HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

Templator2

Bachelor Thesis

HSR—University of Applied Sciences Rapperswil
Institute for Software

Spring Term 2015

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

Abstract

C++ allows the usage of templates to build functions and classes with generic types. This gives the advantage that classes and functions only need to be defined once and can be used for many containing data types without duplicating code. 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.

Based on the passed, deduced arguments, defined functions, and class templates the compiler selects different code that will be executed. Programmers using Eclipse CDT do not have easy access to the instantiated templates and thus to information about select function overloads and class template specializations. Programmers want to know what the compiler finally chooses, especially in the case of nested template instantiations.

The goal of our bachelor theses is to extend the plug-in for Eclipse CDT we developed in our term thesis. The existing plug-in is able to show the programmer simple function template instantiations and should now be extended to support class template instantiations. The outcome is a view that helps the programmer to examine function templates and their deduced arguments, class templates and nested function calls. It offers interactivity to recursively resolve function calls and class template instantiations for an arbitray nesting level. The UI assists the user with a search function, jumping to the definition in the C++ editor and displaying the resulting instantiations in a tree like hierarchy with many UI features.

Management Summary

This bachelor thesis builds on the results of our term project Templator [BS14]. The goal of the term project was to write an Eclipse CDT plug-in that lets a user visualize function template instantiations in his C++ code. In our bachelor thesis we want to extend the functionality to also support class templates.

Motivation

The C++ programming language provides templates to build generic functions and classes using compile-time parameters. A compiler will instantiate these templates by replacing their parameters with actual arguments at compile-time. This internally generated code can result in further nested template instantiations in the template function's or class' body. However, a programmer is not able to see this compiler-generated code unless it generates a compile-error message. The language rules of C++ employed by the compiler during template instantiation with respect to function overload resolution, template argument deduction and class template specialization selection are complex and hard to apply by a developer in his head. As a result, the invisible code resulting from template instantiations can contain compile errors, or in the worst case, unintended run-time behavior that is very hard to diagnose by the developer.

Programmers using the C/C++ Integrated Development Environment Eclipse CDT should be able to obtain information about selected function overloads and class template specializations, even in the compiler-internal only code resulting from template instantiations.

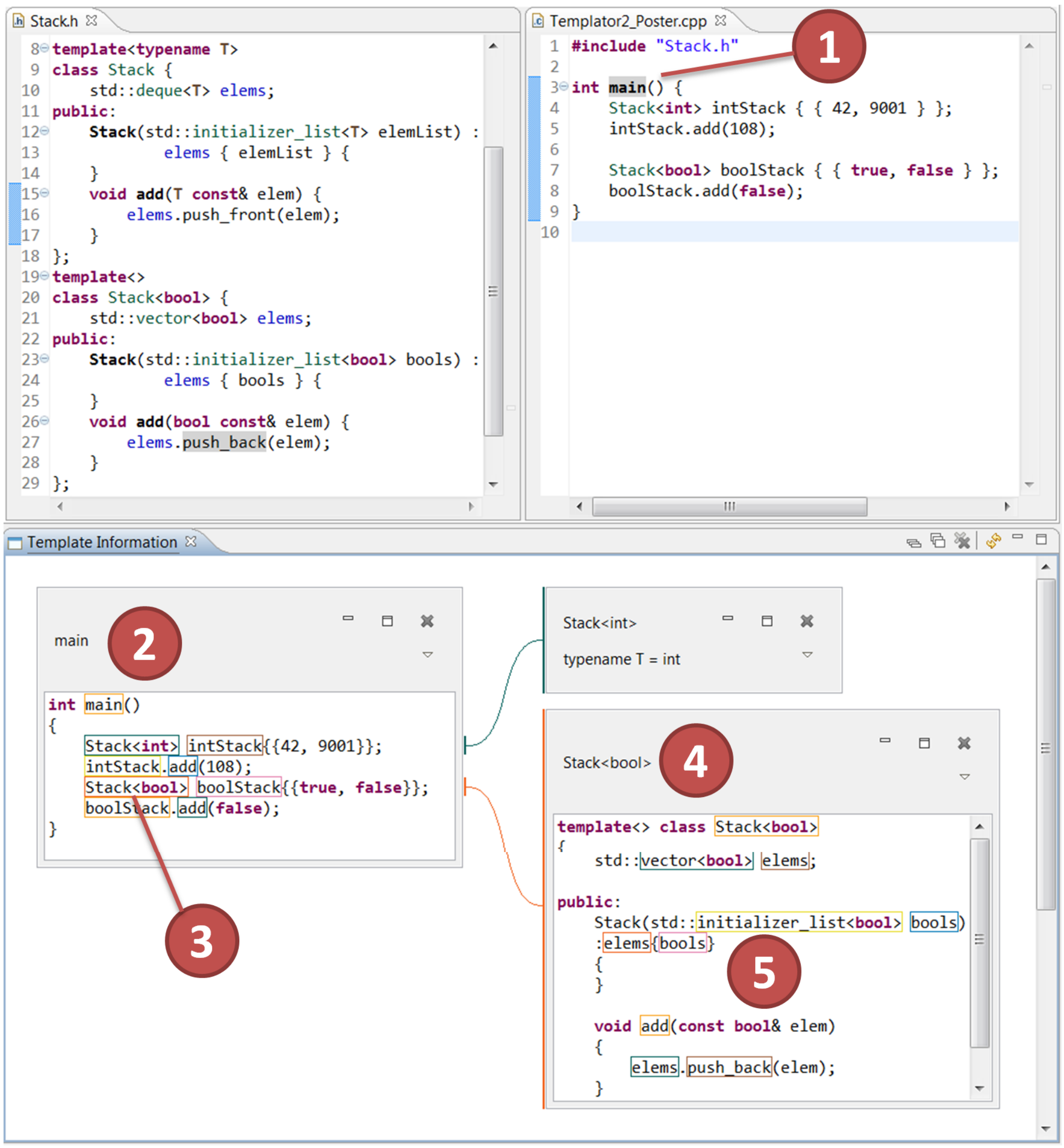
Approach/Technologies

The Templator2 is a CDT plug-in extending the functionality of our previous Templator plug-in resulting from our term project [BS14]. Templator visualized function

template instantiation and function overload selection, where Templator2 now extends that functionality to class template instantiations and specialization selection. This required extensive rework of the user interface to increase its usability and adapt it to the increased functionality. To achieve the template instantiation visualization we analyze the Abstract Syntax Tree (AST) of the template's code for further template instantiations and functions calls. They are then resolved to the finally chosen template definition or function overload. From that information the underlying usually compiler-internal-only class or function definition is formatted and shown to the developer.

Result

The Templator2 plug-in extends CDT with a view of template instantiations. A developer can select a starting point such as a function call or class template use for the visualization in their C++ editor. One is now able to see the chosen function body or class template instantiation. The developer can navigate deeper into nested function calls and template instantiations to get a better understanding of what code the compiler generates internally.



```
template<typename T>
class Stack {
    std::deque<T> elems;
public:
    Stack(std::initializer_list<T> elemList) :
            elems { elemList } {
    }
    void add(T const& elem) {
        elems.push_front(elem);
    }
};
template<>
class Stack<bool> {
    std::vector<bool> elems;
public:
    Stack(std::initializer_list<bool> bools) :
            elems { bools } {
    }
    void add(bool const& elem) {
        elems.push_back(elem);
    }
};
```

```
#include "Stack.h"

int main() {
    Stack<int> intStack { { 42, 9001 } };
    intStack.add(108);

    Stack<bool> boolStack { { true, false } };
    boolStack.add(false);
}
```

main

```
int main()
{
    Stack<int> intStack{{42, 9001}};
    intStack.add(108);
    Stack<bool> boolStack{{true, false}};
    boolStack.add(false);
}
```

Stack<int>

typename T = int

Stack<bool>

```
template<> class Stack<bool>
{
    std::vector<bool> elems;

public:
    Stack(std::initializer_list<bool> bools)
    :elems{bools}
    {
    }

    void add(const bool& elem)
    {
        elems.push_back(elem);
    }
```

1 Selected starting point

2 Starting point is opened

3 Expand a sub statement

4 Selected specialization

5 Further sub statements

The final Templator2 plug-in in Eclipse.

Contents

1 Task Description

This chapter contains the description of our project and our goals for it. The already implemented functionality by Templator plug-in from [BS14] is described first. After that, problems that still exist even with the Templator are described and and finally what our solutions for these problems are.

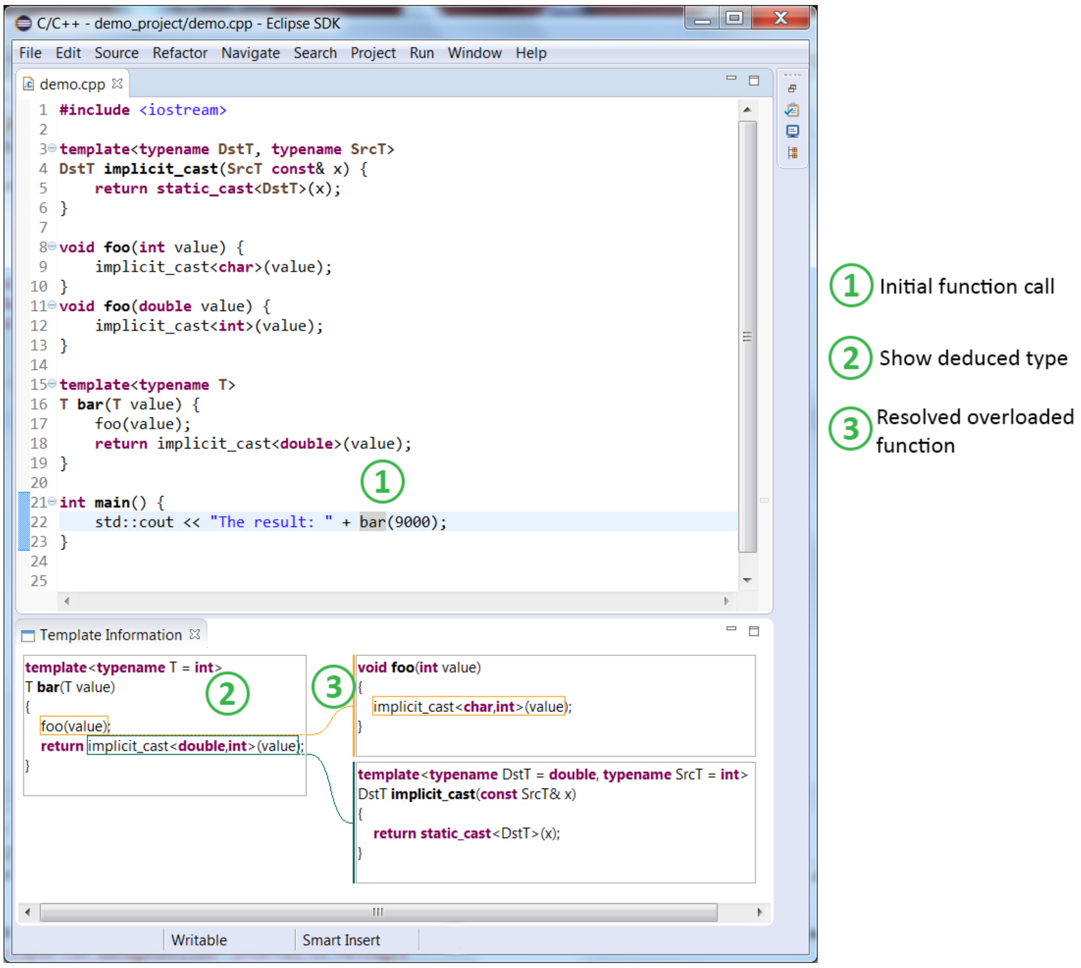
1.1 Previous Work

This bachelor thesis builds on the results of our term project Templator [BS14]. We developed an *Eclipse C/C++ Development Tooling (CDT)* plug-in in the Java programming language to add functionality to visualize template instantiations in C++. While the main goal there was to support the visualization of function template instantiations, the focus in this work are class templates and to improve the existing code.

The first version of the Templator plug-in has the following functionality and is shown in Figure 1.1.

- Resolving a function call to the finally chosen function definition.
- Resolve all further function calls in the chosen function depending on the chosen template arguments in the case of function template calls for an arbitrary number of nesting levels.
- Visualizing the chosen function definitions in a view in a tree-like hierarchy.
  - Show the chosen template arguments.
  - Show the function definition code with syntax highlighting.

– Interaction for the user to see further function calls, show their definition, and jump to the code in his *CDT* editor.


C/C++ - demo_project/demo.cpp - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
demo.cpp
Template Information
Writable
Smart Insert
1 Initial function call
2 Show deduced type
3 Resolved overloaded function

```cpp
#include <iostream>

template<typename DstT, typename SrcT>
DstT implicit_cast(SrcT const& x) {
    return static_cast<DstT>(x);
}

void foo(int value) {
    implicit_cast<char>(value);
}
void foo(double value) {
    implicit_cast<int>(value);
}

template<typename T>
T bar(T value) {
    foo(value);
    return implicit_cast<double>(value);
}

int main() {
    std::cout << "The result: " + bar(9000);
}
```

```cpp
template<typename T = int>
T bar(T value)
{
    foo(value);
    return implicit_cast<double,int>(value);
}
```

```cpp
void foo(int value)
{
    implicit_cast<char,int>(value);
}
```

```cpp
template<typename DstT = double, typename SrcT = int>
DstT implicit_cast(const SrcT& x)
{
    return static_cast<DstT>(x);
}
```

Figure 1.1: The first version of our Templator plug-in from our term thesis ([BS14]).

1.2 Problem

C++ offers more template related functionality than just function templates. Classes can also be parameterized with template parameters. This offers a developer the ability to implement classes where the type is still open and will be determined at compile time. This helps prevent duplicated code and is thus also as often used as function templates.

Class templates are more complicated than function templates because they offer a primary template, partial specializations and explicit specializations. The class template specialization gets chosen based on the specified arguments and works like overloads for functions where the most specialized version is selected by the compiler.

- Explicit (full) specialization

```
#include <string>
template <>
class Stack <std::string> {};
```

Listing 1.1: Fully specialized class template

This template gets chosen when instantiated via

```
Stack<std::string> stringStack {};
```

Listing 1.2: Instantiating a fully specialized class template.

- Partial specialization

```
template <typename T>
class Stack <T*> {};
```

Listing 1.3: Partially specialized class template

This template gets chosen when instantiated via

```
Stack<char const *> charSack {};
```

Listing 1.4: Instantiating a partially specialized class template.

Instead of `char const *` any other pointer type can be used to instantiate this class template.

- Primary template

```
template <typename T>
class Stack {};
```

Listing 1.5: Primary class template

This template gets chosen when none of the two above templates matches, e.g.

```
Stack<int> otherStack {};
```

Listing 1.6: Instantiating the primary class template.

The difficulty arises when working with nested class template instantiations. The finally chosen specialization by the compiler can be hard to find out. Listing 1.7 shows a still rather easy case. `makeStack` on *line 30* shows a function template that creates a `Stack` based on the type of the passed arguments. `Stack` has a primary template (Listing 1.5) defined from *line 6–15* and an explicit class template (Listing 1.1) for `std::string` from *line 17–26*. Both constructors expect an `std::initializer_list`—a sequential list of homogenous values.

```
1  #include <vector>
2  #include <deque>
3  #include <initializer_list>
4  #include <string>
5
6  template<typename T>
7  class Stack {
8     std::vector<T> elems;
9  public:
10    Stack(std::initializer_list<T> elemList) :
11          elems { elemList } {}
12    void add(T elem) {
13       elems.push_back(elem);
14    }
15 };
16
17 template<>
18 class Stack<std::string> {
19    std::deque<std::string> elems;
20 public:
21    Stack(std::initializer_list<std::string> strings) :
22          elems { strings } {}
23    void add(std::string elem) {
24       elems.push_front(elem);
25    }
26 };
27
28 template<typename F>
29 Stack<F> makeStack(std::initializer_list<F> elemList) {
30    return Stack<F> { elemList };
31 }
32
33 int main() {
34    auto vec = makeStack( { 4, 8, 15, 16, 23, 42 });
35    vec.add(108);
36    auto deck = makeStack( { std::string {"Hello"} });
37    deck.add(std::string{"World"});
38 }
```

Listing 1.7: `makeStack` instantiates a `Stack` based on the passed argument type.

`makeStack` in Listing 1.7 is called twice. Since it is a function template it automatically deduces the type of the passed argument ([BS14, 2.1.1, p. 13]) which the template parameter `F` will be substituted with.

- *line 34*: `makeStack` is called with a list of numbers and `F` is automatically deduced to `int`. So the compiler creates a statement `return Stack<int> elemList` where `F` was replaced by `int`. This statement will instantiate the primary `Stack` template (*line 6–15)*). Further `std::vector<T>` with `T=int` is instantiated and `add` will finally call `std::vector<int>`s member function `push_back`.

- *line 36*: `makeStack` is called with `std::string`s, thus the compiler will select the explicit class template `Stack<std::string>` (*line 17–26*). Every `deck.add(...)` call will result in `std::deque<std::string>`s `push_front()` to be called.

After these explanations and a bit C++ knowledge it is clear what `vec.add(108)` on *line 35* and `deck.add` on *line 37* will call. But this code could be nested inside other templates and the programmer would have problems finding out which `Stack` will be instantiated and what `add` finally calls.

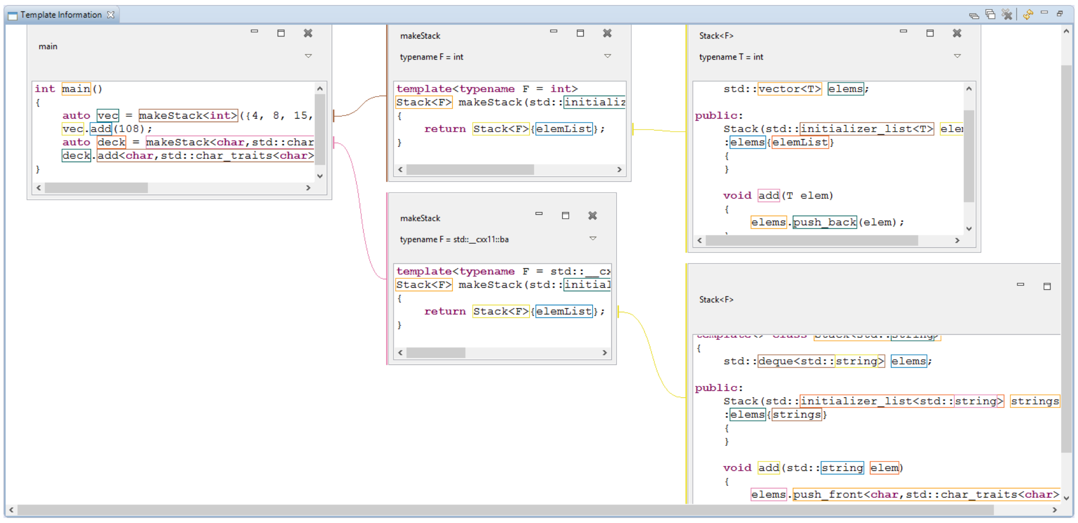
Our existing Templator plug-in only works for function templates. So the call `makeStack` could be resolved and `F` deduced but not which template will be instantiated. *CDT* is only able to resolve class templates for the first hierarchy level like function templates. Here is where the new Templator2 plug-in will help *CDT* users.

1.3 Solution

Our Templator2 plug-in adds a new view to *CDT* to mainly visualize class template instantiations and class template member function calls. If a *CDT* user wants to know which `Stack` will be instantiated in `main` in Listing 1.7, he or she can visualize the `main` definition with any further function calls and template instantiations. Figure 1.2 shows this visualization in the "Template Information View". There are two paths to see.

1. `auto vec = makeStack({4,8,14,16,23,42})` statement with the further instantiation of the primary `Stack` template in the upper part.

2. `auto deck = makeStack( { std::string {"Hello"} });` statement with the further instantiation of the specialized `Stack<std::string>` inside `makeStack` in the lower part.



```
int main()
{
    auto vec = makeStack<int>({4, 8, 15,
    vec.add(108);
    auto deck = makeStack<char,std::char
    deck.add<char,std::char_traits<char>
}
```

```
template<typename F = int>
Stack<F> makeStack(std::initializ
{
    return Stack<F>{elemList};
}
```

```
template<typename F = std::__cx
Stack<F> makeStack(std::initial
{
    return Stack<F>{elemList};
}
```

```
    std::vector<T> elems;

public:
    Stack(std::initializer_list<T> ele
    :elems{elemList}
    {
    }

    void add(T elem)
    {
        elems.push_back(elem);
```

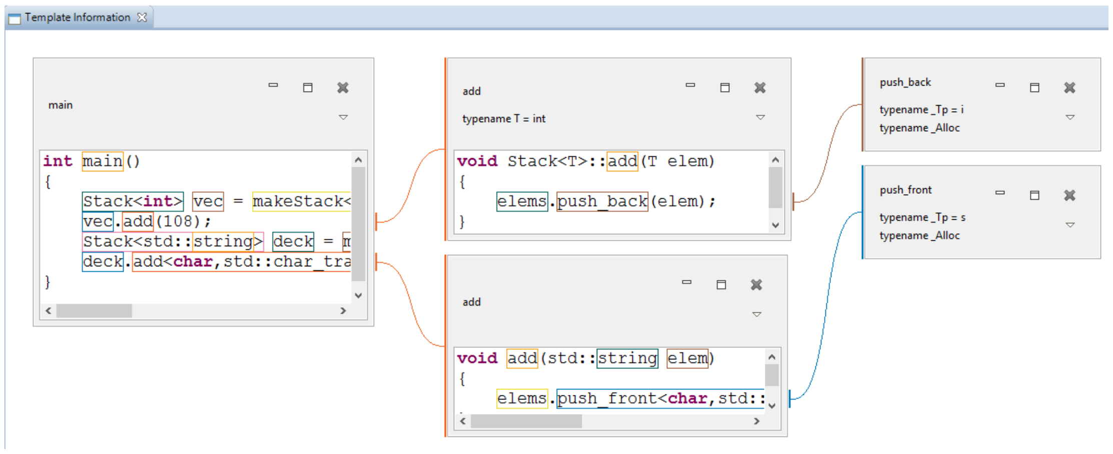
```
{
    std::deque<std::string> elems;

public:
    Stack(std::initializer_list<std::string> strings
    :elems{strings}
    {
    }

    void add(std::string elem)
    {
        elems.push_front<char,std::char_traits<char>
```

Figure 1.2: Listing 1.7 class template instantiations visualized in the new Templator2 plug-in view.

Our plug-in also assists the user by showing which member function is called with the chosen template arguments from the surrounding class template. Figure 1.3 shows both calls to `add` from Listing 1.7. This also shows two known issues. First, our Templator2 plug-in needs a template-id like `Stack<int>` to deduce the type of `vec` and `deck`. It does not work with `auto`. Secondly, the `add` for `Stack<std::string>` does not contain the class name in the function declarator. The lower `void add(std::string elem)` in Figure 1.3 should also contain the class name where it is defined—`void Stack<std::string>::add(std::string elem)`.

CDT users now have the possibility to visualize many class template instantiations, (overloaded) member function calls and with the already done work in our term thesis ([BS14]) also function template instantiations. Working with templates in C++ is now easier and unintentionally executed code may be reduced if the programmer uses our Templator2 plug-in.

The target audience is every C++ programmer writing template code with *Eclipse CDT*. The plug-in can also be used as educational tool in lectures about templates. Future HSR students may see the Templator2 in use or use the plug-in themself to better understand what the compiler instantiates and the compiler error messages.



```
int main()
{
    Stack<int> vec = makeStack<
    vec.add(108);
    Stack<std::string> deck = m
    deck.add<char,std::char_tra
}
```

```
void Stack<T>::add(T elem)
{
    elems.push_back(elem);
}
```

```
void add(std::string elem)
{
    elems.push_front<char,std::
```

Figure 1.3: Listing 1.7 member function calls visualized in the new Templator2 plug-in view.

1.4 Our Goal

The goal of our project is to extend our existing Templator plug-in to also show information about class template instantiations and their member function calls. Especially for nested instantiations in a function template or class template body if the instantiation depends on a template argument.

If there is enough time, we plan to implement

- That the user can select a template-id, a variable of class template instance type or a class template member function call to visualize the template instance in our *Eclipse* view.
- Integrate the new functionality into the existing plug-in so it works also for nested class template instantiations inside function templates and vice versa.
- Show the user possible other specializations beside the chosen one in the existing or even a new view.
- A flow chart like process that shows why a specific class template specializations gets chosen instead of another that could be used

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 by a following work.

1.4.1 Focus for This Project

For this work our goal is to implement the above mentioned objectives partially. We will not have enough time for everything but the goals provide ideas for the reader of what still could be implemented in the future.

The main focus is on the two first points to support class templates without additional features.

1.4.2 Postponed Requirements and Ideas

While the Templator2 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. The Templator2 cannot yet handle alias templates or qualified type aliases. Further, there is Substitution Failure Is Not An Error (SFINAE) which is something the compiler does but not yet implemented by *CDT*. Substitution Failure Is Not An Error (SFINAE) should now be easier to implement with the existing Templator2 plug-in since nested template instantiations are supported and members in class templates can also be found. The last point is variadic templates that are not yet supported.

1.5 Time Management

Our project started on the 16th of February, 2015. It takes 17 weeks and ends on June the 12th, 2015 at 12:00 noon which is when the final release has to be submitted.

2 Analysis

This chapter gives an overview about some of the possiblities a C++ developer has to use class templates and instantiate them. It is not complete and only covers cases we discovered and are relevant for the Templator2 plug-in. For each example, the created Abstract Syntax Tree (AST) for the code is shown and described what *CDT* is already able to resolve correctly. This helps to better understand some of the example in the following chapters. At the end of this chapter is explained what the Templator2 plug-in is able to find and resolve and how the existing Templator from [BS14] had to be changed to support these new features.

An AST in *CDT* is a representation of C++ code and allows for easy analysis of code. The definition of an AST and how it works in *CDT* is described in [BS14, 3.3, p. 25]. All examples are defined in a source and not a header file where class templates are normally declared. This is to hold the code more readable and the created AST smaller.

The AST is needed in the Templator2 plug-in to find relevant statements. Template argument dependent statements and function calls should be shown to the user so he can see what code is finally executed. All operations to find those relevant statements and resolved them are done based on the created AST by *CDT* for the user's C++ code. This is why a deep understanding of nodes and their relation is needed. This chapter is hard to difficult to read just at the start of this thesis because AST nodes are already used but the chapter after this one builds on the knowledge from this chapter.

2.1 Class Template Instantiations in C++

When using a class template the compiler replaces its template parameters with the chosen concrete types—called template arguments. This replacing process is called instantiation and results in an instance of a template [VJ03, p. 11]. Such a template instance is only valid when all template arguments have a value. There exist 3 ways to specify a template argument.

- Specify when using a class template.
- A template parameter has a default value.
- A partial specialization specifies a part of the template parameters with template arguments.

First, template-ids are described since they are necessary to instantiate class templates. After, these 3 described variants from above are explained in detail with the created AST for the sample code.

2.1.1 Template-Id & Nested Template-Ids

A template-id is the class name of a class template with the used set of template arguments listed in angle brackets `<>`. They are needed for an instantiation of a class template since they specifiy the template arguments. One exception exists that is later explained.

Listing 2.2 shows an example of a simple template-id on *line 5*.

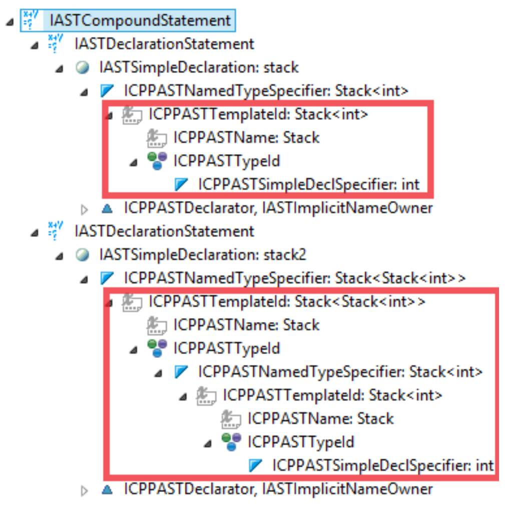
```
1 template <typename T>
2 struct Stack {};
3
4 int main() {
5    Stack<int> stack {};
6    Stack<Stack<int>> stack2 {};
7 }
```

Listing 2.1: Usage of a template-id `Stack<int>` that tells the compiler to instantiate `Stack` with `T=int`. `Stack<Stack<int»` is a nested template-id and means that `T` will be `Stack<int>`.

It is also possible that template-ids can occur as template arguments in other template-ids. This can be nested up to an arbitary level. Each template-id must be found independently. This means the Templator2 plug-in must be able to find all those template-ids and maybe instantiate class template recursively to determine a template argument. Listing 2.2 shows also the usage of a nested template-id on *line 6* and Figure 2.1 shows the corresponding AST for the whole example code. To see is that the first `ICPPASTTemplateId` from *line 5* just contains an `ICPPASTSimpleDeclSpecifier` for `int` while the second contains again an `ICPPASTTemplateId` with an `int`.

```
1 template <typename T>
2 struct Stack {};
3
4 int main() {
5     Stack<int> stack {};
6     Stack<Stack<int>> stack2 {};
7 }
```

Listing 2.2: Usage of a template-id `Stack<int>` that tells the compiler to instantiate `Stack` with `T=int`. `Stack<Stack<int»` is a nested template-id and means that `T` will be `Stack<int>`.



```
IASTCompoundStatement
  IASTDeclarationStatement
    IASTSimpleDeclaration: stack
      ICPPASTNamedTypeSpecifier: Stack<int>
        ICPPASTTemplateId: Stack<int>
          ICPPASTName: Stack
          ICPPASTTypeId
            ICPPASTSimpleDeclSpecifier: int
      ICPPASTDeclarator, IASTImplicitNameOwner
  IASTDeclarationStatement
    IASTSimpleDeclaration: stack2
      ICPPASTNamedTypeSpecifier: Stack<Stack<int>>
        ICPPASTTemplateId: Stack<Stack<int>>
          ICPPASTName: Stack
          ICPPASTTypeId
            ICPPASTNamedTypeSpecifier: Stack<int>
              ICPPASTTemplateId: Stack<int>
                ICPPASTName: Stack
                ICPPASTTypeId
                  ICPPASTSimpleDeclSpecifier: int
      ICPPASTDeclarator, IASTImplicitNameOwner
```

Figure 2.1: The AST for the two type-specifiers from Listing 2.2. Marked are the two `ICPPASTTemplateIds` and their sub nodes.

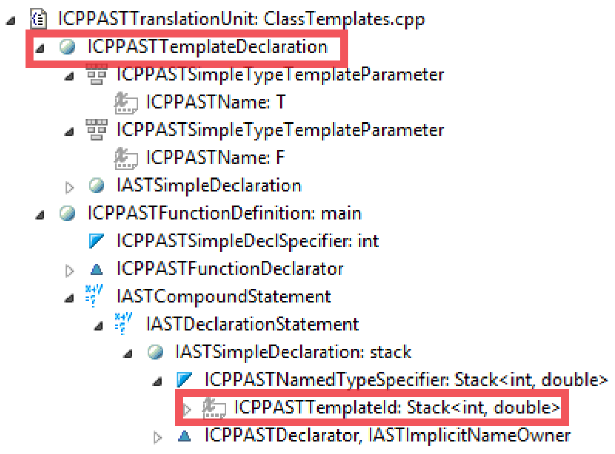
2.1.2 Explicitly Specifiy a Template Argument on Usage

When using a class template it will be instantiated automatically by the compiler. A developer does not need to start the instantiation process explicitly. The template arguments can be specified inside angle brackets `<>` and separated by commas after the class name as shown in Listing 2.3. `Stack<int, double>` instantiates `Stack` with `T=int` and `F=double`. Meaning all occurences of `T` and `F` will be replaced with the chosen argument by the compiler at compile-time.

Figure 2.2 shows the created AST from this code. The definition of the class template is inside an `IPPClassTemplateDeclaration` and its sub nodes.

```cpp
1 template <typename T, typename F>
2 class Stack {};
3 int main() {
4     Stack<int, double> stack {};
5 }
```

Listing 2.3: Primary class template with two template parameters. The template `Stack` is instantiated with `int` and `double` on *5*.



```
ICPPASTTranslationUnit: ClassTemplates.cpp
  ICPPASTTemplateDeclaration
    ICPPASTSimpleTypeTemplateParameter
      ICPPASTName: T
    ICPPASTSimpleTypeTemplateParameter
      ICPPASTName: F
    IASTSimpleDeclaration
  ICPPASTFunctionDefinition: main
    ICPPASTSimpleDeclSpecifier: int
    ICPPASTFunctionDeclarator
    IASTCompoundStatement
      IASTDeclarationStatement
        IASTSimpleDeclaration: stack
          ICPPASTNamedTypeSpecifier: Stack<int, double>
            ICPPASTTemplateId: Stack<int, double>
          ICPPASTDeclarator, IASTImplicitNameOwner
```

Figure 2.2: The AST for the primary class template definition from Listing 2.3. Marked are the node for the template definition and the template-id where the template arguments `int` and `double` are specified.

2.1.3 Default Template Arguments

A default template argument can be specified right in the template parameter declaration. This template argument is used for instantiation if no one is specified when using the class template. This is used by the `std::vector` for the second template parameter where a allocator could be specified. In most cases the C++ developer does not care about the allocator so a `std::allocator` is used as default template argument.

```
template<typename _Tp,
typename _Alloc = std::allocator<_Tp> > class vector :
   protected _Vector_base<_Tp, _Alloc> {/* ... */};
```

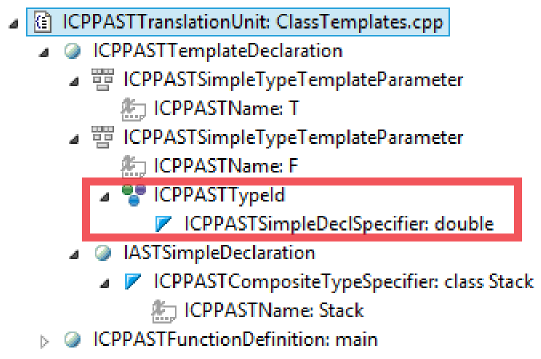
Listing 2.4: `std::vector` primary template definition with default template argument

A simpler example is shown in Listing 2.5. Again for our `Stack` which still declares two template parameters `T` and `F` but when using this class template one only needs to specify `T` since still are template arguments have a value.

```
1 template <typename T, typename F=double>
2 class Stack {};
3 
4 int main() {
5    Stack<int> stack {};
6 }
```

Listing 2.5: Primary class template with two template parameters that is instantiated with `int` and `double` on *5*.

Figure 2.3 shows again the created AST for Listing 2.5 but with the `definition` collapsed because it is almost the same as in Figure 2.2. New is the `ICPPASTTypeId` for the default argument.



```
ICPPASTTranslationUnit: ClassTemplates.cpp
  ICPPASTTemplateDeclaration
    ICPPASTSimpleTypeTemplateParameter
      ICPPASTName: T
    ICPPASTSimpleTypeTemplateParameter
      ICPPASTName: F
      ICPPASTTypeId
        ICPPASTSimpleDeclSpecifier: double
    IASTSimpleDeclaration
      ICPPASTCompositeTypeSpecifier: class Stack
        ICPPASTName: Stack
  ICPPASTFunctionDefinition: main
```

Figure 2.3: The AST for the primary class template definition from Listing 2.5. Marked is the node with the type for the default template argument.

Requirement for a Template-Id

A template-id is always needed for declaring a class template type , except for one special case which is explained after this example that does not compile. Listing 2.6 shows a case where both template parameters have a default type. Trying to instantiate it with using `Stack` creates a compiler error and `<>` are needed even though they are empty.

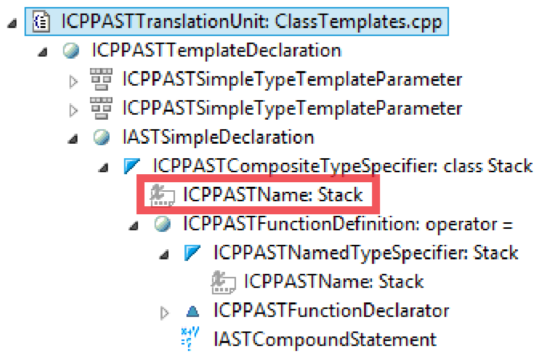
```
1 template <typename T=int, typename F=double>
2 class Stack {};
3
4 int main() {
5    Stack stack{}; // error but Stack<> works
6 }
```

Listing 2.6: Primary class template with two default arguments so all template arguments are known. But angle brackets `<>` are still needed for the instantiation on *5*.

The usage of a class template name without angle brackets `<>` is only allowed inside a class template declaration for the same type. This can be used for example for return types for member functions as shown in Listing 2.7. Using only `Stack` is equivalent to `Stack<T,F>`, meaning it uses the current context for `T` and `F`. The created AST is shown in Figure 2.4.

```
template <typename T, typename F>
class Stack {
  Stack& operator=(const Stack& other) {}
};
```

Listing 2.7: The type-specifier for the return type is allowed without angle brackets. The member function will return and accept as parameter argument a template instance equal to `Stack<T, F>`.



```
ICPPASTTranslationUnit: ClassTemplates.cpp
  ICPPASTTemplateDeclaration
    ICPPASTSimpleTypeTemplateParameter
    ICPPASTSimpleTypeTemplateParameter
    IASTSimpleDeclaration
      ICPPASTCompositeTypeSpecifier: class Stack
        ICPPASTName: Stack
        ICPPASTFunctionDefinition: operator =
          ICPPASTNamedTypeSpecifier: Stack
            ICPPASTName: Stack
          ICPPASTFunctionDeclarator
          IASTCompoundStatement
```

Figure 2.4: The AST for Listing 2.7. Marked is the node with the type-specifier for the return type which is an `IASTName` and not an `ICPPASTTemplateId` in this special case.

2.1.4 Partial Specialization

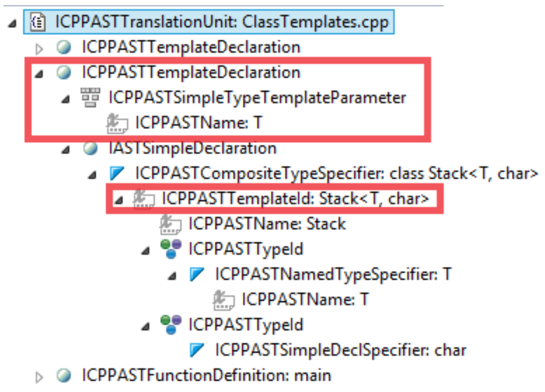
When using a template they still need to be specified. But a partial class template can explicitly define them so this specialization is chosen when the template is instantiated with this argument. Some template parameters must still be defined when using them. For the example in Listing 2.8, the template-id on *line 8* is still a template-id with two arguments.

```
1 template <typename T, typename F>
2 class Stack {};
3 template<typename T>
4 class Stack<T, char> {};
```

```
int main() {
  Stack<int, char> stack{};
}
```

Listing 2.8: Primary class template with two template parameters that is instantiated with `int` and `double` on *5*.

Figure 2.5 shows the created AST for Listing 2.8.



```
◢ ICPPASTTranslationUnit: ClassTemplates.cpp
  ▷ ICPPASTTemplateDeclaration
  ◢ ICPPASTTemplateDeclaration
    ◢ ICPPASTSimpleTypeTemplateParameter
        ICPPASTName: T
    ◢ IASTSimpleDeclaration
      ◢ ICPPASTCompositeTypeSpecifier: class Stack<T, char>
        ◢ ICPPASTTemplateId: Stack<T, char>
            ICPPASTName: Stack
          ◢ ICPPASTTypeId
            ◢ ICPPASTNamedTypeSpecifier: T
                ICPPASTName: T
          ◢ ICPPASTTypeId
              ICPPASTSimpleDeclSpecifier: char
  ▷ ICPPASTFunctionDefinition: main
```

Figure 2.5: The AST for the partial class template definition from Listing 2.8. Marked are the `ICPPASTTemplateDeclaration` that now only contains one template parameter declaration and the class name is now an `ICPPASTTemplateId`.

2.1.5 Specialization Selection and Class Template Instantiation in CDT

The selection of the specialization and the instantiation is already handled by *CDT* for the first hierarchy level. Meaning if templates are not nested. For any further nesting level, Templator2 needs to provide *CDT* with the chosen template. With it, *CDT* can still select the correct specialization and instantiate a template. Letting *CDT* know about the template arguments is the difficult part and is later explained in Subsection 4.4.3 (Class Template Instantiation on page 68).

Members are another language feature for class templates and can be declared or defined inside class templates as for normal classes. Features like member functions, member templates, and type aliases are explained in the next section.

2.2 Class Template Members

Class template members can be:

- Member functions that may use the template arguments from the class template or not.
- Member templates that introduce new template arguments additionaly to the ones from the class template. Like a function template inside a class template.
- Other members that define new template parameters like other class templates or even a normal class, alias templates, or variable templates. They are not described here as they are not supported by Templator2.
- Member variables and type aliases to ease the use inside the definition and for users of this class.

Members do not have to use template arguments and can be completely independent of them. A class template can contain other members like a normal class. This section lists possible members we found with an example and the created AST for the C++ code. This section describes some of the above listed possible members with their AST representation.

Just to note: The used example class `Stack` will from now on only have one template parameter `T` because the second `F` is not needed anymore to demonstrate something. The omission of `F` does not have another reason.

2.2.1 Member Functions

This subsection contains the description of how member functions can be augmented with template arguments. In general, a member function call is appended with `.` or `->` to a variable of a class template instance type. Also a member function inherits the template arguments from the class template. They can be used as parameters and return types for member functions.

For member functions, the template arguments must not be repeated before the definition as long as the function is defined inline. Even if the member function is defined in a source file the template arguments only have to be written for the template class.

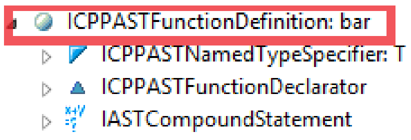
Listing 2.9 shows a member function `bar` that is called twice in the `main`. Once with the `.` syntax and the second time on a pointer with `->` that first dereferences the pointer and then calls `bar`.

```cpp
template <typename T>
struct Stack {
    T bar(T value) {
      return value;
    }
};
int main() {
   Stack<int> foo {};
   foo.bar(42);

   Stack<int>* fooPrt = &foo;
   fooPrt->bar(42);
}
```

Listing 2.9: Simple class member function call.

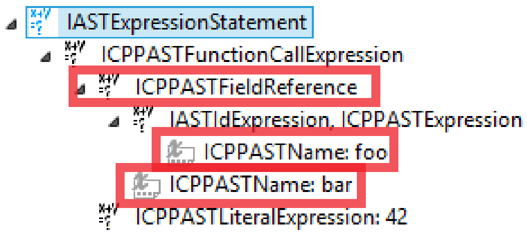
Figure 2.6 shows the AST for the member function definition and Figure 2.7 the AST for the first call `foo.bar(42)`. The one for the call with `->` on the pointer is the same disregarding the call operator difference for `.` and `->`. Interesting is, how the `foo.bar` is modeled. The member function name `bar` has an `ICPPASTFieldReference` as parent. From there the variable class template instance type `foo` can be extracted. From there on, one will be able to extract the type-specifier for `foo` which is `Stack<int>`.

CDT is able to resolve member functions if their owner type is known. This is the template instance the member function is called on (`foo` in `foo.bar()`). Meaning if the owner depends on a template argument then *CDT* cannot resolve the call. Also if a member function is overloaded *CDT* cannot find the correct function overload after the first hierarchy level.



```
ICPPASTFunctionDefinition: bar
    ICPPASTNamedTypeSpecifier: T
    ICPPASTFunctionDeclarator
    IASTCompoundStatement
```

Figure 2.6: The AST for the member function definition `bar` in Listing 2.9. Marked is the node for the member function definition, the same as for a normal function definition outside of a class.



```
IASTExpressionStatement
    ICPPASTFunctionCallExpression
        ICPPASTFieldReference
            IASTIdExpression, ICPPASTExpression
                ICPPASTName: foo
            ICPPASTName: bar
        ICPPASTLiteralExpression: 42
```

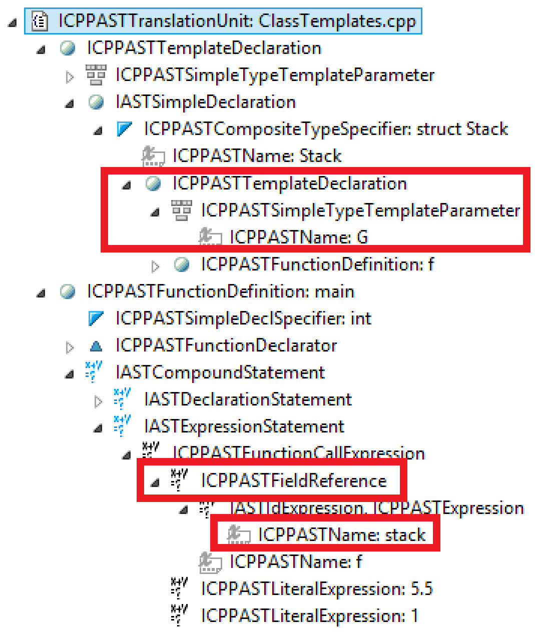
Figure 2.7: The AST for the member function call `foo.bar(42)` in Listing 2.9. Marked are the owner node `foo` and the member function name `foo`. Both are sub nodes of `ICPPASTFieldReference`.

2.2.2 Member Templates

Member Templates are like function templates inside class templates. They can introduce additionaly template parameters to the ones from the class template. The additional arguments can be explicitly specified, implicitly deduced from the calls site or given as default template arguments. They work the same way as template arguments for function templates which are described in [BS14, 2.1, p. 13].

Listing 2.10 shows a defined member template `f`. The member template uses `T` from `Stack` and introduces the new template parameter `G`. `G` is automatically and exclusively deduced for the shown call. Meaning another call `stack.f(false, 1);` will result in `G=bool`.

```
template <typename T>
struct Stack {
   template<typename G>
    void f(G param, T value) {
```



```
ICPPASTTranslationUnit: ClassTemplates.cpp
  ICPPASTTemplateDeclaration
    ICPPASTSimpleTypeTemplateParameter
    IASTSimpleDeclaration
      ICPPASTCompositeTypeSpecifier: struct Stack
        ICPPASTName: Stack
        ICPPASTTemplateDeclaration
          ICPPASTSimpleTypeTemplateParameter
            ICPPASTName: G
          ICPPASTFunctionDefinition: f
  ICPPASTFunctionDefinition: main
    ICPPASTSimpleDeclSpecifier: int
    ICPPASTFunctionDeclarator
    IASTCompoundStatement
      IASTDeclarationStatement
      IASTExpressionStatement
        ICPPASTFunctionCallExpression
          ICPPASTFieldReference
            IASTIdExpression, ICPPASTExpression
              ICPPASTName: stack
            ICPPASTName: f
          ICPPASTLiteralExpression: 5.5
          ICPPASTLiteralExpression: 1
```

Figure 2.8: The AST for the member template definition and call from Listing 2.10. Marked are the `ICPPASTTemplateDeclaration` with the new template parameter `G` and the call again with the `ICPPASTFieldReference`.

```cpp
    }
};

int main() {
    Stack<int> stack {};
    stack.f(5.5, 1);
}
```

Listing 2.10: Member template call. `T` is still `int` and `G` is automatically deduced to `double`.

Figure 2.8 shows the created AST. Instead of `ICPPASTFunctionDefinition` for normal member functions, member templates are defined by an `ICPPASTTemplateDeclaration`. The `IASTName` for the function name `f` again has a parent of type `ICPPASTFieldReference`.

CDT can also resolve member templates if the owner is known and the passed function arguments does not depend on template arguments.

2.2.3 Member Variables and Type Aliases

A class template can declare and define variables like a normal class. The only difference is, that they can additionally depend on template arguments.

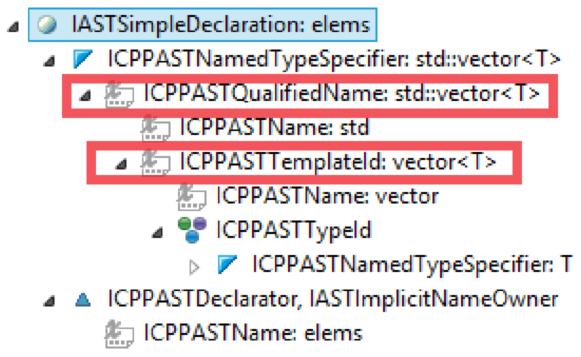
Listing 2.11 shows the class template `Stack` with its member variable `elems`. The variable's type is `std::vector<T>`. This means the instantiation for `Stack<int>` will also trigger the instantiation of `vector<int>`.

The created AST shown in Figure 2.9 introduces a new node implementing `IASTNode`—an `ICPPASTQualifiedName`. A qualified name is separated by `::` into different segments. Here it is used to tell the compiler that the definition of `vector` is in the namespace `std`. This builds the first segment of the name. The second segment is again an `ICPPASTTemplateId` with a template argument that depends on the chosen argument for `T`.

```cpp
#include <vector>
template<typename T>
class Stack {
    std::vector<T> elems;
};
int main() {
    Stack<int> mystack {};
}
```

Listing 2.11: Class template with a member variable that depends on the template argument for `T`.

One does not always want to write `std::vector<T>` when using this type. For this reason C++ offers the creation of aliases via the `using` and `typedef` keyword. Listing 2.12 shows both of them in action. Both `using` and `typedef` are equivalent when defining an alias and running the program. But as seen in Figure 2.10 they create different AST nodes. A `using` statement creates an `ICPPASTAliasDeclaration` while a `typedef` creates an `IASTSimpleDeclaration`. But they contain the same sub node for the type-specifier which is an `ICPPASTNamedTypeSpecifier`. This means when retrieving the type-specifier for the aliased type `std::vector<T>` they need to be distinguished.



```
IASTSimpleDeclaration: elems
  ICPPASTNamedTypeSpecifier: std::vector<T>
    ICPPASTQualifiedName: std::vector<T>
      ICPPASTName: std
      ICPPASTTemplateId: vector<T>
        ICPPASTName: vector
        ICPPASTTypeId
          ICPPASTNamedTypeSpecifier: T
  ICPPASTDeclarator, IASTImplicitNameOwner
    ICPPASTName: elems
```

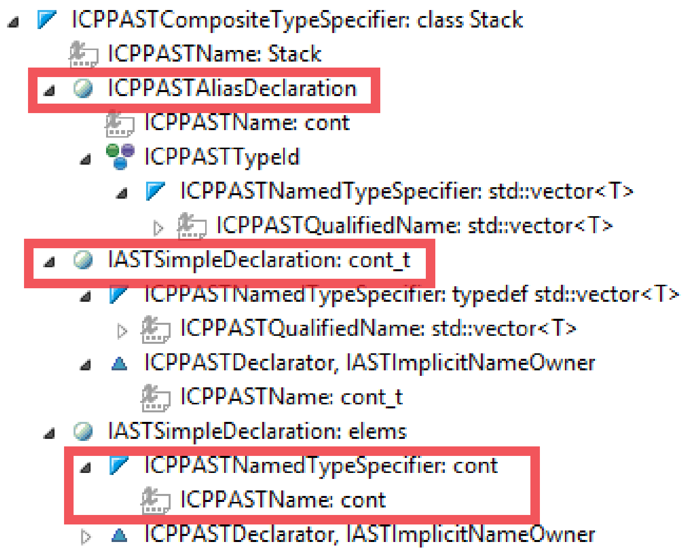
Figure 2.9: The AST for member variable `elems` with type `std::vector<T>` from Listing 2.11. Marked are the type-specifier in form of an `ICPPASTQualifiedName` and the containing `ICPPASTTemplateId` which depends on `T`.

```cpp
#include <vector>
template<typename T>
class Stack {
   using cont=std::vector<T>;
   typedef std::vector<T> cont_t;
   cont elems; // cont_t elems is equivalent
};

int main() {
   Stack<int> mystack {};
}
```

Listing 2.12: Class template with a type alias for `std::vector<T>` that is used for the variable declaration.

Type aliases can make the code shorter and more readable. They are especially useful if further members from `vector` are often used and can be aliased. A constructed example in Listing 2.13 shows what is also possible with type aliases. This is more interesting since it shows the complexity one can use his type aliases with to produce code that may be hard to read but generally shorter.

Determining the type for `elems` is now more difficult and takes the following steps for the compiler.



```
ICPPASTCompositeTypeSpecifier: class Stack
  ICPPASTName: Stack
  ICPPASTAliasDeclaration
    ICPPASTName: cont
    ICPPASTTypeId
      ICPPASTNamedTypeSpecifier: std::vector<T>
        ICPPASTQualifiedName: std::vector<T>
  IASTSimpleDeclaration: cont_t
    ICPPASTNamedTypeSpecifier: typedef std::vector<T>
      ICPPASTQualifiedName: std::vector<T>
    ICPPASTDeclarator, IASTImplicitNameOwner
      ICPPASTName: cont_t
  IASTSimpleDeclaration: elems
    ICPPASTNamedTypeSpecifier: cont
      ICPPASTName: cont
    ICPPASTDeclarator, IASTImplicitNameOwner
```

Figure 2.10: The AST for Listing 2.12. `using` and `typedef` statements create different nodes. The newly created alias `cont` is used as type-specifier for `elems`.

1. Get the type-specifier for `elems` which is `cont`.
2. `cont` itself is an alias for `_b::const`.
3. Get the type-specifier for `_b` which is `Base<T>`.
4. Instantiate `Base<T>` where `T` is the template argument `Stack` has been instantiated which is `int`.
5. Now that `_b` is known and instantiated the compiler can search for a member named `cont` in the class template definition for `Base`.
6. The aliased type for `Base::cont` is `std::vector<T>` where `T` is again `int` because the compiler instantiated `Base<int>`.

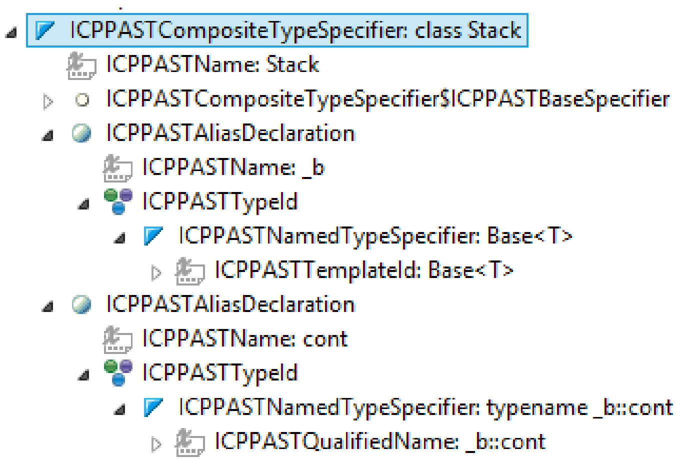
```cpp
#include <vector>

template<typename T>
struct Base { using cont=std::vector<T>; };
```

```

template<typename T>
class Stack : Base<T>{
   using _b = Base<T>;
   using cont = typename _b::cont;
   cont elems;
};

int main() {
   Stack<int> mystack {};
}
```

Listing 2.13: Purposely complex example with type aliases to show what is possible.

This is often used in the C++ standard library and creates a challenge for *CDT* and the Templator2 plug-in. Having a look at the relevant created AST nodes in Figure 2.11 reveals that again `ICPPASTQualifiedName`s were created. `ICPPASTQualifiedName` offers `getLastName()` which returns the `IASTName` for the last segment. For `_b::cont` this is `cont`. *CDT* is already able to resolve the correct type for `cont` if no segment depends on `T`—which is not the case here. This means the Templator2 plug-in needs to perform the above described steps to get the correct type for `elems`. This theoretically works endless, so some kind of recursion is needed to resolve this. This is described in Section 4.4.3 (Deduce the Type for Any Template Argument on page 69). Compilers have a limit when they stop instantiating nested templates.

C++ offers a wide range of language features for class templates. Based on the done analyis and the created ASTs we will try to implement as many described features to support class templates in the Templator2 plug-in so the user can visualize them. *CDT* is only able to instantiate them on the first hierarchy level if a statement does not depend on a template argument. This is where the Templator2 will help the user.



```
ICPPASTCompositeTypeSpecifier: class Stack
    ICPPASTName: Stack
    ICPPASTCompositeTypeSpecifier$ICPPASTBaseSpecifier
    ICPPASTAliasDeclaration
        ICPPASTName: _b
        ICPPASTTypeId
            ICPPASTNamedTypeSpecifier: Base<T>
                ICPPASTTemplateId: Base<T>
    ICPPASTAliasDeclaration
        ICPPASTName: cont
        ICPPASTTypeId
            ICPPASTNamedTypeSpecifier: typename _b::cont
                ICPPASTQualifiedName: _b::cont
```

Figure 2.11: The AST for Listing 2.12. `using` and `typedef` statements create different nodes. The newly created alias `cont` is used as type-specifier for `elems`.

2.3 Supported Features in Our Templator Plug-in

The Templator2 plug-in supports:

- Instantiations of class templates that depend on template arguments.
- Correct specialization selection for any nesting level.
- Selection of correct member function overload.
- Argument deduction and instantiation for member templates.
- In some cases the resolving of aliased types.

Every point is only valid if the template argument does not come from a nested class template, alias template or variable template. They are not supported yet respectively only work in rare cases.

This chapter gave us the insight about the architecture changes we needed to adapt from our term thesis' class hierarchy and process. The next section describes our process of developing an architecture that supports both function templates and class templates.

2.4 Supporting Function Templates and Class Templates

With the architecture from our term thesis [BS14] we had various difficulties implementing the new functionallity to support class templates with the Templator2 plug-in. We explain the problems to support both of them with the existing architecture and what we finally implemented.

2.4.1 The Existing Architecture

The existing process was to use a visitor to find all `ICPPASTFunctionCallExpressions` and then store them in our own class `FunctionCall`. This class encapsulated the data and functionality that represented a single function call. `FunctionCall` then had a list of `FunctionCalls` that reperesented the found sub calls that were found in the parent's function definition.

The sub calls were strictly of type `FunctionCall`. This way we could not have class template instantiations in function templates and vice versa. Our first idea was to implement both worlds parallel which meant to write an own class `ClassTemplate` that could have `ClassTemplates` as children. The plan was to introduce an `abstract` base class so `FunctionCall` definitions and `ClassTemplate` could be mixed. The base class then could have had children of itself. However, when we started writing methods to search for further possible children we generated many duplicated lines of code. Because the searching for functions and class templates in a definition was different. Searching and accepting a found statement was also different for the first hierarchy level and for all others. For the first level we want a statement that does not depend on an unknown template argument but for later those statements should be found. This was the point we noticed that it should be the same code for both cases

At this point we decided to write a new class that could support normal functions, function templates, and class templates and especially search for every possible sub statement with the same algorithm.

2.4.2 The New Architecture

We started with a simple class that only consisted of a list of children of its own class type. The key to make this work the same in the whole project was to make a factory method. This factory then can be called from outside for the first level of deduction and from the inside for all following levels. This way the deduction code was the same for all cases.

Now came the problem to make this code working for all cases. We searched for `FunctionCall`s with a visitor for `ICPPASTFunctionCallExpression`s as mentioned earlier. Searching for a general AST node did not work for class templates. Resolving the binding and maybe get the type specifier for a variable was necessary to find out if a statement is class template related. We found out that we could extract all information from `IASTName`s AST nodes. Still searching for function calls with a visitor for `ICPPASTFunctionCallExpression`s was still easier. However, this would have meant to traverse some `IASTNode`s twice.

So we built a visitor that traverses all `IASTName`s for a definition and then decides if the statement is relevant for us. This way we were able to find functions, function templates and class templates with the same algorithm and without duplicated code. This list of `IASTName`s is all we needed for further processing.

Figure 2.12 shows the final class hierarchy we used to model the statements. Each concrete class knows how to get the definition for the set binding. A factory method in `AbstractStatementInfo` creates the concrete sub class. `AbstractStatementInfo` offers `searchSubstatements()` to start searching for further sub statements.

The following chapters describe how we found the needed information to display in the User Interface (UI) and stored them into these classes. The gained knowlegde from this chapter helped us to extract these information from the AST.

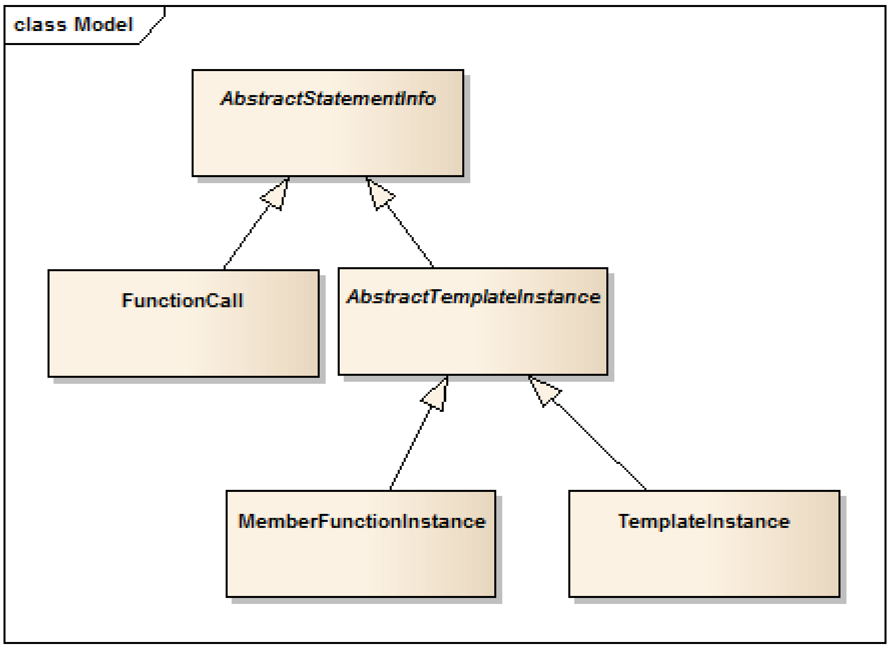


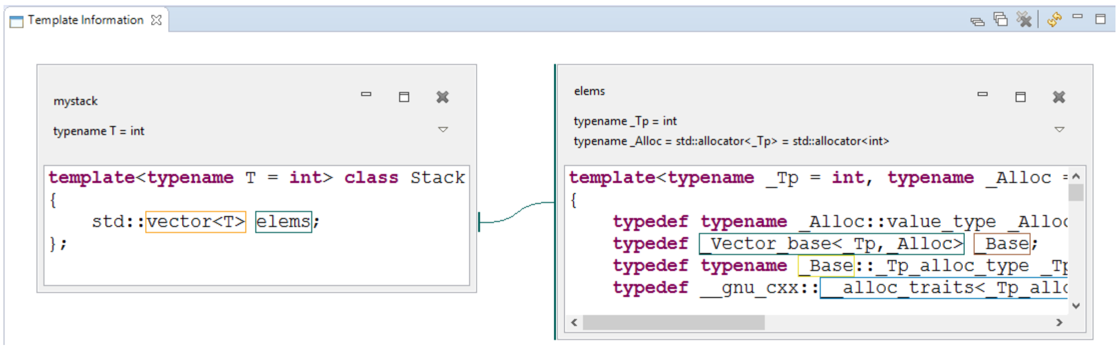
Figure 2.12: Our final class hierarchy to model found statements.

3 Finding Further Relevant Statements

How does a user get from his code in Listing 3.1 in his *CDT* C++ editor to the “Template Information View” containing information about the `mystack` variable and further statements shown in Figure 3.1?

```
#include <vector>
template<typename T>
class Stack {
    std::vector<T> elems;
};
int main() {
    Stack<int> mystack {};
}
```

Listing 3.1: Own class `stack` contains a `std::vector` which depends on the chosen template argument for `Stack`.



typename T = int

```
template<typename T = int> class Stack
{
    std::vector<T> elems;
};
```

typename _Tp = int
typename _Alloc = std::allocator<_Tp> = std::allocator<int>

```
template<typename _Tp = int, typename _Alloc =
{
    typedef typename _Alloc::value_type _Alloc
    typedef _Vector_base<_Tp, _Alloc> _Base;
    typedef typename _Base::_Tp_alloc_type _T
    typedef __gnu_cxx::__alloc_traits<_Tp_allo
```

Figure 3.1: Resulting “Template Information View” contributed by the Templator2 plug-in when visualizing the `mystack` variable from Listing 3.1

First, the user selects a name he or she wants to visualize. This triggers the execution of 4 steps in the Templator2 plug-in until the definition with further template instantiations and function calls will be shown. They are shown in Figure 3.2. The goal is to create an instance of our own class `AbstractStatementInfo` with other `AbstractStatementInfo`s as sub statements. These 4 necessary steps to create this instances are described in the following 4 chapters.

1. Getting the defintion for the selected statement and finding further statements inside this definition. Save the results in our own `AbstractStatementInfo` class. This is described in this chapter.

2. Create an `AbstractStatementInfo` for each statement and resolve it to the selected specialization or function. This is described in Chapter 4 (Template Resolution on page 55).

3. Use the `ASTWriter` to match found sub statements from step 2 to find the regions they will appear in the view. This position is needed to show the user where he can click to open further sub statements.

4. Use the `AbstactStatementInfo` from step 1 and all sub entries from step 2 and visualize them in the "Template Information View". This is described in Chapter 6 (User Interface on page 78).

This chapter first describes how the definition for the user selected statement is found and then continues how we find further instantiations and function calls inside this definition. The terms identifier, `IASTName`, bindings and binding resolution are frequently used in this chapter and are explained in [BS14, p. 28].

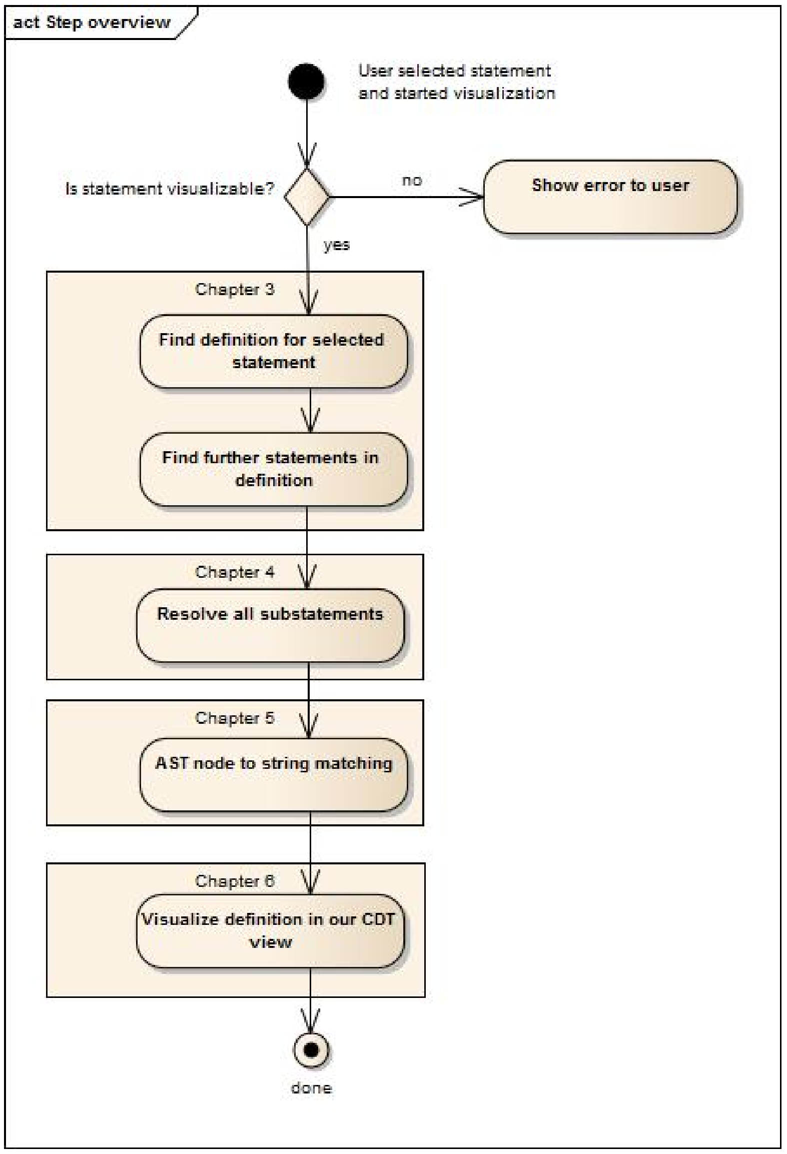

act Step overview
User selected statement and started visualization
Is statement visualizable?
no
Show error to user
yes
Chapter 3
Find definition for selected statement
Find further statements in definition
Chapter 4
Resolve all substatements
Chapter 5
AST node to string matching
Chapter 6
Visualize definition in our CDT view
done

Figure 3.2: Rough overview of the steps performed by the Templator2 plug-in to get from the selected statement to the final view.

3.1 Starting Point

A Templator2 user can select a statement and then start the visualization. We first need to find the corresponding definition to the selected statement to start the visualization. However, there may be statements that result in an ambiguous definition.

3.1.1 Possible Starting Points for Visualization

There are restriction on what can be selected as starting point. The statement has to be an instance of the *CDT* class `IASTName` and has to be fully resolvable by `CDT`, i.e. the name cannot depend on a template argument. Listing 3.2 serves as example code to demonstrate what the user can select as starting point and what not.

```
 1 #include <vector>
 2 
 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 3.2: `Stack<int>` or `mystack` can be used as starting points. `std::vector<T>` and `elems` cannot because they depend on a yet unknown template parameter `T`.

The following `IASTNames` cannot be used as starting point.

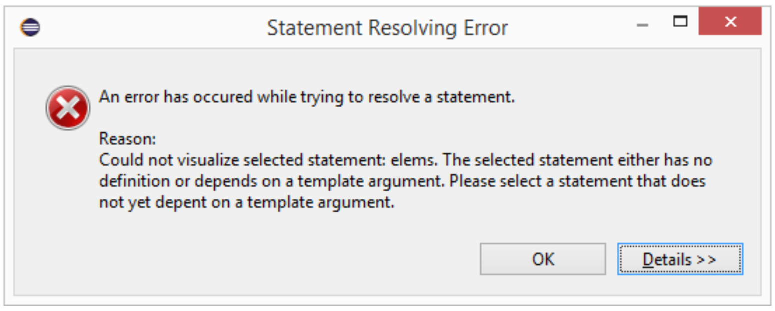
- *line 1*: The `IASTName` in `#include <vector>` marks an include possibly containing type declarations and can thus not be selected. It does not identify a type or function.

- *line 3*: `T` in `typename T` could potentially be a type that can be visualized but at this point it is unknown what the current context is. `T` has not been substituted by anything at this point and just serves as declaration for the template parameter.

- *line 4*: The class name `Stack`. This 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. `elems`' type-specifier 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 user will get an error dialog (Figure 3.3) if one of the above `IASTNames` was used as starting point.



An error has occured while trying to resolve a statement.

Reason:
Could not visualize selected statement: elems. The selected statement either has no definition or depends on a template argument. Please select a statement that does not yet depent on a template argument.

Figure 3.3: Shown error dialog when the user selects an invalid starting point when trying to visualize the statement with the Templator2 plug-in.

On the other hand, the remaining names can be used as starting point.

- *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 *CDT*.

- *line 9*: `Stack<int>` and `mystack` have the same type: A template instance for `Stack<int>`, meaning the current context is now `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 without bloating up the example code.

However, there are still other cases we need to consider. The general algorithm for determining if a `IASTName` is described in Section 3.2 (Find Relevant Sub Statements on page 48) and should now be easier to understand after the above explanation for each case. The goal is always to get a type-specifier represented by an `ICPPASTTemplateId`, at least for the first hierarchy level. Later it is possible that a class name is used without the angle brackets <> specifying the arguments.

After this step, our Templator2 plug-in knows that the selected name is resolvable and will result in a definition we can find.

3.1.2 Find the Definition for the Starting Point

Now that we have a IASTName that *CDT* can fully resolve, the corresponding type identifier has to be found first. As described in Subsection 4.4.3 (Class Template Instantiation on page 68), we need a template-id which is a represented by an ICPPASTTemplateId to instantiate the class template. After having the template instance we can use our DefinitionFinder.findDefinition(IBinding) to get the IASTName from the class template definition.

3.1.3 Retrieve Type-Specifier for a Statement

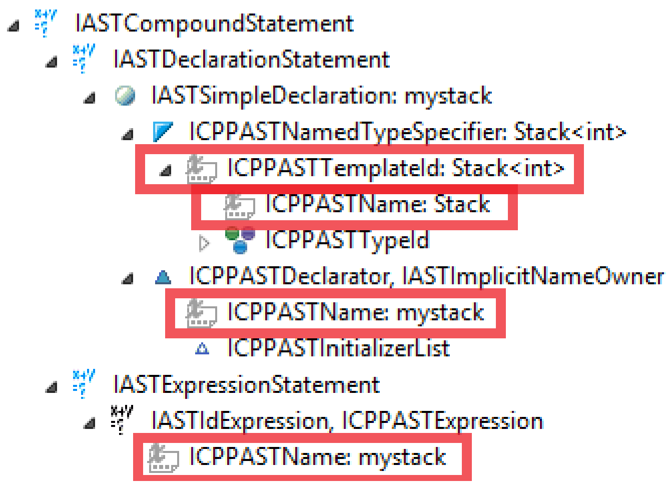
Listing 3.3 shows three possible starting points from Listing 3.2. The relevant part of the AST for these two lines is shown in Figure 3.4. When selecting any of these 3 IASTNames (Stack<int>, mystack in the first or second line), they should all have Stack<int> as type so they resolve to the class template specialization for Stack<int>. To achieve this for any selected IASTName in the example code the following needs to be done.

```
Stack<int> mystack {};
mystack;
```

Listing 3.3: Excerpt with *line 9* and *10* from Listing 3.2.

CDT offers functionality to get the nearest enclosing node as described in [BS14, 5.2.2, p. 53]. The user does not have to select the whole node, for example for Stack, it is enough if the cursor is inside the Stack.

- User selects Stack<int>: Nothing to be done. This is already an ICPPASTTemplateId. Resolving this template-id will result in a CPPClassInstance where T=int. This



```
◢ IASTCompoundStatement
  ◢ IASTDeclarationStatement
    ◢ IASTSimpleDeclaration: mystack
      ◢ ICPPASTNamedTypeSpecifier: Stack<int>
        ◢ ICPPASTTemplateId: Stack<int>
            ICPPASTName: Stack
          ▷ ICPPASTTypeId
      ◢ ICPPASTDeclarator, IASTImplicitNameOwner
          ICPPASTName: mystack
          ICPPASTInitializerList
  ◢ IASTExpressionStatement
    ◢ IASTIdExpression, ICPPASTExpression
        ICPPASTName: mystack
```

Figure 3.4: Created AST for the two statements in Listing 3.3.

is the result of instantiating a class template.

- Selecting only `Stack`: This `IASTName` is a sub node of the `ICPPASTTemplateId`. Resolving this `IASTName` would result in a `CPPClassTemplate` which represent the class template `Stack` itself but not a specific instance of it. For this case we can just get the parent node from the AST to get the template-id and through it the type which resolves to a template instance.

- `mystack` from the first line: The result when resolving this `IASTName` is a variable represented by `CPPVariable`. A `CPPVariable` knows its type and we would be able to get the `CPPClassInstance` for `Stack<int>` but we need the template-id for later resolving—especially in the case of nested template instantiations. Getting the type from the `CPPVariable` would only work for the first hierarchy level. So we need to get the `ICPPASTTemplateId` to get the type and thus the template instance.

  To achieve this we need to extract the type-specifier, which is `ICPPASTTemplateId` for this declaration statement from the AST. This is shown in Listing 3.4.

- `mystack` from the second line: The resulting binding is also the same `CPPVariable` like in the last case. But to get the `ICPPASTTemplateId` we first need to get the `mystack IASTName` from the first line. For this we used our existing functionality from [BS14, 4.2, p. 38] which we changed so we could pass any binding to get the definition name from and not just functions. Our clas `DefinitionFinder` offers `findDefinition(IBinding)` and returns the `IASTName` in the definition statement for any binding. So we can call `findDefinition` with the `CPPVariable` which is an `IBinding`. In return we get the same `IASTName mystack` from the last described case from the first line. After that, the procedure is the same as described above to get the `ICPPASTTemplateId` with Listing 3.4.

```
IASTSimpleDeclaration decl = firstAncestorByType(varName,
   IASTSimpleDeclaration.class, 3);
IASTNamedTypeSpecifier spec = decl.getDeclSpecifier();
ICPPASTTemplateId type = spec.getName().getLastName();
```

Listing 3.4: Simplified version of our code to get the type-specifier from a variable `IASTName`. All other cases to find a specific AST node from a given point work like this. First find the ancestor with a specific type and after that use the nodes methods to get specific sub nodes.

After we retrieved the type-specifier we can call `resolveBinding` which is existing *CDT* functionality to select the correct class template specialization and instantiate it.

Get the Definition for a Class Template Instance

Now that we have the correct binding for a class template instance represented by an `ICPPClassSpecialization` it is easy to get the definition for it. There are only two things to consider.

First, we need to distinguish instances for fully specialized class templates and the rest. As described in this cdt-dev mailinglist entry [Rid15a], a fully specialized class template results in an `CPPClassInstance` instead of an `CPPClassSpecialization`. This is something we did not expect because a fully specialized class template like `template<> struct Stack<std::string>` does not need to be instantiated since it contains no template parameters. As further described in the mailinglist entry this specialization can be distinguished from real template instances with `CPPClassInstance.isExplicitSpecialization()`. This is needed for the next step to find the template declaration.

Secondly, we need to get the specialized binding for template instances but not for explicit specializations. For explicit specializations we can just call our `DefinitionFinder.findDefinition` with the `CPPClassInstance`. As result we get the `IASTName Stack` for the fully specialized `template<> struct Stack<std::string> {};`. From there on we can just find the first ancestor with type `ICPPASTTemplateDeclaration` with our `ASTTools.findFirstAncestorByType` to get the whole class declaration.

For template instances—i.e. the ones that really get instantiated when using the primary or partially specialized class template—the specialized binding is needed. A `CPPClassInstance` represents a specific instance but we need the declaration for the used class template that was instantiated. This is the specialized binding of type `CPPClassTemplate` for an instance we need to find the template declaration. Listing 3.5 shows an example when resolving the `IASTName Stack` in the template declaration on *line 2* and `Stack<int>` on *line 3*.

```
1 template <typename T>
2 struct Stack {}; // CPPClassTemplate=specialized Binding
3 Stack<int> m{}; // Stack<int> resolves to CPPClassInstance
```

Listing 3.5: Resolving the `IASTName Stack` results in a `CPPClassTemplate` which is the specialized binding for the `CPPClassInstance` when resolving `Stack<int>`.

The `CPPClassTemplate` will be set as specialized binding in the `CPPClassInstance` and to find the definition we need the `CPPClassTemplate`. `CPPClassInstance` implements `ICPPSpecialization` which offers the method `getSpecializedBinding()`. Since class template instances can be inside other class template instance, we need to call `getSpecializedBinding` recursively to get the innermost specialized binding. Now we can do the same thing as for explicit specializations to find the definition and then search the `ICPPASTTemplateDeclaration` ancestor.

3.1.4 Save the Information

Now we have every information we need for the starting point. An instance of our `AbstractStatmentInfo` is created with the following information.

- The original selected `IASTName` by the user,
- The type-specifier that is finally used for resolving to get the `CPPClassInstance`,
- the `CPPClassInstance` itself and
- the `ICPPASTTemplateDeclaration`

This `AbstractStatementInfo` serves as context for the containing statements in the `ICPPASTTemplateDeclaration`. `CPPClassInstance` offers `getTemplateParameterMap()` containing a mapping from template parameters to template arguments which is used to instantiate further statements that depend on a template argument. Further relevant statements can now be searched in the template declaration since we have the `ICPPASTTemplateDeclaration` which is described in the next section.

3.2 Find Relevant Sub Statements

The class template definition is now found and saved in an `AbstractStatementInfo` and further relevant statements can be searched in this definition. Found relevant sub statements are later shown to the user as clickable rectangles (Section 6.2.4 (Rectangle on page 89)) to open the definition for this sub statement. Note that this example here works with a class template definition represented by an `ICPPASTTemplateDeclaration` but works exactly the same for other definitions like ones from functions or function templates.

First is described what a relevant statement is for the Templator2 because the described steps after that are easier to understand.

The whole process to determine if a statement is relevant for us is visualized in Figure 3.5. Actually the same algorithm is used to determine if a user can visualize the selected statement or not that is described in Subsection 3.1.2 (Find the Definition for the Starting Point on page 44). The only difference there is that we deferred bindings are not allowed ther but they are as sub statements. This algorithm is implemented in our `ASTAnalyzer.extractResolvingName(IASTName, boolean)` where the second parameter is a flag for accepting deferred bindings or not.

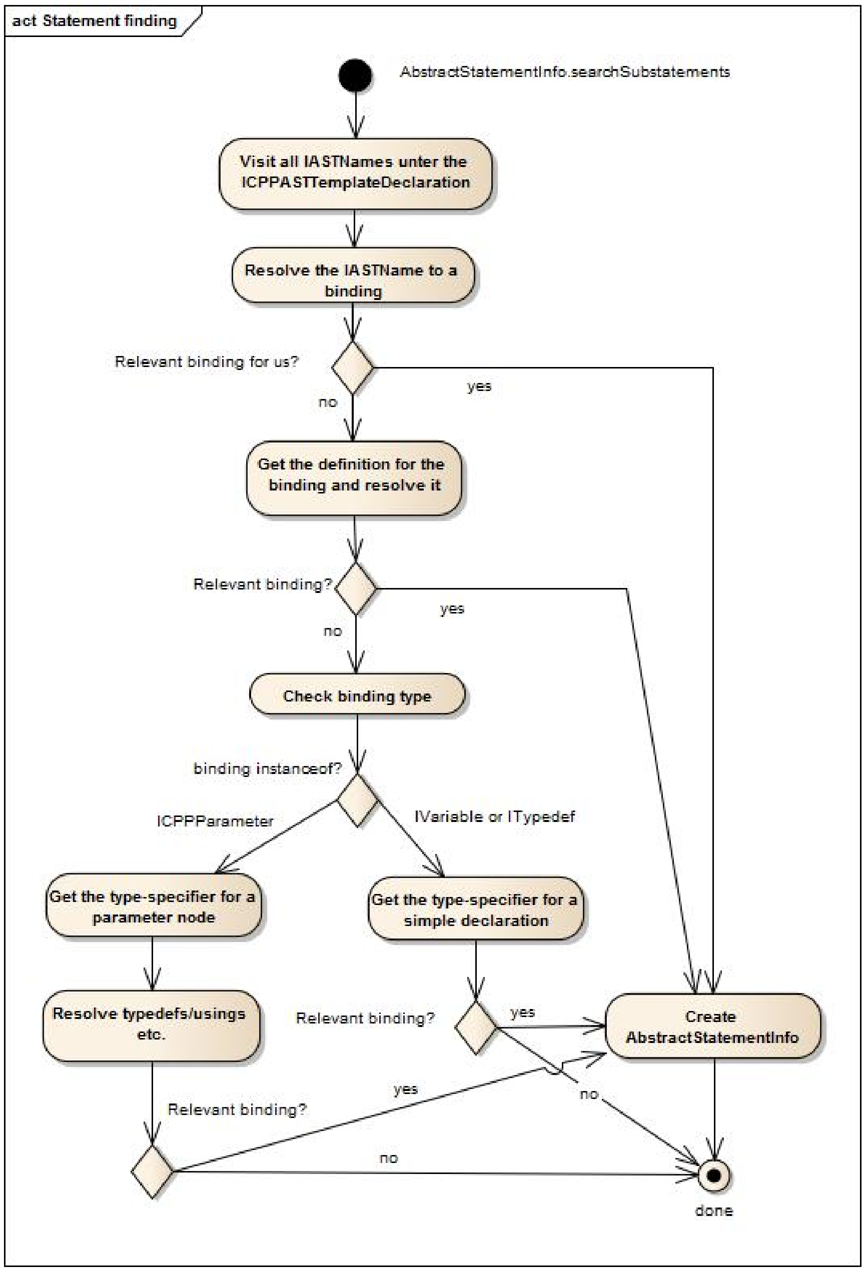

act Statement finding
AbstractStatementInfo.searchSubstatements
Visit all IASTNames unter the ICPPASTTemplateDeclaration
Resolve the IASTName to a binding
Relevant binding for us?
yes
no
Get the definition for the binding and resolve it
Relevant binding?
yes
no
Check binding type
binding instanceof?
ICPPParameter
IVariable or ITypedef
Get the type-specifier for a parameter node
Get the type-specifier for a simple declaration
Resolve typedefs/usings etc.
Relevant binding?
yes
Create AbstractStatementInfo
no
Relevant binding?
yes
no
done

Figure 3.5: Process the determine if a `IASTName` is relevant. If the name is relevant an `AbstractStatementInfo` is created, otherwise `null` meaning not relevant.

3.2.1 Determining if a Statement is Relevant

A sub statement is actually also an `IASTName` and the Templator2 decides based on the type of the resolved binding if it is relevant for us. `IASTName.resolveBinding()` returns a class implementing `IBinding`. Resolving `param` in `void foo(int param)` for example returns an `ICPPParameter`. Using Java's `instanceof` operator we check for its relevancy for our Templator2 plug-in.

- `IFunction`: Every function resolves to an `IBinding` implementing `IFunction`. Since we use `IFunction` and not `ICPPFunction` we are also able to find and later resolve C and not only C++ functions.

- `ICPPSpecialization`: Every instance where the template **parameters** can be determined implements this. Meaning every function template that has been fully resolved and every class template binding. Every class template has the same template parameters thus the `ICPPTemplateParameterMap` can already be built but contains types dependending on the chosen template arguments. Partial and explicit specializations class templates cannot introduce new template parameters but can specialize the already existing ones.

- `ICPPUnknownBinding`: If a function call is deferred and depends on a template argument then it resolves to a class implementing `ICPPUnknownBinding`. This binding does not implement `ICPPSpecialization` because it is unknown if it finally resolves to a normal function or a function template.

We do not yet support

- `ICPPUnknownMemberClass`: Every member that is defined inside a class template that is yet deferred, meaning the template has to be instantiated first The exception are unknown member functions `CPPUnknownMethod` we are able to find and resolve.

- `ICPPUnkownMemberClassInstance`: Represents a class template inside a yet deferred class template.

The Templator2 is already able to resolve most of them but our current architecture with `AbstractStatementInfo` and subclasses does not support them unfortunately. This resolving is described in Section 4.4.3 (Deduce the Type for Any Template Argument on page 69)

If an `IBinding` is relevant according to the above list, we process it further. The User can select the `IASTName` that resulted in this `IBinding` at the end in the "Template Information View". If not, the `IASTName` is ignored and the next one is processed.

3.2.2 Find the Relevant `IASTName` to Check for Relevancy

Now to be able to check if an `IASTName` is relevant we first need to get the correct `IASTName` to resolve the binding on.

The algorithm shown in Figure 3.5 is easier to understand with examples. Especially why we need to check three times if a binding is relevant. The first example shows a simple variable that is used and the second one with a parameter where the type is an alias that does not work yet with our Templator2 plug-in.

Working Example With Variable of Class Template Instance Type

Suppose the user selected `intStack` as starting point in Listing 3.6. We know that `ICPPASTTemplateDeclaration` and its sub nodes for `Stack` from *line 3–10* needs to be searched for further sub statements. This `ICPPASTTemplateDeclaration` was found in the last steps and stored in `AbstractStatementInfo`.

```
 1  #include <vector>
 2
 3  template <typename T>
 4  struct Stack {
 5     std::vector<T> elems;
 6
 7     void add(T elem) {
 8        elems.push_back(elem);
 9     }
10  };
```

```
11
12 int main() {
13    Stack<int> intStack {};
14 }
```

Listing 3.6: `elems` is an instance variable for with type `std::vector<T>`.

This example especially shows the processing of `elems` on *line 8*. All other nodes are processed in a similar way. We iterate over all `IASTName`s in `ICPPASTTemplateDeclaration` with the visitor pattern. Now when we get to `elems` we do the following steps.

1. Resolve the binding and check if it is already relevant. This identifier resolves to a `CPPField`, so not relevant yet.

2. Get the definition for this `CPPField`. This returns the `IASTName` `elems` from the definition on *line 5*.

3. We resolve the identifier again which yields the same `CPPVariable` as before but this resolving is needed for other cases.

4. Now check if the `CPPField` implements `IVariable` which is the case. We now know that an ancestor of `elems` is an `IASTSimpleDeclaration`.

   - This is why the distinction between parameters and normal variables is necessary. The type-specifier for parameters is retrieved differently from the AST than for variables and typedefs.

5. Retrieve the type-specifier `std::vector` from the `IASTSimpleDeclaration`.

6. Create an `AbstractStatementInfo` with the original `IASTName` from *line 8*, the type-specifier `std::vector<T>` and the resolved `ICPPDeferredClassInstance` for the type-specifier.

7. Return the created `AbstractStatementInfo` meaning the original `IASTName` is relevant and we will resolve it later.

Non Working Aliases

Now to an example where we find out if an `IASTName` is relevant or not but cannot resolve it later.

```
 1  #include <vector>
 2  
 3  template <typename T>
 4  struct Stack {
 5     // std::vector<T> = finally resolved binding
 6     // which is relevant
 7     using container_type = std::vector<T>;
 8  
 9     void copy(container_type other) {
10        other; // IASTName to check
11     }
12  };
13  
14  int main() {
15     Stack<int> stack{};
16  }
```

Listing 3.7: `other` is a parameter and its type is an alias for `std::vector<T>`.

Suppose the user selected `stack` as starting point in Listing 3.7. We already know from the created `AbstractStatementInfo` that the `ICPPASTTemplateDeclaration` for `Stack` from *line 3– 12* needs to be searched for further sub statements.

The `IASTName param` resolves to an `ICPPParameter`. That means the definition name for this binding is needed. Calling our `DefinitionFinder.findDefinition` with the `ICPPParameter` returns the `IASTName` in `void copy(container_type param)` from *line 9*. Knowing that the found `IASTName` is inside a parameter declaration we can extract the type-specifier `container_type`.

Now here is where our problem lies. We use *CDT*'s existing method `SemanticUtil.getUltimateType` to resolve all `typedefs` and `usings`, pointers and so on to get the ultimate type behind a binding. This way we check if `std::vector<T>` resolves to a relevant binding but without having the actual used type-specifier `std::vector<T>`, we only have the resolved binding of it and do not look for the used `IASTName` that resulted in this binding. So `SemanticUtil.getUltimateType` returns an `ICPPDeferredClassInstance std::vector<#0,std::allocator<#0»` where `#0` means that the template argument is unknown. This means we have to resolve it later which will fail because we have saved `container_type` as type-specifier.

The alias in Listing 3.7 would be possible to handle and would fit into our classes. But this would have caused further ugly code that was just inserted to be able to resolve something quickly rather than fully refactored into nice methods. And we already had enough of this ugly code at the end we were not able to refactor so we decided against implementing this feature. Also, an alias can be defined depending on other aliases and can be qualified. This means this would not have fitted into our existing architecture because we cannot model qualified name segments besides the last name of an `ICPPASTQualifiedName`. For `typedef _Base::_Tp_alloc_type::other` we can only model `other` but not seperately `_Base` or `_Tp_alloc_type`. We only considered this cases 3 weeks before delivery and were not able to change the existing code to support them nicely.

3.2.3 Conclusion

We are able to find many relevant `IASTName` but not all of them can be resolved later because we save the wrong type-specifier in some cases. To determine the correct type-specifier we would need to change the our classes and architecture which we did not have time for unfortunately. But we still added the finding of these `IASTName`s for later because at the end of our thesis it was already clear that we will expand the Templator2 after this thesis.

Every `IASTName` where the type is not an alias (`typedef` or `using`) or a qualified name is correctly found for later and will be resolved.

4 Template Resolution

Now that all relevant sub statements have been found, they need to be resolved in the next step further. The saved binding in `AbstractStatementInfo` may still be deferred if they depend on a template argument. The final resolving of deferred bindings is explained in this chapter. The correct binding is needed to know which class template specialization or function overload is chosen so the corresponding template declaration for the sub statement can be shown in the Templator2 “Template Information View”.

CDT is already able to resolve nested data members and member functions inside a class template instance which is explained first. But the Templator2 plug-in needs to resolve far more `IASTNames` than just members. Template names can appear inside of functions, as type specifiers in parameters, or return types. This chapter continues with the description why the existing functionality was not enough and how we resolve these bindings.

4.1 Existing Resolving Functionality

CDT already offers functionality to get all data members and member functions for a class. This also works for class templates, i.e. template dependent data member and member functions that depend on a template argument are also specialized and resolved correctly.

```
#include <vector>
template<typename T>
class Stack {
   std::vector<T> elems;
```

```
 5 };
 6
 7 int main() {
 8    Stack<int> intStack {};
 9    Stack<int> boolStack {};
10 }
```

Listing 4.1: `vector<T> elems` depends on the chosen argument for the surrounding template instance for `Foo`.

The example in Listing 4.1 will completely be resolved by *CDT*. Meaning that the `IASTName` `intStack` on *line 9* will be resolved to a `CPPClassInstance` and the containing `std::vector<T> elems` on *line 5*, that depends on the chosen template argument `T`, can be resolved to a `CPPClassInstance` `vector<int>`. Listing 4.2 shows how to get the `CPPClassInstance` for the `elems` data member in the `Stack<int>` instance.

```
1 IASTName name = getSelectedName();
2 IBinding binding = name.resolveBinding();
3 IType classType = ((ICPPVariable) binding).getType();
4 IField[] fields = ((ICPPClassType) classType).getFields();
5 IField elemsMember = fields[0];
6 CPPClassInstance instance = (CPPClassInstance)
     elemsMember.getType();
```

Listing 4.2: Getting the binding for the template argument dependent name `elems` in Listing 4.1. Simplified without type checking and casts.

- *line 1*: Retrieve the selected name, in this case `intStack`.
- *line 2*: Resolve the name to get the corresponding variable binding. Bindings are explained in [BS14, p. 28].
- *line 3*: Get the type of this variable, i.e. the resolved binding from the type-specifier `Stack<int>` which is a `CPPClassInstance`.
- *line 4*: Get all class members (=fields) from the template instance for `Stack<int>`.

- *line 5*: Retrieve the first data member named `elems`.
- *line 6*: Get the type of this variable as on *line 3*, i.e. resolving the binding for `std::vector<T>`. Because the binding is resolved inside a `CPPClassInstance`, *CDT* knows that the template parameter `T` was substituted by `int`. So `instance` now holds a `CPPClassInstance` for `vector<int>`.

The same can be done for the `IASTName` `boolStack`. `instance` then is a `CPPClassInstance` for `vector<bool>` which is a fully specialized template for `vector`.

4.2 Resolving Other Template Names

But template names can appear at other places than just type-specifiers for class members. For example, as type of a parameter. *CDT* offers functionality to specialize them but does not do it by default.

Listing 4.3 shows the usage of the template dependent `initializer_list<T>` on *line 8* which is represented by an `ICPPASTTemplateId`. If the name `std::initializer_list<T>` will be normally resolved by *CDT*, it creates a deferred binding indicating that the name depends on a template argument and thus cannot be fully resolved now. If `T` is known later it can be specialized.

```
 1 #include <vector>
 2 
 3 template<typename T>
 4 class Stack {
 5    std::vector<T> elems;
 6 public:
 7    Stack(std::initializer_list<T> elemList) {
 8       elems = std::vector<T> { elemList };
 9    }
10 };
11 
12 
```

```
int main() {
   Stack<int> intStack { 4, 8, 15, 16, 23, 42 };
}
```

Listing 4.3: Nested class template that dependends on a template argument.

On the other hand, nested template names that do not depend on a template argument can be resolved by *CDT*. `initializer_list<int>` on *line 8* in Listing 4.4 does not depend on `T`. Resolving this template-id results in a `CPPClassInstance`, so the binding is not deferred.

```
 1  #include <initializer_list>
 2
 3  template<typename T>
 4  class Stack {/*... */};
 5
 6  template<>
 7  struct Stack<int> {
 8     Stack(std::initializer_list<int> elemList) {}
 9  };
10
11  int main() {
12     Stack<int> intStack { 108 };
13  }
```

Listing 4.4: Nested class template with a concrete type that does not depend on another template argument.

The summary is that *CDT* is now only instantiating a class template if it does not depend on other template parameters or is a class member. So the same as with function templates as described in [BS14]. But for the Templator2 plug-in to work, class templates need to be instantiated for any nesting level, i.e. even if they are dependent on template arguments from the surrounding function or class.

The next section describes that there is already *CDT* functionality to also resolve `initializer_list<T>`in Listing 4.3 but why this was not sufficient for us and what we implemented to further resolve more function templates, fix errors from the existing Templator plug-in, and resolve class templates and their members correctly.

4.3 Problems With Existing Functionality

Using the already built in functionality described in Section 4.1 came with the problem, that the template declaration for the `IBinding` could not be found in a later step. Thus resulting in no showable code in the view. It is first described how the instantiation of nested template dependent names would have worked and second why this was no solution for us.

4.3.1 Instantiating Nested Template Dependent Names

The `initializer_list<T>` on *line 8* in Listing 4.5 could already be resolved correctly with `ICPPClassSpecialization.specializeMember`.

```
1 #include <initializer_list>
2
3 template<typename T>
4 struct Stack {
5   Stack(std::initializer_list<T> elemList) {}
6 };
7
8 int main() { Stack<int> intStack { 4, 8}; }
```

Listing 4.5: Nested class template that dependends on a template argument. Shortened version from Listing 4.3 (Nested class template that dependends on a template argument on page 57).

To try and resolve this with existing *CDT* functionality the following steps need to be done. Listing 4.6 shows the code to specialize the nested class template followed by a description of the code.

```
1 ICPPASTTemplateId intStack = getStackIntTemplateId();
2 CPPClassInstance stack = intStack.resolveBinding();
3 ICPPASTTemplateId initList =
     getDependentInitializerListName();
4 IBinding deferredInitList = initList.resolveBinding();
5 IBinding intInitList =
     stack.specializeMember(deferredInitList, initList);
```

Listing 4.6: Getting the specialized binding for the template argument dependent name `initializer_list<T>` in Listing 4.5. Shortened without type checks and casts.

- *line 1 and 2:* Get the `Stack<int>` name and resolve it.
  - Resolving the `ICPPASTTemplateId` `Stack<int>` results in a binding of type `CPPClassInstance`.
  - `CPPClassInstance` implements `ICPPClassSpecialization` which provides `specializeMember(IBinding, IASTNode)`. So a class instance serves as context (the template parameter `T` is `int`) and provides a method to specialize any binding inside the class `Stack` where `T` is substituted with `int`.
- *line 3 and 4:* Get the `ICPPASTTemplateId` for `initializer_list<T>` and resolve it to a `ICPPDeferredClassInstance`.
- *line 5:* Specialize the deferred binding.
  - Since it is called on the `CPPClassInstance` for `Stack<int>`, `T` will be substituted with `int`.
  - The result is an `ICPPClassSpecialization` for `initializer_list<int>`. `ICPPClassSpecialization` is a sub class of `ICPPClassInstance`.

4.3.2 Problems

This seems like we could use it but there are two problems with this solution which are described in the following.

1. **It may influence other plug-ins because the binding is cached.**

   Each `CPPClassInstance` has a cache that maps the original binding with the specialized. So a call `instance.specializeMember(originalBinding, node)` will put `originalBinding` with the specialized returned binding in a map. This should speedup further member specializations for the same `originalBinding`. However, this may influence *CDT* or other plug-ins that may call `specializeMember` with the exact same `originalBinding` on the same `instance`. So they might get a wrong specialized binding for the wrong context. If the Templator2 plug-in adds a deferred binding for the context `T=int` to the map, any other plug-in resolving exactly the same `ICPPASTTemplate initializer_list<T>` and calling `specializeMember` on the same `CPPClassInstance` will also get a binding for `initializer_list<int>`—even though `T` might be substituted for something else in their context. And our plug-in should not influence other plug-ins.

2. **The template declaration for the binding cannot be found.**

   `specializeMember` just calls `CPPTemplates.createSpecialization` besides the caching. So only calling this method to get the correct binding would solve the above problem. But as described in Section 3.1.3 (Get the Definition for a Class Template Instance on page 46), sometimes `getSpecializedBinding()` needs to be used to find a template definition. `createSpecialization` sets the binding returned by `getSpecializedBinding` just to the original binding we passed, i.e. the deferred binding. *CDT* cannot find the definition for a deferred binding because it is unknown which class template got chosen.

Finding the template declaration is essential for our plug-in because otherwise no further template instantiations and function calls can be visualized for the user. If this would not have worked, the Templator2 plug-in would have only been able to show the first hierarchy level—which is exactly what *Eclipse* is now able to do.

CDT is able to instantiate them correctly in the background so this might be a feasible solution in the future. But the way we call and used it caused the above problems. Since we already had the solution that is described in the following section, and did not have enough time, we did not further follow the above solution.

4.4 Instantiation of Templates in Templator2

This section describes the solution to instantiate nested templates in Templator2 and specialize members.

To be able to show the user the resolved template for Listing 4.3 (Nested class template that dependends on a template argument on page 57), we must instantiate a template with consideration of the current context. Resolving a template name in *CDT* means that the template will be instantiated in the background resulting in an `ICPPTemplateInstance`.

The problem we needed to solve was to resolve a template argument dependent `IASTName` to get a Java class of type `ICPPTemplateInstance` that is not an `ICPPUnknownBinding`. Meaning the binding is not deferred so it is actually known which template specialization or function overload was chosen so we can get the template definition (=code) and show it to the user in our view.

We first describe how we improved the resolving of function templates from our term thesis ([BS14]) and then the resolving of class templates via reflection to call *CDT* `private` methods.

4.4.1 Resolving and Instantiating Function Templates

The old solution to get the template parameter map ([BS14, 4.1.4, p. 33]) was to build it by ourselves as described in [BS14, 4.1.5, p. 34]. This caused some function templates to only work on the first hierarchy level. Because the template arguments for function template calls are sometimes deduced ([BS14, 2.1.1, p. 13]) from the call site there are many cases to consider. And our code to deduce the arguments did only handle the simplest cases. If there were further dependend function templates inside this call, they could not be resolved. This cases are described in the last paragraph in [BS14, 2.2.2, p. 20].

Because the three described cases can be crucial, this was an unacceptable situation and we already planned to switch to *CDT* functionality to build the template parameter map in our term thesis ([BS14, 7.7.2, p. 68]).

4.4.2 Instantiating Function Templates

CDT already offers the `private` method `CPPTemplates.instantiateForFunctionCall` to instantiate a function template and returns on success an `ICPPSpecialization` instance. `ICPPSpecialization` offers the method `getTemplateParameterMap()` with the correctly deduced template arguments, the ones we deduced by ourselves until now.

Implementing the same functionality again by ourselves when there is already a *CDT* method that does the same was counterproductive. We decided to call the `private` method `CPPTemplates.instantiateForFunctionCall` via reflection. Even thoug reflection can be bad since method signature changes do not result in a compile time error but in a runtime error, it was the better solution than implementing the whole deduction process again.

Following now two cases that are documented in [BS14, 2.2.2, p. 20] because they did not work yet in our term thesis. By using this new method instead of our template deduction process the first two cases now worked. And with it probably many other function template related argument deductions like the occurence of pointer and references in template arguments.

CDT implements the described deduction from the C++ standard described in [ISO11, [temp.deduct.call]].

Deducing Non-Type Template Arguments

C++ offers other template parameter types than `template <typename T>`. Non-type parameters like `template <int n>` can be used to deduce other information.

A usage example is the first case from [BS14, 2.2.2, p. 20] which is shown in Listing 4.7 to deduce the size of an array.

```cpp
template <typename T,unsigned n>
unsigned array_size(T (&value)[n]) { return n; }

template<typename T=int>
void outer() {
        int arr[] = {1,2,3};
        array_size(arr); // array_size<int,3>
}
int main() { outer(); }
```

Listing 4.7: Array size deduction with a function template that did not work in our term project but does now.

The deduction of these non-type parameters now works because of the newly used method from `CPPTemplates.instantiateForFunctionCall`.

Function Template Calls As Function Arguments

A programmer can pass the return value of a function call as argument for another function call as shown in Listing 4.8. This already worked in our term thesis.

```cpp
double sqr(double d) {
   return d*d;
}
void calculate(double d) { /* ... */ };

int main() {
   calculate(sqr(2.4));
}
```

Listing 4.8: The passed argument to `calculate` is the returned value of `sqr`.

However it did not work if the return type was dependent on a template argument as described in the second case in [BS14, 2.2.2, p. 20] and shown in Listing 4.9. The user could not inspect the `inner` function definition because `outer` could not be resolved. The Templator plug-in did not know if the normal function `outer(int)` was chosen or the function template. This now also works with the newly used method.

```cpp
template<typename T>
void outer(T value) {}
void outer(int value) {}

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

template<typename T>
void inner(T value) { outer(id(value));}

int main() { inner(5); }
```

Listing 4.9: The call to `outer` depends on the return value of `id` which depends on the template argument for `T`.

Selecting Function Overload Based on the Value Category

Every expression in C++ has a value category as described in [ISO11, [basic.lval]]. The compiler will consider this category when selecting a function overload.

Our advisor gave us a C++ quiz from [Mau14, Quiz #3] which is shown in Listing 4.10.

```cpp
#include <iostream>
#include <utility>

void y(int&) { std::cout << '1'; }
void y(int&&) { std::cout << '2'; }

```

```
template <typename T> void f(T && x) { y(x); }
template <typename T> void g(T && x) { y(std::move(x)); }
template <typename T> void h(T && x) {
    y(std::forward<T>(x)); }

int main() {
    int i = 42;
    y(i), y(42); // 1 2
    f(i), f(42); // 1 1
    g(i), g(42); // 2 2
    h(i), h(42); // 1 2
}
```

Listing 4.10: Pubquiz #3 from [Mau14] where functions are called based on the value category.

The results are written as comments beside their call. The trick is that `std::move` and `std::forward` may change the value category based on the category for the passed argument. But the details do not matter here but rather that the existing Templator2 did resolve the last 4 calls (`g` to `h`) wrong even after the description in the last section (Section 4.4.2). The calls from `y` to `f` were correctly resolved since they did not depend on the return type of a function template.

So using `CPPTemplates.instantiateForFunctionCall` did not solve all overload selection errors yet. The problem was that the value category for the returned type was not set. As briefly described in [BS14, 4.1.5, p. 34] we use our class `TemplateContextLookupData` to inject data about nested function template instantiations into the *CDT* resolving process. `TemplateContextLookupData` extends the existing `LookupData` and is used by `CPPSemantics.resolveFunction(LookupData, ICPPFunction[], boolean)))` to get the passed arguments for a function call. `TemplateContextLookupData` now replaces the returned type when depending on a template argument.

This was the status after the term thesis. Each call just returned the correct type but did not store any information in the `TemplateContextLookupData` about the replaced types. `resolveFunction` also uses the value category of the passed arguments to find the best matching overload. Since the replaced types were not stored in the `TemplateContextLookupData`, this means the yet unknown value category for the template parameter was used which defaulted to an lvalue. The solution included to go through all passed arguments, replace them with the correct template arguments or the return types of another function call, and set the value category and other data based on these new types. Pubquiz #3 can now be fully resolved wtih this extension.

Overload Selection for Member Functions

Extending `TemplateContextLookupData` by replacing other members of `LookupData` also helped to resolve other bindings. The resolving of overloaded member functions also works with `resolveFunction` and uses `LookupData.fImpliedObjectType` which is the type of the owner, where the member function is called on. The owner binding is already resolved in another part but was also not yet stored in `TemplateContextLookupData`. However just replacing the `fImpliedObjectType` is not enough. The qualifiers of this type needs to be preserved as they are used for overload selection. Listing 4.11 shows such an example. `stack.foo()` calls the first `foo` because `stack` (=fImpliedObjectType) is not `const`. The second one is called on `con` which is `const` hence the second `foo` is called.

```cpp
#include <iostream>
template <typename T>
struct Stack {
   void foo() { std::cout << "1"; }
   void foo() const { std::cout << "2"; }
};

int main() {
   Stack<int> stack {}; stack.foo(); // 1
   const Stack<int> con {}; con.foo(); // 2

}
```

Listing 4.11: Member function overload selection based on qualifiers.

A type like `Stack<int>` can be wrapped into a container like a `CPPPointerType`, `CPPReferenceType`, or `CPPQualifierType` that saves the original type `Stack<int>`. The `fImpliedObjectType` needs to be replaced considering all possible containers and set member attributes to preserve all qualifiers. `SemanticUtil.replaceNestedType` luckily already offers exactly this to replace only the innermost type in a type container.

Such overloads as shown in Listing 4.11 now also work for any nesting level with the replacement of the `TemplateContextLookupDate.fImpliedObjectData`.

4.4.3 Class Template Instantiation

The best and easiest solution to resolve and instantiate class templates is again by reflection. We did not find any other method in *CDT* to easily resolve deferred class templates where the Templator2 could replace template parameters with template arguments. The next sections describes what the Templator2 does to resolve `ICPPDeferredClassInstances`.

`CPPTemplates.createBinding(ICPPASTTemplateId` does exactly what was needed to resolve a type-specifier. Expanding this method would have been convenient. Getting code into *CDT* is difficult, especially when we cannot guarantee that it works for all class templates. That is why we needed another solution. One with injecting something like our `TemplateContextLookupData` as we do for function template did not work for this method. The method checks at one point if a template argument depends on another template argument. If this is the case, an `ICPPDeferredInstance` is returned—the one we already have and want to fully resolve. Our final solution to this was to

- Copy the whole `CPPTemplates.createBinding` method into a class of our own.
- Replace all `private` method that `createBinding` calls in `CPPTemplates` with reflection calls for these `private` methods.
- Replace the dependent types and instantiate the class template.

The disadvantage of this solution is the maintainability. Every time `CPPTemplates.createBinding` changes, our method has to be updated. The whole extension with the replacing of an argument is extracted into a single method which will make the updating easier but it is still an ugly solution and hard to maintain.

We distinguish three different cases for the argument to replace in our added code to `createBinding`.

- The argument is of type `ICPPTemplateParamater`. For this case the corresponding argument from the current context (=the parent `AbstractStatementInfo`) will be retrieved.
- The argument is a `ICPPASTTemplateId`. We instantiate all `ICPPASTTemplateIds` recursively to get the ultimate type for these nested template-ids.
- The argument is defererred but not an `ICPPASTTemplateId`. We try to retrieve the type from the a parameter, variable, aliased type etc. This algorithm works recursively and is described in the next section.

Deduce the Type for Any Template Argument

Deducing the type for any `IASTName` can be hard. This could resolve to a parameter, a variable, or an aliased type or a parameter/variable with an aliased type. This is a feature that we implemented in the second last week before the final release. The implementation is not yet finished but serves as good starting point to resolve other types in the future and caching them will also be easy. There was just no time before the final release to implement all this.

Lets consider Listing 4.12. The template argument for `std::vector<cont>` is `cont` which has to be fully resolved first. An `IType` is needed so the existing template argument can be replaced.

```
#include <vector>

template<typename F>
struct Base {
   using cont=std::vector<F>;
};


```

```
 9 template<typename T>
10 class Stack : Base<T>{
11    using _b = Base<T>;
12    using cont = typename _b::cont;
13    std::vector<cont> elems;
14 };
15 
16 int main() {
17    Stack<int> mystack {};
18 }
```

Listing 4.12: Aliased type

Our `StatementTypeDeducer` executes the following steps to deduce the type of `cont` to `std::vector<int>` so we finally have `std::vector<std::vector<int>`.

1. Resolve the `IASTName` `cont` which results in an `ITypedef`.
2. Get the `IASTName` for the type alias in the alias declaration with `DefinitionFinder.findDefininition`. This returns `cont` from *line 11*.
3. Now that we know we resolved an `ITypedef` we try to get the type-specifier for the aliased type.
4. We notice that the type-specifier is an `ICPPASTQualifiedName`. So iterate over all name segments.
5. First check `_b` and also get the type-specifier for this aliased type which is `Base<T>`.
6. Instantiate `Base<int>` because we instantiated `Stack<int>` and thus `T=int`.
7. Save this `Base<int>` template instance because the next qualifier segment `cont` is a member of this class template instance.
8. Instantiate the member with `CPPTemplates.instantiateBinding` which does not have the described problems from Subsection 4.3.1 (Instantiating Nested Template Dependent Names on page 59) and is our solution to this.

9. Create a new `AbstractStatementInfo` for the created `Base<int>` instance that will serve us as new context. The next step explains why.

10. Find the definition for the instantiated member binding `cont`. This returns the `IASTName cont` from *line 5*. Now all further templates must be instantiated with `F=int` because we switch to a new `ICPPASTTemplateDeclaration` which introduces a new parameter `F`.

11. Resolve the binding to see that it is an `ITypedef`.

12. Retrieve the aliased type-specifier `std::vector<F>` from *line 5*.

13. Instantiate `std::vector<F>` with `F=int`. The result is an `CPPClassInstance` which implements `IType`.

14. Replace the argument `cont` in `std::vector<cont>` on *line 12* with this type `CPPClassInstance std::vector<int>`.

This case was rather easy. Our `StatementTypeDeducer` can also handle nested classes and alias templates. The difficult part is handling these instantiations and context changes recursively. Section 2.3 (Supported Features in Our Templator Plug-in on page 35) describes that nested class templates and alias templates are not supported. Also, that such qualified type aliases do not yet work with the Templator2 plug-in. The correct wording is: Many of these language features can be correctly resolved but they are not shown to the user. This was the last implemented feature and to support these language feature to be shown to the user, some bigger architecture changes were necessary. There was no time for this. This means for the user that he may need to define an alias type and use this aliased type as template argument to see what it resolves to.

Resolving Without a Template-Id

A class template name can be used without `<>` inside the class template. This is described in Section 2.1.3 (Requirement for a Template-Id on page 24) and means the current context from the surrounding class template is used. `Stack` is used as type-specifier in Listing 4.13, so the corresponding node is an `IASTName` but the Templator2 needs an `ICPPASTTemplateId` to resolve the class template binding.

```
template <typename T, typename F>
class Stack {
   Stack& operator=(const Stack& other) {}
};
```

Listing 4.13: Type-specifier for the return type is allowed without angle brackets. The member function will return and accept as parameter argument a template instance equal to `Stack<T, F>`.

One solution would have been to just clone the parent `AbstractStatementInfo`. We decided us to get the type-specifier from the parent which will resolve to the current class template again. The current solution resolves the `ICPPASTTemplateId` again but caching can be implemented easily later.

4.4.4 After the Resolving

This step was a success if the binding is not deferred or unknown. If this is still the case the `AbstractStatementInfo` is thrown away and the user will not be able to click on this statement later in the view. However, he will see in the "Template Information View" that the resolving was not successful for this `IASTName`.

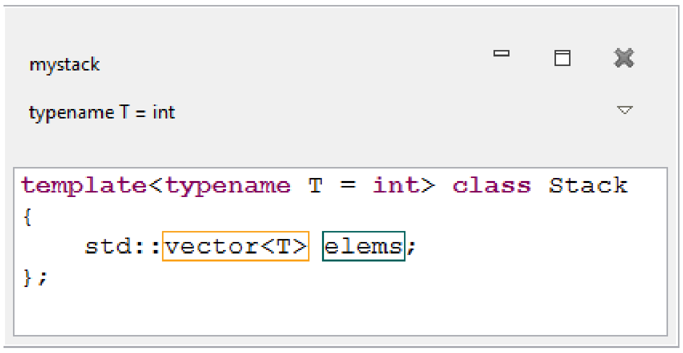
If the statement is resolved to its final binding considering template argument, the definition for it is searched as described in Section 3.1.3 (Get the Definition for a Class Template Instance on page 46). Now the `AbstractStatementInfo` for a sub statement is final. One thing needs to be done before the `AbstractStatementInfo` for the starting point is shown in the UI. The matching information of AST nodes of sub statements with their string representation is needed. This is described in the following chapter.

5 AST Node and Text Matching

To show the user which function gets called or which class template is instantiated, we show the function or class definition in an *Eclipse* view. This is done by printing an `IASTNode` with the existing `ASTWriter.write(IASTNode)` which just returns a string. This string is then shown in the view. The final goal is to frame template depending nodes to indicate the user that he can click on them. We needed to achieve a match between the `IASTNode`s and their string representations.

This section describes why we had to change our existing solution from the term thesis [BS14], what we tried, why it did not work and finally the implemented solution.

The result is shown in Figure 5.1. At the end the region (offset and length) for each sub statement should be found so it can be passed to the UI to draw rectangles. They indicate to the user that he or she can click on it to open the definition.



```
template<typename T = int> class Stack
{
    std::vector<T> elems;
};
```

Figure 5.1: The goal of this chapter: Find the regions for `vector<T>` and `elems`. Finding the position for each sub statement to later draw a rectangle around it.

5.1 The Existing Solution

The solution implemented in [BS14] was string matching. This was enough because we formatted the text so, that each call had the template-id even if it was not in the original call.

This lead to a search string in the form of `function name` + `template id` + `passed arguments`. Listing 5.1 shows the existing solution with their search strings as comments.

```
 1  template<typename T>
 2  T id(T value) {   return value; }
 3
 4  template<typename T>
 5  T sqr(T value) { return value*value; }
 6
 7  template<typename T>
 8  void start(T value) {
 9                   // searchstring:
10     id(value);    // id<int>(value)
11     sqr(value);   // sqr<int>(value)
12     id(value);    // id<int>(value)
13  }
14
15  int main() { start(4); }
```

Listing 5.1: Code example with the existing search strings.

After each found string, we continued searching for the string after the last found string position. With this approach `id<int> (value)` is correctly found twice on line 10 and 12. This method only failed if this exact same string was in a string literal like Listing 5.2.

```
std::string s {"id<4>(value)"};
```

Listing 5.2: Search string in string literal where the existing solution failed.

This problem only occured if the literal is exactly between the last matched string and the actual function call.

This solution worked for almost all function template calls except for the above mentioned string literals.

5.2 Problem With Class Templates

This method however was not enough for class templates. Our plug-in should be able find declarations, class template instance variables, the class name used as type inside the class and more that depends on template arguments. The latter two, instance variables and class names, had a very high fault rate. In our manual tests in our test classes, we were still able to find most strings correctly but in `std::vector` for example we had around 10% wrong matches.

Considering the declaration for the `std::vector` copy constructor in Listing 5.3.

```
vector(const vector& __x);
```

Listing 5.3: `std::vector` copy constructor declaration.

The parameter type `const vector&` is dependent on the template argument. So we use "vector" as search string and find the constructor name instead of the parameter type. This could happen on many locations because the search string is so short and not as unique as with function template calls.

Since the region matching needed to be fully deterministic, we had to find another solution.

5.3 Search for the Parent Nodes String Representation

The next idea was to call `IASTNode#getParent()` to get the direct parent or even grandparent node and use its string representation as search string. We then used this search string to search in the whole class. Figure 5.2 shows that the parent is the node for `const vector` and its parent is the whole parameter declaration `const vector& __x`.

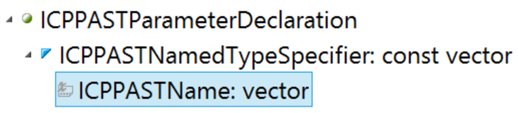

ICPPASTParameterDeclaration
ICPPASTNamedTypeSpecifier: const vector
ICPPASTName: vector

Figure 5.2: AST for parameter type `vector` in Listing 5.3

This solution worked for this and many other cases but it was still possible that the grandparent node would have been the whole `IASTTranslationUnit`. Because of this the search result would have been worse. Usually the search offset was always increasing. In the case when we found the whole translation unit this offset was reset to 0. This made the search offset decreasing. In this case the search of the next string would have started at the beginning of the translation unit again.

This solution reduced the number of errors but we wanted a solution where we could guarantee that each node we frame is certainly correct. This was not possible with this solution.

5.3.1 Writing Node for Node

The optimal solution was to know the offset for each written `IASTNode`. So a mapping from each `IASTNode` to the offset in the written string by `ASTWriter.write(IASTNode)` would be possible. The problem with the `ASTWriter` is that the class has no configuration options or flags that could be set.

It was assumed that it would not work but we wanted to be shure. We tried to write node for node instead of the whole `ICPPASTTemplateDeclaration` to be able to build a map in the background that stores the offset for each node. We tried to write only the leaf nodes but it failed. Obviously, the parent nodes contain more information which is lost when only writing the leaf nodes.

When printing only the leaf nodes for the copy constructor declaration (Listing 5.3) the result would have been `vector vector &__x`. The indentation, newlines, parentheses and other missing characters would have been missing with this approach.

5.3.2 Hooks in the ASTWriter

Our only possibility was to use the `ASTWriter` directly because it was writing the code and so it would be the only way to somehow gain a perfect mapping between `IASTNodes` and regions.

The `ASTWriter` uses the `ChangeGeneratorWriterVisitor` to traverse the AST. We created a sub class of it that overrode the `visit(IASTName)` function. In this function, before we call `super.visit(name);` we could finally make a deterministic mapping between the `IASTNode` and the current offset in the text. The current offset was easy to determine. We could read it out of the `scribe` with the function `scribe.getBuffer().length()`.

We stored the mapping between the `IASTNodes` and their offset in a map. Since the `ASTWriter` was processing a copied AST, it was also necessary to store the original node for every `IASTName`. This was finally the deterministic mapping that we were looking for.

There is one drawback to this solution. Names in macros are not processed in the `ChangeGeneratorWriterVisitor`. If the `ChangeGeneratorWriterVisitor` processed a macro, it copies the source code of the macro from the source file and writes it. This is done with the code in Listing 5.4.

```
protected int writeMixedStatement(IASTStatement statement) {
    String code = statement.getRawSignature();
    scribe.println(code);
    return ASTVisitor.PROCESS_SKIP;
}
```

Listing 5.4: The `ChangeGeneratorWriterVisitor` simply copies macros from the source file.

Because of this, template dependant names in macros are not found with the Templator2 plug-in. Besides this issue, this is the perfect solution for this problem.

6 User Interface

Eclipse's Standard Widget Toolkit (SWT) is a heavyweight widget toolkit for Java. In addition, *JFace* adds an additional abstraction layer and richer widgets with even more UI classes. But none of the existing widgets had the functionality we needed to show the template instantiation to the user, so we had to build it ourselves.

This chapter describes the evolution of our UI. First we discuss why we had to rework the UI in this semester. Then we go over the concepts we followed when implementing the view and finally we will talk about the challenges that we encountered during the implementation.

6.1 From the Old to the New View

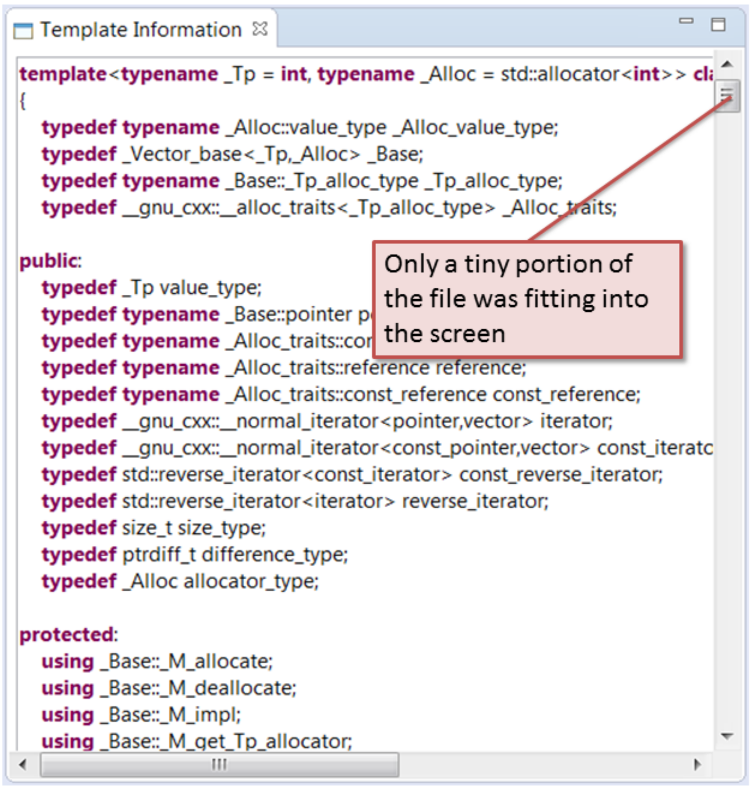
In this section we will discuss why we had to rework the UI in this semester thesis. It explains the problems that we found with our old view and gives an overview what we wanted to accomplish with our new version of the view.

6.1.1 Problems with Our Existing View

The view as it was in the final version of the plugin from the term thesis was already very robust. In this thesis during the first few weeks we could research class templates and use the existing view to visualize our findings. This was mainly possible because of a single inteface `ViewData` that coupled the data with the UI. The interface only contained a handful of functions. We just had to implement these functions to have the possibility to visualize class templates. For the first few weeks of the semester this was absolutely sufficient.

But when we started to examine files from the standard library we started to notice that the view was very confusing for big files (Figure 6.1). This was because the connection lines between the nodes went from the originating rectangle to the middle of the destination entry in the next column. For files with for example 1000 lines of code, these connection lines where almost vertical and the user could not gain any information out of them. The problem is illustrated in Figure 6.2.

This was bad, because the connection lines are our main way to communicate the tree like structure of the template deduction. It was also very circumstantial to close a link because the user had to click the correct rectangle in the originating entry to close an open link. This forced the user to scroll hundreds of line to find the correct rectangle.



```
template<typename _Tp = int, typename _Alloc = std::allocator<int>> cl
{
  typedef typename _Alloc::value_type _Alloc_value_type;
  typedef _Vector_base<_Tp,_Alloc> _Base;
  typedef typename _Base::_Tp_alloc_type _Tp_alloc_type;
  typedef __gnu_cxx::__alloc_traits<_Tp_alloc_type> _Alloc_traits;

public:
  typedef _Tp value_type;
  typedef typename _Base::pointer p
  typedef typename _Alloc_traits::cor
  typedef typename _Alloc_traits::reference reference;
  typedef typename _Alloc_traits::const_reference const_reference;
  typedef __gnu_cxx::__normal_iterator<pointer,vector> iterator;
  typedef __gnu_cxx::__normal_iterator<const_pointer,vector> const_iteratc
  typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
  typedef std::reverse_iterator<iterator> reverse_iterator;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;
  typedef _Alloc allocator_type;

protected:
  using _Base::_M_allocate;
  using _Base::_M_deallocate;
  using _Base::_M_impl;
  using _Base::_M_get_Tp_allocator;
```

Figure 6.1: Screenshot of the old version of view

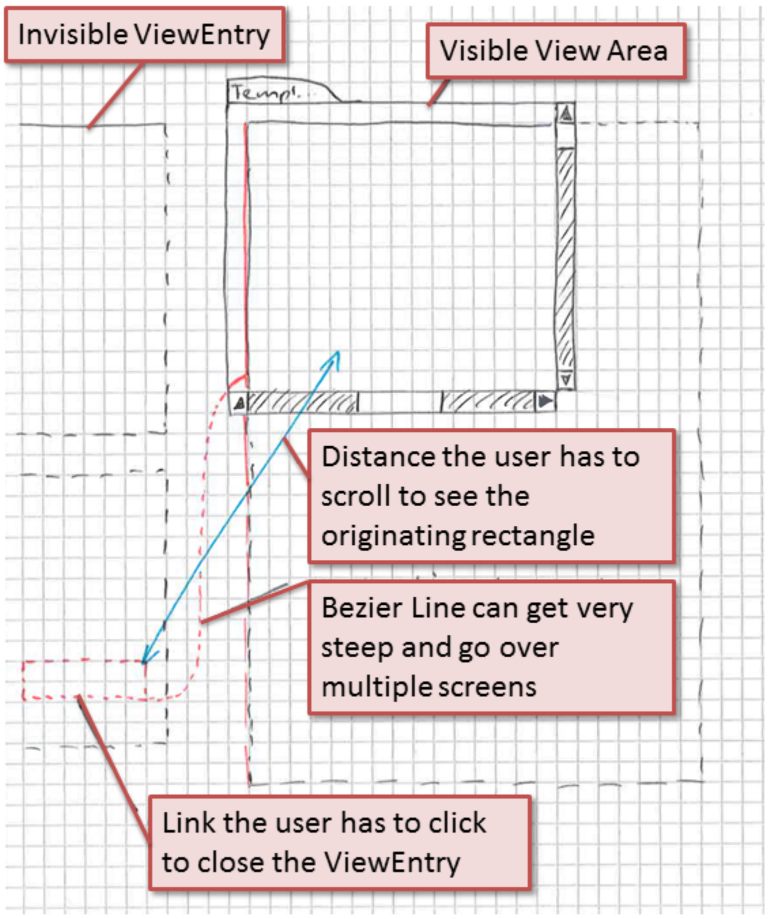

Invisible ViewEntry
Visible View Area
Templ.
Distance the user has to scroll to see the originating rectangle
Bezier Line can get very steep and go over multiple screens
Link the user has to click to close the ViewEntry

Figure 6.2: Illustration of the problem with big files that we encountered with the old version of view

So we had noticed that our view was working fine for small classes and functions but was not sufficient for big classes or functions. We now had to find a solution that was almost the same for small files but offered more flexibility and comfort for big files. This was due to the fact we wanted to keep the majority of the code from our old view. Since the code was already very clean we only wanted to rewrite a few of the existing classes. But at this point it was clear that we had to get our hands dirty in this semester thesis for UI. The usability of out plug-in was always a big focus for us next to the logical functionallity.

6.1.2 Paper Prototyping the View Concepts

We created a paper prototype that showed several concepts for the view. We discussed these proposals with Thomas Corbat. He agreed that the current version of our UI was not sufficient to offer an acceptable support for class templates. He preferred variant 2 of the sketches that are illustrated in Figure 6.3. He gave us green light to start working on a prototype. With a interactive prototype it would be a lot easier to imagine how the view would turn out.

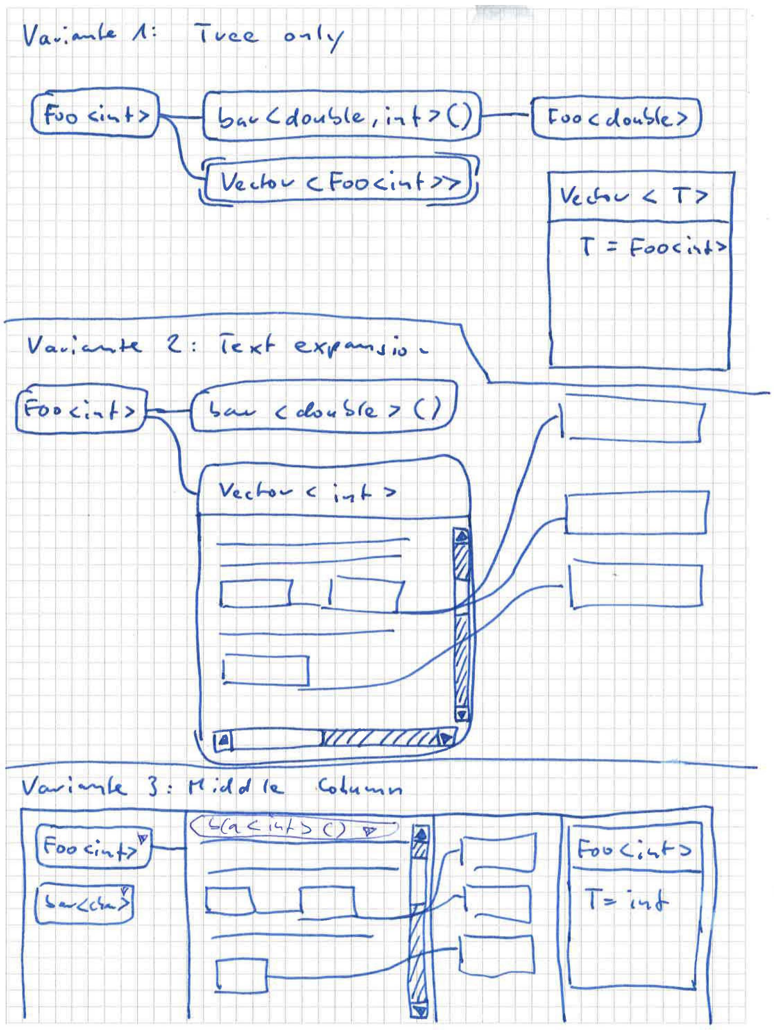
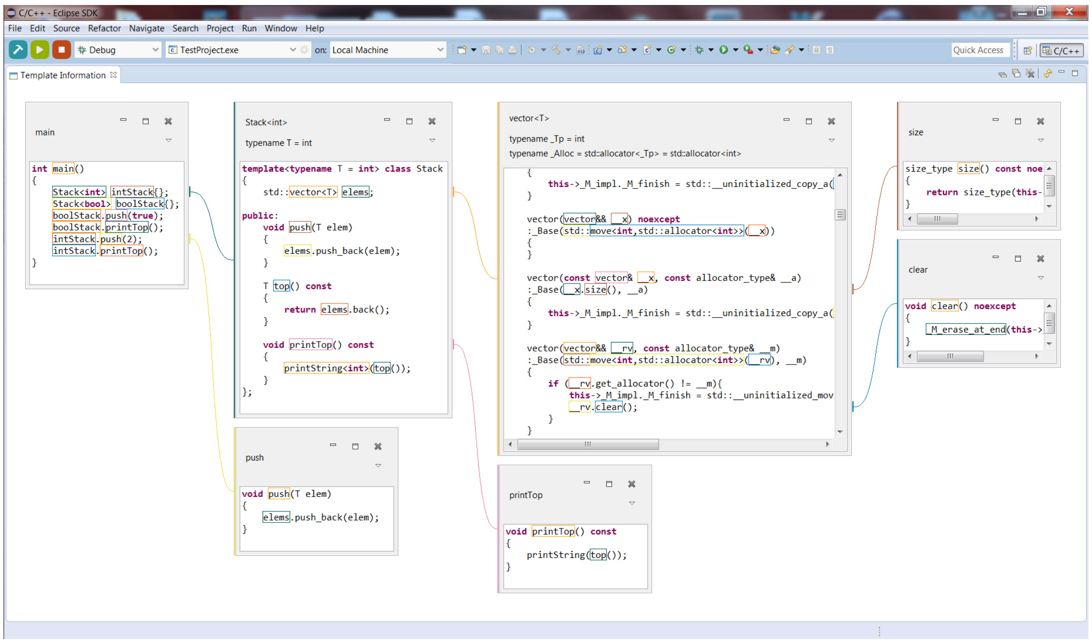


Figure 6.3: Paper Prototyping of UI improvement planning

Vision of the New View—Resizable Source Boxes

The vision for our new view was to visualize the code in small Boxes that are connected with Bézier lines. It must be possible to resize the boxes and add a simple way to minimize and maximize them. When all boxes are minimized, the visualization should look like a tree to give the user an overview of the call hierarchy. He then can further examine nodes that he is interessted in.



main

```
int main()
{
    Stack<int> intStack{};
    Stack<bool> boolStack{};
    boolStack.push(true);
    boolStack.printTop();
    intStack.push(2);
    intStack.printTop();
}
```

Stack<int>

typename T = int

```
template<typename T = int> class Stack
{
    std::vector<T> elems;

public:
    void push(T elem)
    {
        elems.push_back(elem);
    }

    T top() const
    {
        return elems.back();
    }

    void printTop() const
    {
        printString<int>(top());
    }
};
```

vector<T>

typename _Tp = int

typename _Alloc = std::allocator<_Tp> = std::allocator<int>

```
    {
        this->_M_impl._M_finish = std::__uninitialized_copy_a(
    }

    vector(vector&& __x) noexcept
    :_Base(std::move<int,std::allocator<int>>(__x))
    {
    }

    vector(const vector& __x, const allocator_type& __a)
    :_Base(__x.size(), __a)
    {
        this->_M_impl._M_finish = std::__uninitialized_copy_a(
    }

    vector(vector&& __rv, const allocator_type& __m)
    :_Base(std::move<int,std::allocator<int>>(__rv), __m)
    {
        if (__rv.get_allocator() != __m){
            this->_M_impl._M_finish = std::__uninitialized_mov
            __rv.clear();
        }
    }
```

size

```
size_type size() const noe
{
    return size_type(this-
}
```

clear

```
void clear() noexcept
{
    _M_erase_at_end(this->
}
```

push

```
void push(T elem)
{
    elems.push_back(elem);
}
```

printTop

```
void printTop() const
{
    printString(top());
}
```

Figure 6.4: An overview of the new view

In the next chapter we are discussing the concepts that we followed when we where implementing the new version of the UI. It also gives a brief overview about the process of the implementation.

6.2 The Evolution of the view

In this chapter we give a overview of the developement process of the new UI. We explain the evolution from the old to the new view and describe the design of the new view.

6.2.1 Problems with Old View

The top level layout is the outermost layout that defines how the components of the view are arranged and how the view reacts to scaling. Most applications use a layout manager like for example a grid layout for their top level layout. The advantage of this is that that the layout manager calculates the position and size of all components of the UI itself. Therefore layout managers are very easy to use.

But this simplicity comes with the cost of flexibility. Mainly the fact that the layout will resize all its content on its own makes it impossible to achieve the resizing feature that we where planing. Because of this we needed a new approach for the top level layout.

In our previous version of the view, we where using nested grid layouts to achieve the column layout, that is characteristical for our view. This solution was woking but very complex to implement. Also the realization of our very simple needs with an existing layout manager lead to a hudge overhead in code because the layout manager was not designed to solve our needs. And now that we additionally needed to have resizable components in the view, nested grid layouts where not an option anymore.

We also wanted to reuse most of our component classes. Most of them where offering a specific functionality that was needed for the new view as well. Since the majority of the components are encapsulated in small classes we could leave most of them the way they are because they where designed to be interchangable. They could be reused more or less directly in our new view. The remains where a few top level layout classes that we had to rewrite.

6.2.2 Developing a New Top Level Layout

After the first few weeks in this thesis we knew pretty well what we wanted to achieve with the new view. When we started with the implementation of the new view we concentrated on the most important part of the UI - the top level layout. As a first step we made it possible to places composites of different sizes on a `ScrolledForm` which we used as root composite. `ScrolledForm` is offered by SWT and has the advantage that is has scroll bars built-in. We did not have a layout manager for this form. We calculated the coordinates of all composites ourselves. This is what a layout manager does.

The piece of code that substitutes the layout manager is a relatively simple nested for loop that is shown in Listing 6.1. It iterates over all columns of entries that need to be shown (Line 5). It then iterates over all entries of this column (Line 9). For every entry it sets its position according to following rules:

- Left offset: Accumulated width of every previous column where the width is the width of the broadest entry in the column (Line 13-15).
- Top offset: Accumulated height of every entry in the current column. The top offset is reset to zero after every column (Line 6).

This assures that the columns do not overlap. Figure 6.5 shows a sample of how the view could look like with a few uneaven sized entries.

```
 1 public void recalculateLayout() {
 2     int currentLeft = BORDER_MARGIN;
 3 
 4     List<TreeSet<TreeEntry>> columns = entries.getColumns();
 5     for (TreeSet<TreeEntry> column : columns) {
 6         int maxWidth = 0;
 7         int currentTop = BORDER_MARGIN;
 8 
 9         for (TreeEntry entry : column) {
10             entry.setLocation(currentLeft, currentTop);
11             currentTop += entry.getSize().y + MARGIN;
12 
13             if (entry.getSize().x > maxWidth) {
14                 maxWidth = entry.getSize().x;
15             }
16         }
17         currentLeft += maxWidth + CONNECTION_COLUMN_WIDTH;
18     }
19 }
```

Listing 6.1: Entry coordinate calculation.

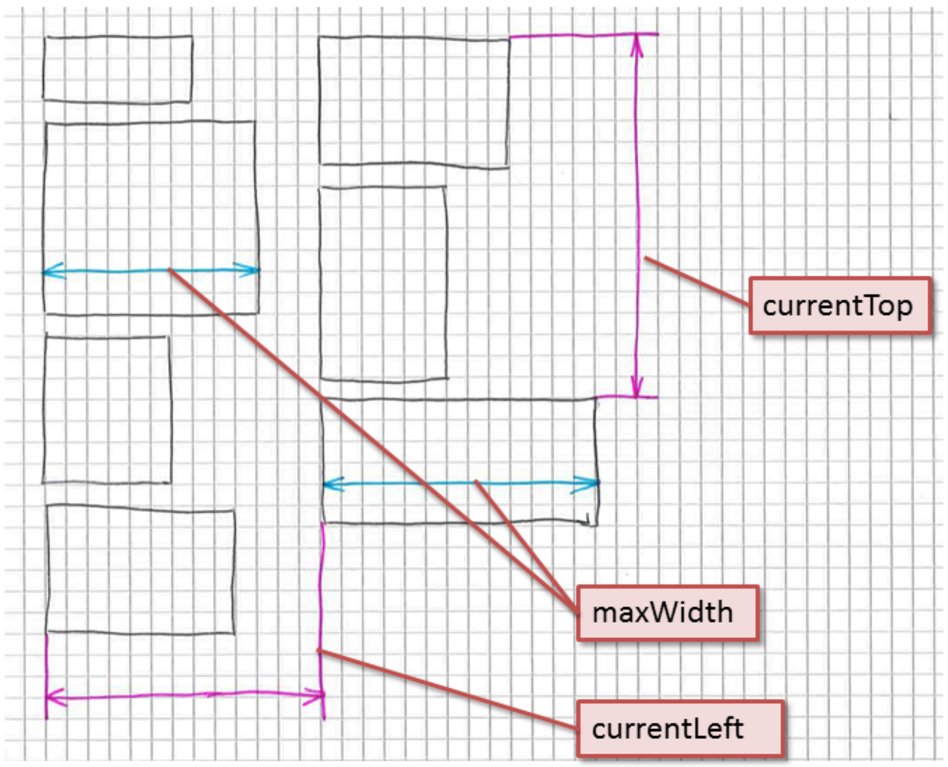


Figure 6.5: Visual explanation of the algorithm variables

Since this piece of code took care of almost all of our layout problems we could throw a big chunk of code away that was responsible to manage the top level layout in the old version of the view. We did not take this step immediately after implementing the new version as we explain in Subsection 6.2.3.

6.2.3 Keeping Support for Our Old Layout

To have as few changes as possible we first extended our plugin. We introduced the possibility to have multiple views on the same data and the view was easy to interchange. This facilitated us to work on the resolving and deduction of templates and on the new view at the same time because the old view was still fully operational. But we did not add simultaneous support for both views because eventually the old view was removed anyway. Instead we had one point in code where we could easily change between the two (Listing 6.2).

```java
public static void showTemplateInfoUnderCursor() {
     TreeTemplateView view =
    openView(TreeTemplateView.VIEW_ID);
    //TemplateView view = openView(TemplateView.VIEW_ID);
     initTemplateView(view);
}
```

Listing 6.2: Interchanging the view.

The possibility to make the view interchangable seemed like a good idea to us since it would later be possible to build a new view on the same data.

Originally we planed to keep the old view in the plugin. But this would have meant that we had to refactor several components to support both versions of the view. This would have lead a lot of gratuitous work.

Finally we decided to get rid of the old view completely and optimized the components for the new view.

6.2.4 Components of New View

In this setion several terms will be explained that are used frequently in the UI documentation.

This is a list of the most important components of the UI. Each one will now be explained in its own paragraph.

- View
- ViewEntry
- SubEntry
- Rectangle
- Link

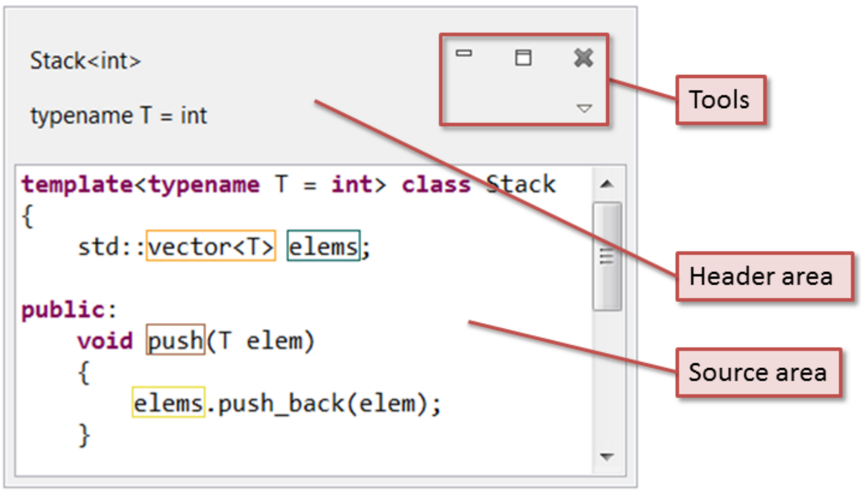
View

The view is the visual representation of our plugin. It is a tab in eclipse and can be arranged according to the affectation of the user. The view can be understood as an empty plane that can be filled with content (Figure 6.6).

Figure 6.6: The empty view

ViewEntry

A ViewEntry is one code box (Figure 6.7). The view typically consists of multiple ViewEntrys. These are the only components that our view can contain. The ViewEntry consists of a header area and a source area. The source area shows the pice of code the ViewEntry is representing. The header area displays additional information about the code and also several tools to assist the user. These tools also help the user to find out about actions he can perform with the ViewEntry.

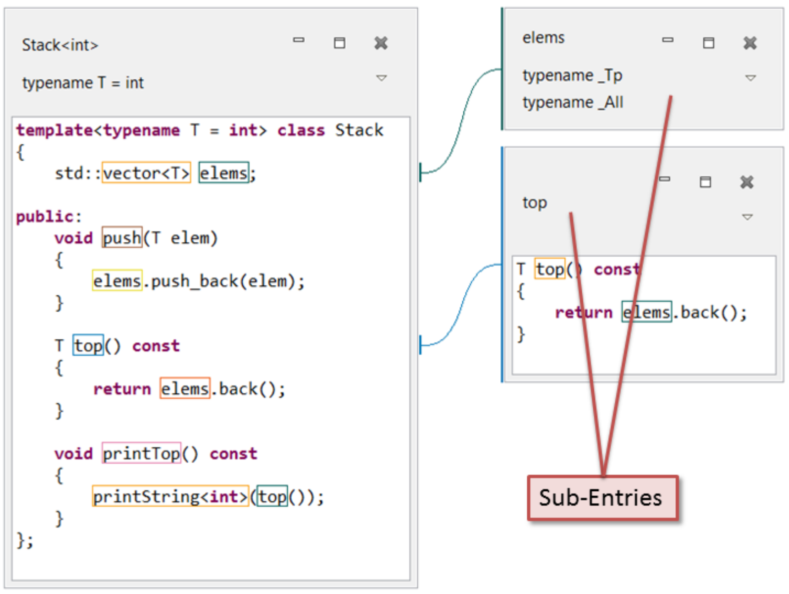


```
template<typename T = int> class Stack
{
    std::vector<T> elems;

public:
    void push(T elem)
    {
        elems.push_back(elem);
    }
```

Figure 6.7: A single ViewEntry

Sub Entry

Sub entires are also ViewEntrys. The difference is, that they belong to a parent ViewEntry (Figure 6.8). This means that they are hard wired with their parent. If a ViewEntry is closed all its sub entries are closed with it.



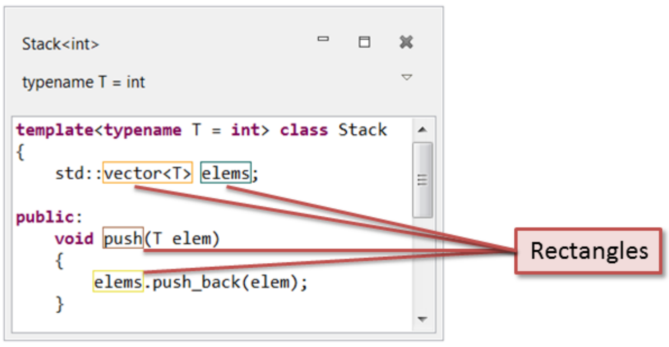
```
template<typename T = int> class Stack
{
    std::vector<T> elems;

public:
    void push(T elem)
    {
        elems.push_back(elem);
    }

    T top() const
    {
        return elems.back();
    }

    void printTop() const
    {
        printString<int>(top());
    }
};
```

```
T top() const
{
    return elems.back();
}
```

Figure 6.8: A minimized and a normal sub entry

Rectangle

Rectangles (Figure 6.9) mark text sections in the source code area that are interessting for the user. Their visible representation is a colored border around the text. They offer interactivity to the user and are therefor clickable.

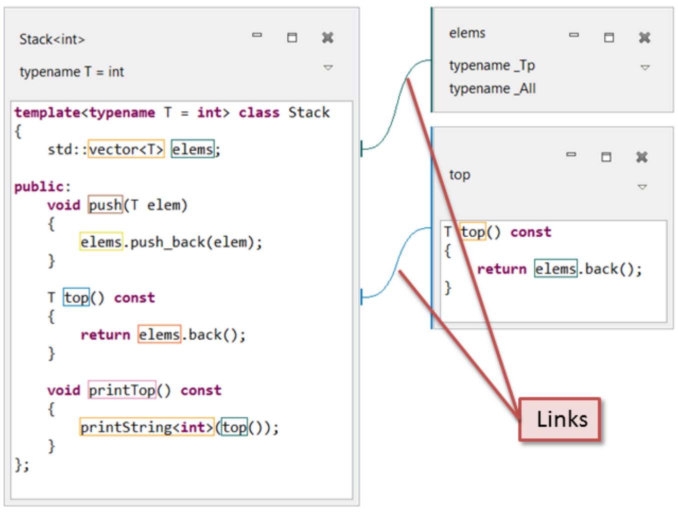


```
template<typename T = int> class Stack
{
    std::vector<T> elems;

public:
    void push(T elem)
    {
        elems.push_back(elem);
    }
```

Figure 6.9: Rectangles in a `ViewEntry`

Link

A link (Figure 6.10) is a connection between a `ViewEntry` and its sub entires. They are not interactive. They are often called Bézier lines. The vertical line at the origin of a link marks the line in the source code where the originating rectangle is located.



```
template<typename T = int> class Stack
{
    std::vector<T> elems;

public:
    void push(T elem)
    {
        elems.push_back(elem);
    }

    T top() const
    {
        return elems.back();
    }

    void printTop() const
    {
        printString<int>(top());
    }
};
```

```
T top() const
{
    return elems.back();
}
```

Figure 6.10: Links between a `ViewEntry` and its sub entries

6.2.5 Visual Improvements

Out goal was to make a plug-in that is visually appealing and comfortable to use. This implies that every action that the user can perform needs a visual feedback. This is the main reason an application is easy to learn und intuitive to understand. Visual feedback was sometimes difficult to achieve because it felt wrong when the look and feel differed to moch from a standard *Eclipse* plug-in. That is why we invented new visual concepts for some features.

The most obvious visual improvement are the scroll animations. When a new `ViewEntry` is opened the view smoothly scrolls to this entry opposed to just jump to it. This is important because jumping makes the user loose track of where his attention is in the view. This is the most important visual improvement.

6.2.6 Sorting of Columns

One major requirement for the view is the sortig of entries in one column. This was already disussed in our term thesis thesis ([BS14, 5.3.3, p.58]). The problem remained the same and since we had already solved it in our term thesis it was relatively simple to adapt the solution for the new view.

The sorting relies on a array of integers for each `ViewEntry` that represents its weight. The integers in this weight array are derived from the indices of all rectangles that where opened to reach the actual `ViewEntry`. This is shown in Figure 6.11.

We will explain the sorting with a simple example: The first entry (root entry) is alone in its column and therefore has a weight array of size 0 because it is not originating from a rectangle. Lets say the sub entry number 3 of the root entry was opened. The new `ViewEntry` will be added to column number 1 and therefor have a weight array of length one. The weight array contains the single number [3]. If then the sub entry number 4 of the just opened `ViewEntry` is opened it is added to column 2 and has a weight array of length 2 that contains [3,4]. The wight arrays are also shown in Figure 6.11.

It is simple to sort the `ViewEntry`s in one column with these weight arrays. To compare two `ViewEntry`s all the integers in the weight array are compared from left to right. As soon the weight integer differs the order of the `ViewEntry`s is defined. The smaller weight comes before the heigher weight.

This ordering technique also offers some nice side effects. The length of the weight array corresponds with the index of the column that contains the `ViewEntry`. The last weight integer in the weight array corresponds with the originating sub entry index in the parent `ViewEntry`. These side effects help to gain performance in the `ViewEntryCollection` that is described in Subsection 6.2.7.

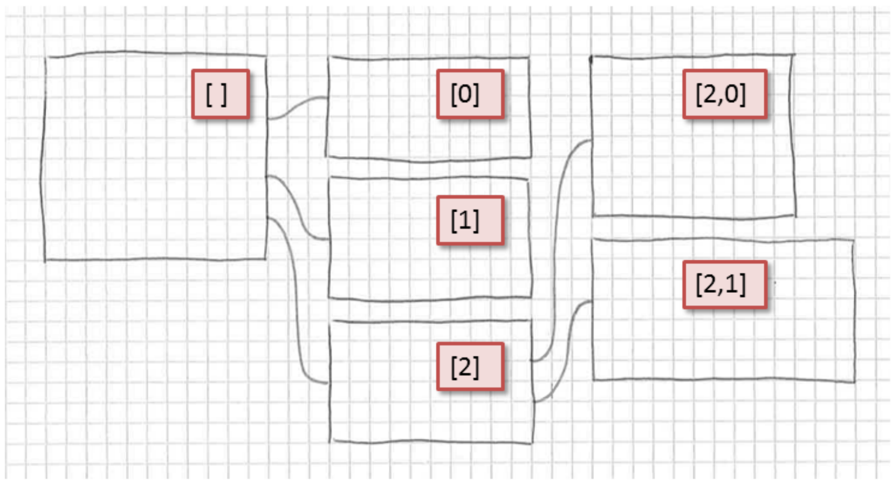


Figure 6.11: Weight arrays of `ViewEntry` that are used for ordering

6.2.7 Entry Collection

To keep the `TemplateView` class simple we introduced the `ViewEntryCollection` class that is a specialized collection to store `ViewEntrys`. It encapsulates the sorting code and offers a two dimensional list of `ViewEntrys` to the view (Listing 6.3). This list is already sorted and the view can diretly arrange the `ViewEntrys` in the view according to this list. The arranging algorithm is described in Listing 6.1.

```
public List<TreeSet<TreeEntry>> getEntries();
```

Listing 6.3: Get the `ViewEntry` list from the `ViewEntryCollection`.

The `ViewEntryCollection` also offers a set of simple functions to add and remove `ViewEntrys`.

Internally the `ViewEntryCollection` operates with `ViewEntryNodes`. These nodes encapsulate the tree like hierarchy of regular `ViewEntrys`. The `ViewEntry` itself has no knowledge of its ancester and children.

Since a `ViewEntryNode` knows its parent and children it is very simple to perform a recursive remove operation. This is neccesary because without a parent `ViewEntry` a sub entry cannot exist. That is why all sub entires have to be removed from the view as well when a `ViewEntry` is closed. The simplified removal algorithm is shown in Listing 6.4. This is the main purpose of the `ViewEntryNode` class.

```java
private void remove(ViewEntryNode node) {
    removeChildren(node);
    //destroy the node
}

private void removeChildren(ViewEntryNode node) {
    for (ViewEntryNode childNode : node.getChildren()) {
        remove(childNode);
    }
}
```

Listing 6.4: Recursive removal of `ViewEntryNode`.

6.3 New UI Features

After we have been working on the new view for about two weeks, we had all features from the old view integrated in the new view. At that time we had the base functionallity and could start adding new features to the UI.

6.3.1 Using CSourceEditor

Originally where where using a basic `StyledText` from SWT to display the code. The `StyledText` class allows to format the text that it is displaying.

We used this feature to implement syntax highlighting since it is a must have feature when displaying code. The problem with a native `StyledText` was that we had to use a regex to find the correct words to highlight. Also the user settings for the editor where not affecting our view.

We could solve this by using the *CDT* built-in class `CSourceEditor`. This class performed syntax highlighting on its own and all the user settings from the editor where taken into account automatically.

Although the `CSourceEditor` was causing two problems and one of them we could not solve.

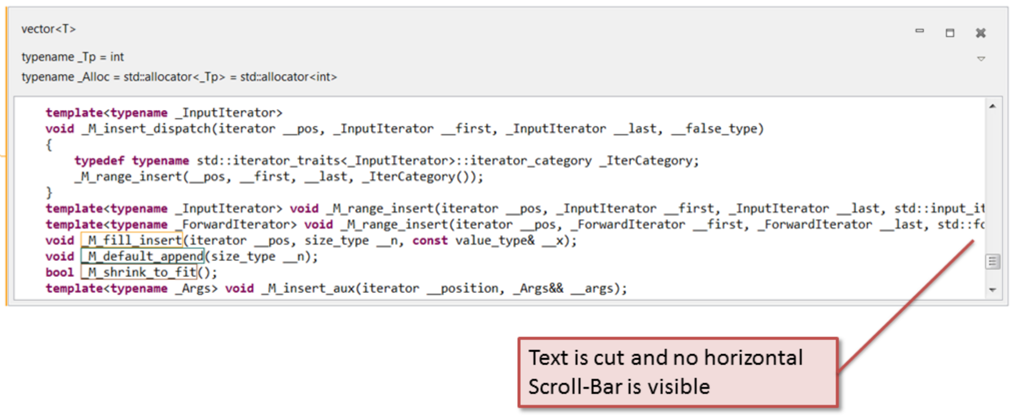
The first problem was caused by the built-in scroll bars of the `CSourceEditor`. The scroll offset of the built-in scroll bars was not added to the rectangle coordinates. Because of this the repainting of the rectangles did not works as expected. This could be seen as a weird effect where the rectangles position did not match the text.

Our solution for the first problem was to use a `ScrolledComposite` that encapsulated the `CSourceEditor`. The scroll bars of the `ScrolledComposite` where taken into account by the rectangle painter just fine. This structure was solving the rectangle paint problem.

But the second problem remained. Since the `CSourceEditor` had no scrollbars itself anymore, it was messing up the calculation of the maximal line length. Then it was cutting all code that is longer than this maximal line length (Figure 6.12). We could not find out how the `CSourceEditor` was calculating this maximal line length. It seemed that it was only taking the first few lines into account because long lines at the beginning of a piece of code are working just fine.

Since files rarely have such long lines, we decided leave this bug in the plug-in. For normal use, this should never be a problem. We also did not want to waste to much time researching this issue since it was happening very seldom.

Also the user still has the possibility to resize the `ViewEntry` to an extent that he can see the cut code.


vector<T>

typename _Tp = int
typename _Alloc = std::allocator<_Tp> = std::allocator<int>

```
template<typename _InputIterator>
void _M_insert_dispatch(iterator __pos, _InputIterator __first, _InputIterator __last, __false_type)
{
    typedef typename std::iterator_traits<_InputIterator>::iterator_category _IterCategory;
    _M_range_insert(__pos, __first, __last, _IterCategory());
}
template<typename _InputIterator> void _M_range_insert(iterator __pos, _InputIterator __first, _InputIterator __last, std::input_it
template<typename _ForwardIterator> void _M_range_insert(iterator __pos, _ForwardIterator __first, _ForwardIterator __last, std::fo
void _M_fill_insert(iterator __pos, size_type __n, const value_type& __x);
void _M_default_append(size_type __n);
bool _M_shrink_to_fit();
template<typename _Args> void _M_insert_aux(iterator __position, _Args&& __args);
```

Figure 6.12: Text is cut after a certain line length

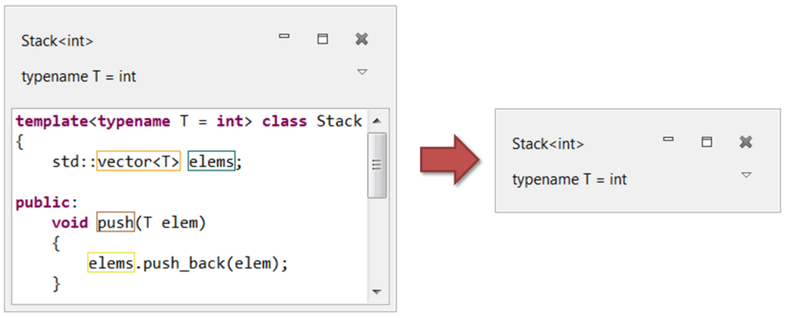
6.3.2 ActionButtons

Every `ViewEntry` has a set of `ActionButtons`. They offer the possibility to minimize, maximize and close a `ViewEntry`.

Or intention was to give the user the functionality that he is accustomed to from the Microsoft and Apple operating systems. We tried to make the `ActionButtons` as similar to those operating systems as we could. In the next paragraphs we will explain the difference to the regular behaviour.

The close button works straight forward. It closes the `ViewEntry`. It only closes the visual representation of a `ViewEntry` but not its underlying Data. This has the effect that opening an entry for the second time is significantly faster.

The minimize button resizes an entry to the smallest possible size. On a minimized entry only the header is visible. The user can still see the name of the entry but the code is hidden. We decided to implemented this in a very simple way. Upon pressing the minimize button the `ViewEntry` changes its size to the minimal size it can be. The outcome is the same as if the user would have resized the `ViewEntry` with the mouse. This means that it is still possible to resize the `ViewEntry` with the mouse when it is minimized. We found a more complicated solution unnecessary because we would have had to introduce a special mode for a minimized `ViewEntry`. It would have been

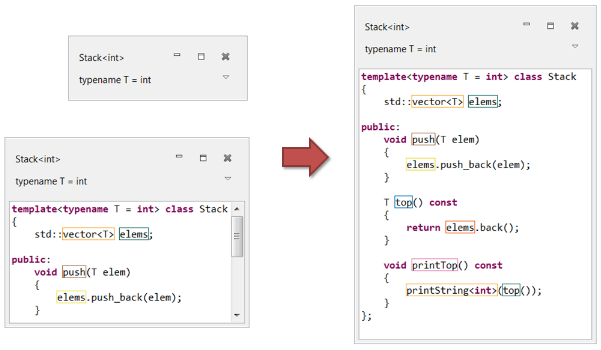


```
template<typename T = int> class Stack
{
    std::vector<T> elems;

public:
    void push(T elem)
    {
        elems.push_back(elem);
    }
```

Figure 6.13: Minimizing a `ViewEntry`

complicated to add and remove the source area and therefor we implemented the simple solution. In the end, both solutions would have looked the same anyway but the simple solution also offers more flexbility. The outcome of a minimize operation is shown in Figure 6.13.

The maximize button changes the size of the `ViewEntry` to its optimal size. If possible the whole code is visible. The optimal size of the `ViewEntry` is determined by the optimal size of the source area and the size of the header area. If the code is very large the size caps at the default size. In this case, the scrollbars are shown. When the `ViewEntry` is maximized the user can still increase its size manually. The outcome of a maximize operation is shown in Figure 6.14.



```
template<typename T = int> class Stack
{
    std::vector<T> elems;

public:
    void push(T elem)
    {
        elems.push_back(elem);
    }
```

```
template<typename T = int> class Stack
{
    std::vector<T> elems;

public:
    void push(T elem)
    {
        elems.push_back(elem);
    }

    T top() const
    {
        return elems.back();
    }

    void printTop() const
    {
        printString<int>(top());
    }
};
```

Figure 6.14: Maximizing a `ViewEntry`

Look and Feel of ActionButtons

Our goal was to make the `ActionButton`s as similar to the original *Eclipse* minimize and maximize buttons as possible. The problem was that the standard SWT buttons have a completly different look and feel. They are permanently visible and have the three different modes: normal, hover and pressed. The *Eclipse* minimize and maximize buttons however only have their outline visible when the user hovers them with the mouse. Any other time, only the icon without the outline is visible. This makes the buttons look much more lightweight.

We wanted to have this effect for our `ActionButton`s as well. But SWT is not offering a native solution for this button style. This is why we had to come up with a solution our own. It was also not possible to just use the *Eclipse* solution. The *Eclipse* minimize and maximize buttons are tightly integrated in the *Eclipse* view. They do not offer a way to use them outside a view.

In the end we found a tricky solution that resulted in the look end feel for the `ActionButton`s we where aiming for. Per default we hid all `ActionButtons`. On the same location as the button we just paint the exact same icon that is shown on the button. Then we introduced a `MouseMoveListener` to determin if the Mouse was entering the Area where the invisible button is. When that happens we just make the button visible. The visible button then is covering the icon that is painted below it. It is important that the position of the painted icon and the icon on the button match exactly. For the user, it is looking that just the outline of the button is appearing when he hovers the button. Finally when the mouse is leaving the area of the button, it has to be set invisible again. This could be done with a `MouseTrackListener` that fires an event when the mouse is leaving the button area. Sadly the mouse track listener is only working for the exit event since it does not fire the enter event on a invisible button. The outcome of our solution is shown in Figure 6.15.

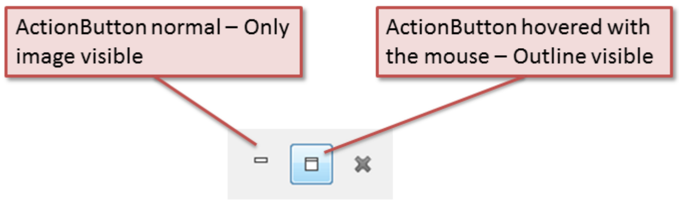

ActionButton normal – Only image visible
ActionButton hovered with the mouse – Outline visible

Figure 6.15: Different modes of the `ActionButtons`

This solution lead to a small problem that we did not solve because it is not occuring very often and it is not disturbing the user experience. If the view is scrolled automatically and the cursor lands on a `ActionButton`, it is not made visible. The reason for this is that the enter event is only fired when the mouse is moved over the button.

This problem also exists for the drop down menu. When the user clicks the menu button, the menu it is shown below it. This is also the way it looks in *Eclipse*. If the user does not select a menu option and closes the menu by clicking on the exact same spot again, the button is not shown again. It stays hidden until the mouse is moved again. Becase this is a minor issue that are not crucial for the user experience, but would take a lot of effort to solve, we did not solve it.

ActionButtons Dropdown Menu

To offer some additional functionallity for every `ViewEntry` we added one additional `ActionButton` that opens a dropdown menu. It offers further options for the corresponding `ViewEntry` (Figure 6.16). The menu makes it possible to add an arbitary number of additional actions for every `ViewEntry`. This menu is the only possibility to perform actions on the `ViewEntry` itself.

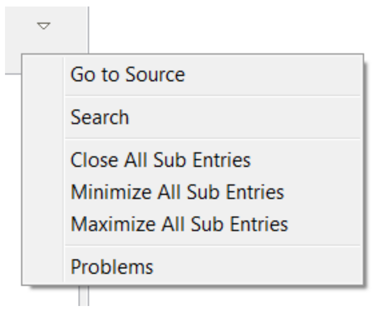


Figure 6.16: DropdownMenu of the `ActionButtons`

Since it is good practice, we made all possible UI actions avalible in dropdown menu.

The most important option is the navigate to source option. Until now it was only possible to navigate to sub entries via their rectangles. This was very inconvenient in the old view since the user had to find the originating rectangle to navigate to the source.

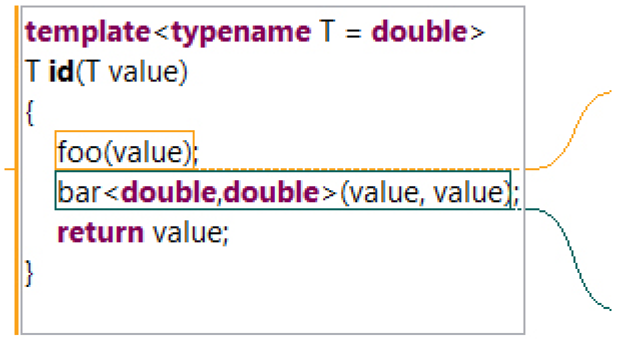
The search option displays the search bar for the `ViewEntry`. Another possibility to display it is the shortcut Crtl-F. The search bar is explained in Subsection 6.3.11.

The other dropdown commands are pretty self explaning: Close, minimize and maximize all sub entries. Minimizing and maximizing of `ViewEntrys` is explained in Subsection 6.3.4.

6.3.3 Improved Bézier Lines

The Bézier lines where already feature of the old view. But because of the big class problem that is explained in Subsection 6.2.1 they where not particulaty useful in the old version of the view. They would extend over multiple screens and the origin was not easy to find. In our new view this problem solved itself because the `ViewEntrys` are a lot smaller by default and because of that the Bézier line look a lot more natural.

In our term thesis we had the promlem that the Bézier lines started outside of the view entry. This made it hard to tell from wich rectangle the Bézier line originated. Back then we connected the rectangle with a dotted line with the bezier line. This line went horizontlly through the whole `ViewEntry` (Figure 6.17). This was looking very ugly

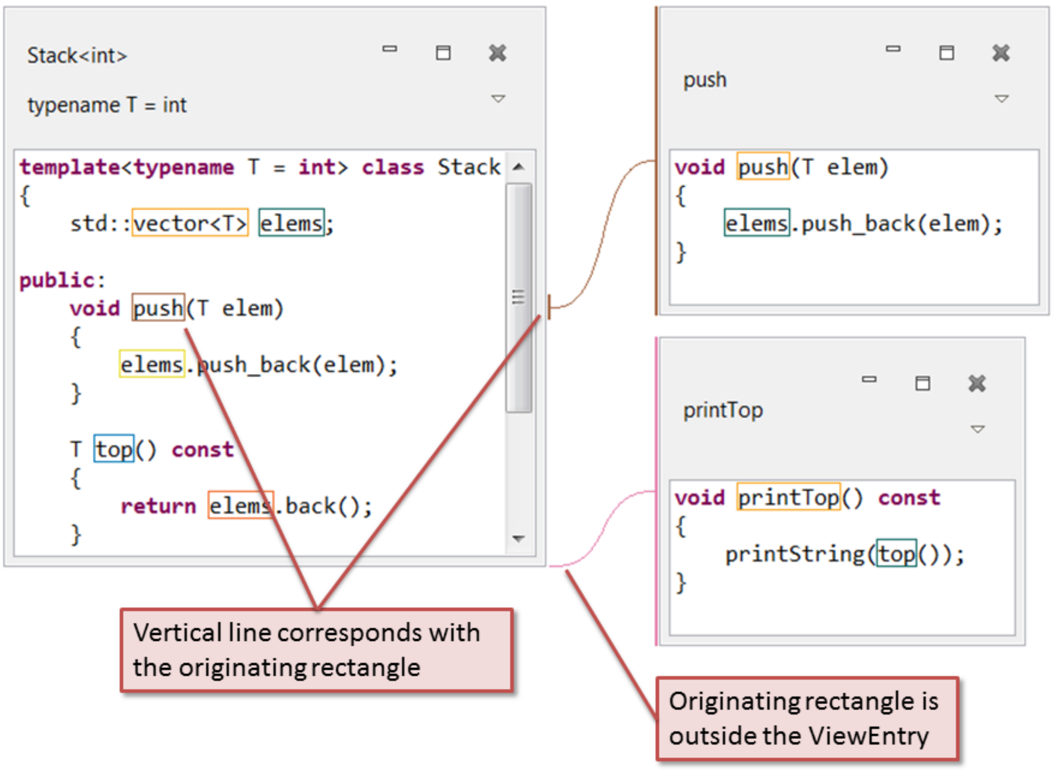


```
template<typename T = double>
T id(T value)
{
    foo(value);
    bar<double,double>(value, value);
    return value;
}
```

Figure 6.17: Old unreadable version of the horizontal Bézier line connections

and also not very understandable . In the case where alot of lines went through the `ViewEntry`, they where crowding the code.

In this semester thesis a classmate had an awsome idea when we showed him our plugin. Because we are displaying vertical lines to indicate the destination view entry of a open link he mentioned that he was missing this lines on the originating side of the link. That brought us to the idea to mark the link on the originating side also with a vertical line. This apporach is very readable and looks very nice (Figure 6.18).



Stack<int>
typename T = int
push
printTop
Vertical line corresponds with the originating rectangle
Originating rectangle is outside the ViewEntry

```
template<typename T = int> class Stack
{
    std::vector<T> elems;

public:
    void push(T elem)
    {
        elems.push_back(elem);
    }

    T top() const
    {
        return elems.back();
    }
```

```
void push(T elem)
{
    elems.push_back(elem);
}
```

```
void printTop() const
{
    printString(top());
}
```

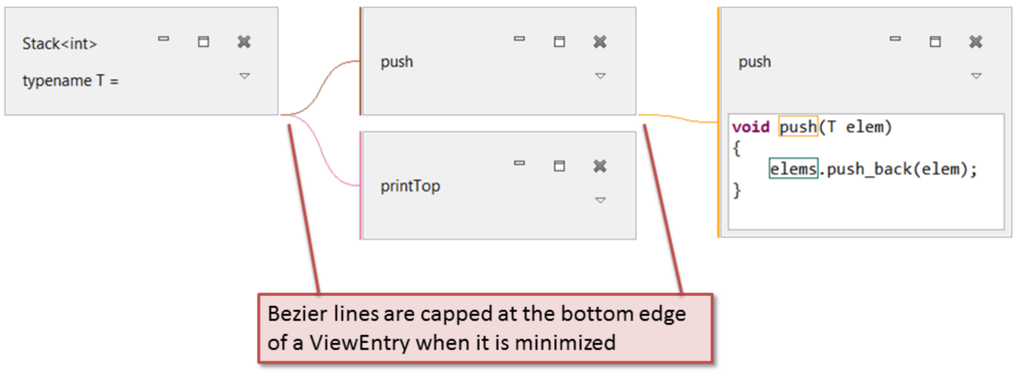
Figure 6.18: New version of the Bézier lines

To improve the look and feel of the view we also added the ability for the Bézier lines to change their start and end point dynamically. This is something that the user is expecting because the resizing of a `ViewEntry` would change origin and destination points of the Bézier lines. To make this possible, we have to repaint the whole view when a `ViewEntry` is resized. This leads to a slight performance loss, because of the increased number of drawcalls.

The dynamic repainting of the Bézier lines also caused another problem. The lines started to flicker because the number of the high number of draw calls. The solution for this was very simple because SWT offers the possibility to make a component double buffered. With bouble buffering enabled, all the Bézier lines are painted on a offscreen image and is drawn to the screen all at once. This made the flickering vanish entirely.

Scrolling the Bézier Lines

Because in the new version of the view, a `ViewEntry` could be scrolled we had to implement another feature for the Bézier lines. Now it was possible that the origin of a Bézier line would not always be inside the visible section of the source area of a `ViewEntry`. We had to improve the rendering algorithm to support the case when the origin of the Bézier line was off screen. We solved this i a very natural way. When the line start was scrolled out of the `ViewEntry` we capped it on the top or bottom edge of the `ViewEntry`. This also lead to the effect that when a `ViewEntry` is minimized, all Bézier lines start at its bottom edge (Figure 6.19). We decided to leave it this way because it was looking good and it was the native solution.


Stack<int>
typename T =
push
printTop
push
Bezier lines are capped at the bottom edge of a ViewEntry when it is minimized

```
void push(T elem)
{
    elems.push_back(elem);
}
```

Figure 6.19: Bezier lines for minimized `ViewEntry`

6.3.4 Resizing ViewEntries

Resizing is the core feature of the new view. Resizing caused one show stopping issue that we had to solve to make the view vaiable. The problem occured when the user wanted to resize one of the outermost `ViewEntry`. With the native SWT resizing solution this problem was impossible to solve.

The first implementation of the resize feature was relying on the native SWT built in resize functionality. when the user was holding down the mouse button anywhere on the border of the `ViewEntry`, the resizing gizmo was shown. This gizmo had the shape of a rectangle and the size of the `ViewEntry` that was resized. The user could move the mouse to change the size of the gizmo. When the mouse was released, the size of the resizing gizmo was set to the `ViewEntry`. This can be seen in Figure 6.20. With this resizing functionality it was not possible to resize a `ViewEntry` over the screen edge.

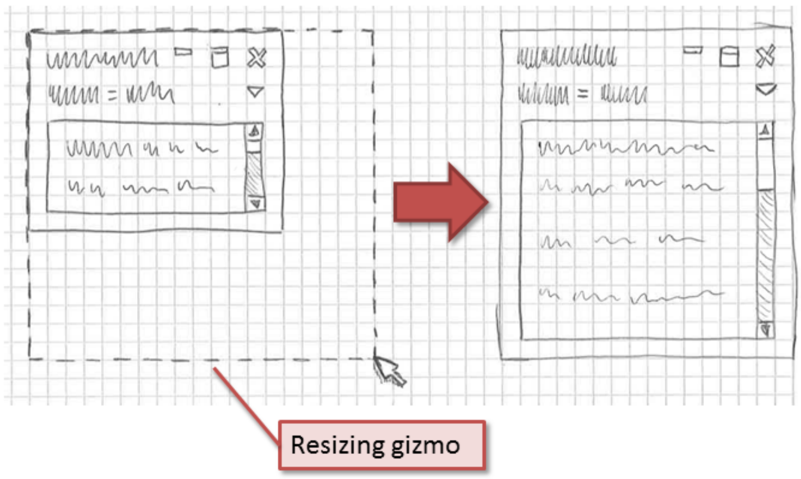


Figure 6.20: SWT built in resizing functionality

Another akward thing about the SWT resize feature is, that after the mouse is pressed to start the resize, the mouse cursor jumps to a corner of the `ViewEntry`. The corner is determined by first mouse movement after the mouse was pressed. For example if the user was pressing on the top left corner and then moved the mouse down, the mouse was jumping accross the whole `ViewEntry` to the bottom right corner. This behaviour felt very strange.

The most simple way to solve these problems was to implement the resize feature completely ourself. This was possible with only a few lines of code. With our own resize functionality we gained alot of flexibility that we needed to implement the crucial 'resize a `ViewEntry` over the border' feature.

The outcome of our own resizing feature is very nice. It behaves exactly as the user would expect it. The user can now start a resizing operation by clicking the right or bottom edge of the `ViewEntry`. The mouse is not jumping anymore and the cursor is changing according to the resize operation that is possible.

With the new resize feature it was now possible to drag the edge of a `ViewEntry` over the screen edge. As soon as the cursor approaches the edge of the screen, the size of the view is smoothly extended and the `ViewEntry` is resized. This lead to a very natural user experience for resizing that the user might know from other programs.

We added another resizing feature that is very convenient for the user. He can double click on the border or the header of the `ViewEntry`. This action has the same effect as the maximize `ActionButton`.

Introducing a Minimal Size

The most challenging feature of the new view was to introduce a minimal size for a `ViewEntry`. This was because it was interfering with alot of other areas of the UI. When we finally had it implemented we saw the problems it caused.

The motivation to introduce a minimal size in the first place was because when the user can freely change the size of a `ViewEntry`, nothing would keep him from reducing the size to zero. With a size of zero the `ViewEntry` was invisible. In that state it could not be closed or resized again because it was not clickable anymore.

In the first approach to implement the resize feature we changed the size according to the relative move of the mouse. This approach lead to a lot of variables to store all the relevant data like the initial mouse position when the resize started, the location of the entry, the origin of the view and so on. The crux was that we had calculated all coordinated relative to the screen instead of relative to the view what made the calculations very complex. The code got very messy what propably would have lead to a lot of bugs and problems. At this point it was easier to throw away this big ball of mud and implement a clean solution. The clean solution was done very quickly because during the developement the resize feature all its requirements became very clear.

In the final solution of the resize feature we followed an approach that was possible because of a nice effect of SWT's `MouseListener` and `MouseMotionListener`. When the resize operation was started, the mouse down event was fired. After this event the mouse move event was fired as long as the mouse button was held down. On releasing the mouse button, the mouse up event was fired. The most important thing here was, that the events kept firing after the mouse down event regardless of the mouse leaving the Entry or not. This was exactly what we needed, because now we had absolute coodinates of the mouse relative to the `ViewEntry`. This coordinates now exactly matched the new size of the `ViewEntry` without any calculation. Another advantage of this approach was that the coordinates could be capped easily to minimal size.

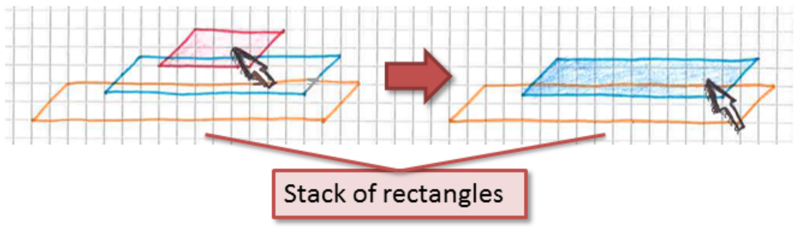


Figure 6.21: Ordering of overlapping rectangles: If the mouse leaves the innermost rectangle the next rectangle on the stack is highlighted

6.3.5 Highlighting Rectangles

Our advisor Prof. Peter Sommerlad had the idea to make the hovered rectangles much more visible by highlighting their background in the color of their borders. Although it is a simple feature we experienced some issues during the implementation. The reason for the issues was the `CSourceViewer` that we use to show the code. The `CSourceViewer` is relying on the SWT component `StyledText` to alter the font and do syntax highlingting according to the user preferences. The `CSourceViewer` internally uses `StyleRanges` to accomplish syntax highlighting. This is why we could not just create a new `StyleRange` for the background since this would have overwritten the syntax highlighting.

To solve this we had to store all existing `StyleRange`s so we would later be able to reset them. Then to change the background color, we could take the original `StyleRange`, alter it as needed and set it again for the correct text range.

So now, when the mouse is entering a rectangle, we copy the original style, alter it and set it again. When the mouse is leaving again, we set the original style again.

There was one additional problem that we encountered. We also had nested rectangles and their `StyleRange`s would overlap. So if we directly changed between nested rectangles the style of the outer rectangle was overriding the inner style and on mouse exit, the inner style was not correctly reset.

We solved this by introducing a stack like collection that kept track of all currently colored rectangles that where overlapping. When a rectangle was left, we could then find out what rectangle was lying under it and recolor it again. This is shown in Figure 6.21.

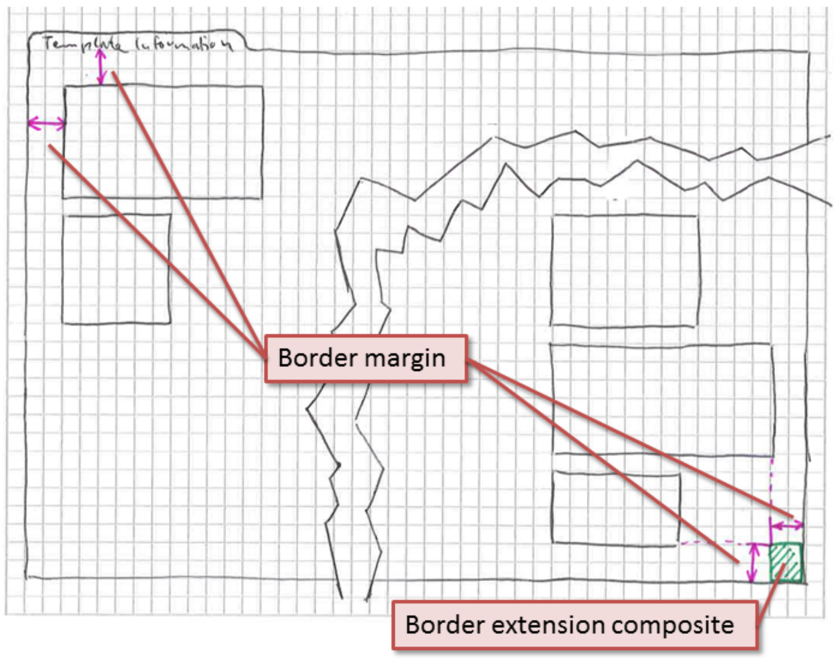


Figure 6.22: Border margin at the edge of the view

6.3.6 Border Margin

Originally, the view was over right behind the last `ViewEntry`. This made it not very comfortble to resize `ViewEntrys` when they where sticking at the edge of the view. A simple but very helpful improvement was to introduce a small border at the edge of the view so `ViewEntrys` at the edge could be resized easier. For the top and left margin it was very simple. All we had to do was to add the border width to the calculated coordinates of the `ViewEntrys`. For the right and bottom margin it was a little bit more complicated because the view calculates its size to fit its content exactly. Because of this we added a invisible composite with the shape of a quare and the size of the border margin. It was placed behind the last enties as shown in Figure 6.22.

6.3.7 Mouse Wheel Global Scrolling

Originally it was only possible to scroll the code inside a `ViewEntry` with the mouse wheel and the trackpad. Since we where really missing the possibility to scroll the whole view, we made it possible that the user can click on the empty space in the view and when scroll the whole view with the mouse wheel.

We also added a nifty little feature where the user can hold down the Ctrl Button to scroll the view horizontally. This might not be very intuitive for the user to find out, but it is a very helpful feature since it it often necessary to scroll the view horizontally. Sadly we could not test this feature on an Apple computer.

When we tested horizontal scrolling on a Linux operating system, we discovered that pressing Ctrl does not disable the standard scrolling behaviour. This has the efect so the view is scrolled in both directions vertical and horizontal at the same time. This makes the feature useless on Linux operating systems. We did not found out why this is happening but it appears that SWT behaves different on different opperating systems in this area.

6.3.8 Scrolling Labels

For testing new UI features we always used small test classes. When we then tested the UI with a large class with a very long template argument we found an annoying bug that we had to solve. If the text in header labels was very long, the `ActionButton`s disappeared because the where outside the `ViewEntry`. Fixing this was a lot of work.

The bug is caused by SWT's `GridLayout`. The user would expect the `ActionButton`s to be on the right side of the header area and always visible. To accomplish that we put them in the second column of the grid layout and the labels for title and description in the first column. Upon resizing, the rightmost column disappears first. This means that as soon as the width of the `ViewEntry` was smaller then the width of the title label, the `ActionButton`s in the second column would disappear. This was very inconvenient because the user would expect the `ActionButton`s to be always visible. Especially because the template arguments are often very long the `ActionButton`s disappeared very frequently. This problem is shown in Figure 6.23.

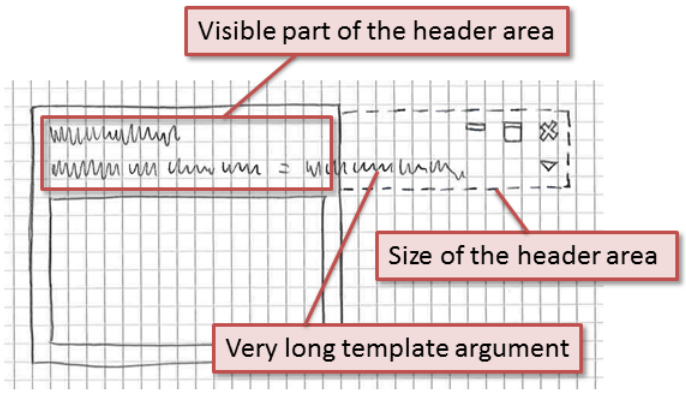


Figure 6.23: `ActionButtons` are not visible because the label text is too long

We now had a dilemma that there is no perfect solution for this problem. It was clear that the `ActionButtons` need to be always visible, but this meant, that the label had to be cut in some way. It would have been confusing to indicate it with three dots (...) to the user that the text is cut because that could have been mistaken for variadic templates. We decided to just cut the text. Since there is no perfect solution for this problem anyway, we found that this is the best compromise.

Now that we needed to make the label smaller we encountered another problem. Usually a label in a layout would always adjust its size according to its content. Now since the labels must be smaller so the `ActionButtons` would stay visible, we needed to find a way to make the label smaller. The solution for this was to calculate the optimal width that label would have if all text was displayed. This width was in proportion to the full number of characters of the full length text. We could calculate the width that the label needs to have so the `ActionButtons` would stay visible. With these results we then could calculate the number of chars that must be displayed so the label would have the correct size. This lead to the simple equation:

$$\frac{optimalWidth}{neededWidth} = \frac{totalNumberOfChars}{neededNumberOfChars}$$

With this simple equation it was possible to calculate the number of chars that needed to be displayed in the Label so the width would be correct. So the label would store the original string but only the calculated `neededNumberOfChars` is shown (Figure 6.24).

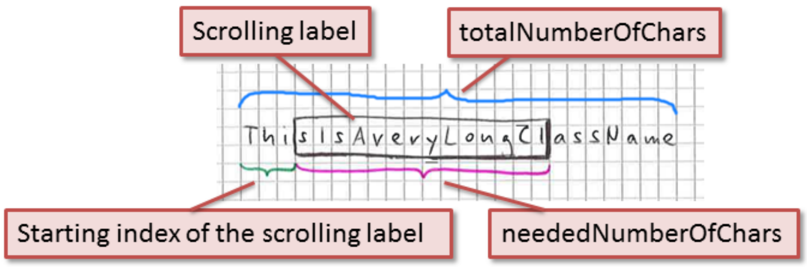


Figure 6.24: Only a portion of the text is visible on the label

To further assist the user we implemented the feature that gave the labels their name: `ScrolligLabel`. Since we had the full text stored, it was simple to adjust the starting index of the displayed text so the user can scroll through the text. We only needed a way to make this interactive. To achive that we implemented a `MouseListener` that reacted on mouse clicks and a `MouseMoveListener` that reacted on mouse movement. When the user had clicked the label the text scrolled according to the mouse movement. With the mouse movement and a scroll speed, the new staring index of the displayed text could be calculated. This made it possible for the user to scroll through the text.

One problem was still remaining. No user would ever try to click and drag the label. To conquer this we abused the mouse cursor. When the user hovers over a `ScrollingLabel` with the mouse (In case the text is cut) the cursor changes to an open hand to indicate that the mouse can be presses. If the user then presses the left mouse button, the mouse cursor changes to a closed hand to indicate that the user can drag the text.

One thing is still a little bit awkward with this solution. The user will always try to drag the mouse vertical and not horizontal as inteded. Dragging vertically will have no effect and this will appear like a bug to the user. We hope that the user will find out about the feature himself sooner or later.

6.3.9 Global Toolbar

As another helpful feature we added some buttons to the global toolbar. The advantage of this is that those buttons do not occupy space in the view because they reside in the tab bar of the view.

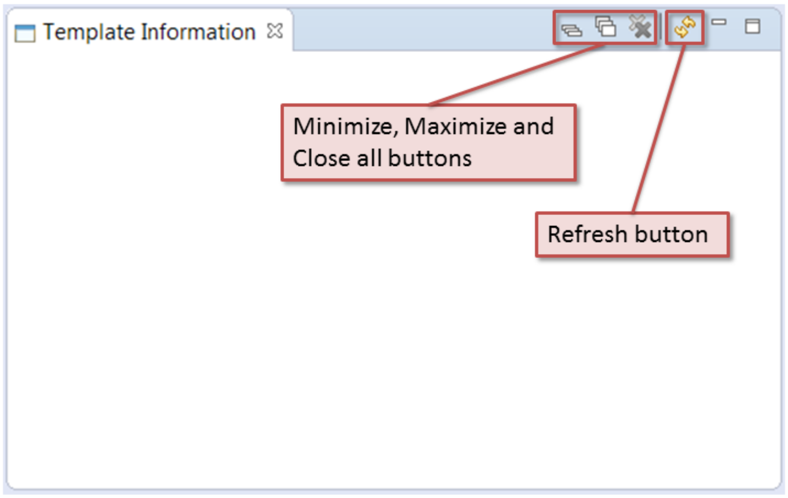


Figure 6.25: Buttons in the global toolbar

The buttons offer the possibility to Minimize/Maximize/Close all `ViewEntry`. We also added a refresh button that loads the view with the currently selected name in the editor again. This is also helpful for a new user to find out that he can open template dependent names with the Templator2.

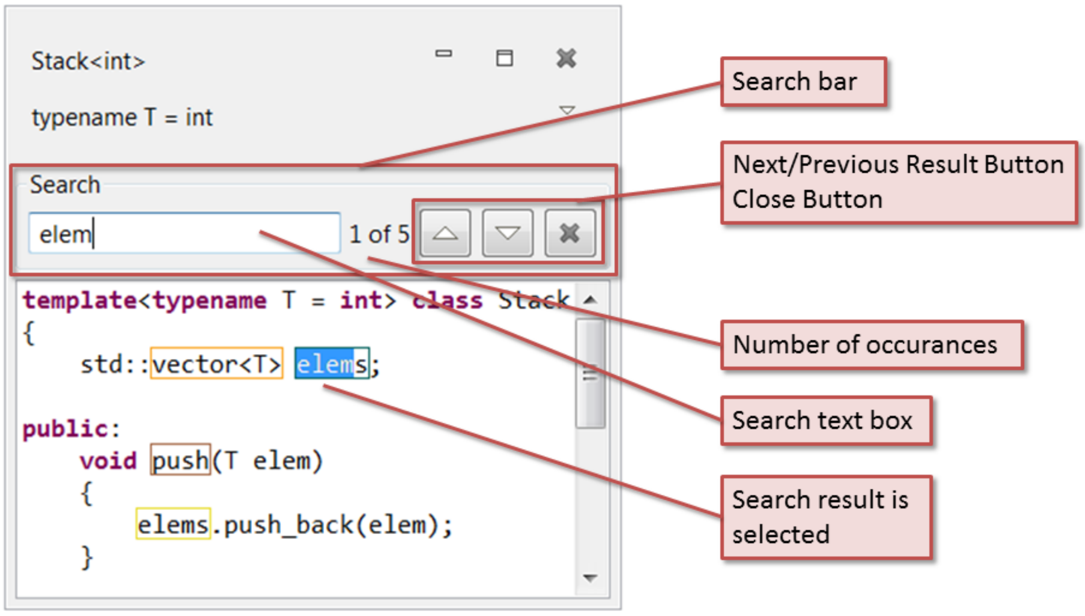
6.3.10 Auto Scrolling After Opening a New ViewEntry

One feature that was on our list for a very long time was auto scrolling. Until now the user had nearly no feedback where a newly opened `ViewEntry` was located. The only thing that happened was a change in in scroll bar selection. This was almost not noticeable.

It was therefore mandatory that we implemented a scrolling mechanism that navigated the view to a newly opened `ViewEntry`. If the view would have just jumped to the new `ViewEntry` in an instant, the user would have lost track of where he is in the view. To tackle this we introduced a scrolling animation that was slowly sliding the view to the newly opened `ViewEntry` in a short timespan. We also added a bit of acceleration the scroll animation to give it a more natural feeling and also that the user does not have to wait all that long for the animation to finish.

6.3.11 Searching in a ViewEntry

As a final feature we implemented the possibility for the user to perform a text search in a `ViewEntry`. There are literaly thousands of possibilities of how to realize the search. We decided to make it similar to the search in the Chrome browser. The search bar can be seen in Figure 6.26.



```
template<typename T = int> class Stack
{
    std::vector<T> elems;

public:
    void push(T elem)
    {
        elems.push_back(elem);
    }
```

Figure 6.26: Components of the `SearchBar`

Showing and Hiding the Search Bar

To begin, we needed to be able to show and hide the `SearchBar`. The most natural way seemed to show it below the header area and above the source area. The problem with this was that it is not possible to set the size of a composite in a GridLayout to zero. This means that we could not make the `SearchBar` invisible by changing its size. The solution is to create a new `SearchBar` anytime it is shown. After the creation of the `SearchBar` it needs to be moved up in the rendering order so it is drawn between the header area and the source area.

We offer two ways of displaing the `SearchBar`. The first os obviously the Ctrl+F shortcut. For the shortcut to work the cursor has to be in the source area of the `ViewEntry`. If the `SearchBar` is invisible and the shortcut is pressed, it is shown. When it is already visible, the focus is set to the search text field in the search bar. The second way to show the search bar is to access it via the ViewEntry Context menu.

The Search bar can be closed either by pressing ESC or the X button on the search bar.

How the search works

Whenever the user alters the text in the search text field, the whole source area is searched for that text.

If there is any search result, the occurance label is updated and the first occurance is marked in the source area. The user then can navigate through the results either by pressing the Next and Previous buttons or by pressing Enter.

It would have been nice if we could have colored all findings in the source area with a gray background but this would have had interfered with the coloring of the hover rectangles (Subsection 6.3.5). It would have been a lot of effort to get both of them up and running parallel.

Scrolling to Search Result

Scrolling the search results, that are selected into the visible area of the source area is a problem that has no perfect solution. As a rule of thumb we decided that if the result that has to be marked is already inside the visible area, no scrolling is be performed at all.

As an outcome of this, there are two situations where scrolling is needed. The first situation is when the search result is below or above the visible area of the source area. In this case the view is scrolled that the search result is on the first line of the source area (Figure 6.27).

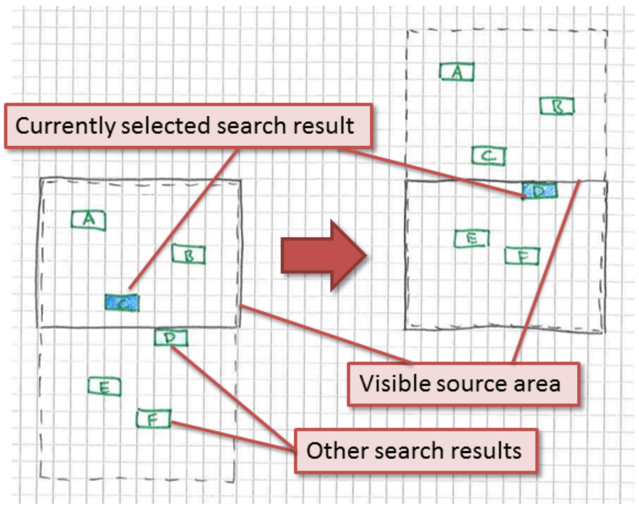


Figure 6.27: Search result outside the visible source area: vertical case

The second situation is when the search result to mark is outside the right border of the source area. In this case the view is scrolled to the right just as much so the reach result is in the visible area again (Figure 6.28). But whenever possible the source area gets left aligned again.

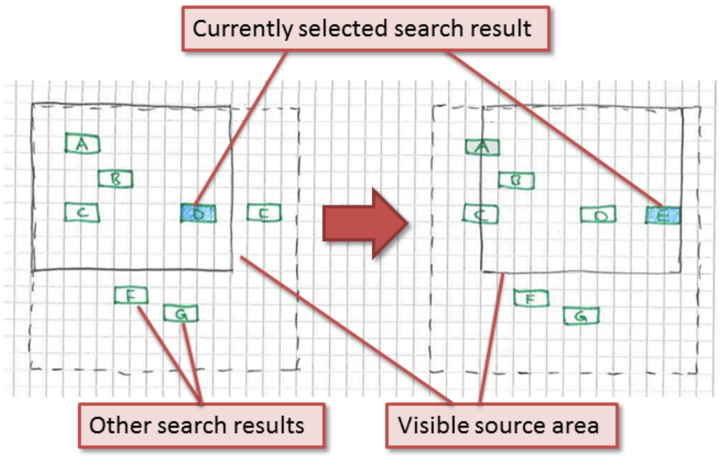


Figure 6.28: Search result outside the visible source area: horizontal case

This scrolling approach tries to scroll as few times as possible to cause as little disturbance for the user as possible.

6.4 Background Loading

One requirement was that *Eclipse* would not freeze when something is opened with the Template Information view. At one point we had to tackle this problem, so we researched how we could make our view more responsive.

We found out, that several more or less time consuming tasks that had to be performed to open one `ViewEntry`:

- Deduce and resolve the clicked name itself
- Search deduce and resolve all substatements
- Format the AST
- Rewrite the code
- Create a `ViewEntry`

At first is seemed nice to do all this tasks in the background so we would not block the UI thread at all. But it would have meant that the `AbstractStatementInfo` that was holding all the template relevant imformations would not have been completly initialized after construction. This was something we did not wanted so we took a deeper look in the loading process.

It turend out that the majority of time was consumed while all substatements where examined. The examining task consumes so much time that all the other tasks can be ignored regarding time consumption.

Originally the search sub templates function was called in the constructor of `ViewData` that was holding the `AbstractStatementInfo`. The easiest solution was to move this expensive function and all the minor tasks that followed after it to the `prepareForView` function. Now this function could be called from outside. With this in place the `ViewData` could be created very fast end the time consuming load task was extracted to the `prepareForView` function. The `prepareForView` function now is the long running task that can be processed asynchronous.

With this in place everything was set up for background loading. The loading is done with a standard *Eclipse* background job. As a nice side effect it also shows up in the *Eclipse* process monitor.

6.4.1 Background Loading Visualisation

Our plan was to show a empty `ViewEntry` with a loading indicator that could be added to the view instantly. After the loading is finished, the final components of the `ViewEntry` should be created and filled with the data that was loaded asynchronous.

For the loading indicator we tested the spinning loading indicator that is well known from the iPhone. But we soon noticed that it was difficult to get the animation running smooth and so we decided to use the standard eclipse progress bar the show the loading progress. Now we only needed some loading progress callback from the `ViewData` when it was loaded. Since almost all time was consumed during sub template deduction and resolving it was not a very difficult task to get the progress relative to the number of sub statements. The only thing that we had to change was to alter the visitor to return a flat list of all names that had to be further examined. This made the visitor rather simplistic because before it was alrady preselecting names that where interessting for us. The loding bar can be seen in Figure 6.29.

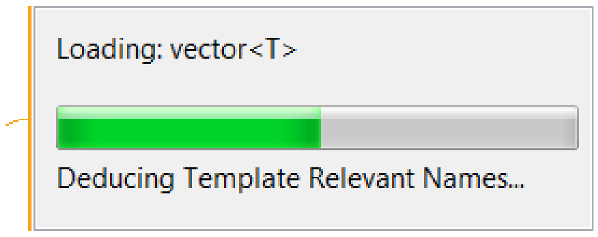


Figure 6.29: Loading bar that is shown when a new `ViewEntry` is loaded

In the end, introducing background loading of `ViewEntry`s was a simpler task then we imagined. The steps that where described above could be implemented in a few days.

But by the nature of parallel programming, there where alot of problems that we did not thought of in the first place. The major problem that we discovered occurs when the user closes the view while a `ViewEntry` is loaded in the background. In this case two things happen.

The first one is relatively easy to cover. As the background job is finished it tries to write the result to the view. Since the view is disposed at that time, there is nothing the result can be written to. Thankfully the references to the view objects still exist and the callback can detect that they are disposed. In this case no action is taken and the callback does nothing. Only when *Eclipse* is closed completly then those references do not exist anymore. In this case we can not to anything. But in that case the user does not care about the data anyway.

The second problem is a lot more severe. When the view is closed, our AST helper classes are cleaned up. At this point the background job can't access them anymore. This leads to a lot of problems.

We decided to not solve the second problem because if the user closes the view he most likely dosn't care anymore about the data. But the problem should be carefully examined before the plugin would ever go live. But this research is out of scope for our Bachelor Thesis. It is always bad when exceptions happen in the background that are not caught. We think that nothing serious is happening but we have no proof for this.

7 Testing

We describe how we tested our application and on which systems we tested the Templator2 plug-in.

7.1 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. Prof. Peter Sommerlad mostly created with the unit testing framework CUTE (`http://cute-test.com/`).

We tested with Windows and libstdc++ from gcc Version 5.1 and with libc++ 3.5-2 on Linux Ubuntu. We set CDT so that C++11 was used. For both operating systems we used *Eclipse* 4.4 and Java 7.

7.2 Unit testing

We extended some test classes from the term thesis [BS14] to test some changed and newly implemented methods. Since we changed our whole class hierarchy and architecture (Subsection 2.4.2) almost all existing unit tests had to be rewritten.

Unfortunately we did not have enough time to write extensive tests for the new functionaly to resolve class template. Only the correct selection of class template specializations is unit tested.

8 Conclusion

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

8.1 Achievements

The result is a useful plug-in for *Eclipse CDT* that helps C++ developers visualizing hard to understand nested template instantations, selected overloads, and specializations. It adds a new view to *CDT* where the user can visualize almost all function calls and many class template instantiations.

The Templator2 is robust and user-friendly and therefore usable by users that were not involved in the development of this plug-in. We hope it will be used by many developers and helps preventing unintended run time behaviour. Because the plug-in does not crash but rather collects the errors, the plug-in can be used for demonstrations and integrated in the near future into Cevelop (`http://cevelop.com`)—an Integrated Development Environment (IDE) based on *Eclipse CDT*.

We managed to already implement some features for future workers on this plug-in. Many features are already implemented and just need to be exposed to the user later on.

8.2 Future Work

As mentioned in the last section, our plug-in supports many function templates and class templates. However, there are some more complex C++ language features the plug-in does not handle yet and cannot show the user the correct definition. This section describes how the plug-in could be extended in a future project or by another thesis.

8.2.1 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 [VJ03, p. 106]. Removing the function from the candidate list is not done by our plug-in yet and can be added to show the actually called function by the compiler and not the one selected by *CDT* that might be wrong.

Adding support for SFINAE should be possible with the current Templator2 functionality. The Templator2 is able to find all identifiers for a template paramater. Trying to specialize the member could then be used to check if all member exists in the chosen template argument and if not, remove this template from the candidates list.

8.2.2 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. Based on the wanted visualization (showing each instance vs. showing only the definition from the C++ editor) this is also formatting feature that the plug-in should implement.

Templator2 passes a default value of `0` as pack offset for some methods to instantiate function templates. This pack offset can be used to correctly instantiate the variadic template for the correct amount of remaining template arguments. *CDT* should be able to select the correct overload for every pack offset. This is a guess and not tested.

8.2.3 Support for Normal Classes

The plug-in could easily be extended to also support non-template classes. The Templator2 would then work as visualization of a call hierarchy for every possible statement with a definition. This already exists in *CDT* but is not able to resolve templates for an arbitrary nesting level and the user has a UI that helps him. The existing *CDT* functionality is just a textual tree hierarchy.

8.3 Known Issues

The following list is for known issues where our plug-in does not work like expected. This list is not about new features that could be implemented but only about plug-in behaviours that seem like bugs.

8.3.1 Declarations With `auto`

The Templator2 plug-in needs a `ICPPASTTemplateId` to be able to instantiate class templates. When using the `auto` keyword to automatically deduced the type by the compiler, there is no template-id and thus the class template cannot be instantiated.

8.3.2 Member Function Calls Where the Owner Is a Template Argument

Member function called on a type that is a template argument will not be resolved.

8.3.3 Unknown Member Function Calls in Non-Template-Classes

A function call resolves to an unknown member function if the owner is a template argument. This unknown method is treated as class template member function even thought it may be defined in a non-class-template. This messes up the formatter that is responsible for writing out the template-id with the deduced template arguments.

8.3.4 Formatting of Member Function Declarator in Class Template Explicit Specialization

When opening a member function definition by itself (not the whole class declaration) the class name with the template paremeters is added to the function declarator. However, the template-id is not added if the function definition is inside a class template explicit specialization.

This is because a member function call for a class template explicit specialization is wrongly tagged as normal function call by us. Changing this would require a bigger change in our class hierarchy or many more `instanceof` checks. This is something we noticed in the last week before the final release and we did not want to potentially break existing functionality just for this to work.

8.3.5 Rectangles for Deduced Template Arguments for Function Templates

Automatically deduced template arguments for function templates will be added to the function call expression. If a deduced argument is a class template, the user cannot click on the `IASTName`. But if the same `IASTName` were somewhere other, the name would be found, resolved and the user click on it. This is because the relevant names are searched before the formatting happens.

Added arguments in the formatting process need to be added manually to our found list of relevant `IASTName`s if they are template argument dependant.

8.3.6 Definitions Outside of Class Template Definition

While the declaration of a member function must be inside the `ICPPASTTemplateDeclaration` for a class template it can be defined outside the class with the classes name qualifier. The Templator2 only processes and shows the class definition and does not consider this outside definitions. The functionality to find those definitions is already implemented in our `ASTAnalyzer.searchFunctionDeclarationsToDefinitions` but not yet used. The time to format these definitions into the `ICPPASTTemplateDeclaration` and consider the definitions when searching for sub statements was not available.

Bibliography

[BS14] Jonas Biedermann and Marco Syfrig. Templator, 2014.

[Cla09] Eric Clayberg. *Eclipse plug-ins*. The eclipse series. Addison-Wesley, Upper Saddle River, NJ, 3rd ed edition, 2009.

[GS14] Fabian Gonzales and Toni Suter. CharWars Rise of the fallen strings: Replace C-String Library calls with C++ std::string Operations, 2014.

[IFS] HSR IFS. CDTTesting git repository. `https://github.com/IFS-HSR/ch.hsr.ifs.cdttesting`. [Online; accessed 10-June-2015].

[ISO11] ISO-IEC. *Programming Languages—C++, ISO/IEC 14882:2011(E) International Standard*, Schweiz. Normen-Vereinigung SNV edition, 2011.

[Mau14] Olve Maudal. C++ pub quiz. `http://www.pvv.org/~oma/PubQuiz_ACCU_Apr2014.pdf`, April 2014. [Online; accessed 8-June-2015].

[Rid15a] Nathan Ridge. [cdt-dev] CPPClassInstance vs CPPClassSpecialization. `http://dev.eclipse.org/mhonarc/lists/cdt-dev/msg29188.html`, May 2015. [Online; accessed 9-June-2015].

[Rid15b] Nathan Ridge. [cdt-dev] Instantiating templates with dependent arguments. `http://dev.eclipse.org/mhonarc/lists/cdt-dev/msg29138.html`, April 2015. [Online; accessed 21-April-2015].

[swt] SWT snippets. `https://www.eclipse.org/swt/snippets/`. [Online; accessed 10-June-2015].

[VJ03] David. Vandevoorde and Nicolai M. Josuttis. *C++ templates: the complete guide*. Addison-Wesley, Boston, 2003.

List of Abbreviations

AST	Abstract Syntax Tree
CDT	C/C++ Development Tooling
IDE	Integrated Development Environment
HSR	Hochschule für Technik Rapperswil
SFINAE	Substitution Failure Is Not An Error
SWT	Standard Widget Toolkit
UI	User Interface