HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

INSTITUTE FOR
SOFTWARE

Templator

Student research project thesis

HSR—University of Applied Sciences Rapperswil
Institute for Software

Fall term 2014

Authors: Jonas Biedermann, Marco Syfrig
Advisor: Prof. Peter Sommerlad

Abstract

C++ allows the usage of templates to build functions with generic types. This offers the advantage to define a function for many different data types without duplicating code and also supports compile time polymorphism and is thus used often. Template arguments are often automatically deduced from the function call site.

Function templates can be hard to work with because the compiler instantiates templates during the compilation process which results in code that the developer cannot see. Programmers using Eclipse CDT do not have easy access to the instantiated templates and thus to information about the deduced template arguments and selected function overloads. Programmers want to know what overload the compiler finally chooses, especially in the case of nested template instantiations.

The goal of our term project is to build a plug-in for Eclipse CDT that helps the programmer see the called functions and deduced template arguments. The outcome is a view that helps the programmer to examine function templates, nested function calls and show the deduced template arguments. It offers interactivity to resolve recursively nested functions to an arbitrary nesting level.

Management Summary

This thesis contains an analysis of the various ways how function templates can be used in C++. Additionally, it describes the development of an Eclipse CDT plug-in that lets a CDT user see the finally called functions for his C++ code.

Motivation

C++ is a programming language that is widely used to build software. There is a very powerful feature called function templates in C++ that allows the creation of generic functions.

What a function template actually executes depends on the concrete type that is used to call the function. For each concrete passed argument type, the compiler creates a function body exactly for this type where further overloaded functions can be called. In the following code, the outer function template is called with a double as argument and now depending on this type, the overloaded inner function is called. In this case the one with a double.

```
1    void inner(int i) {}
2    void inner(double d) {} // gets called
3
4    template<typename T>
5    void outer(T value) { inner(value); }
6
7    int main() { outer(2.5);  }
8
```

So the difficult part is that one function call can map to different functions. The

programmer now wants to know which function and thus which statements finally get called for his call.

Programmers use an IDE (Integrated Development Environment) to write code. Compared to Java, C++ has a much richer set of features and it is therefore harder to provide IDE support for C++. No known IDE assists the user when dealing with function templates and overloaded functions but it would be possible to extract this information from the users C++ code.

Goal

We want to provide support for the Eclipse CDT to show a programmer which function finally gets called. Additional functionality can be added to Eclipse by creating a plug-in so we want to develop a plug-in called Templator.

The goal is to offer visualization of the finally called functions and thus executed statements for a user and his C++ codebase. He should be able to see what will be executed for all his function template calls before running the program.

The Templator plug-in shall reduce the time a programmer needs to write function templates and analyze existing code because he can visualize all his calls and see the called functions for an arbitrary nesting level.

It would be good if the finished plug-in would be flexible enough to later support other types of templates as well.

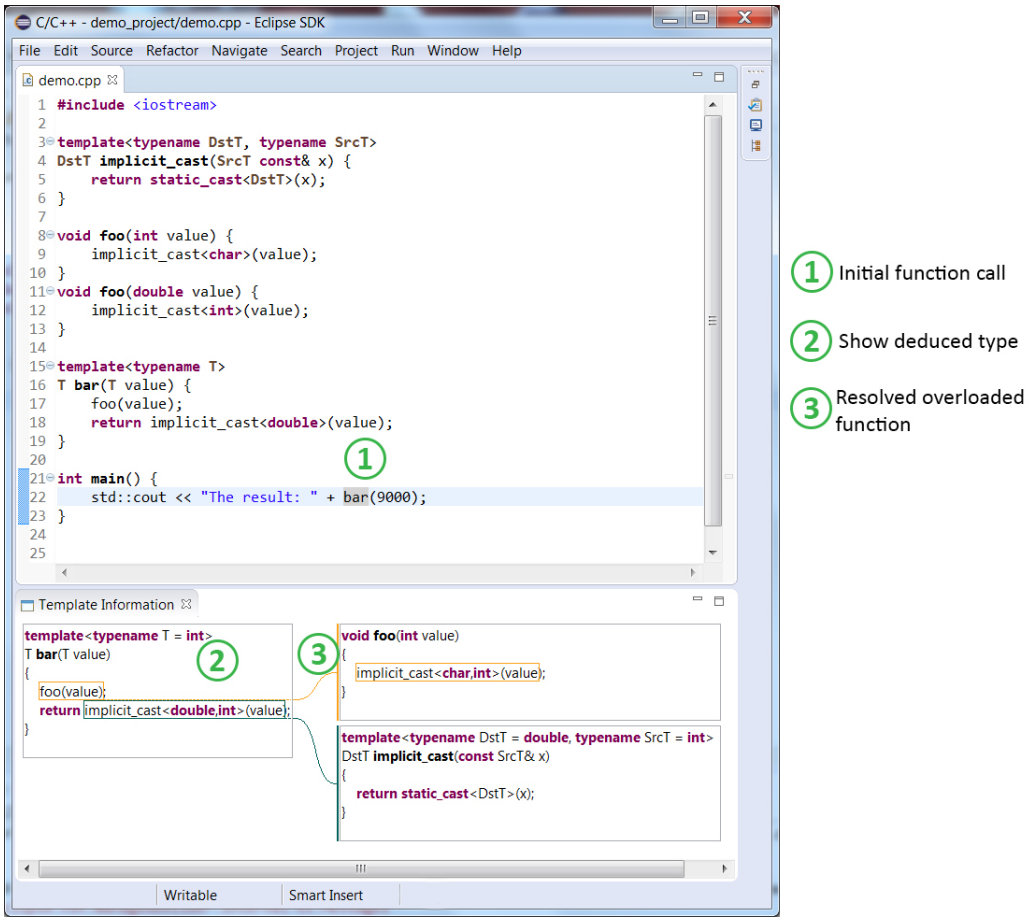
Results

The outcome is an Eclipse CDT plug-in named Templator. Its functionality can be divided in two parts. The first part is the resolution of the correct function calls. It can determine what concrete data type is used for the function. With this information the plug-in can decide which function overload is selected.

The second part is the visualization. It includes showing the concrete data types for the function template call. The user can select a function in his source code that he wants to explore. The plug-in also offers interactivity for the user to explore the call hierarchy of this selected function.

A CDT user can now examine his function calls in a tree like hierarchy. CDT now has one more unique features that differentiates it from other C++ IDEs.

The final Templator plug-in in Eclipse.



Contents

1 Task description

This chapter contains the description of our project and our goals for it.

1.1 Problem

C++ allows the usage of templates to build functions with generic types. This offers the advantage to define a function for many different data types without duplicating code and also compile time polymorphism and is thus often used.

While templates are a nice feature to have, it can be hard to work with them, especially if the function template calls overloaded functions or other function templates based on template arguments. The programmer quickly loses the overview which function is called or it is difficult to understand error messages when using the wrong instantiation.

Programmers using *Eclipse C/C++ Development Tooling (CDT)* do not have enough information about the deduced template arguments and overloads. Programmers want to know what the compiler finally chooses, especially in the case of nested template instantiation. This is bad since *Eclipse CDT* always has some of this information but does not provide a way to show it.

1.2 Solution

The solution is to visualize template instantiations for function templates and show the deduced template arguments to the *Eclipse* user. To our knowledge no other Integrated Development Environment (IDE) offers such functionality yet nor is there any specific code for exactly this problem in *CDT*.

1.2.1 Simple function template

Listing 1.1 code shows a simple function template that calls another function template which returns itself and is then casted to the specified type, an extended version from [1, :105]. Figure 1.1 shows the resulting view that shows the deduced arguments and which function finally got called.

```
1  #include <iostream>
2
3  template<typename T>
4  T id(T value) {
5      return value;
6  }
7
8  template<typename DstT, typename SrcT>
9  DstT implicit_cast(SrcT const& x) {
10     return id(x);
11 }
12
13 int main() {
14     std::cout << implicit_cast<int>(4.5);
15 }
```

Listing 1.1: Simple template function

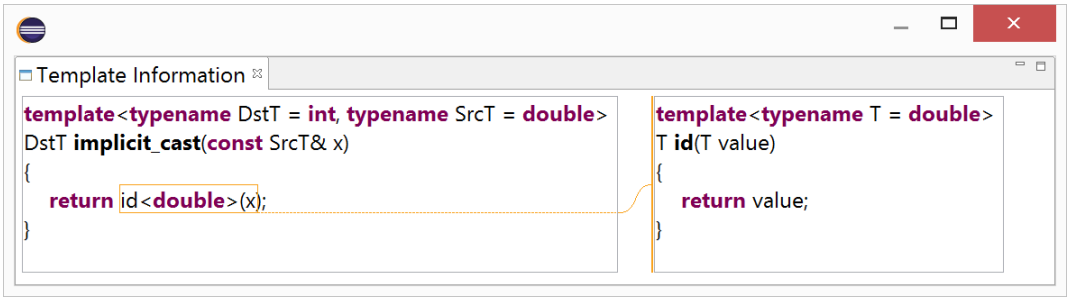


Figure 1.1: Resulting view for Listing 1.1

1.3 Our goal

The goal of our project is to eliminate this nuisance of *CDT* having information but not showing it. We will develop an extension to *CDT* as a plug-in in Java that allows a developer to visualize deduced template arguments, function overloads and selected template specializations. A final goal would be to be able to visualize template instantiations recursively to provide a way to debug template meta programs.

Our goal is not strictly defined. We will implement as much as possible in the given time in an easily extensible way so the remaining requirements can be implemented in another thesis, possibly by ourselves (Jonas Biedermann, Marco Syfrig).

1.3.1 Focus for this project

For this work our goal is to implement the above mentioned objectives partially.

We plan to implement

- A new view to show information about a function template instance.
- Resolve function calls (function template and non-template) recursively.
- Interactivity for the user to show the definition for the called function, the deduced template arguments and resolving further calls.

1.3.2 Postponed requirements and ideas

While the plug-in already provides useful functionality there is much room for improvements.

There are cases where extended functionality will not be handled by the plug-in. This omission could be implemented in the future. Passing a function template as argument to a function template call requires resolving all further function calls recursively to determine the return type. Further, there is Substitution Failure Is Not An Error (SFINAE) which

needs to be considered and handled. Additionally, there will be most likely many other cases we did not even consider.

Class templates and variadic templates are not supported yet. And with class templates there are specializations that should be presented to the user.

1.4 Time management

Our project started on the 15th of September, 2014. It takes 14 weeks and ends on December the 19th, 2014 at 5pm (noon) which is when the final release has to be submitted.

2 Analysis

This chapter describes the many possibilities that a C++ developer has to instantiate a function template and what cases our *Eclipse CDT Templator plug-in* is capable of handling.

2.1 Function template instantiations in C++

A function template can be instantiated in C++. Such a template instance is only valid when all template arguments have a value. There are 3 ways on how to achieve this and instantiate a function template.

For easier identification, the first template parameter is always named `T` and the corresponding function parameter named `first` while the second, if existing, is named `F` and `second` and so on.

2.1.1 Template argument deduction

The template argument is often automatically deduced from the function call site. Listing 2.1 shows the most simple case where just one template parameter exists. Here the compiler has to deduce the passed argument from the call site, in this case an `int`.

```
1 template<typename T>
2 T id(T first) {
3    return first;
4 }
```

```
5
6 int main() {
7     id(4); // <int>
8 }
```

Listing 2.1: Calling a function template where the compiler deduces the type

2.1.2 Explicit template argument specification

C++ allows to qualify the function template name with a list of template arguments. Listing 2.2 shows an instantiation where the type is specified. `add<double,int,double>` is a template-id. The first `double` is just for the return type which cannot be deduced since there is no parameter for it. The two added values are casted to `double` on *line 3*. The remaining two arguments specify the types of `T` (=`int`) and `F` (=`double`). The template arguments are specified in the same order as the template parameters, so the first one matches `typename RETURN`. The compiler does not have to deduce the type but needs to cast the passed arguments to the specified types.

```
1 template<typename RETURN, typename T, typename F>
2 RETURN add(T first, F second) {
3     return static_cast<RETURN>(first+second) ;
4 }
5
6 int main() {
7     add<double,int,double>(4.5, 4); // <double,int,double>
8 }
```

Listing 2.2: Calling a function template where a template-id specifies the argument types

2.1.3 Default template arguments

A function template parameter can specify a default template argument. This is used if there is no function parameter to deduce from and it is not specified in the template-id.

Because of this it fits into the two descriptions above and is theoretically also a deduction but our plug-in needs to handle it separately. Listing 2.3 shows an example where the template parameter `T` has a default type `int` that is neither deduced from the call site nor explicitly specified.

```
1 template<typename T=int>
2 void foo(){}
3
4 int main() {
5    foo(); // <int>
6 }
```

Listing 2.3: Function template parameter with a default argument

2.1.4 Combinations and special cases

C++ allows to combine the three variants. Also the declaration of the function parameters does not have to be in the same order as the function template parameter declarations. Additionally, there can be other, non-template parameters, between template parameters.

Listing 2.4 shows a combination of the three above mentioned variants and additionally changing the function parameter declaration order and adding a non-template parameter.

```
1 template<typename T,
2        typename F,
3        typename G=unsigned,
4        typename H>
5 G combined_variants(H forth, int i, F second) {}
6
7 int main() {
8    // <int,double,unsigned int,char>
9    combined_variants<
```

```
10                      int,
11                      double>
12                      ('c',
13                       42,
14                       108);
15  }
```

Listing 2.4: Function template instantiation showing a combination of ways to specify the argument

- *Line 3* says that the argument for the template parameter `G` is `unsigned int` and is not overwritten because there is no function parameter with `G` as type.
- *Line 10* says that the template parameter `T` is of type `int`.
- *Line 11* says that the template parameter `F` is of type `double` and the last passed argument `108` on *line* 14 is therefore casted to `double`.
- *Line 12* says that the first function parameter `forth` is of type `char` which maps to the template parameter `H`. So the argument of `H` is `char`.
- *Line 13* is just a normal `int` argument for a non-template parameter.

Listing 2.5 shows the last example to instantiate a function template in this chapter. But there would be are many more possible, like a list of template parameters as template parameter. In the example below, a function template is used to deduce the array size. C++ allows the usage of non-type template parameters like `n` which could be used for deducing other information.

```
1  template <typename T,unsigned n>
2  unsigned array_size(T (&value)[n]) {
3     return n;
4  }
5
6  int main() {
7     int arr[] = {1,2,3};
```

```
8     array_size(arr); // <int,3>
9 }
```

Listing 2.5: Using a function template with a non-type template parameter to deduce the array size

2.2 Supported features in our Templator plug-in

Our Templator plug-in supports most features listed above (see Section 2.1 Function template instantiations in C++ on page 13) fully and some at least partially. We are supporting C++11 only. It might work with older versions because the standard only added and did not remove new features like default arguments for function templates in the C++03 standard. But the latter is not explicitly tested.

2.2.1 Showing explicit specified template arguments

Templator supports the usage of template-ids to specify template arguments explicitly (described in Subsection 2.1.2 Explicit template argument specification on page 14). This works for template-ids where all arguments are specified and also for function template calls where some arguments are specified and some are deduced from the call site or from the default template argument.

It is important to note that we do not check if the passed arguments on the function call site can be casted to the specified types or if the parameter can be substituted with the given argument without compile errors (Substitution Failure Is Not An Error a term first coined in [1, :106]). So if the AST is correct we just pick the best matching function based on the parameters.

2.2.2 Deduced template arguments

The plug-in supports the visualization of deduced template arguments and the deduction itself as described in Subsection 2.1.1 Template argument deduction on page 13. The deduction works for the following cases on all hierarchy levels.

The supported features have each an example. Listing 2.6 shows a frame and each supported features shows the inner function template instantiation (the content for *line 14*).

```
1  struct MyClass{};
2  typedef long mytype;
3  int pinkElephant{42};
4  double id(double d) { return d; }
5  int id(int i) { return i; }
6
7  template<typename T, typename F>
8  void inner(T first, F second) {}
9
10 template<typename T>
11 void outer(T templateParam, int nonTemplate) {
12    char c{'1'}; mytype l{108};
13    char& refC{c};
14    //<inserted line>
15 }
16
17 int main() { outer(3.14, 0); }
```

Listing 2.6: Frame to demonstrate the supported features

- Passing literals like `1.5` or `'c'` as template argument.

```
inner(1.5, 'c');
```

- Passing local and global variables as template argument.

```
inner(c, pinkElephant); // local and global variable
```

- Passing a function template parameter in a nested function template call as template argument.

```
inner(templateParam, templateParam);
```

- Passing a non-template parameter as template argument.

```
inner(nonTemplate, nonTemplate);
```

- Passing the return type of a non-function template as template argument or any other expression that does not depend on a template parameter.

```
inner(id(5), id(15.23));
```

- All variables and parameters can further be wrapped in other data types. This is something *CDT* specific since an `int const &` is an `int` wrapped inside a qualifier type wrapped inside a reference type but the core type is really of type `int` which needed to be considered.

```
inner(refC, new MyClass{});
```

- Aliases like `typedef`s as types.

```
inner(l, l); // will be resolved as <long,long>
```

- The usage of default template arguments.

```
1  template <typename T=int>
2  T inner() {}
3  template <typename T>
4  int outer(T value) { return inner(); }
5
6  int main() { outer(1); }
```

The following cases will only work for the first hierarchy level. Each point shows an example of code that the Templator plug-in is **not** able to resolve.

- Deducing and passing non-type template parameters like the array size. This does not work in the example even though the array is not passed as template argument to `outer`

```
1  template <typename T,unsigned n>
2  unsigned array_size(T (&value)[n]) { return n; }
3
4  template<typename T=int>
5  void outer() {
6          int arr[] = {1,2,3};
7          array_size(arr);
8  }
9  int main() { outer(); }
```

- Passing the return type of function template calls which depend on the instantiated arguments as template arguments. This would require to instantiate all further nested function templates until the return type could finally be deduced.

```
1  template<typename T>
2  void outer(T value) {}
3  void outer(int value) {}
4
5  template<typename T>
6  T id(T value) { return value; }
7
8  template<typename T>
9  void inner(T value) { outer(id(value));}
10
11 int main() { inner(5); }
```

- Passing class templates as template arguments. Does not work and will show an

error that this feature is not supported.

```
1  #include <vector>
2  template<typename T, typename Alloc, template
       <typename, typename> class V>
3  void inner(V<T, Alloc> &con){}
4
5  template<typename T=int>
6  void outer() {
7     std::vector<int> v{1,2,3};
8     inner(v);
9  }
10
11 int main() { outer(); }
```

These three cases work for showing the deduced arguments on all levels but not for the deduction itself.

3 Eclipse CDT

This chapter describes the Eclipse platform and mostly how the *Eclipse CDT* works.

Eclipse is an IDE that is extensible via plug-ins. A bundle of plug-ins, delivered as *Eclipse CDT*, brings the functionality to develop, refactor, run and debug C/C++ (in the following abbreviated to just C++) programs to *Eclipse*.

Eclipse is not an a single program, but rather a small kernel containing a plug-in loader surrounded by hundreds of plug-ins. [2]

Our plug-in uses the existing *Eclipse CDT* platform to extend it with functionality to inspect function template calls. This chapter gives a brief overview of *Eclipse CDT* features that are relevant for our thesis. We first describe how the plug-in mechanism works and can be used to dynamically extend the *Eclipse* platform. Then following *CDT* specific features like the parsing, Abstract Syntax Tree (AST), the index, what bindings are and how the are resolved. Finally, we describe how all these features work together.

3.1 The plug-in mechanism

Eclipse itself consists of some core plug-ins and almost all of its functionality is added via other plug-ins which use extension points to add functionality, e.g. to add an extra view. *Eclipse* plug-ins can be written in a JVM compatible language like Java or Scala and are at the end just JAR files that reside in an *Eclipse* folder.

3.1.1 Extension points

An extension point can be used to add dynamically functionality to *Eclipse*. They are defined entry points in an XML file and describe the attributes one can use. They can for example be used as hooks to add entries to a context menu. The menu is created dynamically by checking all plug-ins that use this extension point if they want to contribute something for the current context. Or there is for example an extension point for views that we use to make our view accessible and known in *Eclipse*.

```
1  <extension
2      point="org.eclipse.ui.views">
3  <view
4      allowMultiple="false"
5      category="org.eclipse.cdt.ui.views"
6      class="ch.hsr.ifs.templator.plugin.ui.TemplateView"
7      id="ch.hsr.ifs.templator.plugin.ui.TemplateView"
8      name="Template Information View"
9      restorable="true">
10 </view>
11 </extension>
```

Listing 3.1: Using an extension point to add the template view

Now if the user opens the *Show View* dialog (see Figure 3.1) to see all available views, the name "Template Information View" pops up in the `org.eclipse.cdt.ui.views` category. This is a defined category in the `org.eclipse.cdt.ui` plug-in with the name "C/C++". The view entry is greyed out because it is already open and *allowMultiple="false"* is set.

3.1.2 Equinox

To be able to implement such a modular system, *Eclipse* uses an OSGi implementation called *Equinox*. OSGi is a specification that describes a modular system as extension for standard Java Virtual Machine (JVM) programs that cannot do this by default. Equinox

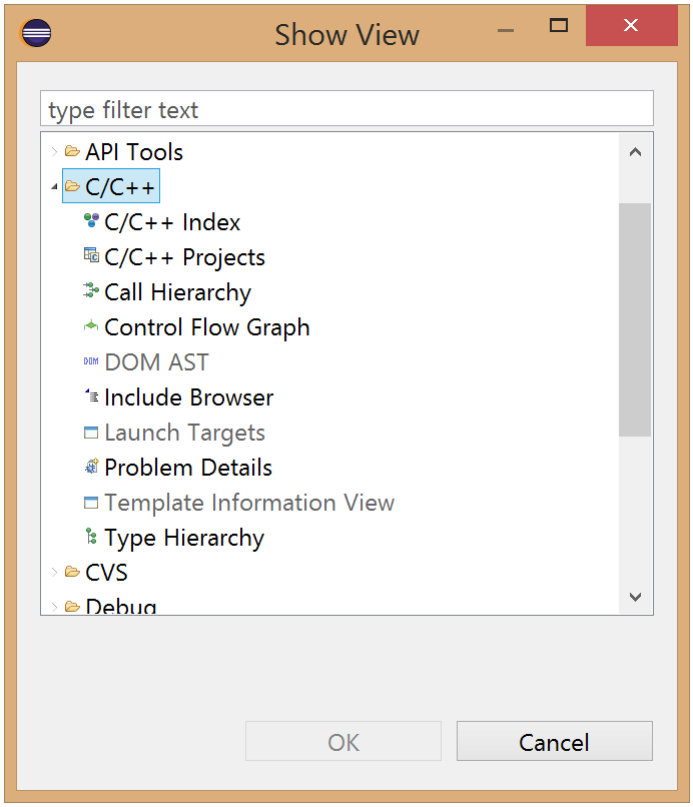


Figure 3.1: *Show View* dialog with our defined "Template Information View"

is the Open Service Gateway initiative (OSGi) reference implementation developed by the Eclipse Foundation, it loads all plug-ins and handles the lifecycle.

3.2 Parsing

While writing C++ code, *Eclipse CDT* parses the code subsequently to assist the user with autocompletion, syntax highlighting, showing function definitions, quickfixes and more.

A so called lexer reads every character (skipping whitespaces and collecting comments) until it has a full token, like a keyword, operator or C++specific punctuation which it passes to the parser. The parser knows what context it is in and what it can possibly

expect as next token. For example, if there is no open bracket and the `template` keyword token, the next token is expected to be an opening angle bracket `<`, then an optional `template` or `class` token and so on. So it checks if the tokens fits in the defined grammar.

3.3 The AST

The *CDT* parser does more than checking the grammar. It builds an AST, which contains nodes for each part of a grammar rule. The AST allows for easy analysis, transformation and searching in the background. A simple C++ programs is shown in Listing 3.2, implementing a function template and Figure 3.2 shows the corresponding AST.

```
1 template<typename T>
2 T id(T first) {
3     return first;
4 }
5 int main() {
6     id(5);
7 }
```

Listing 3.2: AST example of a simple function template

The root node is of type `IASTTranslationUnit` and represents a single source code file with all its direct includes. Further down are more specific nodes. The root node includes the whole translation unit. Each node knows its parent through a property and vice versa, there are no hardlinks between them. Each node can also have several children. Traversing the AST and doing `instanceof` checks with the visitor pattern now allows to analyze and modify the C++ codebase very quickly.

CDT distinguishes 76 AST nodes for C++ alone but uses 101 classes for both C and C++. All these nodes implements the `IASTNode` interface and begin with `IAST...` respectively with `ICPPAST...`. The C++ standard defines the grammar rules which these nodes follow. So an `IASTBinaryExpression` for example has an operator and two `IASTExpression` as operands. But an expression can further contain many other nodes like the following line

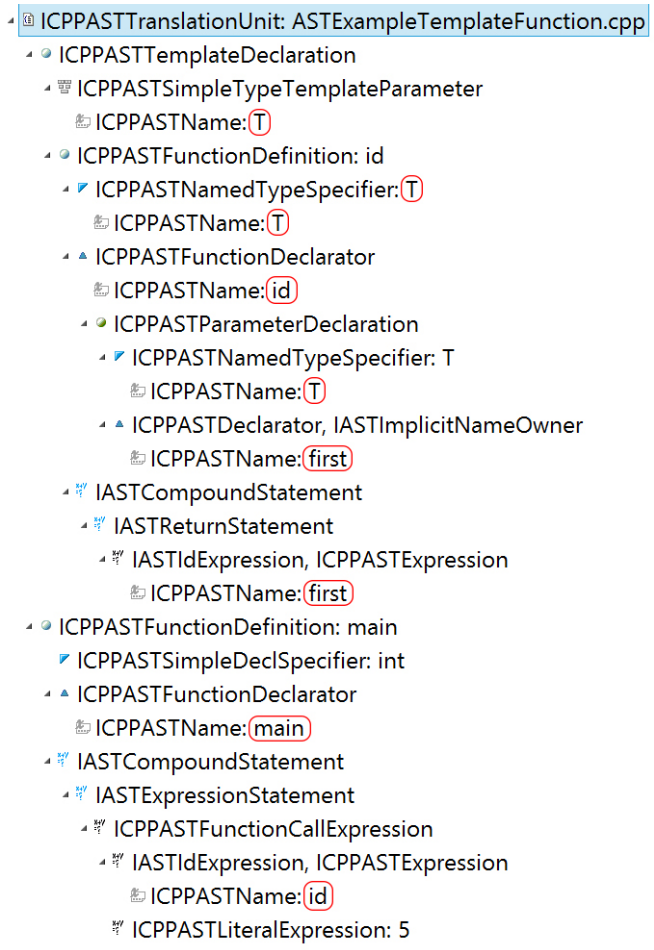


Figure 3.2: AST for Listing 3.2. All `IASTNames` that can be resolved to a binding are marked.

of code shows. The right hand side node contains many more nodes.

```
1 + func1(otherFunc(5));
```

An AST exists for each source file, so saving a file results in building only the AST again for this file and not all included and referenced files. Function calls to defined functions in other files can be resolved to a binding (see Section 3.5 Bindings on page 28), so this two ASTs are independent of each other.

3.4 The index

A C++ project can include many header files outside of the workspace. For example the header files from the standard library or other libraries. The linker combines them with object code during the compile process.

When writing code in the *Eclipse CDT*, a programmer wants to see these library functions and classes in the autocompletion popup. But there may be dozens of libraries and hundreds of files and building and storing an AST for all of them would be too time and memory consuming. *Eclipse* is not a front end compiler but an IDE, so not everything has to be known. It is enough to know the declaration and have position information so the user can jump to the definition. *CDT* has a solution for this without parsing the whole code and resolving all bindings again.

Copied and modified from [3].

- This information is stored in a Persistent Document Object Model (PDOM) file, called *the index*, in the workspace directory.
- With each save, *the index* is updated.
- The parser is able to skip included header files.
- No semantic analysis like type checking is done, so the parser can skip the header files that may contain this information.
- *The index* can be used to build the AST, so a translation unit does not need to be fully reparsed.
- But the containing `IIndexBinding` and `IIndexName` have less information, e.g. an `IIndexName` resolves to a `PDOMCPPFunction` and does not know its body, only the parameters and return type.

3.5 Bindings

A binding represents an instance of information about an identifier. For a function `foo()`, there is a single binding that represents `foo` and knows the declaration, definition and all places where this function is called. The same applies to variables.

Listing 3.3 contains an `IASTName` for the function call in the `main`. Each `IASTName` contains a `resolveBinding()` method to get the `IBinding` for a name.

```
1 int id(int i) {
2    return i;
3 }
4 int main() {
5    id(42);
6 }
```

Listing 3.3: Function call and definition. `id` on line 5 represents an `IASTName` that can be resolved to an `ICPPFunction`

So resolving the function name in the call returns a `CPPFunction` which knows the `IASTNode` for the function declaration and definition.

Listing 3.4 shows another example. Here the two `IASTNames` `side` in the function parameter declaration (*line 2*) and `side` in the function definition (*line 3*) are two separate instances. However, the binding resolution algorithm resolves both to the same `ICPPTemplateParameter`.

```
1 template<typename T>
2 T area(T side) {
3     return side*2;
4 }
```

Listing 3.4: Resolving `side` in the parameter declaration and in the body returns the same binding

An identifier resolves either to a subclass of `IBinding` in the `org.eclipse.cdt.internal.`

`core.dom.parser.cpp` package like `CPPFunctionTemplate` or one in `org.eclipse.cdt.internal.core.pdom.dom.cpp` like `PDOMCPPFunctionTemplate`. The first one means the binding is found in the AST and the function definition is accessible. In the latter, the binding is found in the index which means only the function declaration but not the definition is accessible. See Subsection 4.2.2 Index searching on page 39 how we solved this problem in our Templator plug-in.

3.6 Interaction between AST, index and binding resolution

This last section to *Eclipse CDT* describes how the AST and index nodes work together and how they affect the binding resolution.

The index is finally just a stored PDOM file in the current *Eclipse* workspace while the AST is always an in-memory construct. The AST is usually built by parsing the whole translation unit again. But there is a way in *CDT* to build it from the existing index data. This results in an AST where some bindings resolve to a binding in the index and some directly in the AST as described in the last section.

Figure 3.3 shows a perfect image to describe this behavior from a *CDT* presentation[4]. If the AST is built using the index data, then all function definitions from the included header files resolve to an index binding. Function definitions from the same translation unit like `circumference` resolves to an AST binding.

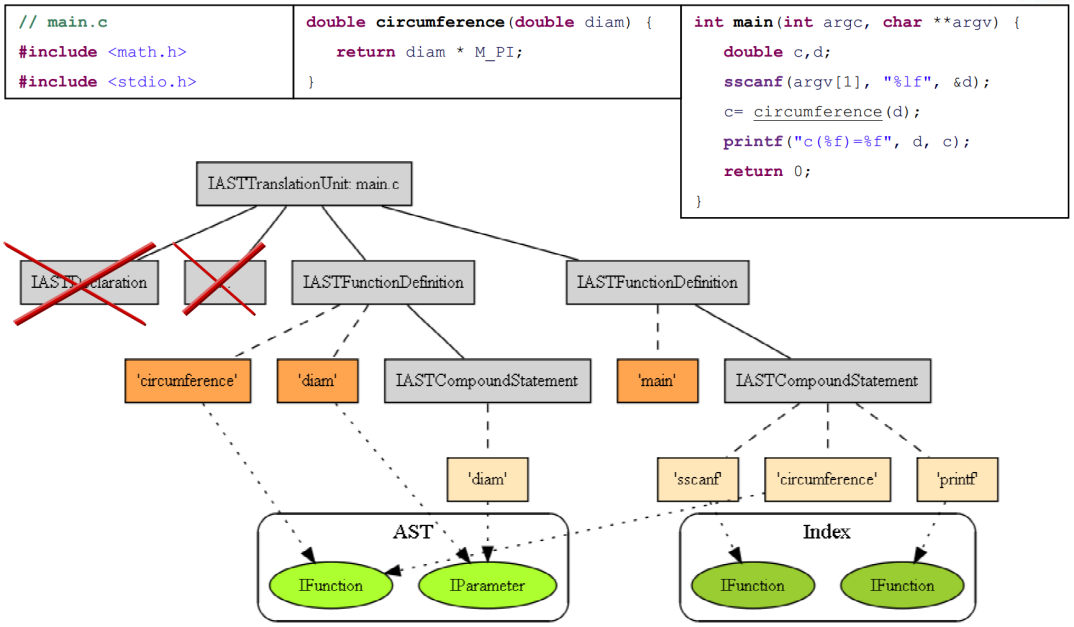


Figure 3.3: Result when resolving bindings with an index based AST

4 Implementation

This chapter describes how we used the infrastructure explained in Chapter 3 Eclipse CDT on page 22 and expanded it to implement an *Eclipse* view to show information about a function template instance. We developed our plug-in with the Java language.

This chapter also describes what functionality *Eclipse CDT* already has for function templates and what we had to implement for our plug-in to work.

As stated in the chapter Analysis (see Section 2.2 Supported features in our Templator plug-in on page 17) some things only work in the first hierarchy level even if they seem to solve the same problems in all hierarchy levels. The Templator plug-in will support the visualization of an arbitrary number of nesting levels and we could use existing *CDT* code for some parts. The arguments for the first level will always be deduced by the existing *CDT* infrastructure and all further levels by our plug-in. So all arguments will correctly be deduced in the first hierarchy level but some of them are not for nested function template calls.

In order to prepare the data for the UI there are 3 stages to run through. First, we find the correct binding for the called function using the potentially deduced template arguments. Then we search the definition of the function for this binding. Finally, we modify the AST to show the deduced arguments in the rendered source code.

4.1 Find the called function

The main task of our plug-in is to resolve a function call to the correct definition to find the next function calls so a user can jump around the function calls and see the code behind it.

4.1.1 Already implemented functionality in CDT

CDT is already able to resolve nested function calls that are not template argument dependent. So in the following Listing 4.1, `inner(i)` can be resolved.

```
1  void inner(int i) {}
2  void inner(double d) {}
3
4  template<typename T>
5  void outer(T value, int i) {
6     // do something with value
7     inner(i);
8  }
9
10 int main() {
11    outer(2.5, 1); // <double>
12 }
```

Listing 4.1: Template function calls another function that does not depend on the argument

4.1.2 Template argument dependent functions

CDT is not yet able to resolve functions that depend on at least one template argument and the called function is overloaded as seen in Listing 4.2.

```
1  void inner(int i) {}
2  void inner(double d) {}
3
4  template<typename T>
5  void outer(T value) {
6     inner(value);
7  }
8
```

```
9  int main() {
10     outer(2.5); // <double>
11 }
```

Listing 4.2: Function template that calls another function depends on a template argument

Here the binding resolution algorithm in *CDT* will return an `ICPPDeferredFunction` with the two `inner` functions as candidates. The context what template argument is substituted for `T value` is missing for *CDT*. Function template instantiations are just done for the first hierarchy level.

4.1.3 Context for binding resolution

The binding resolution algorithm uses a context with namespace scope, variables and a mapping from template parameters to arguments and much more. This context is an instance of type `LookupData` and is already correctly instantiated—except for function templates after the first level where the template parameter map is empty, hence the name resolves to a deferred binding.

4.1.4 Template parameter map

This template parameter map is an instance of `ICPPTemplateParameterMap` and maps a template parameter to the finally deduced type. The map for Listing 4.3 shows a simple function template with two template parameters and an instantiation with an `int` and a `double`.

```
1  template<typename T, typename F>
2  void foo(T first, F second) {}
3
4  int main() {
5     foo(23, 42.0); // <int,double>
6  }
```

Listing 4.3: Function template with 2 template parameters to show the template parameter map content

The resulting template parameter map looks like this.

```
{#0,0: int, #0,1: double}.
```

Since templates can be nested, the index before the comma indicates the nesting level while the second indicates the position in this level. `#0,0 int` means the first template parameter (in nesting level 0) `T` was deduced to an `int`.

The mapping of parameters and their arguments is done by parameter positions and not bindings. differ between different class template specializations but the positions are the same.

4.1.5 Building the template parameter map

So to solve this problem, we needed to create our own `ICPPTemplateParameterMap` instance and wrap it in a `LookupData` class. Since the latter is not intended to be done in *CDT*, an extension to `LookupData` was needed so it overrides methods that return the arguments for the function call. The code for the extended `LookupData` class was contributed by Thomas Corbat, *CDT* committer and IFS employee.

Building the template parameter map that is needed to pass to the newly created `LookupData` class `TemplateContextLookupData` is complex. C++ allows a developer to instantiate a function template in many ways (see Section 2.1 Function template instantiations in C++ on page 13). That needs to be considered and is discussed in the following.

We create our own class `TemplateArgumentMap` for storing the data and also deducing the arguments. This class extends `CPPTemplateParameterMap` which is an implementation of `ICPPTemplateParameterMap`. This way we can pass it to the `LookupData` context.

Figure 4.1 shows the simple overview for the creation. There are generally 2 steps. Adding the explicit arguments and then deducing the rest that are explained in detail in the following paragraphs.

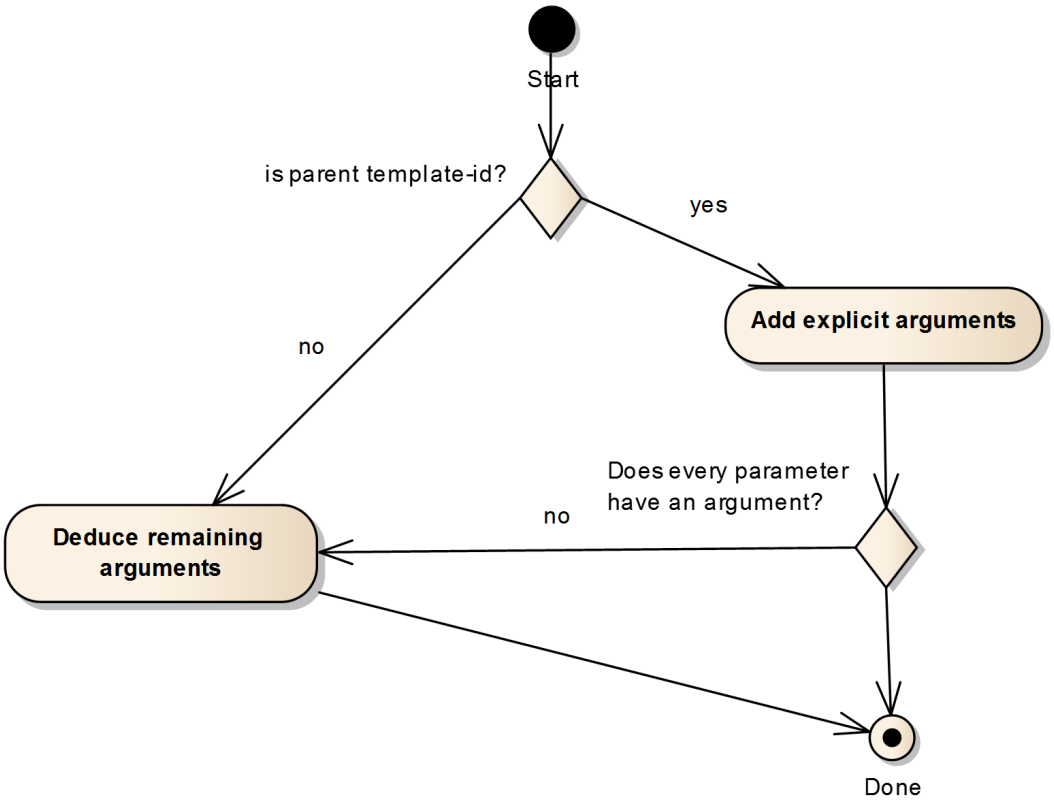


Figure 4.1: In the first part, all explicit arguments from the template-id are added.

Explicit arguments

Template arguments for the instantiation can be specified in a template-id like `int` in `func<int>(108)`. These arguments map in the exact same order to the template parameter declaration.

So at first a check is done if there is a template-id. This is the easy part because there is a `public` method `org.eclipse.cdt.internal.core.dom.parser.cpp.semantics.TemplateArgumentDeduction.addExplicitArguments` in *CDT* to add these arguments to a passed `ICPPTemplateParameter`.

After that a check is done to see if there are remaining template parameters without a substituted argument. We do this by counting the template parameters in the function template declaration and the number of arguments in the map. If they match, the following steps can be skipped.

Now we iterate over all remaining template parameters and deduce their type. This is either done by getting the default template argument or deducing the argument from the function call site.

Default arguments

Next there is a differentiation between template parameters that have a mapping function parameter. There are also 2 ways to go in this step as seen in Figure 4.2

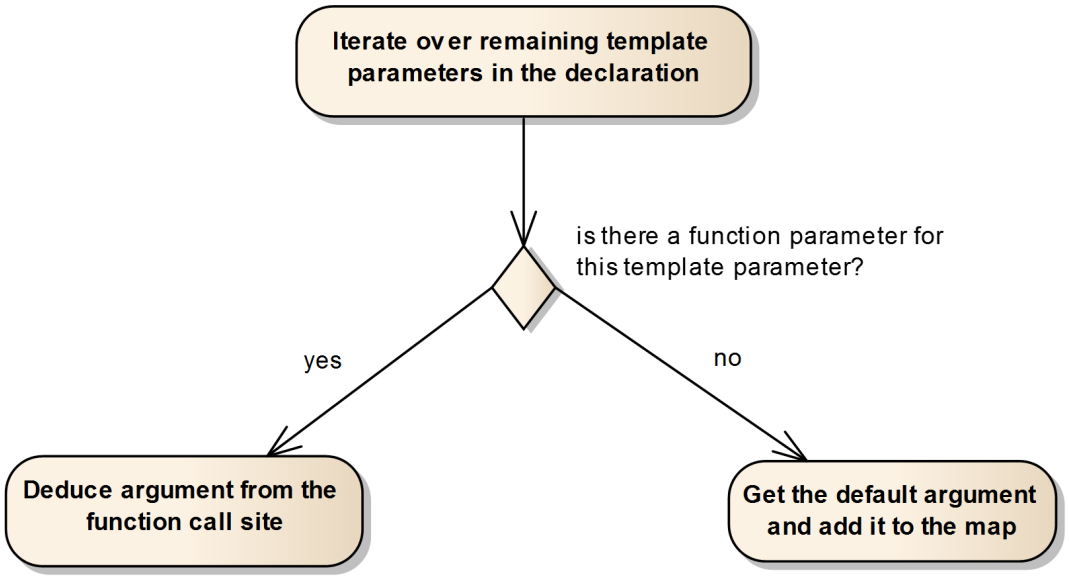


Figure 4.2: Then a check is made if the default template argument needs to be added to the map

```
1 template<typename T=int, typename F>
2 void foo(F value) {}
3
```

```
4 int main() { foo(45.8); }
```

This example shows that the template parameter `T` has no argument passed from the call side. So a parameters default type is added to the template parameter map if the argument is not overwritten on the function call site. This is done by checking if the argument for this template parameter is already in the map and if not, checking if there is a function parameter where this argument can be deduced from.

Deduction

Finally the arguments need to be deduced if they are not added yet. This code serves as example for the below enumeration.

```
1  template<typename T, typename F, typename G>
2  void inner(T first, F second, G third) {}
3
4  template<typename T>
5  void outer(T first, int index) {
6      double d{2.7};
7      inner(index, first, d);
8  }
9
10 int main() { outer('c', 9); }
```

We determine the first function parameter position where the template parameter serves as type. With the position we are able to get the passed argument on the function call site. This is of type `IASTInitializerClause`. Now there are 3 cases for the passed argument. The passed argument is a normal function parameter, a template parameter, a variable or a literal/function call with a return value.

- The argument is of type IASTIdExpression which means it was also a function parameter on the call site.

    1. Then we resolve the parameter name to a binding and check again the type of

the binding.

2. If of type `IVariable`: this means it is a non-template parameter and we resolve all aliases like `typedef`s und `using`s. This is the passed `index` in the listing above.

3. If of type `ICPPTemplateParameter`: this means it was a template parameter on the call site and we get the template argument for this parameter from the call site template parameter map. This is the `first` in the listing.

- If the passed argument is only of type `IASTExpression` we get the expression type. This is for local and global variables, literals and functions passed as arguments. This is for the `d` in the listing.

Now the complete template parameter map is filled and can be used with the `LookupData` instance to get the called function with `org.eclipse.cdt.internal.core.dom.parser.cpp.semantics.CPPSemantics.resolveFunction`.

4.2 Find the function definition

Once we had the correctly resolved binding we needed to find the AST node representing the function definition for this binding, an `ICPPASTFunctionDefinition`. There are two different cases that can occur. Either the binding is found directly in the AST or needs to be searched in the index which results in reparsing the containing file to get the AST.

4.2.1 AST searching

First the binding is searched in the AST for the currently opened file in the *Eclipse* editor. The content is dependent on the flags the programmer set for his indexer. So we cannot rely that the binding is found here because the default *CDT* index flags are set to not include header files in the AST. There is an existing method `org.eclipse.cdt.core.dom.ast.IASTTranslationUnit.getDefinitionsInAST(IBinding)` to get the function definition from a binding which we used.

4.2.2 Index searching

If the function is in an included header file, it resolves to an `IPDOMBinding`. These kind of bindings can not be found in the AST because the index is logically stored in a file in the workspace and the AST is just an in memory representation of the current code. So the `IPDOMBinding` is not found in the AST.

This can be in the same project, in any referenced project or even outside the workspace for headers from the C++ Standard Library.

If the binding is not found in the AST there has to be an `IIndexName` for this binding in the index. Otherwise the programmer has not included the file in his translation unit or the index is out of date and we show the user a warning that he or she should try to rebuild his index.

The difficult part is that the index only stores function declarations and not the function body, because for code completion it is enough to know the function name, the return type and number of parameters and their names. So it is not yet possible in *CDT* to access this functionality so we had to implement that functionality.

The `IIndexName` knows the translation unit (=file) which it belongs to and it is easy to get the AST for a file in the *Eclipse* workspace. But files can also be outside the workspace like C++ Standard Library headers. In this case the file must be parsed again to create an `IASTTranslationUnit` and then the already existing binding can be searched for in this new AST. We implemented a failsafe solution if the binding is still not found. The `ICPPASTFunctionDefinition` node which represent the function definition is selected via a node selector. A node selector just selects a node based on file offsets and length which are known from the `IIndexName`.

4.3 Formatting the deduced template arguments

After we had the function template specialization with all arguments and the finally called function definition, we needed to prepare the data to show it in our view. Our goal was to show compilable code which can be copied and compiled again in *CDT* for example. The final code to create a string from a node is then called in the User Interface

(see Chapter 5 User Interface on page 44), this section describes the creation of the data needed for that.

The goal was to get from Listing 4.4 to Listing 4.5.

```
1  template<typename T>
2  T id(T value) {
3     return value;
4  }
5
6  template<typename T>
7  T outer(T value) {
8     return id(value);
9  }
10
11 int main() {
12    outer(42);
13 }
```

Listing 4.4: Original code before the AST modification and formatting to show the deduced arguments

```
template<typename T=int>
T id(T value) {
   return value;
}

template<typename T=int>
T outer(T value) {
   return id<int>(value);
}

int main() {
   outer(42);
}
```

Listing 4.5: Modified code after the AST modification and formatting to show the deduced arguments

The easiest way to achieve this is copying the AST because it is frozen, modify it and finally print the code representing this modified AST. All building blocks for this functionality are already in *CDT*.

4.3.1 Copy existing function definition

Each `IASTNode` has a `copy` method which copies all child nodes recursively so we do not need to copy the whole `AST`. We need to copy the existing `ICPPASTTemplateDeclaration` node because the AST is frozen and cannot just be modified. Figure 4.3 shows how the `IASTNode` subclasses map to C++ code.

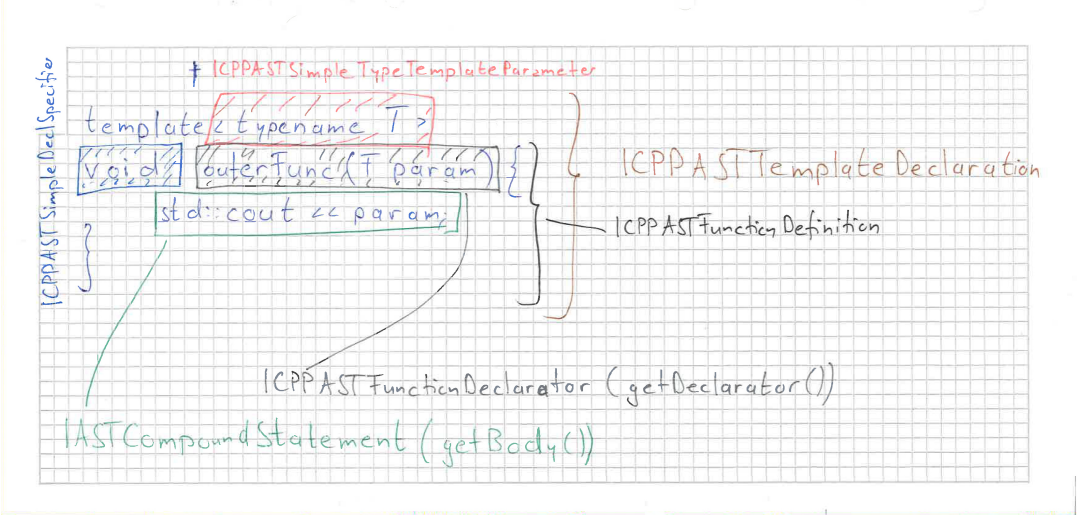


Figure 4.3: Sketch with explanation for `IASTNode`s in a function template declaration we used the whole term

In the previous step Section 4.2 we got the correct `ICPPASTFunctionDefintion` node but now we need to differentiate between function template and non-template function definitions. If the parent of the `ICPPASTFunctionDefintion` node is of type `ICPPASTTemplateDeclaration` then we have a function template definition. So the parent template declaration needs to be copied because we need to add default arguments to template parameters and these `IASTNode`s are in the template declaration. Otherwise it is enough to copy the function definition `ICPPASTFunctionDefinition` node and its child nodes because it is a non-template function.

4.3.2 Add template arguments

A copied node can now be modified to show the template arguments as default arguments in the template parameter declaration and on the function call site as template-id which we describe in the following paragraphs.

Function call site

A function call is represented as `ICPPASTFunctionCallExpression` and has an `IASTName` that provides the called function with a name. Figure 4.4 shows a function call without a template-id and the representing AST nodes (upper `ICPPASTFunctionCallExpression`) and one with a template-id (lower). The lower `IASTName outer` has a parent of type `ICPPASTTemplateId`.

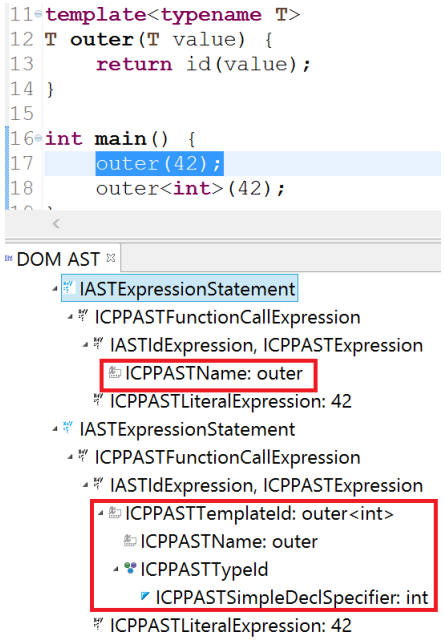


Figure 4.4: Function call with and without a template-id and the representing AST nodes

This `ICPPASTTemplateId`, containing the original name, needs to be created, the template arguments added and replaced in the `ICPPASTFunctionCallExpression`. However, there are some special cases. It is possible that there is already an existing template-id that is empty, containing some arguments or even all arguments.

So there are the following 3 cases:

- No template-id: create a template-id with a node factory and add all template arguments.
- Template-id where not all arguments are added: identify the number of already

added arguments and add the remaining.

- All arguments are already in the template-id: nothing to do.

Template parameter default argument

The deduced template arguments are shown as default arguments in the template parameter declaration.

`PARAM` as in `template<typename PARAM>` is an `ICPPASTTemplateParameter` and there are also 3 cases to differentiate (there are 3 subclasses) because for each case, the default type needs to be set differently.

- `ICPPASTSimpleTypeTemplateParameter` is a template parameter like the above `PARAM`. A parameter with a name and the most used one: A `TypeId` must be created with a declarator and then be set as default type.

- `ICPPASTParameterDeclaration` for function parameters or non-type template parameters: They can be initialized with a more complex initializer and we create a new `IASTEqualsInitializer` with the typename as expression so the equal sign is created when the code is created for the view.

- `ICPPASTTemplatedTypeTemplateParameter` nested template parameters as `V` in `template<template<typename T> typename V> void foo()`: format all nested template parameters with the above 2 methods recursively and then create an expression to set it as default value.

4.3.3 Pretty printing the modified AST

CDT already has a class to create the code behind an AST node and all its children: `org.eclipse.cdt.internal.core.dom.rewrite.astwriter.ASTWriter`. This code is called in the UI that is described in the next chapter but uses the above described modified nodes.

5 User Interface

Eclipse offers with Standard Widget Toolkit (SWT) a heavyweight widget toolkit for Java and in addition, *JFace* adds an abstraction layer and richer widgets. And for itself also has many UI classes. But none of the existing widgets had the functionality we needed to show the template instantiation to the user, so we built it ourselves.

This chapter describes the evolution of our UI. Then we go over the concepts we followed when implementing the view and finally we will talk about the implementation in detail.

5.1 How the view evolved

This section describes how our UI evolved over the 14 weeks. At first we showed the deduced arguments in a simple mouse hover which then became a standalone view.

5.1.1 The first approach—A mouse hover

As a first approach we implemented a simple mouse hover that extends the default hover that has already been shown. It was shown when the mouse cursor remained over a function template call for a brief second as shown in Figure 5.1.

This was realized by creating a hook into the extension point `org.eclipse.jdt.ui.javaEditorTextHover` and creating a class that implemented `ITextHover`. This class was used to determine if the mouse cursor is over a function template instance and therefore show the hover. This was sufficient to make first experiences with resolving function templates. At this time the template logic behind it was much more important than the view itself.

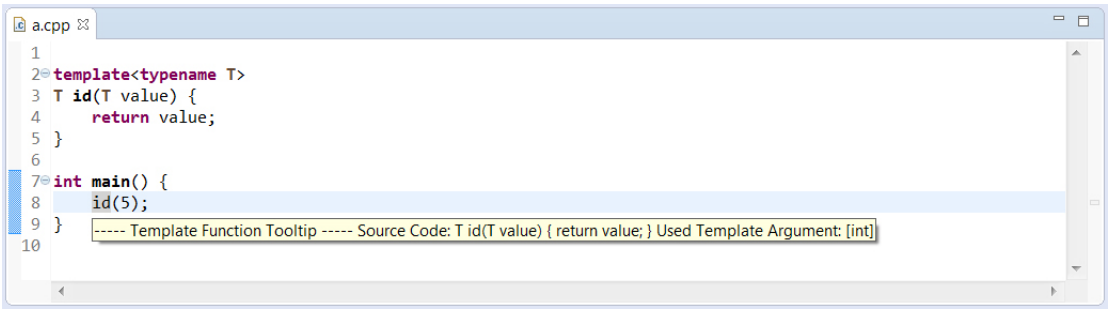


Figure 5.1: First version of showing the user information—a mouse hover

After a while the need for a separate UI grew. The hover solution did not provide enough flexibility. This was mainly due to the fact that we could not do any formatting in the shown hover such as syntax highlighting or even line breaks. We were unable to find out what the existing hovers in *CDT* did different than our plug-in. Another problem was that we could not add any interactivity to the hover window and that was what we needed at this point to expand subcalls of function templates. From there on it was clear that we needed a more sophisticated view that would offer more flexibility.

5.1.2 Using a view

To gain more flexibility we decided to show the template information in a specific view that was able to stay open and allowed us to add interactions to it.

Starting the template deduction process

Since we could not rely on the hover interaction to show the template information any more, we needed a new way to initiate the template argument deduction process and show the result in the view. *Eclipse*s extension points `org.eclipse.ui.commands`, `org.eclipse.ui.handlers` and `org.eclipse.ui.bindings` were exactly what we needed. They allowed us to open the view via an action and offered us a way to bind this action to a shortcut.

- The `org.eclipse.ui.commands` extension point is used to describe an abstract

command which represents the concept of a User Interface (UI) function with a name. A command can then appear in context menus, toolbars and normal menus.

- `org.eclipse.ui.handlers` declares a concrete implementation of a command and allows it to implement the `IHandler` interface. This determines when the command is enabled and what exactly is executed. A command can have multiple handlers so in a different context the command executes something different.

- `org.eclipse.ui.bindings` associate a command with a key binding. So a command can be executed in many ways.

We implemented this by writing the `ShowTemplateInfoHandler`, which is our concrete implementation of the `IHandler` interface. This class determines if the command is enabled by checking if the mouse cursor is over a function template instance.

The last thing was to offer the user a way to trigger the command. Therefore, we used a key binding that was linked with the command and the user could set a shortcut by himself. We chose a shortcut randomly but that was not a good idea, because our choice of *Ctrl+7* was already used for comment in/out. Meanwhile we changed the default shortcut to *Alt+F8* which is more reasonable because it is unused.

The first version—A Tree view

We had a lot of ideas of how the view should look like, but since the data was strictly organized hierarchical in the form of a tree, a tree view seemed like a good solution. Another benefit was that it was very easy to implement and SWT offered all the required classes out of the box. We had the tree view up and running very quickly and could show the results to our supervisors in some earlier meetings. Figure 5.2 show this approach.

But we tested the tree view only for relatively small number of examples and did not notice that the view was becoming crowded very quickly. This caused the user to loose track of the original expanded function because it was necessary to scroll a lot to find the wanted function definition.

Also we had no way to handle nested functions properly and before wasting to much time in this feature, we decided to follow a different approach.

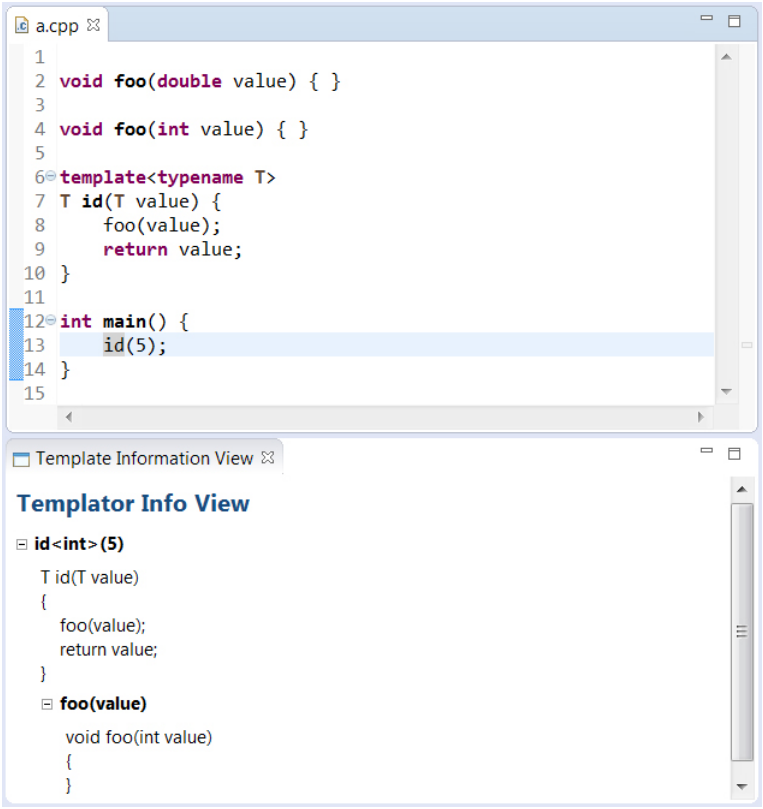


Figure 5.2: Our second UI approach—a tree structure in an own view

Using the Eclipse built in compare Editor

When Prof. Peter Sommerlad saw the problem with the tree view, he proposed to expand subcalls in horizontal direction very much like the *Eclipse* compare view as seen in Figure 5.3. We then tried to used the compare editor for our purpose.

The problem with this approach was that the compare editor supported only two columns in total. It was also designed to have a source file behind it since it was inheriting from the Java editor. Since we only had artificial code, it was not possible to use the comparison view as it is. It would have meant that we would have to store our artificial code in a temporary file and keep it in sync with the artificial AST. This did not appear to be a good solution for us. The reason was that we would have to write a lot of code around the core problem to get the inflexible compare view up and running.

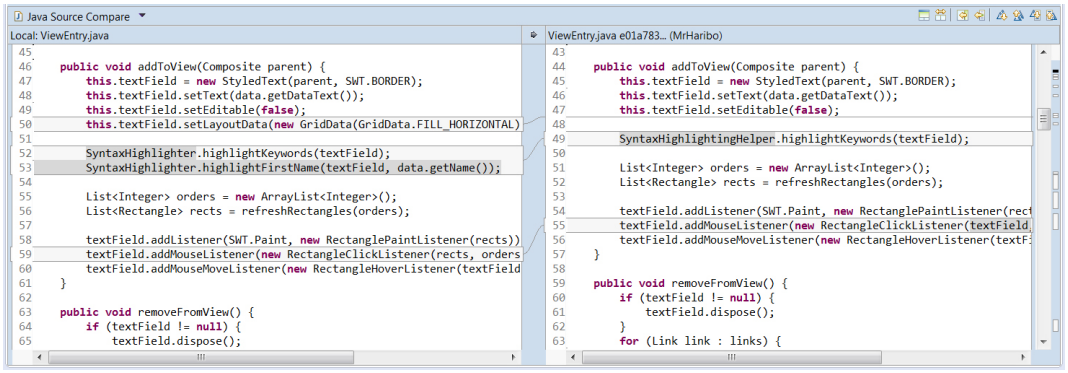


Figure 5.3: The *Eclipse* comparison editor we used as inspiration for our links

The next idea was to try and use some parts of code from the compare editor. It turned out that it was hard to find exactly the right code that we needed. We only wanted the code for the connection between the compare nodes in the compare editor. After a little bit of research we found out that SWT was offering a path class that was able to draw a cubic path with just a few lines of code. Since that was everything that we needed we decided to build a new view from scratch to fulfill our specific needs.

Building a new view from scratch

It was easy to build up a new view by using the extension point `org.eclipse.ui.views`. From hereupon we had our playground to try out different approaches to show the template information in our view.

We did not need a lot of features in the first place so we could quickly develop a fully functional version in a short time. The basic functionality could be very much taken from the first approach with the tree view. The only difference was that it expanded horizontally and not vertically. We quickly implemented a first version with clickable function calls and a Bézier path between the function call and the definition that popped up on the next column to the right.

Since we did not have much experience programming with SWT at that time, we did not spent much effort in developing a clear code structure in the first place. Thus the view code was not very well structured and continuously growing. That made it difficult to

add new functionality because it would made the code even more ugly.

On the other hand we had a simple prototype to show very quickly and we could develop a feeling of how our UI would look like. We could also show it to our supervisor to gain early feedback.

The limits of the first version

At this time we had a discussion whether we would have to implement any big features in the view that could not be done with our simple version. Although it was a bit complicated, it solved its purpose very well up to this point to show all the needed information about function template instances.

The thing that broke our neck was a feature we purposed that we though, was nice to have and our advisor Prof. Peter Sommerlad agreed. The feature was about opening links in the order they occurred in the parent functions body. A link is a function call that is marked with a rectangle and can be clicked to expand its function definition. Up to this point, newly expanded links where simply shown at the bottom of the next column and forgot their ancestors (discussed in Subsection 5.3.3 The ViewEntry class on page 58. It also made it impossible to close the link again and resort the entries since we did not store any information about ordering.

The difficult part to implement this was that one function definition entry would need a lot of information for sorting. This would be information about preceding entries in the same column to determine its index in the column. It would only be possible to gather this information if thew whole tree until the root function definition was traversed. For example if there are 5 nesting levels, the position on which a function definition should appear depends on all 4 preceding levels and how many function calls each function definition has. This information was not available in our existing view and it would have made the already complicated code even more complicated. Simply refactoring of the code or just breaking it up in smaller parts was not an option because the view grew evolutionary with our need. This had resulted in some legacy code that defined the way how we did things. This first version of the view was simply not suited for an expansion of this scale.

The new view

This lead us to do an all over refactoring to build up a new view that could support all the features that where requested and would make the maintenance in the future a lot easier. Since we also knew at this point that we would write our bachelor thesis about class templates it was the best to write the new view that it was prepared to support class templates as well. The old view was very tightly coupled with function templates and this was something that was really annoying for us at this point.

A good thing about the old prototype was that we had a lot of working code already that only had to be altered slightly and moved to new and better locations, and so the code was better structured according to the Software Engineering principles.

The end result is quite pleasant and we are looking forward to work with our clean solution during our bachelor thesis. As seen in Figure 5.4 our UI developed into a easy extensible skeleton that will support class templates and is visually pleasant.

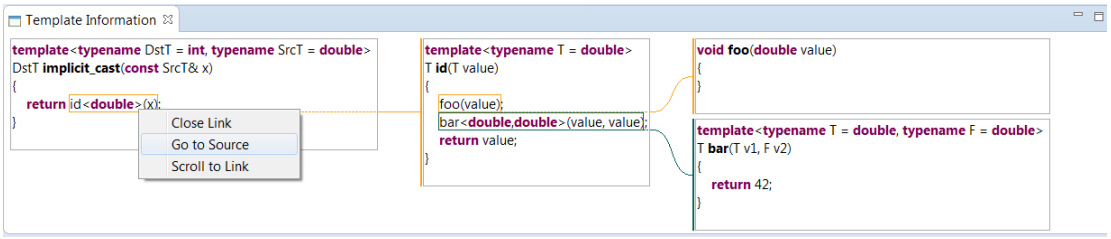


Figure 5.4: Our final view

5.2 The view in detail

We will now talk about the concepts that we followed while were implementing the view. Especially how it fits into the *Eclipse* user workflow.

5.2.1 Showing the TemplateView

The entry point for the view is the `TemplateView` class. It derives directly from view parts that can be registered in `plugin.xml` under the extension point `org.eclipse.ui.views`. The view can then be shown in many ways. We offer 2 different ways of opening the view to the user.

Executing the command

The first one is via the command handler that is also registers in the `plugin.xml` (see Section 5.1.2 Starting the template deduction process on page 45). This approach has the advantage that we can instantiate the view and inject data to it after it was instantiated. This is done with the following commands:

```
1 IWorkbenchWindow win =
     PlatformUI.getWorkbench().getActiveWorkbenchWindow();
2 IWorkbenchPage page = win.getActivePage();
3 IViewPart view = page.showView(TemplateView.VIEW_ID);
```

Listing 5.1: Opening a view in Eclipse

The data then is injected like so:

```
1 TemplateView templateView = (TemplateView)view;
2 templateView.setRootData(viewData);
```

Listing 5.2: Injecting the view data. The class cast exception handling is hidden.

`ViewData` is independent from its underlying data (e.g. `FunctionTemplateViewData`) that means, all the data needed can be injected this way and the view does not need to have any references to template specific classes.

This approach remained the same for almost the whole project duration since it is very stable and simple at the same time.

Show In... context menu

The "Show in..." context menu is the second way to show our view. It is used in *Eclipse* to show the current selection in a view and registered with the extension point `org.eclipse.ui.perspectiveExtensions`. This enables the "Show in..." command for an *Eclipse* perspective, in our case `org.eclipse.cdt.ui.CPerspective`. Our view then implements the interface `IShowInTarget` which provides the method `boolean show(ShowInContext context)` that is called, when the view is opened.

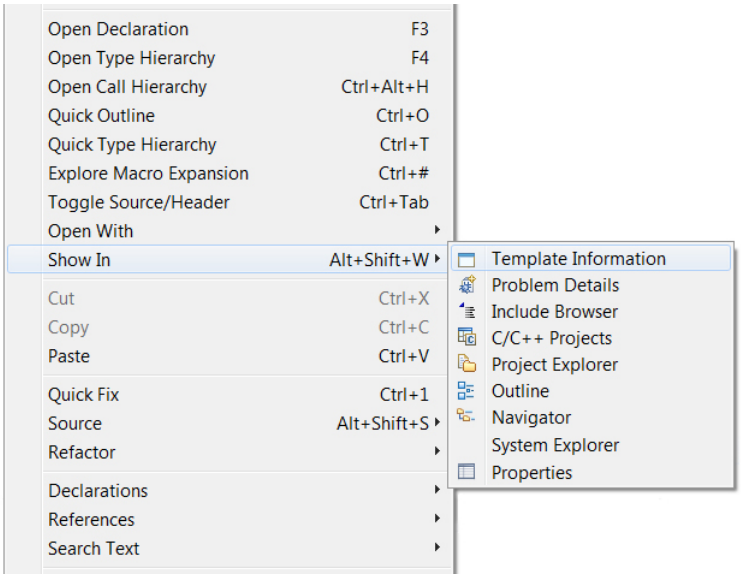


Figure 5.5: Opening our view via context menu "Show in...".

This approach is very nice for new users and helps them to find the plug-in. Although it has a few disadvantages. The major disadvantage is that the context that is given to the show method is not specific enough. If a name in a function body is selected then we get a `ShowInContext` instance that encapsulates the region of the surrounding function (e.g. the main function for a function call within main). This is useless for us and we cannot find out which name was clicked initially because this information is lost. At least under Windows, a right click does not position the cursor at the clicked coordinates but stays at the its previous location. This brings us to a solution which is very ugly from a Software Engineering point of view because the view has to ask a superior class for the data it needs. This superior then reopens the view from the outside and injects the data properly.

Since we implemented this approach really late we had no time to find a way to solve this problem nicer according to Software Engineering principles. This is something that we definitely want to improve during our bachelor thesis.

5.2.2 Finding the IASTNode for which the template deduction is executed

As a root for template resolving, we use `IASTNode`. To find the node that is selected, we use the selection from the `IEditorPart`. This is achieved by the following code:

```
1 ISelectionProvider provider =
     editorPart.getSite().getSelectionProvider();
2 ITextSelection textSelection = (ITextSelection)
     provider.getSelection();
```

Listing 5.3: Getting the current selection in an Eclipse editor. Simplified from a helper function without exception handling

If we get a selection we try to get the corresponding node out of the AST:

```
1 IASTNodeSelector nodeSelector = ast.getNodeSelector(null);
2 IASTNode node =
     nodeSelector.findEnclosingNode(region.getOffset(),
     region.getLength());
```

Listing 5.4: Getting the `IASTNode` for a editor selection

If the cursor was over a function template instance, we try to build an instance of our own class `FunctionCall` with the found `IASTNode`. We then generate the `TemplateFunctionViewData` out of the that then can be injected into the view.

If a problem is occurring during any of this stages, the process is stopped and the user gets an error message with a brief description of what happened.

If the `TemplateFunctionViewData` could be built successfully it is passed to the view as `IViewData`. The view does not have any knowledge about the data in `IViewData` itself

and is only working with it over interface functions. This is also done with the reuse of the view in our bachelor thesis in mind. This means that we can focus on class templates and adding new usability features rather than spending much time for the view (see Section 7.2 Future work on page 68).

5.2.3 A view out of columns

Since we wanted to make our view look similar to the comparison view, we built our view around the concept of columns as we call it. The difference between the comparison view and our view is that we not only have 2 columns but theoretically an infinite number of columns.

Another difference is that in the compare editor, each column contains the source code of exactly one source file that is displayed in one `TextField` which is divided into sections that are connected with the sections in the second column. In our case, we needed multiple independent `TextField`s that can be further divided into sections that serve as origins for the links. The endpoints for the links then are the `TextFields` in the next column. Figure 5.6 shows an early draft we created for this column based UI.

5.2.4 How the columns map with function templates

The way we put the data into the columns is pretty straight forward. One `TextField` in a column contains exactly one function definition. Inside this `TextField` we mark the interactive parts of the text with rectangles that the user can distinguish them from the rest of the source code. These rectangles serve as link starts. The link's end is one function definition in the next column. This is the core functionality that is required for the view and everything else we have made is just for usability.

5.2.5 Mapping of function call and its definition

Inspired by the comparison view in *Eclipse*, we wanted to have something that looks similar to the links in the comparison view that connects 2 pieces of code together. In

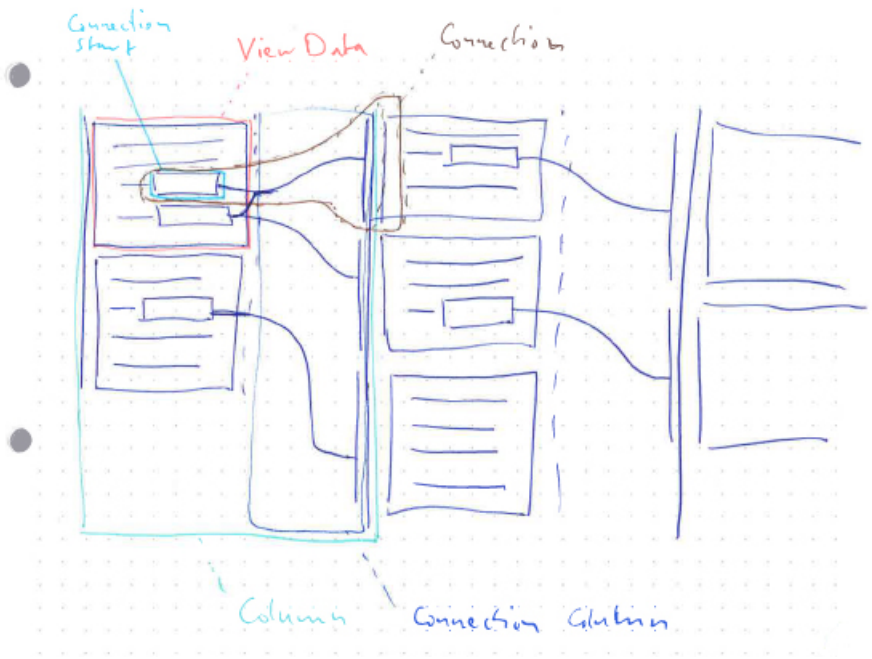


Figure 5.6: Draft we created for our column based UI

our case the link should connect a function call with the definition of the function in the next column.

5.2.6 Using nested GridLayouts for the hierarchy

The approach that worked very well to build the hierarchy was to use a nested `GridLayout`. The outer `GridLayout` is the first level of the hierarchy and consists of a single row with multiple columns. Every column then has its own `GridLayout` with a single column but multiple lines. This is the second level of the hierarchy. Every line of the inner `GridLayout` then contains exactly one `TextField` that contains one function definition. These are the end nodes of the hierarchy that can be understood as leaves of a tree.

Because a tree always has a single root, the first column will always have only one entry since the starting column never has an ancestor. Further from there every column could have an arbitrary number of children. These children have exactly one entry in the previous column as their originating entry (see Subsection 5.3.3 The ViewEntry class). If

an entry has no originating entry, it is the root entry.

Encoding of the entrys

To avoid ugly bidirectional references between the columns and their entries, we simply use integers for positioning. An entry knows in what hierarchy level (=column index) it is and what its index in this column is. With this information the `IViewController` can find again the column to which the entry belongs. Since the nesting level of an entry never changes this approach will always work as simple as it is.

5.3 The implementation of the view

The most important classes of the view are `TemplateView`, `ViewColumn` and `ViewEntry`. These 3 classes represent the 3 levels of our view hierarchy. The remaining view classes encapsulate often used code and can be used from the 3 major classes to solve specific tasks.

A `TemplateView` contains a list of `ViewColumn`s which again contain multiple references to `ViewEntry`s. Because the column has no information about its position in the layout, the view inserts the entries into the correct column. Figure 5.7 shows a simplified overview without interfaces and Figure 5.8 their visual representation in the view.

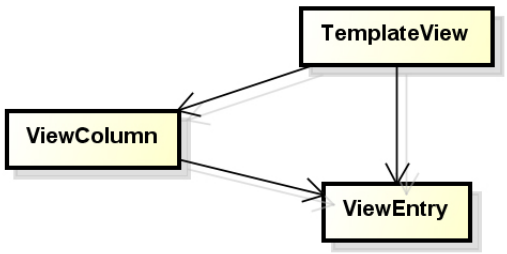


Figure 5.7: Simplifed overview of how the UI main classes reference each other

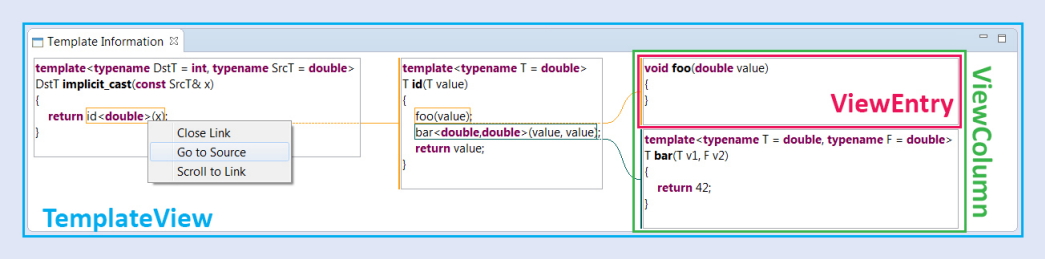


Figure 5.8: Visual representation of the 3 main UI classes

5.3.1 The TemplateView class

The `TemplateView` class serves as the root container for the whole view. The function `createPartControl(Composite parent)` is called from the `IViewPart` interface when the view is initialized. Here we can do our initial setup of the view.

A scrolled form is serving as a body of the view. All additional composites have this body as parent directly or indirectly via another component. That means, when the form is disposed, all of its children will be disposed with it.

It also holds references to the first level of the hierarchy of composites that contains the tree of function calls in their call order. Each layer in the hierarchy is responsible to create and destroy all of its child composites. It is stored as a list of `ViewColumn`s that is kept sorted to correspond with the nesting level of the functions.

To provide functionality like reflowing the nested grid layouts, the template view implements the `IViewController` interface. This interface works as a controller for deeper nested widgets that do not have access to the top level widget.

5.3.2 The ViewColumn class

This class is the second level of the hierarchy and holds a list of `ViewEntry` references. It does not need to know anything about the order of the entries since the template view holds that information and the `addEntry(ViewEntry entry)` function also gets this knowledge injected by the `IViewController`.

The main responsibility of this class is to resort the entries when a new entry is added because the order of the entries can change, and when that happens we have to rebuild the container.

It also takes care of refreshing the connections when the order of the entries changes because it stores a reference of the connection column where the connections are drawn and passes it to the entry upon refreshing. Like the order of the entries, the links always change after adding or removing an entry.

5.3.3 The ViewEntry class

This is the most important class of the view. It represents one entry in a column. It has knowledge of the index of the column it is within and the index of its row. The first is necessary to be able to add a new entry in the next column and the latter is important for ordering. It also contains a reference to data it encapsulates. This is wrapped in the `IViewData` interface which provides functions to get out the data to fill the content into the `TextField`, no matter what data it represents. This will come in handy when we will support class templates next semester. Finally it contains a list of links that represent the clickable links to the nested function calls that can be opened and added to the next column.

The entry implements the `IComparable` interface that is necessary to sort the entries in the `ViewColumn`. The sorting is the feature that gave us the most trouble. Since it was a requirement to order `ViewEntry`s the way they appeared in the calling functions body, we needed to know where the entry was opened from. Because if the parent entry was not the first in its column, all children that originated from an entry that came before the parent in it column entry, had to be in front of this very entry. The sketch in Figure 5.9 shows the view before closing a link and Figure 5.10 shows the view after a link was closed. This action caused a reordering.

To achieve that we need a reference to the parent link from which the entry was opened. The link then knows which index of function call it has. If there is no parent link, we are at the root and the entry is the only one in its column. For this to work, the comparator simply gets the index of the parent entry in its column. If this index is not equal, the order of the compared entries is already clear. If it equals, the index of the parent link

determines the order. With this concept, the entries of one column can be sorted with ease.

The rest of the `ViewEntry` defines its rendering and also receives callbacks for clicks. For matching of clicks to links, the references to the rectangles that is a member of the link are compared.

Requests to open new function definition entries can be passed to `IViewController`.

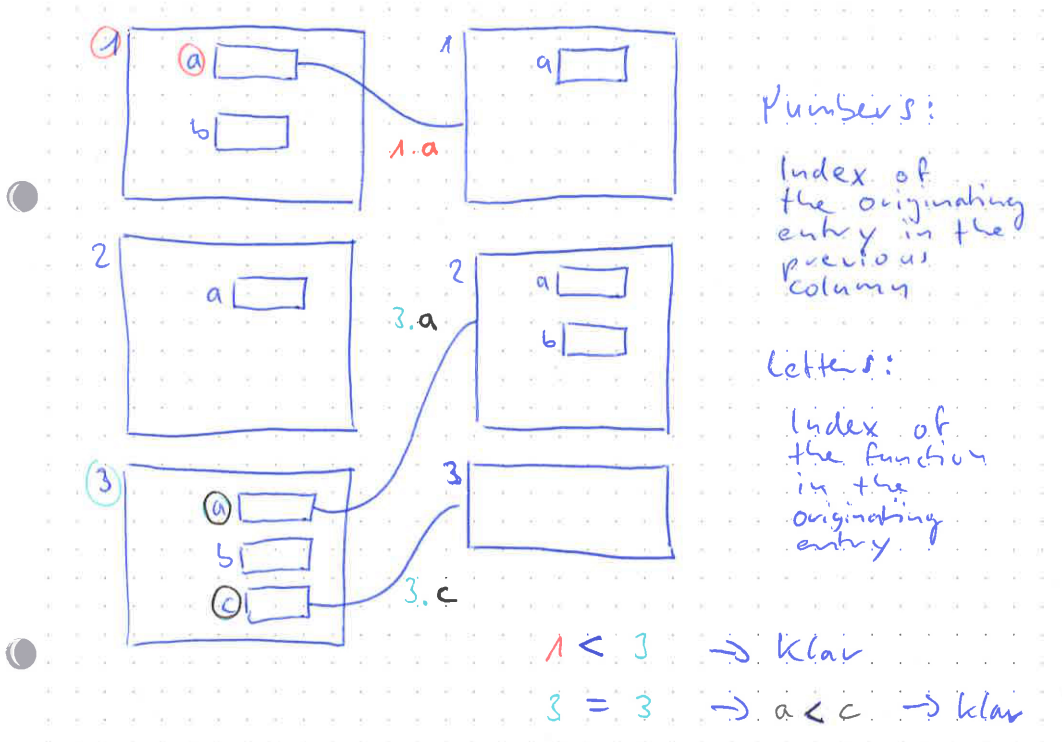


Figure 5.9: Sketch illustrating the ordering of function definition before closing of link 3.a

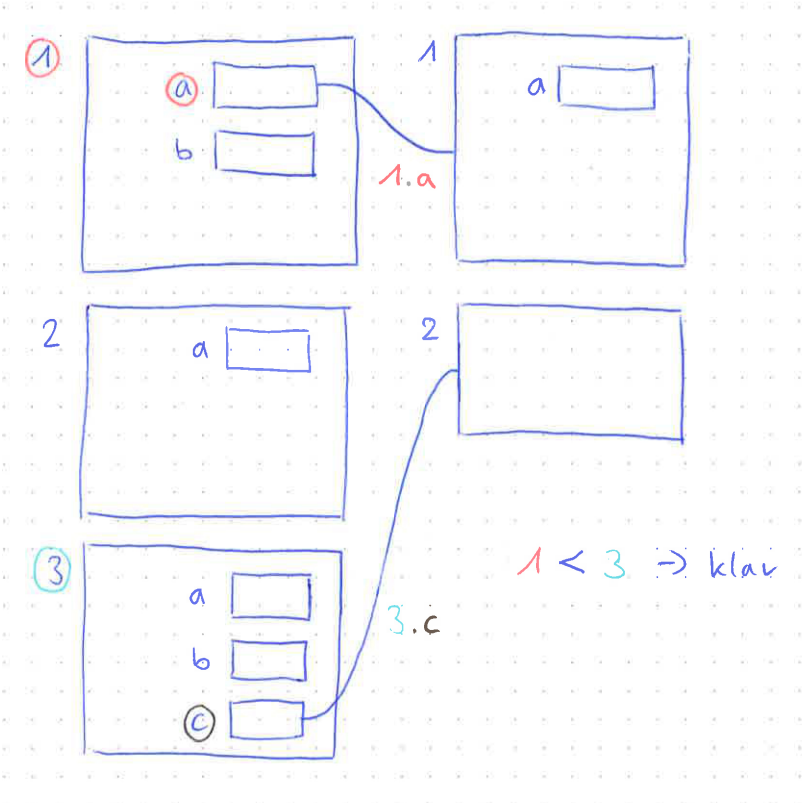


Figure 5.10: Sketch illustrating the ordering of function definition after closing of link 3.a. Link 3.c becomes the second entry now.

6 Testing

We describe how we tested our application and what features we automatically unit test.

6.1 Strategy

At the beginning we planned to follow a TDD cycle. This was too complex and most importantly, consumed too much time. Running a JUnit-Plug-in test takes about 30 seconds to create the *Eclipse* instance and run the test on our computers.

So we created JUnit tests for the most important and used classes `ASTTools` and `ASTAnalyzer` after we created and tested them manually.

6.1.1 Manual testing

For the most of the project, we tested our features manually by starting a *CDT* instance with our plug-in loaded and own code examples. These examples were created mostly by us and some by our advisor Prof. Peter Sommerlad. The most complex example containing almost all supported feature is shown in Listing 6.1 and passed the test in our final Templator version.

```
1 struct MyClass{};
2 typedef long mytype;
3 bool its_over(int i) { return i > 9000; }
4
```

```
5  // using class just to show it works
6  template<class TEMPLATE_ID, class SECOND, typename THIRD,
       class STAYS_DEFAULT=mytype, typename FIFTH, typename
       SIXTH=MyClass, typename SEVENTH>
7  TEMPLATE_ID mitAllesOhneScharf(SECOND second, THIRD third,
       FIFTH const fifth, SIXTH sixth, SEVENTH const* seventh)
       {}
8
9  template<typename DOUBLE>
10 void start(DOUBLE first, mytype second) {
11     double d{13.37};
12     // <int,double,bool,long int,double,long int,MyClass *>
13     mitAllesOhneScharf<int>(first, its_over(9000), d,
       second, new MyClass{});
14 }
15
16 int main() {
17  start(5.4, 4); // <double>
18 }
```

Listing 6.1: Test C++ code containg the most supported features that our plug-in supports

6.1.2 Unit testing

We created unit tests for the most used and important methods after we created them. We added the C++ code we used for the unit test as comment above the JUnit test method. With reflection we are able to get the comment from the call stack and parse it with the existing `org.eclipse.cdt.core.parser.tests.ast2.AST2TestBase.parse(String, ParserLanguage)` to get an AST in form of an `IASTTranslationUnit`. This was needed for the basic automated tests which just tested a single method.

We then created automated integration tests with JUnit to support the whole resolving of a function call to the function definition. For this we needed a more complex setup with its own C++ source and header files that could be included. We used the Institute

for Software (IFS) testing framework for *CDT*[5] which offers support to create a C++ file, get the content and also creating `ICProjects` instances which we needed to create an index from. Because in *CDT* an index exist per C/C++ project.

The tests were run automatically on our build server with each commit. This helped us to ensure that we did not break anything existing.

The next section describes in detail what cases and functionality we tested.

6.2 Unit tested features

We created unit tests that check to correct behavior of single, frequently used main methods and then integration tests for the function resolution process. This section explains what we tested and contains examples of our JUnit tests to better understand the testing ideas we had.

6.2.1 Single methods

We tested all `public` functions in our `ASTAnalyzer` and `ASTTools` class. The first class `ASTAnalyzer` contains `static` methods that need access to the whole AST and the index. For example `ICPPFunction getFunctionBinding(IBinding)` to get the correct function binding. For a template binding (class and function) we need to call `getSpecializedBinding()`.

Listing 6.2 shows one of our four tests for the `getFunctionBinding` method. We created our own helper classes and methods to get a specific `IASTName` which was enough for most tests.

```
1 public class GetFunctionBindingTest extends
     TemplatorSimpleTest {
2     @Test
3     // template<typename T>
4     // T bar(T value) {
```

```
5       //      return value * 42;
6       // }
7       // int main() {
8       //      bar<int>(-30.0);
9       // }
10      public void testWithAngleBrackets() throws Exception {
11          String source = getCommentAbove(getClass());
12          IASTTranslationUnit ast = parse(source, CPP);
13          ASTAnalyzer al = new ASTAnalyzer(null, ast);
14          IASTName name = findName(ast, 1, "bar");
15          IBinding binding = name.resolveBinding();
16          ICPPFunction bind = al.getFunctionBinding(binding);
17
18          assertNotNull(bind);
19          assertTrue(bind instanceof ICPPFunctionTemplate);
20      }
21  }
```

Listing 6.2: Unit test for a single method. Shortened the names and the class so it just contains one test

The second class `ASTTools` contains methods where the complete AST and index is not needed. For example `isFunctionTemplateInstance(IASTName)` to check if a name finally resolves to a function template instance as the name says. These tests follow the same pattern like the above Listing 6.2 with parsing the code, calling the method to test and assert the result.

6.2.2 Function resolution

All integration tests follow the same principle and the tested C++ code is almost always a function template instantiation an the `main` function. The called function template calls another function template. Then there is another non-template function with the same name as the nested function template with the same number of parameters. Listing 6.3 shows the C++ code we used to test if a function call with an empty template-id `<>` resolves to the function template and not the non-template function as described in the

C++ Standard[6, , 14.8.1]. This was a late bug we found and after fixing the bug, we created this test for it.

```
1  template<typename T=double, typename F=int>
2  void inner() {}
3
4  void inner() {}
5
6  template<typename T, typename F>
7  void outer(T first, F second) {
8      inner<>(); //<double,int>
9  }
10
11 int main() {
12     outer(42, true); //<int,bool>
13 }
```

Listing 6.3: Test C++ code to test if the `inner<>()` call resolves to the function template

Almost all tests check the same 3 points:

- Did the template parameter map for the function template call in the `main` contain the correct deduced arguments?
- Did the `inner` call resolve to the function template and not the non-template function?
- Did the template parameter map for the `inner` function template call contain the correct deduced arguments?

Listing 6.4 shows the JUnit test class for Listing 6.3. Because all tests follow the same pattern and check the same points we created a class `FunctionTemplateResolutionTest` which each test extended.

```
1  public class
       DefaultArgumentsEmptyTemplateIdWithoutArgumentsTest
```

```
    extends FunctionTemplateResolutionTest {
2
3       @Test
4       public void testOuterArgumentMapIsIntBool() {
5           testOuterArgumentMap(INT, BOOL);
6       }
7
8       @Test
9       public void testSubcallArgumentMapIsDoubleInt() throws
    TemplatorException {
10          testFirstInnerArgumentMap(DOUBLE, INT);
11      }
12
13      @Test
14      public void
    testSubcallResolvedToFuncTemplAndNotNormalFunction()
    throws TemplatorException {
15          testFirstInnerCallResolvesToFirstDefinition();
16      }
17  }
```

Listing 6.4: JUnit test class for Listing 6.3 with shortened class and method names

We created other test to check exactly the opposite and some general test for the whole function resolution process. For example for the above listings that the function call without a template-id resolves to the non-template function and not the function template.

7 Conclusion

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

7.1 Achievements

We analyzed the problem what types of function template instantiations exist in C++ and why it can be hard to see what function finally gets called. We then developed a solution for this problem in form of a *Eclipse CDT* plug-in.

Our Templator plug-in is able to resolve the listed function template instantiations in Section 2.2 Supported features in our Templator plug-in on page 17 and offers the user a new view in *Eclipse CDT*. This view allows interaction to see what function overload the compiler finally chooses and the body of this function.

We managed to implement the features in a way that the code is not too complex, easily extensible and tested. We refactored constantly and used unit tests for automated testing the features. We did not just implement as many features as possible in the given time because we did not want to add a lot of code that becomes legacy code later and is hard to maintain and extend.

We built a generic view so that new functionality—like the one described in the next section—can be added easily.

7.2 Future work

As mentioned in the last section, our plug-in supports just some function template instantiations. There are some more complex C++ language features the plug-in does not handle yet and cannot show the user the correct function definition. This section describes how the plug-in could be extended in a future project that is most likely done by ourselves (Jonas Biedermann, Marco Syfrig) in the form of a bachelor thesis.

7.2.1 Using the CDT scanner

The actual Templator plug-in uses simple string matching and regular expressions for syntax highlighting, finding the starting point of a function body and to find the subcalls the printed code in the view. This causes keywords to be highlighted in string literals and if there is the exact same function call expression in a literal, this string gets framed indicating the user he can click on it. And the original function call expression does is not framed.

This could be improved in a future version by parsing the printed code again and using the *CDT* scanner that provides a stream of tokens. Then a function call expression can be easily found without string matching and syntax highlighting would be correct without highlights in string literals and other places.

7.2.2 Using existing CDT functionality

We built most of our functionality by ourselves. For example the whole deduction of the template arguments is done by our algorithm. *CDT* offers `org.eclipse.cdt.internal.core.dom.parser.cpp.semantics.CPPTemplates` and `org.eclipse.cdt.internal.core.dom.parser.cpp.semantics.TemplateArgumentDeduction` from the same package that already implement what we need to create the correct template parameter map with the deduced arguments. We already use the `public` functions that we could use but most functions are `private` or only callable in the same package. These two classes could be analyzed to check if they really do what the plug-in needs to fully support the C++ standard and cover all possible cases. Then the plug-in could use these functions by

creating a class in the `org.eclipse.cdt.internal.core.dom.parser.cpp.semantics` package or if that is not enough, by calling them via reflection.

Maybe using this *CDT* functions would make the next two points, missing function template instantiations and SFINAE, unnecessary. It is just the template argument deduction that does not work yet for these situations but the support to show them in the view was already added in this thesis.

7.2.3 Missing function template instantiations

As described in Section 2.2 Supported features in our Templator plug-in on page 17, some function template instantiations are just supported on the first hierarchy level. Support for deducing non-type template parameters and passing function template instances with a template argument could be added in the future.

7.2.4 SFINAE

Substitution Failure Is Not An Error is a technique where the compiler removes candidate functions for a template instantiation if a substituted template argument would result in an error [1, :106]. Removing the function from the candidate list is not done by our plug-in yet and can be added to really show the actually called function by the compiler.

7.2.5 Variadic templates

Variadic templates allows compile time typesafe functions with an arbitrary number of arguments. The compiler generates as many function definitions in the background as there are passed arguments. The plug-in is required to create the AST definitions to be able to show them to the user with the right amount of parameters and their correct types. Probably it is enough to just show the parameter pack and the last argument that is used as tail. This should be researched and implemented in a future version.

7.2.6 Class templates

Class templates offer new features. They offer partial and full specializations that are unknown for function templates. A specialization is an implementation of a class template where all arguments are specialized and for a partial specialization only some template arguments are specialized. An example for the latter would be an implementation of a class template for all pointer types. Different class template specializations do not need to have something in common which could possibly make the deduction more difficult. So class templates could be added to the plug-in to offer support to deduce them and then show them in the existing view. And additionally it could be implemented that the user sees what other specializations of this class template exist beside the finally instantiated one.

7.2.7 Showing errors

The current exception handling throws all exceptions with an error message to the view. The error message is usually understandable by the user. We open a dialog with the error message and the stacktrace can be expanded if wanted. This is standard error handling in *Eclipse*. But it can be improved by always showing a message that is not too technical and tells the user what to do to fix the problem.

A second problem is that the "Show in...-> Template Information View" context menu entry is always visible and not only on function template instances. This is a limitation in *Eclipse* we cannot avoid. But if the mouse cursor is on anything other than a function template instance nothing happens and the user gets no feedback. For this to work our `TemplateView` and `IViewData` would need new methods to support store and show an error message.

And the third point that could be improved: If a function binding is not found neither in the AST nor index we throw an exception with the error message, that the user should try refreshing the index for the project. This is shown as error with the stacktrace. In a future version it is advisable to only show a warning or information in this case and not an error.

7.2.8 Enable for non-template and C functions

The plug-in could easily be extended to start the process not only for function template instances but for any function call. Also C and not only C++ could be used for resolving.

Bibliography

[1] D. Vandevoorde and N. M. Josuttis, *C++ templates: the complete guide*. Boston: Addison-Wesley, 2003.

[2] E. Clayberg, *Eclipse plug-ins*, 3rd ed., ser. The eclipse series. Upper Saddle River, NJ: Addison-Wesley, 2009.

[3] "CDT wiki—overview of parsing," May 2011. [Online]. Available: https://wiki.eclipse.org/CDT/designs/Overview_of_Parsing

[4] M. Schorn, "Using CDT APIs to programmatically introspect c/c++ code," 2009. [Online]. Available: https://wiki.eclipse.org/images/c/c7/CDT_APIs_for_code_introspection.pdf

[5] H. IFS, "CDTTesting git repository," Oct. 2014. [Online]. Available: http://cevelop.com/cdt-test-plugins/development/

[6] ISOCPP, "Standard for programming language c++," Tech. Rep., Oct. 2013.

[7] "SWT snippets." [Online]. Available: https://www.eclipse.org/swt/snippets/

List of Abbreviations

AST	Abstract Syntax Tree
CDT	C/C++ Development Tooling
IDE	Integrated Development Environment
IFS	Institute for Software
JAR	Java ARchive file
JVM	Java Virtual Machine
OSGi	Open Service Gateway initiative
PDOM	Persistent Document Object Model
SFINAE	Substitution Failure Is Not An Error
SWT	Standard Widget Toolkit
TDD	Test Driven Delevopment
UI	User Interface
XML	Extensible Markup Language