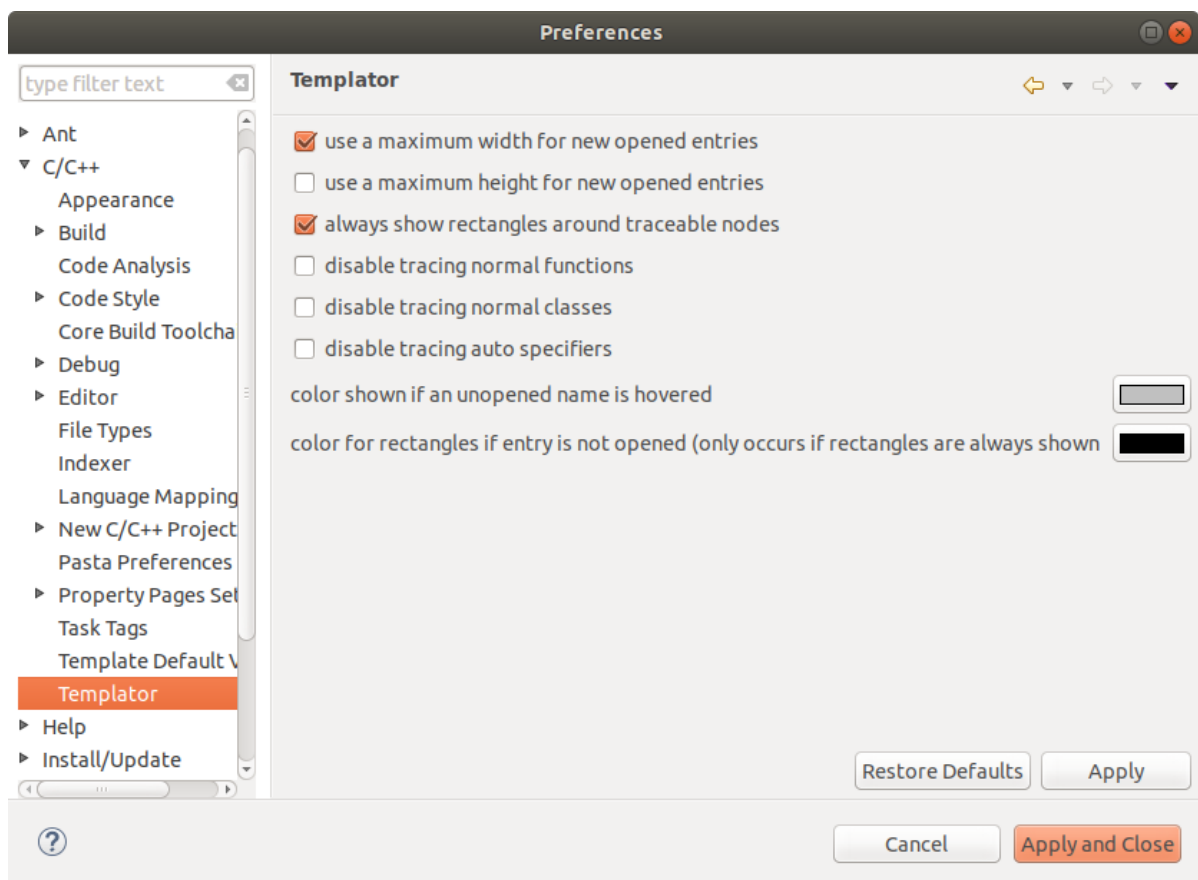


Figure 5.30.: PreferencePage for the Templator plug-in

## 6. Testing

All the tests for the Templator plug-in reside in the package `com.cevelop.templator.test`. Most of the tests extend the functionality offered by `CDTTestingTest` which is included in the plug-in IFS CDT-Testing [IFS]. *Figure 6.1* shows an overview of the test classes used to test the Templator plug-in. Those classes are not the tests itself, the solely provide the infrastructure to write them.

With the help of the following sections, the test classes are explained which makes adding additional tests easier. `CDTTestingTest` is not explained, but if needed, more information about it can be retrieved in [IFS14]. Also `TemplatorSimpleTest` and `FunctionTemplateResolutionTest` are not explained. Both were created during the student research project thesis and remained untouched. A description of those test classes can be found in [BJ14].

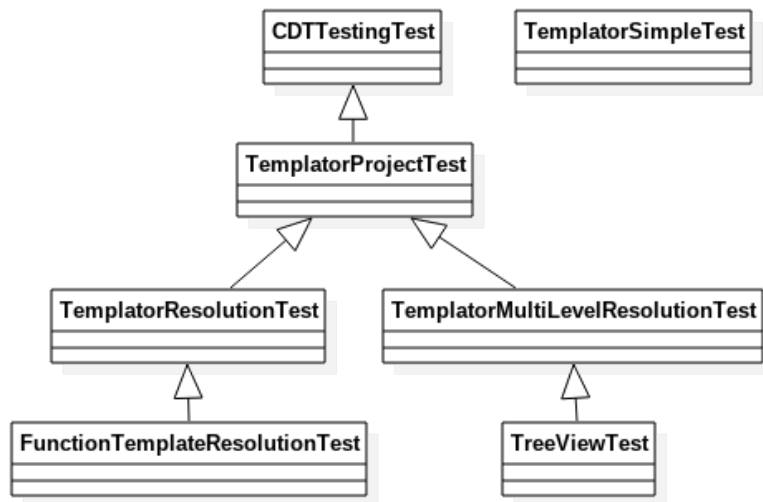


Figure 6.1.: Overview of the test classes

## 6.1. TemplatorProjectTest

The class `TemplatorProjectTest` is responsible for setting up the test environment. It initializes the `ASTAnalyzer` and stores the top level definitions. How a `TemplatorProjectTest` is setup is shown in *Figure 6.3*. The single steps are discussed below.

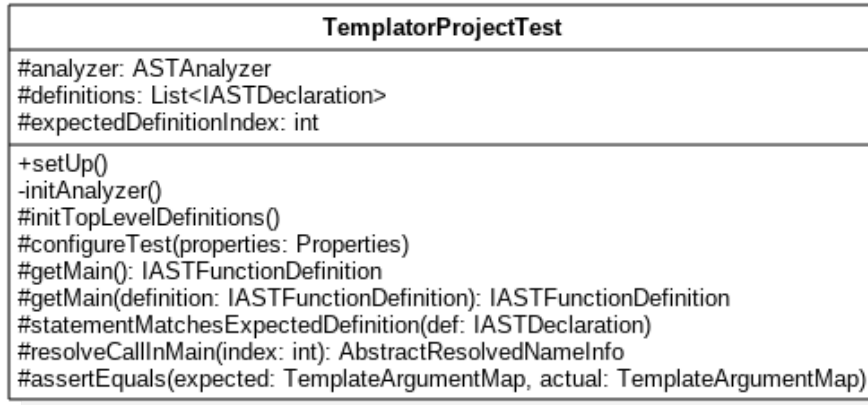


Figure 6.2.: Class diagram of `TemplatorProjectClass`

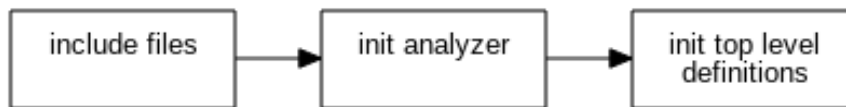


Figure 6.3.: Setup for a `TemplatorProjectTest`

### 6.1.1. Include Files

The folder "filesOutsideWorkspace" is added as an include directory.

### 6.1.2. Init Analyzer

An instance of the class `ASTAnalyzer` is initialized with a translation unit created by parsing the source of the active file. The active file is the file created inside the `rts` file.

### 6.1.3. Init Top Level Definitions

All the declarations can be retrieved from the AST. They are stored to compare them with the definition of the created `AbstractResolvedNameInfo`.

### 6.1.4. Properties

In addition, the `configureTest` method is overridden to allow defining the expected definition directly inside the rts file. This is useful if more than one test is defined in an rts file. An example is shown in *Listing 6.1* where the expected definition in the second test is definition 1 (the definition of the alias template). The first test however does not explicitly define it, therefore the default value 0 is taken.

```
1  //!alias template 1
   //@MyFileName.cpp
3  template<class T>
   using pointer = T*;
5
   int main() {
7     pointer<int> x;
   }
9
   //!alias template 2
11  //@.config
   definition=1
13  //@MyFileName.cpp
   template<typename T1, typename T2, typename T3, typename T4>
15  struct Quadruple {
   };
17
   template <typename T1>
19  using sameTypeQuad = Quadruple<T1, T1, T1, T1>;
21
   int main() {
       sameTypeQuad<int> x { };
23 }
```

Listing 6.1: rts file that uses the definition property

## 6.2. TemplateResolutionTest

In the setup of the `TemplatorResolutionTest`, the first found rectangle (the first possible `IASTName`) in the main function is resolved and stored. All the methods offered by this class are checking this first found `AbstractResolvedNameInfo`.

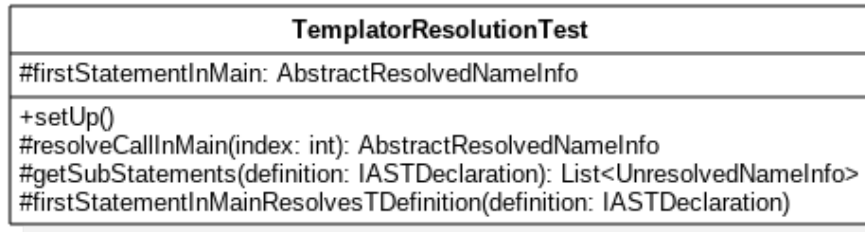


Figure 6.4.: Class diagram `TemplatorResolutionTest`

An example of a test with the input shown in *Listing 6.2* is displayed in *Listing 6.3*.

```
1  //!ExplicitSpecializationSelection
2  //@MyFileName.cpp
3  template <typename T>
4  struct Stack {
5  };
6
7  template <typename T>
8  struct Stack<int> {};
9
10 int main() {
11     Stack<int> stack{};
12 }
```

Listing 6.2: Input for the test

```
1 public class ExplicitClassTemplateSpecializationTest extends
2     TemplatorResolutionTest {
3
4     @Test
5     public void resolvesToFirstDefinition() {
6         firstStatementInMainResolvesToDefinition(definitions.get(1));
7     }
8 }
```

Listing 6.3: Test to check if the explicit specialization is selected

This test checks if the definition saved in the created `AbstractResolvedNameInfo` is the same as the definition at the given position (in this case 1). It checks that `Stack<int>` resolves to the explicit specialization for `T = int` and not to the primary template. The

test class can be used if simple examples are tested, which do check the first rectangle. For more complex tests using `TemplatorMultiLevelResolutionTest` explained in *section 6.3* should be considered.

## 6.3. TemplatorMultiLevelResolutionTest

In comparison to the `TemplatorResolutionTest`, this test offers more flexibility. `TemplatorResolutionTest` always resolves the first rectangle. While this is often what is needed, sometimes the first one does not have any meaning for the test.

TemplatorMultiLevelResolutionTest
#rectangleIndex: int
#configureTest(properties: Properties)
#getSubnameInMain(index: int): AbstractResolvedNameInfo
#getMainAsResolvedName(): ResolvedName
#getSubName(nameInfo: AbstractResolvedNameInfo, index: int): AbstractResolvedNameInfo
#statementMatchesExpectedDefinition(definition: IASTDeclaration, index: int)

Figure 6.5.: `TemplatorMultiLevelResolutionTest`

Another important difference is that `TemplatorResolutionTest` does use its own function to create the `AbstractResolvedNameInfo`. This is done to reduce the work. However, tests are mostly really small and resolving the nodes using `searchSubNames` as described in 3.1.3.1 does not add much of an overhead.

```

2  #!/Fibonacci
3  //@MyFileName.cpp
4  template <int N>
5  constexpr int fibonacci =
6      (fibonacci<N-1>) + fibonacci<N-2>;
7
8  template<>
9  constexpr int fibonacci<1> = 1;
10 template<>
11 constexpr int fibonacci<0> = 0;
12
13 int main() {
14     fibonacci<4>;
15 }

```

Listing 6.4: a



```

1 public class VariableTemplateTest extends
    TemplatorMultiLevelResolutionTest {

3     @Test
    public void fibonacci_4() throws TemplatorException {
5         statementMatchesExpectedDefinition(getSubnameInMain(0).
            getDefinition(), 0);
    }

7

9     @Test
    public void fibonacci_4_1() throws TemplatorException {
        AbstractResolvedNameInfo fibo3 = getSubname(getSubnameInMain(0), 0)
        ;
11         statementMatchesExpectedDefinition(fibo3.getDefinition(), 0);
    }

13

15     @Test
    public void fibonacci_2_2() throws TemplatorException {
        AbstractResolvedNameInfo fibo2 = getSubname(getSubnameInMain(0), 1)
        ;
17         AbstractResolvedNameInfo fibo0 = getSubname(fibo2, 1);
        statementMatchesExpectedDefinition(fibo0.getDefinition(), 2);
19     }
}

```

Listing 6.5: VariableTemplateTest

The first test resolves the first name in main (fibonacci<4>) and checks if the definition is the definition at position 0 (the primary template).

In the second test, the first sub name in the definition of fibonacci(4) is resolved (fibonacci<4-1>). Again, this should match with the primary template.

The last test goes a level deeper. The definition for fibonacci<0> is retrieved by always travelling to the second sub name. This definition should now match to the explicit specialization for  $N = 0$ .

Of course it could also be tested by writing fibonacci<0> in the main directly. But we want to make sure that the non-type template parameters get replaced correctly.

### 6.3.1. Properties

Like the property for the definition, this class additionally parses a property that can be used to specify the rectangle (the index of the sub name).

## 6.4. TreeViewTest

To test the connections and the rectangles the TreeViewTest (*Figure 6.6*) is used. It uses an instance of TreeTemplateView and offers methods to add and remove TreeEntries.

TreeViewTest
<u>-templateView: TreeTemplateView</u>
#createView() #getRootEntry(): TreeEntry #findChild(treeEntry: TreeEntry, index: int): TreeEntry #clickName(treeEntry: TreeEntry, index: int) #closeAllEntries() #closeAllSubEntries(treeEntry: TreeEntry) #closeEntry(treeEntry: TreeEntry) #assertColors() #assertRectangleStates(expected: RectangleStateCollection) #assertTreeViewTestInfo(expected: List<TreeViewTestInfo>)

Figure 6.6.: Test class to test the TreeTemplateView

Because the opening and removal of entries are tested, the tests sometimes have to be executed in a defined order. For example if we want to build a tree of TreeEntries by opening all sub names and test the state after each step. The annotation `@FixMethodOrder` allows the user to choose the order of execution. The only way to sort them reliably is sorting it by name as shown in *Listing 6.6*.

```
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
```

Listing 6.6: Annotation to sort the methods by name

By default the rts file is chosen depending on the class name of the test. But for testing the TreeTemplateView the code inside the TreeEntries does not really matter. Therefore, the same rts files are often used. This can be done by explicitly defining the rts file as shown in *Listing 6.7*.

```
1 @RunFor(rtsFile = "/resources/SourceFiles/integrationtest/view/tree/  
ExampleWithoutPortals.rts")
```

Listing 6.7: Defining the rts file

In 3.1.3 it is explained that loading a TreeEntry is done asynchronously. While this is needed to not block the UI-thread, it is not perfect for testing purposes. A class SyncEntryLoader is introduced which loads an entry synchronously. Instead of passing the AsyncEntryLoader an implementation of IEntryLoader is passed. The IEntryLoader interface only declares a start method which is triggered to load an entry. Like that it is possible to use a synchronous way to load the entries for the tests.

The following example shows a test with the rts file in *Listing 6.8*.

```

1 int foo2() {}
3 int foo1() {
    foo2();
5 }
7 int main() {
    foo1();
    foo2();
9 }

```

Listing 6.8: rts file for testing the removal of connections

The first test *Listing 6.9*, creates the view with the help of the ViewOpener which is described in 3.1.1. The loading of the entries is set to synchronous and the function main is resolved. After that the ViewData is created and set as the root data of the view. This root entry then is retrieved and stored as entry0. The sub entries with index 0 and 1 are opened by simulating a click on the name. Because entry1 is used to open its sub names it has to be found by retrieving the child with id 0 from entry0. Because all this is difficult to imagine, a diagram was added as a commentary above the method. Such diagrams can be created with asciiflow [asc]. After all the entries are opened now, it has to be checked if it is what is expected.

```

//
//      +-----+      +-----+      +-----+
//      |         |      |         |      |         |
4 //      |         +-----+ 1  +-----+      |
//      |         |      |         |      |         |
6 //      |  0    |      +-----+      |  2    |
//      |         |      |         |      |         |
8 //      |         +-----+      |         |
//      +-----+      +-----+
10 //
@Test
12 public void test01_openAll() throws TemplatorException {
    createView();
14    entry0 = getRootEntry();
    clickName(entry0, 0);
16    entry1 = findChild(entry0, 0);
    clickName(entry1, 0);
18    clickName(entry0, 1);

20    List<TreeViewTestInfo> expected = new ArrayList<>();
    expected.add(new TreeViewTestInfo(0, 0, 2, 0, 0));
22    expected.add(new TreeViewTestInfo(1, 1, 1, 0, 0));
    expected.add(new TreeViewTestInfo(2, 2, 0, 0, 0));
24    assertTreeViewTestInfo(expected);
}

```

Listing 6.9: Opening all entries

To be able to compare the `TreeEntries` with something we expect and to offer a way to simply express what should be inside a `TreeEntry` it is simplified. For that reason `TreeViewTestInfo` was introduced. The meaning of the `TreeViewTestInfo` in line 23 has to be interpreted the following (the following numbers correspond to the arguments 2, 2, 0, 0, 0).

- 2: The `TreeEntry` is in the column with the number 2.
- 2: 2 direct connections arrive at the entry.
- 0: No direct connection leaves the entry.
- 0: No portal connection arrives at the entry.
- 0: No portal connection leaves the entry.

The entries in the `TreeTemplateView` are also reduced to this to allow comparison.

```
1 @Test
2 public void test02_checkRectangleColors() {
3     assertColors();
4 }
```

Listing 6.10: Testing the colors

The second test (*Listing 6.10*) tests if the colors are correct. It compares the color of the rectangle with the color of the Bézier curve (only for direct connections) and the color of the destination entry. It passes only if all colors are the same.

```
1 @Test
2 public void test03_checkStates() {
3     RectangleStateCollection expected = new RectangleStateCollection();
4     expected.addEntry(ACTIVE, ACTIVE);
5     expected.addEntry(ACTIVE);
6     expected.addEntry();
7     assertRectangleStates(expected);
8 }
```

Listing 6.11: Testing the rectangle states

By default rectangles are only shown if active. To test this another reduction was introduced. The `RectangleStateCollection` is used to express what rectangle states are expected. (ACTIVE, ACTIVE) for example means that the `TreeEntry` has two rectangles and both are opened at the moment. An empty entry means there is no rectangle to click for this `TreeEntry`. This way it is easily possible to test if the rectangle have the correct state.

```

2  //      +-----+      +-----+
3  //      |         |      |         |
4  //      |         +-----+ 1 |
5  //      |         |         |
6  //      | 0       |         +-----+
7  //      |         |         |
8  //      |         +-----+      +-----+
9  //      +-----+ |         |
10 //          +-----+ 2 |
11 //          |         |
12 //          +-----+
13 //
14 @Test
15 public void test04_removeConnection() {
16     clickName(entry1, 0);
17
18     List<TreeViewTestInfo> expected = new ArrayList<>();
19     expected.add(new TreeViewTestInfo(0, 0, 2, 0, 0));
20     expected.add(new TreeViewTestInfo(1, 1, 0, 0, 0));
21     expected.add(new TreeViewTestInfo(1, 1, 0, 0, 0));
22     assertTreeViewTestInfo(expected);
23 }

```

Listing 6.12: Testing the removal of a connection

The test in *Listing 6.12* removes the connection between entry1 and entry2 (it is easy to see that if both diagrams in the comments are compared). If we analyze the `TreeViewTestInfo` of entry2 again, it is visible that the expected result is a column change and losing 1 direct connection. The connections are not tested directly, but it is possible to see that the correct connection was removed because one entry loses an incoming connection and the other loses an outgoing connection.

```

1 @Test
2 public void test05_checkStates() {
3     RectangleStateCollection expected = new RectangleStateCollection();
4     expected.addEntry(ACTIVE, ACTIVE);
5     expected.addEntry(IDLE);
6     expected.addEntry();
7     assertRectangleStates(expected);
8 }

```

Listing 6.13: Checking if one rectangle is IDLE now

The last test checks the state of the rectangles again. Entry1 has one rectangle and this should now be inactive because the connection just got removed.

The colors of the rectangles could be tested again, but it is not done here because it would be the same as in the second test.

## 6.5. Sleak

There is a platform-specific limit on the amount of allocatable resources. Therefore, it is important to free the resources for graphical objects for example widgets, images and fonts. The mantra of SWT is "if you created it, you dispose it". An indication for a possible resource leak could be the exception below [Ani].

```
org.eclipse.SWTError.No more handles
```

The exact location causing the errors is difficult to find. A tool that helps to find memory leaks in applications using SWT is Steak. It allows monitoring graphics resources and is developed by the team that also develops SWT.

Sleak is contained in SWT Tools which can be installed from the update site [Too]. To examine the Templator plug-in, Sleak has to be installed in the Eclipse instance where the Templator plug-in is available (the Eclipse instance which is started from the Eclipse instance where we edit the Templator code). Additionally, the run configuration of the plug-in has to be adapted. Tracing must be enabled for org.eclipse.ui for debug and trace/graphics. This has to be done in the other Eclipse instance (the instance where the code is altered).

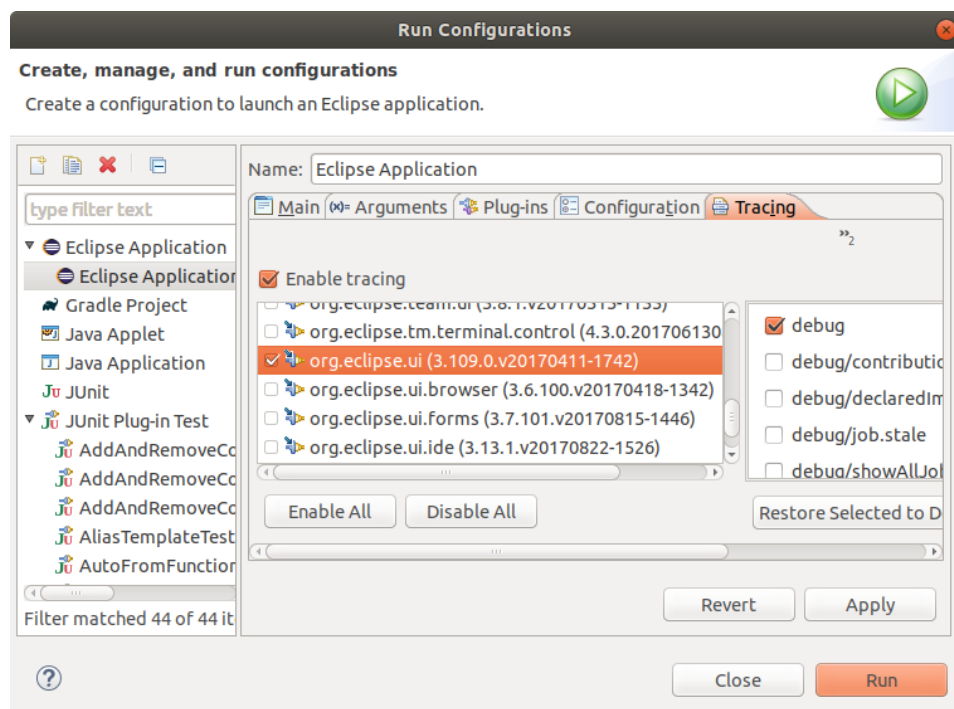


Figure 6.7.: Enabling tracing

The Sleak view offers an option to show a diff. With this it is possible to only show the resources which were allocated by the Templator plug-in. The stack helps to find the location where it was allocated. All this information helps to find a leak which is much harder to do without Sleak.

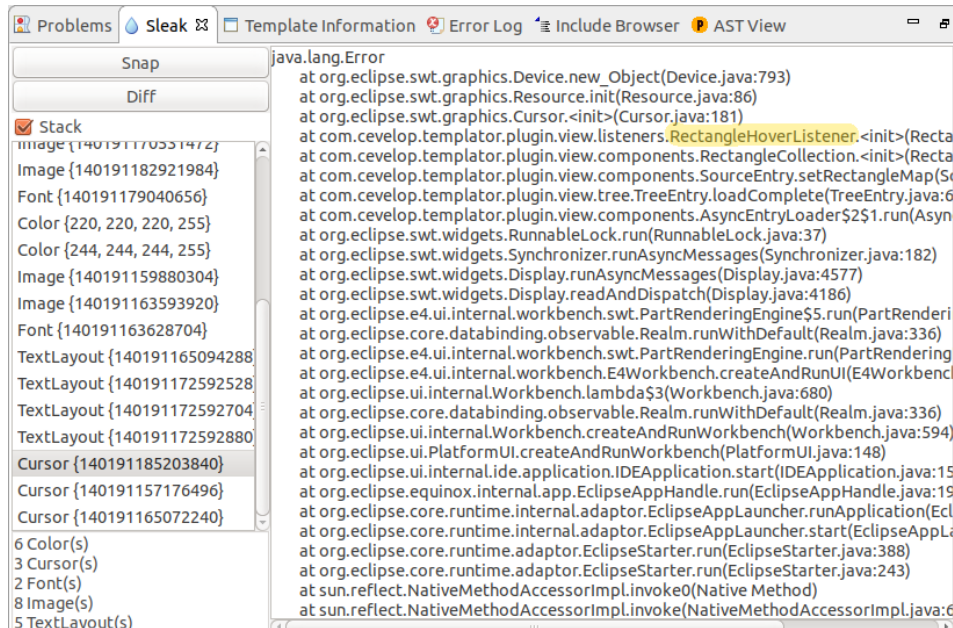


Figure 6.8.: Sleak shows the allocated resources

## 6.6. Analyzing Boost.Hana Examples

Boost.Hana is a modern library for C++ metaprogramming. It uses C++11/14 implementation techniques and idioms like variable templates and automatically deduced return types. That makes it the perfect candidate to test the Templator plug-in with.

The preferences for the rectangles was set to "always show". This way it is directly visible what nodes can be traced.

The following examples are taken from the Boost.Hana tutorial [Dio17].

The first example shown in *Listing 6.14* represents a generic serializer and is quite simple. It iterates over the members of a user-defined type and writes them to a stream.

```

1 #include <iostream>
2 #include <boost/hana.hpp>
3 namespace hana = boost::hana;
4
5 struct Person {
6     BOOST_HANA_DEFINE_STRUCT(Person,
7         (std::string, name),
8         (int, age)
9     );
10 };
11
12 auto serialize = [](std::ostream& os, auto const& object) {
13     hana::for_each(hana::members(object), [&](auto member) {
14         os << member << std::endl;
15     });
16 };
17
18 int main() {
19     Person john { "John", 30 };
20     serialize(std::cout, john);
21 }

```

Listing 6.14: Serialize example

At first glance this should be possible to analyze with the Templator plug-in because classes and lambda expressions are possible to trace now. If we set the starting point to the main function, the situation pictured in *Figure 6.9* is presented.

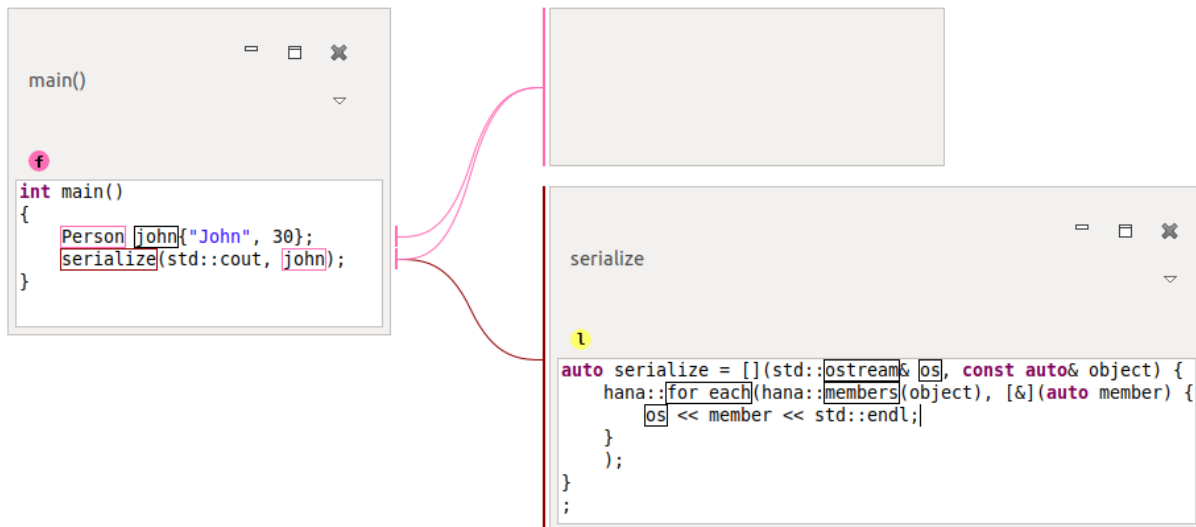


Figure 6.9.: Serialize example analyzed with the Templator plug-in

The class `Person` cannot be displayed because a macro is used. The positions of the rectangle are out of bounds and an empty entry is shown. Another thing that is catching



the eye is the parameter member, that cannot be traced. Boost.Hana often uses generic lambdas. It would be an improvement if the arguments of the generic lambda were replaced just like it is done for templates.

```

1 #include <boost/hana.hpp>

3 namespace hana = boost::hana;

5 template <typename T>
6 struct basic_type {
7     // empty (for now)
8 };
9 basic_type<int> Int{};
10 basic_type<char> Char{};

11
12 template <typename T>
13 constexpr basic_type<T*> add_pointer(basic_type<T> const&)
14 { return {} }; }
15 template <typename T>
16 constexpr auto is_pointer(basic_type<T> const&)
17 { return hana::bool_c<false>; }
18 template <typename T>
19 constexpr auto is_pointer(basic_type<T*> const&)
20 { return hana::bool_c<true>; }
21
22
23 int main() {
24     auto has_name = hana::is_valid([](auto&& x) -> decltype((void)x.name)
25     { });
26     basic_type<int> t{};
27     auto p = add_pointer(t);
28     BOOST_HANA_CONSTANT_CHECK(is_pointer(p));
29 }

```

Listing 6.15: Mixed example

The second example in *Listing 6.15* is only a naive version of the actual functionality provided by hana, but it is enough to show some problems.

In *Figure 6.10*, almost all the nodes can be traced but again a macro is causing problems and `is_pointer` is not traceable. The rectangle for the `()` operator is not positioned correctly.

```

f
int main()
{
    auto has_name = hana::is_valid([](auto&& x) -> decltype((void)(x.name)) {
    }
    );
    basic_type<int> t{};
    auto p = add_pointer(t);
    BOOST_HANA_CONSTANT_CHECK(is_pointer(p))BOOST_HANA_CONSTANT_CHECK(is_pointer(p));
}

```

Figure 6.10.: Rectangle for operator () is off

In *Figure 6.11*, the reason for the offset in the rectangle can be seen. The type of the template argument for F is not printed correctly.

```

is_valid_t
0
struct is_valid_t
{
    template<typename F> constexpr auto operator ()(F&&) const;
    template<typename F, typename... Args> constexpr auto operator ()(F&&, Args...&&) const
};

operator ()<F,Args...>
typename F = {main.cpp:486}::{main.cpp:520}
typename... Args =
?
template<typename F = {main.cpp:486}::{main.cpp:520}, typename... Args>
constexpr auto is_valid_t::operator ()(F&&, Args...&&) const
{
    return type_detail::is_valid_impl<F&&,Args...&&>(int{});
}
0

```

Figure 6.11.: typename F and typename... Args not correct

For the last example a kind of a switch statement that is able to process `boost::any`s was used. Dispatching to the function associated to the dynamic type of the any is the goal. The code can be found in *Listing E.1*. The following comparison shows what nodes can be traced with the old version of the Templator and what is possible now. It also shows that the features that were added are very important to be able to analyze code written with Boost.Hana

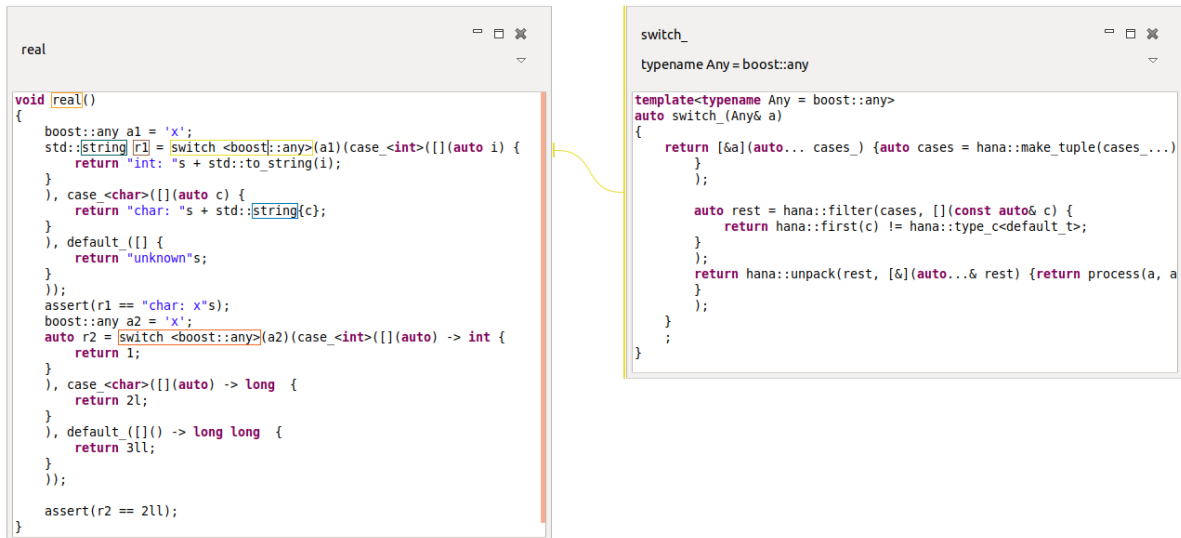


Figure 6.12.: Nodes resolvable with the old version

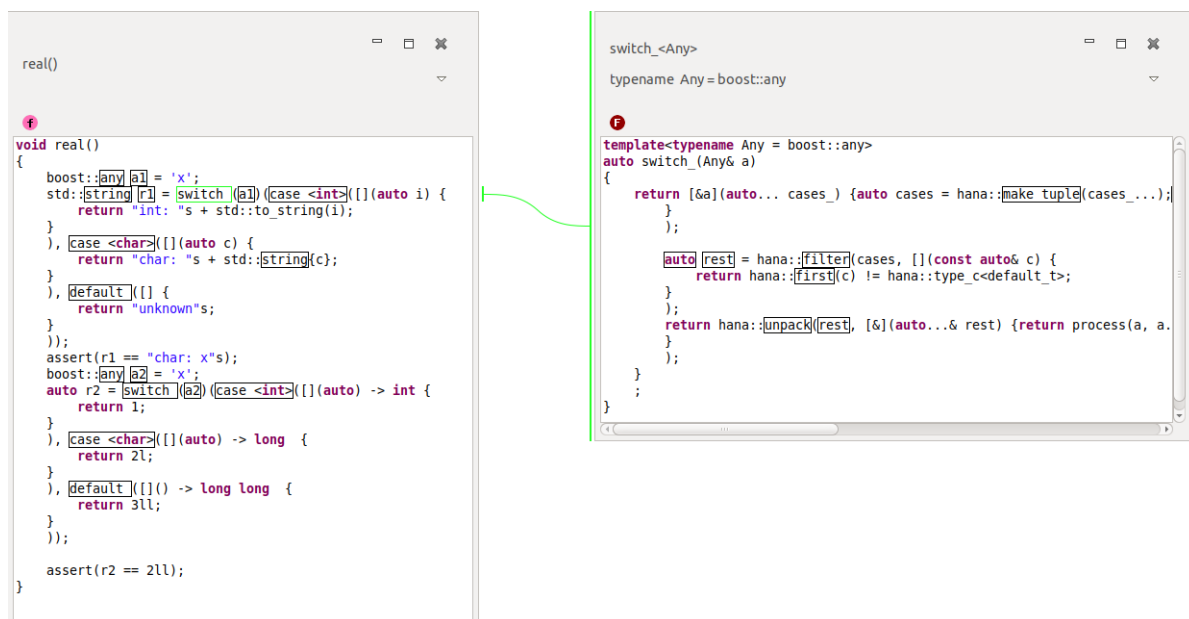


Figure 6.13.: Nodes resolvable with the current version

Even if the picture in *Figure 6.13* looks promising, there are still a lot of cases where the Templator plug-in fails and on the way to the tool for debugging template metaprograms a lot of examples have to be analyzed and step by step debugged through the code of the Templator plug-in.

## 7. Conclusion

This chapter describes the results of this thesis and also what else could be done in the future to extend the plug-in with more functionality.

### 7.1. Results

Support for alias templates and variable templates as well as support for automatically deduced types was added to the Templator plug-in.

It is possible to trace normal classes and lambda expressions now. This means the Templator can also be used to browse code without templates. This is an improvement because the starting point does not always have to be set to a different one.

The user interface was refactored and improved to handle the new challenges while getting less overloaded at the same time. If two nodes resolve to the same type, only one entry is created.

With the help of preferences, the functionality of the Templator plug-in can be restricted. Also boxes around the text are by default not shown for every traceable node anymore, which leads to a better overview. The user can hover a name to see if it can be traced or not. The old way that showed all the rectangles at once can still be selected in the preference page for the Templator.

### 7.2. Further Work

While the functionality of the Templator plug-in is good for small projects, it still has problem with analyzing code from the standard library or template metaprograms. The following list contains improvements that could be done to further increase the power of this tool.

- Adding support for generic lambdas
- Improving the resolving of member functions

Additionally, there are still open topics that could not be solved during this thesis:

- Adding support for class template argument deduction
- View that is capable of showing all instantiations of a selected template inside the project (direct and indirect).

Also, the tests have to be updated to the new version of CDTTesting that was released close to the end of the thesis.

# Appendices

# A. Package Structure

The Templator plug-in is structured into following packages listed below. To increase clarity, the common package `com.cevelop.templator` is omitted intentionally and packages that do not contain anything (just other packages) are not listed. An overview of the package structure can be gained by inspecting the package diagram figured in *Figure A.1*.

## **plugin**

This is the parent package. All other packages reside inside this. It contains the activator class which controls the life cycle of the plug-in.

## **plugin.asttools**

Classes to retrieve information about nodes of the abstract syntax tree, with or without the help of the index, are contained in this package.

## **plugin.asttools.data**

The data package contains classes to store information about function calls and template instances.

## **plugin.asttools.formatting**

The text shown in the entries of the view offered by the Templator plug-in have a separate formatting. All the classes dealing with the formatting of the text are in this package.

## **plugin.asttools.resolving**

Classes focusing on resolving and instantiating deferred bindings are located in this package.

## **plugin.asttools.resolving.nametype**

The nametype package offers the functionality to retrieve the names of definitions and types.

## **plugin.asttools.templatearguments**

This package only contains the TemplateArgumentMap. It maps template parameters to deduced types.

## **plugin.asttools.type.finding**

All the possible classes to represent the found classIASTNames are contained in this package.

## **plugin.handler**

The handler package contains the handlers which are called with the help of commands and a helper to open the view.

## **plugin.logger**

Loggers and exceptions for the Templator plug-in can be found in this package.

## **plugin.util**

This package contains general utilities such as retrieving components from the workbench and helper methods to ease the usage of reflection for example.

## **plugin.view**

The main package contains all the packages and classes that are related to the view of the Templator plug-in.

## **plugin.view.components**

Components that are used for the view like buttons, Bézier curves, loading bars etcetera are located in this package.

## **plugin.view.interfaces**

All the interfaces for the view are contained in this package.

## **plugin.view.listeners**

Listeners to allow handling events on entries and rectangles for example are part of this package.



## plugin.view.rendering

Classes that deal with drawing components on the screen or calculating them can be found here.

## plugin.view.tree

This package offers classes that extend the classes in the package plugin.view.components in order to visualize the data in a tree.

## plugin.viewdata

The package contains the class ViewData which is used to prepare the data for the UI as well as classes to modify the data.

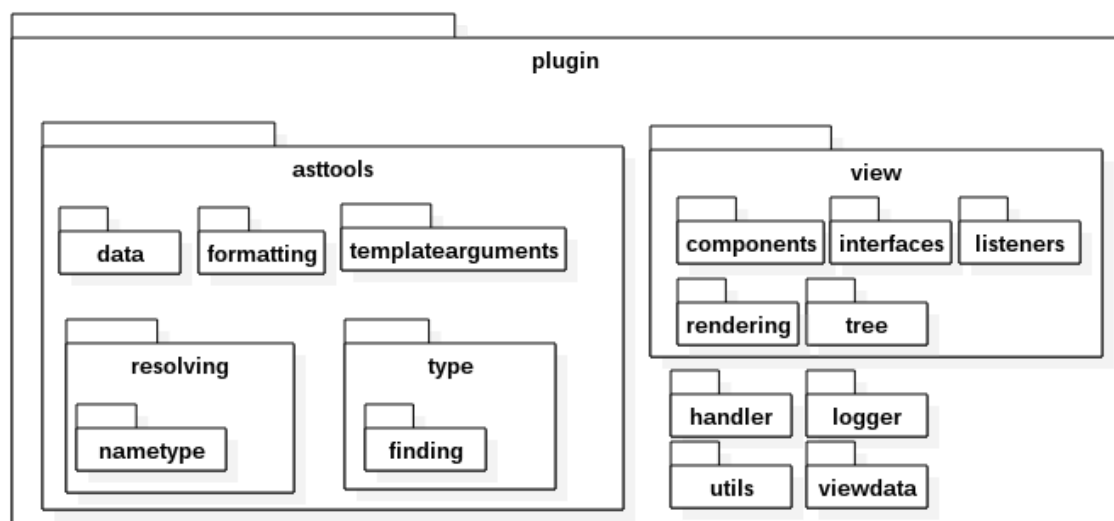


Figure A.1.: Package structure of the Templator plug-in

## B. Templator Classes

This chapter introduces some important classes inside the Templator plug-in.

### B.1. TemplatorPlugin

TemplatorPlugin is an extension of AbstractUIPlugin which is the activator class that controls the plug-in life cycle. In comparison to the activator class created with the wizard, the TemplatorPlugin class takes care of disposing the ColorPalette on stopping the plug-in.

### B.2. ColorPalette

The ColorPalette initializes the colors that are used in the Templator plug-in. Eight colors are initialized for the connections. Those eight colors were not selected randomly. They were chosen to be easily distinguishable even for color blind people. A 15-color palette for color blindness can be found at [MK12]. The colors in *Figure B.1* are colors from the 15-color palette. They are used for the rectangles and the Bézier curves.



Figure B.1.: Colors used to draw the connections

If a rectangle is hovered, a brighter color should be used to hover the text to make it more pleasant to read. Those colors were created by adding more white to the colors selected above.



Figure B.2.: Colors used for highlighting

## B.3. RelevantNameCache

To decrease the loading times to prepare the data for the view, the RelevantNameCache class from the package plugin.asttools.data is used. A mapping from bindings respectively from declarations to RelevantNameTypes is offered. Already available RelevantNameTypes do not have to be created again which increases overall performance. On overview of the class is shown in *Figure B.3*.

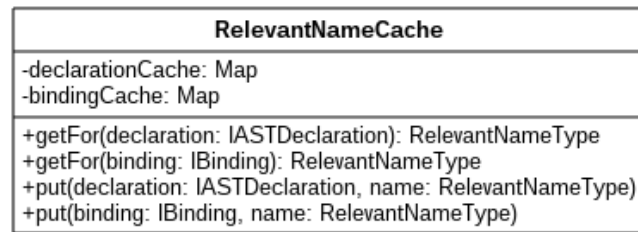


Figure B.3.: Class diagram of the RelevantNameCache

## B.4. ImageCache

The class ImageCache is using an image registry which automatically disposes the registered images when the SWT display is disposed. All images should be registered in here.

## B.5. CursorCache

CursorCache is responsible to initialize and dispose the cursor used inside the plug-in. If additional cursors are used, they should be initialized here.

## B.6. ReflectionMethodHelper

With the ReflectionMethodHelper, non accessible methods and fields can be obtained. Additionally, it offers a method to invoke the methods.

## B.7. EclipseUtil

The class **EclipseUtil** inside the package plugin.asttools.data offers static methods to retrieve the index, the AST, the project or information of the editor. The functionalities offered are figured in the class diagram shown in *Figure B.4*.

### B.7.1. `getActiveEditor`

params:	
returns:	<code>IEditorPart</code>

This method is used to get the active editor which is a component used to edit and browse a document.

### B.7.2. `getCursorPosition`

params:	<code>editorPart : IEditorPart</code>
returns:	<code>IRegion</code>

The text selection of the passed editor is retrieved and a new `IRegion` (a value object which stores an offset and a length) is created which is then returned.

### B.7.3. `getNameUnderCursor`

params:	<code>ast : IASTTranslationUnit</code>
returns:	<code>IASTName</code>

The above mentioned methods are used to get the cursor position. With the region the `IASTName` can be retrieved (B.8.1).

### B.7.4. `navigateToNode`

params:	<code>node : IASTNode</code>
returns:	

With the help of the file location of the passed node, the path is retrieved. Together with the node offset and the length, it is passed to a job called `NavigationToUIJob`. This job selects and reveals the specified range in the editor.

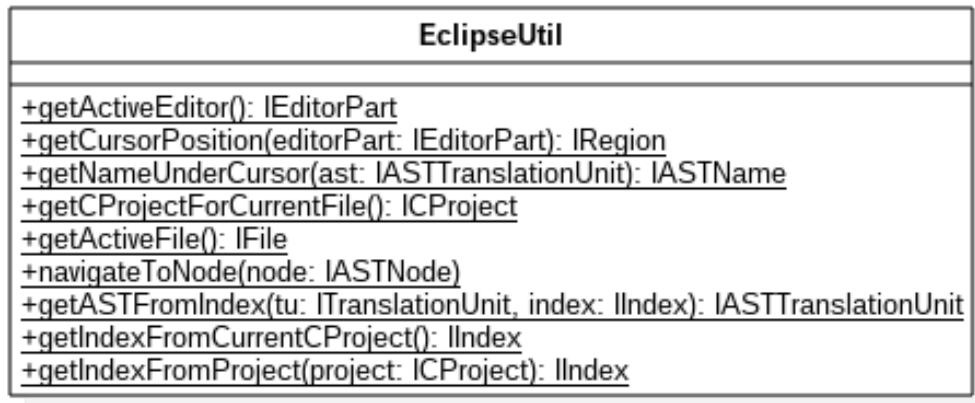


Figure B.4.: Class diagram for EclipseUtil

## B.8. ASTTools

The class ASTTools is part of the package plugin.asttools and offers static methods for IASTNodes that do not require the index.

### B.8.1. getNameAtRegion

params:	region : IRegion, ast : IASTTranslationUnit
returns:	IASTName

To get the IASTName from a region, the IASTNodeSelector is retrieved from the AST. With its help it is possible to find the enclosing IASTNode. If the node is an instance of IASTName, it is casted and returned. If it is not a name, null is returned instead.

### B.8.2. extractTemplateName

params:	templateInstanceName : IASTName
returns:	IASTName

The last name of the IASTName (last name of a IASTQualifiedName or this for a IASTName) is checked. If the last name is an instance of ICPPASTTemplateID it is returned. If this is not the case, the same is done with the parent of the last name. An exception is thrown if the last name has no parent. In all other cases, the name which was passed to the method is returned. This procedure is also shown in *Figure B.5*.

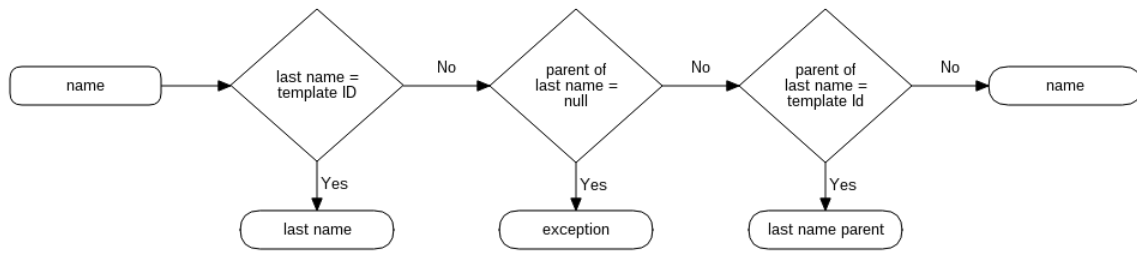


Figure B.5.: Extracting the template instance name

### B.8.3. getName

params:	declSpecifier : IASTDeclSpecifier
returns:	IASTName

There are different overloads to extract the IASTName. The overloads that take an ICPPASTTypeId or an IASTSimpleDeclaration both retrieve the decl specifier first and then use that to get the name. For a function call, the name can be obtained from the template id, id expression or the field reference (depending on the implementation) of the function name expression.

### B.8.4. unwrapTypedef

params:	binding : IBinding
returns:	IBinding

If the passed binding is an instance of IType the ultimate binding is retrieved. For this, the static method getUltimateType from the class SemanticUtil is used, which is offered by Eclipse CDT. If it returns a type which is an instance of IBinding, the binding is returned. In the other case, the binding which was passed is returned.

### B.8.5. isRelevantBinding

params:	binding : IBinding, acceptUnknownBindings : boolean, acceptNormalFunctions : boolean
returns:	boolean

The binding, a boolean to select if unknown bindings should be accepted and a boolean to select if normal functions are relevant as well are passed to the method. A binding is relevant if it is an instance of IFunction and normal functions are accepted (the same applies to classes but is not figured here) or if the binding is template dependent (B.8.6).

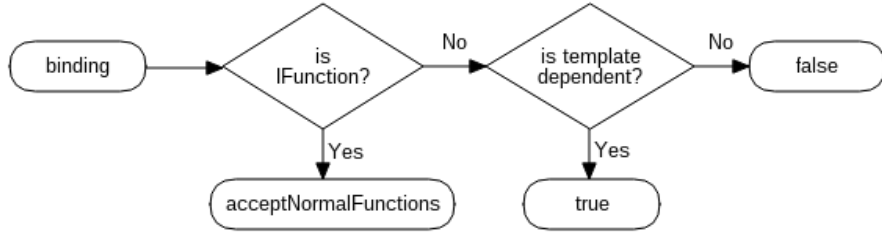


Figure B.6.: Checking if a binding is relevant

### B.8.6. isTemplateDependentBinding

params:	binding : IBinding, acceptUnknownBindings : boolean
returns:	boolean

If the passed binding is an IFunction and an ISpecialization, it is template depending. But that is not the only case. It is also possible if the binding is an IFunction and an ICPPUnknownBinding if they are accepted. The diagram shown in *Figure B.7* shows all the cases. The terminator there stands for the unsymmetrical XNOR whose truth table is shown in the table below.

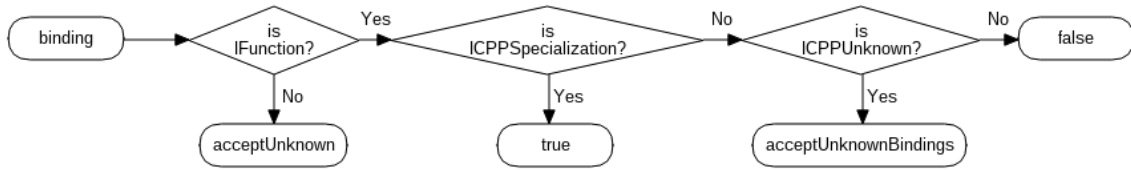


Figure B.7.: Checking if a binding is template dependent

acceptUnknownBindings	is ICPPUnknownBinding	acceptUnknown
false	false	true
false	true	false
true	false	true
true	true	true

Table B.1.: Unsymmetrical XNOR

## B.9. DefinitionFinder

The DefinitionFinder is used to find the definition of a IBinding in the AST or index.

### B.9.1. findDefinition

params:	binding:IBinding
returns:	IASTName

First, the definition is tried to get from the AST by calling the method `getDefinitionsInAST` on the instance of the ast. If the definition could not be retrieved, the definition is tried to get from the index. The `IIndexName` is retrieved from the binding and the AST for it is retrieved. If it is also not possible to, the definition has to be obtained with the help of the `NodeSelector` which is able to find the enclosing `IASTName` with the offset and length of the `IIndexName`.

## B.10. ASTAnalyzer

The ASTAnalyzer of the package `plugin.asttools` is used to find all related information to a template instance like sub calls and function or template definitions. In contrast to the class `ASTTools`, the index is used as well.

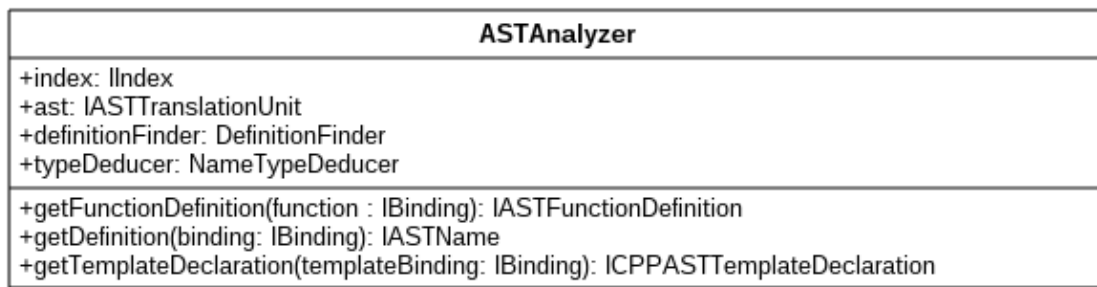


Figure B.8.: Class diagram ASTAnalyzer

### B.10.1. getDefinitionName

params:	originalName : IASTName, acceptUnknownBindings : boolean
returns:	IASTName

If the `originalName` has a parent and a grand-parent node, and the binding of it is relevant (B.8.5) the definition name is retrieved and returned.



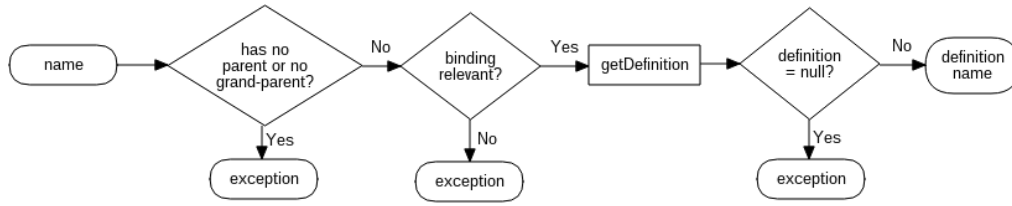


Figure B.9.: Getting the definition name

### B.10.2. extractResolvingName

params:	originalName : IASTName, acceptUnknownBindings : boolean
returns:	RelevantNameType

After retrieving the definition name (B.10.1), it is passed to the create method of the RelevantNameType. If the binding of the type name of the retrieved RelevantNameType is relevant (B.8.5) it is returned.

### B.10.3. getFunctionDefinition

params:	binding : IBinding
returns:	IASTFunctionDefinition

The passed binding is used to obtain the innermost binding. If the innermost binding is an IFunction, the definition name of it is retrieved. The definition name then is used to get the function definition by searching for the first ancestor of type IASTFunctionDefinition.

### B.10.4. getTemplateDeclaration

params:	templateBinding : IBinding
returns:	ICPPASTTemplateDeclaration

If the passed binding is an instance of ICPPClassSpecialization and the ICPPTemplate-Instance is an explicit specialization, the template definition name is retrieved from it. If this is not possible, the innermost binding is obtained and used to retrieve the template definition name. The ICPPASTTemplateDeclaration is the first ancestor of this name.

## B.11. NameTypeKind

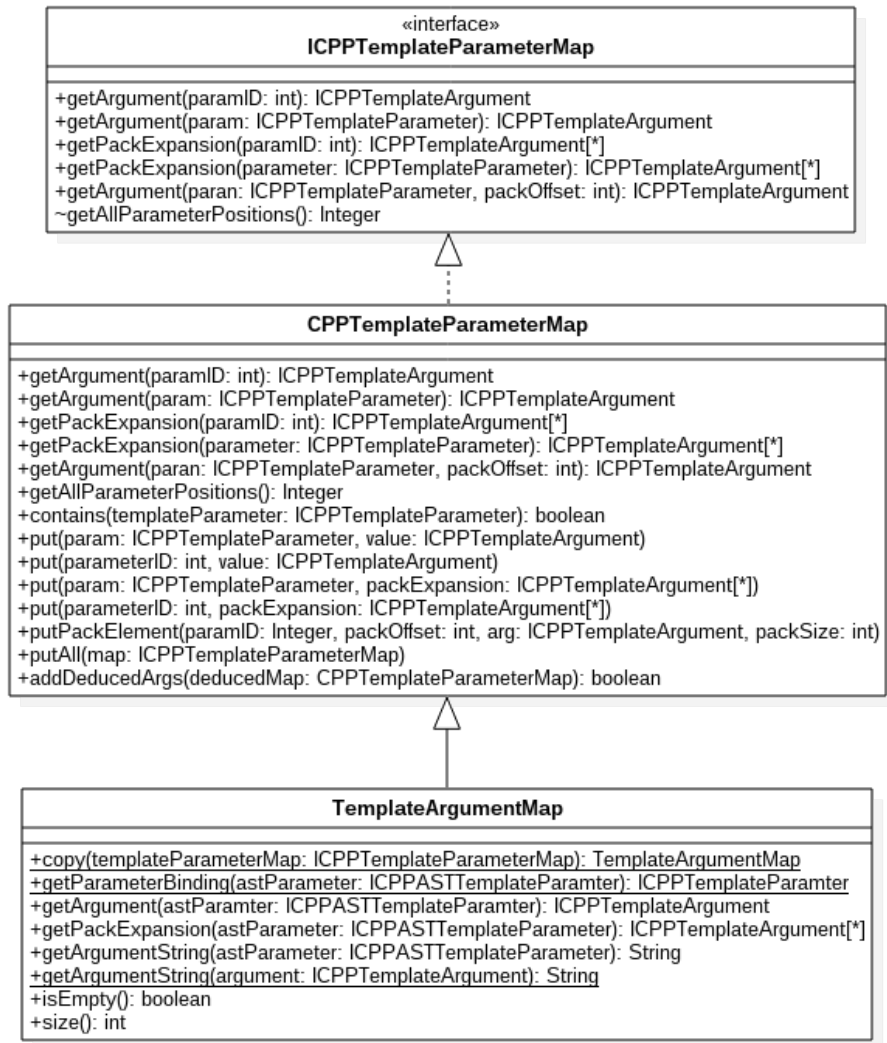
**NameTypeKind** is an enum in the package `plugin.asttools.data` that offers additional methods to set flags (`deferred`, `member`). It also contains static methods to check if an `IBinding` is a specific `NameTypeKind`. Finally, it offers a static method to get the `NameTypeKind` of a binding. The table lists the values of the enum and how the corresponding flags are set.

<b>NameTypeKind</b>	<b>deferred</b>	<b>member</b>
FUNCTION		
CLASS		
LAMBDA		
METHOD		X
FUNCTION_SPECIALIZATION		
FUNCTION_TEMPLATE		
CLASS_TEMPLATE		
VARIABLE_TEMPLATE		
METHOD_TEMPLATE		X
DEFERRED_FUNCTION	X	
DEFERRED_METHOD	X	X
DEFERRED_CLASS_TEMPLATE	X	
DEFERRED_VARIABLE_TEMPLATE	X	
DEFERRED_MEMBER_CLASS_INSTANCE	X	X
UNKNOWN_MEMBER_CLASS	X	X
TEMPLATE_PARAMETER		
ALIAS_TEMPLATE		
DEFERRED_ALIAS_TEMPLATE	X	

Table B.2.: NameTypeKind values

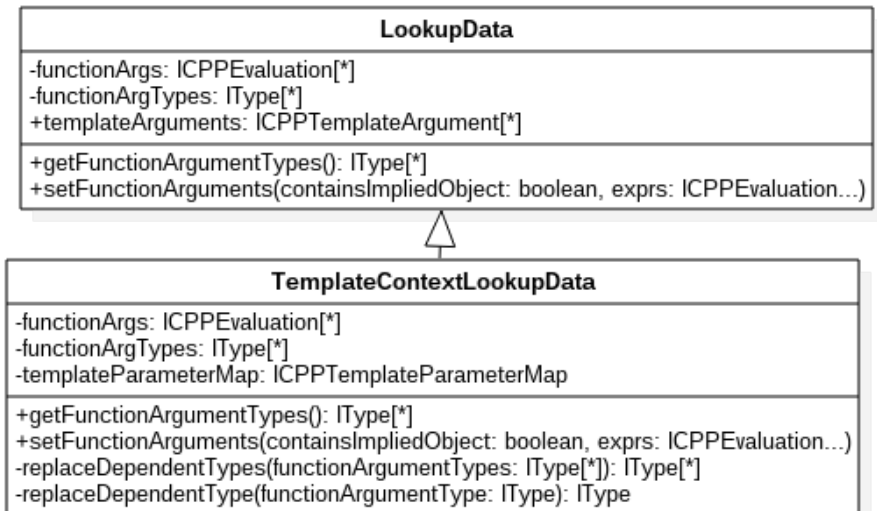
## C. Class Diagram

### TemplateArgumentMap



## D. Class Diagram

### TemplateContextLookupData



## E. Boost.Hana Switch Example

```
1 #include <boost/hana.hpp>
2 #include <boost/any.hpp>
3 #include <cassert>
4 #include <string>
5 #include <typeindex>
6 #include <typeinfo>
7 #include <utility>
8
9 using namespace std;
10
11 namespace hana = boost::hana;
12 //! [cases]
13 template <typename T>
14 auto case_ = [](auto f) {
15     return hana::make_pair(hana::type_c<T>, f);
16 };
17 struct default_t;
18 auto default_ = case_<default_t>;
19 //! [cases]
20 //! [process]
21 template <typename Any, typename Default>
22 auto process(Any&, std::type_index const&, Default& default_) {
23     return default_();
24 }
25 template <typename Any, typename Default, typename Case, typename ...
26     Rest>
27 auto process(Any& a, std::type_index const& t, Default& default_,
28     Case& case_, Rest& ...rest)
29 {
30     using T = typename decltype(+hana::first(case_))::type;
31     return t == typeid(T) ? hana::second(case_)(*boost::unsafe_any_cast<T>(&a))
32         : process(a, t, default_, rest...);
33 }
34 //! [process]
35 //! [switch_]
36 template <typename Any>
37 auto switch_(Any& a) {
38     return [&a](auto ...cases_) {
39         auto cases = hana::make_tuple(cases_...);
40         auto default_ = hana::find_if(cases, [](auto const& c) {
41             return hana::first(c) == hana::type_c<default_t>;
42         });
43         static_assert(default_ != hana::nothing,
```

```

43     "switch is missing a default_ case");
    auto rest = hana::filter(cases, [](auto const& c) {
45         return hana::first(c) != hana::type_c<default_t>;
    });
47     return hana::unpack(rest, [&](auto& ...rest) {
        return process(a, a.type(), hana::second(*default_), rest...);
49     });
    };
51 }
    //! [switch_]
53
54 void real() {
55     boost::any a1 = 'x';
    std::string r1 = switch_(a1)(
57         case_<int>([](auto i) { return "int: "s + std::to_string(i); }),
        case_<char>([](auto c) { return "char: "s + std::string{c}; }),
59         default_([] { return "unknown"s; })
    );
61     assert(r1 == "char: x"s);

62
63     boost::any a2 = 'x';
    auto r2 = switch_(a2)(
65         case_<int>([](auto) -> int { return 1; }),
        case_<char>([](auto) -> long { return 21; }),
67         default_([]() -> long long { return 311; })
    );
69     // r is inferred to be a long long
    static_assert(std::is_same<decltype(r2), long long>{}, "");
71     assert(r2 == 211);
}

```

Listing E.1: Switch statement to handle boost::any

# Bibliography

- [Ani] Chris Aniszczyk. Finding swt leaks with sleak.  
<https://eclipsesource.com/blogs/2009/04/17/finding-swt-leaks-with-sleak/>  
Accessed: 27.Jan.2018.
- [asc] asciiflow.  
<http://asciiflow.com/>.
- [BJ14] Syfrig Marco Biedermann Jonas. Templator. 2014.  
<https://eprints.hsr.ch/410/>  
Accessed: 18.Sep.2017.
- [BJ15] Syfrig Marco Biedermann Jonas. Templator2. 2015.  
<https://eprints.hsr.ch/466/>  
Accessed: 18.Sep.2017.
- [C<sup>+</sup>17] Eclipse Contributors et al. Eclipse documentation - current release. 2017.  
<http://help.eclipse.org/oxygen/index.jsp>  
Accessed: 19.Oct.2017.
- [cpl] cplusplus. Variables.  
<http://www.cplusplus.com/doc/tutorial/variables/>  
Accessed: 20.Dec.2017.
- [Cppa] Cppreference. auto specifier.  
<http://en.cppreference.com/w/cpp/language/auto>  
Accessed: 19.Dec.2017.
- [Cppb] Cppreference. Template parameters and template arguments.  
[http://en.cppreference.com/w/cpp/language/template\\_parameters](http://en.cppreference.com/w/cpp/language/template_parameters)  
Accessed: 15.Dec.2017.
- [Cppe] Cppreference. Templates.  
<http://en.cppreference.com/w/cpp/language/templates>  
Accessed: 8.Dec.2017.
- [Cppd] Cppreference. Type alias, alias template.  
[http://en.cppreference.com/w/cpp/language/type\\_alias](http://en.cppreference.com/w/cpp/language/type_alias)  
Accessed: 15.Dec.2017.

- [Cppe] Cppreference. Userliteral.  
[http://en.cppreference.com/w/cpp/language/user\\_literal](http://en.cppreference.com/w/cpp/language/user_literal)  
 Accessed: 6.Nov.2017.
- [Cre01] Tod Creasey. Preferences in the eclipse workbench ui. 2001.  
<http://www.eclipse.org/articles/preferences/preferences.htm>  
 Accessed: 5.Jan.2018.
- [DA05] Aleksey Gurtovoy David Abrahams. *C++ Templates Metaprogramming - Concepts, Tools and Techniques from Boost and Beyond*. Pearson Education, 2005.
- [dCR] Eclipse documentation Current Release. Interface paintlistener.  
<https://help.eclipse.org/neon/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/events/PaintListener.html>  
 Accessed: 10.Feb.2018.
- [Dio17] Louis Dionne. Boost.hana - your standard library for metaprogramming. 2017.  
[http://www.boost.org/doc/libs/1\\_63\\_0/libs/hana/doc/html/index.html](http://www.boost.org/doc/libs/1_63_0/libs/hana/doc/html/index.html)  
 Accessed: 14.Jan.2018.
- [DV03] Nicolai M. Josuttis David Vandevoorde. *C++ Templates - The complete guide*. Pearson Education, 2003.
- [Fou17] The Eclipse Foundation. Swt: The standard widget toolkit. 2017.  
<http://eclipse.org/swt>  
 Accessed: 25.Oct.2017.
- [IBMa] IBM. Introduction to the c++11 feature: trailing return types.  
[https://www.ibm.com/developerworks/community/blogs/5894415f-be62-4bc0-81c5-3956e82276f3/entry/introduction\\_to\\_the\\_c\\_11\\_feature\\_trailing\\_return\\_types?lang=en](https://www.ibm.com/developerworks/community/blogs/5894415f-be62-4bc0-81c5-3956e82276f3/entry/introduction_to_the_c_11_feature_trailing_return_types?lang=en)  
 Accessed: 12.Dec.2017.
- [IBMb] IBM. Non-type template arguments.  
[https://www.ibm.com/support/knowledgecenter/SSLTBW\\_2.3.0/com.ibm.zos.v2r3.cbclx01/template-non-type\\_arguments.htm](https://www.ibm.com/support/knowledgecenter/SSLTBW_2.3.0/com.ibm.zos.v2r3.cbclx01/template-non-type_arguments.htm)  
 Accessed: 2.Jan.2018.
- [IBMc] IBM. Non-type template parameters.  
[https://www.ibm.com/support/knowledgecenter/en/SS3KZ4\\_9.0.0/com.ibm.xlcpp9.bg.doc/language\\_ref/non-type\\_template\\_parameters.htm](https://www.ibm.com/support/knowledgecenter/en/SS3KZ4_9.0.0/com.ibm.xlcpp9.bg.doc/language_ref/non-type_template_parameters.htm)  
 Accessed: 2.Jan.2018.
- [IFS] IFS. Cdttesting.  
<https://www.cevelop.com/cdt-testing/development/>.



- [IFS14] IFS. Extensible infrastructure for source code transformation. 2014.  
<http://repara-project.eu/wp-content/uploads/2014/04/ICT-609666-D4.1.pdf>.
- [Jen] Jenkins. Open source automation server.  
<https://jenkins.io/>.
- [Mar17] Daniel Marty. Bug 527844 - expressionwriter does not write suffix of iastliteral-expression. 2017.  
[https://bugs.eclipse.org/bugs/show\\_bug.cgi?id=527844](https://bugs.eclipse.org/bugs/show_bug.cgi?id=527844)  
 Accessed: 28.Nov.2017.
- [MK12] Steven Jones Marco Marra Martin Krzywinski, Inanc Birol. 15-color palette for color blindness. 2012.  
<http://mkweb.bcgsc.ca/biovis2012/>  
 Accessed: 4.Feb.2018.
- [Ott09] Sebastian Otte. C++ metaprogrammierung. 2009.  
[https://www.cs.hs-rm.de/~linn/fachsem0809/cppmeta/pdf/otte\\_cppmeta.pdf](https://www.cs.hs-rm.de/~linn/fachsem0809/cppmeta/pdf/otte_cppmeta.pdf).
- [PH<sup>+</sup>10] JosÃ¶l Jeria Sean Champ Patrick Hofer, Mike Morearty et al. Jface. 2010.  
<http://wiki.eclipse.org/JFace>  
 Accessed: 25.Oct.2017.
- [Pri17] Sergey Prigogin. Bug 513105 - need a reliable mechanism to propagate point of instantiation to all methods that need it. 2017.  
[https://bugs.eclipse.org/bugs/show\\_bug.cgi?id=513105](https://bugs.eclipse.org/bugs/show_bug.cgi?id=513105)  
 Accessed: 5.Jan.2018.
- [Red] Redmine. Flexible project management web application.  
<http://www.redmine.org/>.
- [Rid17a] Nathan Ridge. Bug 520722 - [c++17] add support for class template argument deduction. 2017.  
[https://bugs.eclipse.org/bugs/show\\_bug.cgi?id=520722](https://bugs.eclipse.org/bugs/show_bug.cgi?id=520722)  
 Accessed: 18.Jan.2018.
- [Rid17b] Nathan Ridge. Bug 527843 - variable template parsed incorrectly. 2017.  
[https://bugs.eclipse.org/bugs/show\\_bug.cgi?id=527843](https://bugs.eclipse.org/bugs/show_bug.cgi?id=527843)  
 Accessed: 12.Jan.2018.
- [sta] Uml modeling tool.  
<http://astah.net/de>.
- [TeX] TeXstudio. Integrated writing environment for creating latex documents.  
<http://www.texstudio.org/>.

- [Too] SWT Tools. Swt tools update sites.  
<https://www.eclipse.org/swt/updatesite.php>.
- [Zha] Yu Xuan Zhang. Introduction to variable template of c++14.  
[https://www.ibm.com/developerworks/community/blogs/5894415f-be62-4bc0-81c5-3956e82276f3/entry/Introduction\\_to\\_Variable\\_Templates\\_of\\_C\\_14](https://www.ibm.com/developerworks/community/blogs/5894415f-be62-4bc0-81c5-3956e82276f3/entry/Introduction_to_Variable_Templates_of_C_14)  
Accessed: 2.Jan.2018.