



HSR – University of Applied Sciences Rapperswil

Institute for Software

Master Thesis

# ReDHead - Refactor Dependencies of C/C++ Header Files

Lukas Felber  
lfelber@hsr.ch

<http://redhead.ifs.hsr.ch>

supervised by Prof. Peter Sommerlad

July 2010

# Abstract

Even though C++ belongs to the most widely spread programming languages and is used in many different areas very effectively and also very often, it has long ago been outperformed by other languages, most notably Java, in terms of the power of IDEs and their features. There are C++ IDEs which provide a limited support for some features, like for example *refactoring*. But these cannot even come close to what, for example, *Eclipse's Java Development Tools* provide.

In the scope of this master thesis, the *ReDHead* tool is developed, which adds additional features to the C++ IDE *Eclipse CDT*. These features provide functionality to statically analyze the include dependencies of C++ header files and provide suggestions and hints on how the *include structure* of a C++ software project can be optimized.

The aim of these optimizations is to (1) improve code quality, (2) reduce code coupling, (3) lower compile-time and (4) improve the speed of the development process of C++ software.

Real C++ projects often span millions of lines of code which are distributed over several hundred source and header files. Also initially well designed projects develop a complex net of include dependencies over the years, which is often almost impossible to understand and manage. As a side effect, compile time rises significantly. Hence, the possibility to approach such design issues supported by an automated *static include analysis* tool is a crucial advantage.

The name ReDHead origins from the idea to support a software engineer while *Refactor Dependencies of C++ Header Files*.

Following, a list of features provided by the ReDHead tool:

- Organize includes
- Find unused includes
- Directly include referenced files
- Find unused files
- Static code coverage

The last feature mentioned, *static code coverage*, is actually a special type of *code slicing* feature, which is very helpful to understand a program's design and expose unused code parts.

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	Importance of IDE Features . . . . .	6
1.2	Static Include Analysis . . . . .	6
1.3	Choice of IDE . . . . .	7
1.4	Important Terms . . . . .	7
1.5	Static Analysis in General . . . . .	8
1.6	Feasibility Study . . . . .	9
1.7	Document Overview . . . . .	9
<b>2</b>	<b>Abstraction Concept for C++ Source Code</b>	<b>10</b>
2.1	AST . . . . .	10
2.2	System Dependence Graph . . . . .	11
2.3	ReDHead Graph . . . . .	15
2.4	Conclusion . . . . .	16
<b>3</b>	<b>Used CDT and Eclipse Components</b>	<b>20</b>
3.1	Eclipse CDT . . . . .	20
3.1.1	Compile Configurations . . . . .	20
3.1.2	AST and Indexer . . . . .	20
3.2	UI Elements . . . . .	21
<b>4</b>	<b>Dependency Optimization Algorithms</b>	<b>22</b>
4.1	Finding Unused Includes . . . . .	23
4.1.1	Optimal Include Path Selection . . . . .	24
4.1.2	Algorithm Enhancements . . . . .	27
4.2	Organize Includes . . . . .	27
4.3	Directly Include Referenced Declarations . . . . .	28
4.4	Find Unused Files . . . . .	30
4.5	Static Code Coverage . . . . .	32
4.6	Replace Includes with Forward Declarations . . . . .	35
4.6.1	Refactor Towards <i>iosfwd</i> . . . . .	36
4.7	Introduce Redundant Include Guards . . . . .	36
4.8	Finding Optimal Insert Positions . . . . .	37
4.9	Include Substitution . . . . .	39
<b>5</b>	<b>ReDhead Data Structure</b>	<b>40</b>
5.1	Logical and Physical Design . . . . .	41

5.2	Declaration References . . . . .	42
5.3	Declaration Reference Dependencies . . . . .	42
5.3.1	Preprocessor Symbols . . . . .	45
5.3.2	Function-like Macros . . . . .	45
5.3.3	Templates . . . . .	46
5.4	Includes . . . . .	46
5.5	Include Paths . . . . .	47
<b>6</b>	<b>Implementation</b>	<b>48</b>
6.1	Used CDT Functionality . . . . .	48
6.1.1	AST . . . . .	48
6.1.2	Indexer . . . . .	49
6.1.3	INames . . . . .	49
6.2	ReDHead Data Structure . . . . .	49
6.2.1	Data Stores . . . . .	53
6.2.2	Clean Up . . . . .	53
6.3	Optimization Algorithms . . . . .	53
6.4	UI Integration into Eclipse . . . . .	55
6.4.1	Codan . . . . .	56
6.4.2	Problem Feedback . . . . .	56
6.5	Testing . . . . .	56
6.5.1	External Include Directories . . . . .	58
<b>7</b>	<b>User Manual</b>	<b>59</b>
7.1	ReDHead Introduction . . . . .	59
7.2	Usage . . . . .	60
7.2.1	CDT Codan Integration . . . . .	64
7.3	ReDHead Code Analysis Algorithms . . . . .	65
7.3.1	Finding Unused Includes . . . . .	65
7.3.2	Organize Includes . . . . .	66
7.3.3	Auto Organize Includes . . . . .	66
7.3.4	Directly Include Referenced Declarations . . . . .	67
7.3.5	Finding Unused Files . . . . .	69
7.3.6	Static Code Coverage . . . . .	70
7.4	How ReDHead Include Analysis Works . . . . .	71
<b>8</b>	<b>Market Analysis</b>	<b>73</b>
8.1	Similar CDT Features . . . . .	74
<b>9</b>	<b>Challenges</b>	<b>76</b>
9.1	Adapting the CDT Index . . . . .	76
9.2	Synchronism of CDT Index and AST . . . . .	77
9.3	Algorithm Performance . . . . .	77
9.4	Preprocessor Problems . . . . .	80

<b>10 Outlook</b>	<b>83</b>
10.1 Improvements . . . . .	83
10.2 Unimplemented Features . . . . .	88
10.2.1 ReDHead include tag cloud . . . . .	88
10.2.2 ReDHead graph view . . . . .	88
10.2.3 Implement further algorithms . . . . .	88
10.2.4 Combine <i>compile configuration</i> results . . . . .	89
<b>A Continuous Integration Setup</b>	<b>90</b>
A.1 Continuous Integration Introduction . . . . .	90
A.2 ReDHead Project Server . . . . .	90
A.2.1 Git . . . . .	90
A.2.2 Hudson . . . . .	91
A.2.3 Trac . . . . .	91
A.2.4 Apache Configuration . . . . .	92
A.3 Automated Building of the ReDHead Plugin and Its Tests . . . . .	94
A.3.1 Build Scripts . . . . .	95
A.4 Automated Build of the Documentation . . . . .	100
<b>B Eclipse Plugin Samples</b>	<b>101</b>
B.1 UI Menu Integration . . . . .	101
B.1.1 Extending the Main Menu-Bar . . . . .	101
B.1.2 Extending the Navigator's Pop-up Menu . . . . .	102
B.2 Example Problem Marker . . . . .	104
B.2.1 Customized Markers . . . . .	105
B.3 Example Quickfix . . . . .	106
B.4 Example Codan Checker . . . . .	108
B.5 Undo-Redo Operations . . . . .	109
<b>C Organizational</b>	<b>110</b>
C.1 Project Environment . . . . .	110
C.1.1 Development Environment . . . . .	110
C.1.2 ReDHead Build Server . . . . .	111
C.2 Project Plan . . . . .	111
C.3 Time Schedule . . . . .	113
C.4 CDT Bug Tickets . . . . .	114
C.5 Personal Impression . . . . .	114
C.6 Changes Since Term Project . . . . .	116
<b>D Nomenclature</b>	<b>117</b>

# 1 Introduction

This paper was written within the scope of my master thesis at *University for Applied Science* in Rapperswil, Switzerland. The ReDHead project was initially started as a term project [Fel09] and then continued as this master thesis. The project covers a time period of about 900 working hours in the master thesis and 400 hours in the term project.

The Eclipse plugin *ReDHead*, which's development was started during the term project, adds *static includes analysis* features to *Eclipse CDT*. This is done in the hope that developing C++ will get more comfortable and less complex. The overall aim is to help clean up the *include structure* of C++ projects while also reducing compile time.

To read and understand this paper, knowledge about C/C++ [Str97] and software engineering is required. I will not give any introduction into C/C++ or any common software engineering concepts. To comprehend the *Continuous Integration Setup* Appendix A of this paper, basic knowledge about the Linux operating systems and Ant [Apa04] is recommended.

## 1.1 Importance of IDE Features

When developing C++ source code, one can choose from a variety of integrated development environments (short IDE), which try to support developers in doing so. All of these IDEs provide support for syntax highlighting. Many of them provide enhanced searching features. But often, additional features that go beyond this range are rare. Every feature that helps a developer to find information he is looking for in existing code, speeds up software development and thus decreases the complexity of analyzing, understanding and enhancing code. So, the more of these features an IDE provides, the more helpful it becomes in the hands of a capable software engineer, which, instead of spending hours on scanning code manually, can focus on engineering better software.

## 1.2 Static Include Analysis

Compared to the *import* mechanism of Java, the mechanisms that C++ brings along with *include directives*, can, in bigger projects, become quite tedious and add a lot of complexity. Because the *include* mechanism works transitively, it is more complicated and thus adds, inadequately used, a lot of unnecessary complexity to C++ projects. A whole project of C++ code files depicts a complex construct of source and header files, of which often many rely on each other. A project which was not well designed from the start, is often hard to refactor when it comes to cleaning up and maintaining an existing

*include structure*. Often, nobody dares to even try to remove existing *include directives*, because the effect it will have can be very unpredictable.

Because of the amount of information we are talking about is often huge, it makes sense to approach this issue with automated features which can process the information far quicker than a human could when browsing and analyzing code manually. This is where the motivation for a *static include analysis* plugin origins.

### 1.3 Choice of IDE

The reason to choose Eclipse CDT to implement a plugin for was simple. The IDE already provides good support for code navigation and some refactoring support. I do not want to push any commercial solution to develop C++ code, so the open source IDE *Eclipse CDT* was the favorite choice. The features already contained in CDT provide a good code base for a *static include analysis* plugin.

### 1.4 Important Terms

In this paper I often talk about *static include dependency analysis*. Let us find out what this exactly means when considering a given C++ project. In the source code, one uses *C++ identifiers* which here are referred to as *declaration references*. This can, for example, be the name of a called *function*, a *type* or a *variable*. These *declaration references* refer to *declarations* that can be found somewhere in the current file itself or in any included header file. *Static include dependency analysis* means to find such *dependencies* and use this information to optimize a project's include structure in any way.

To better understand what the *declaration* and *declaration reference* term means, have a look at Figure 1.1.

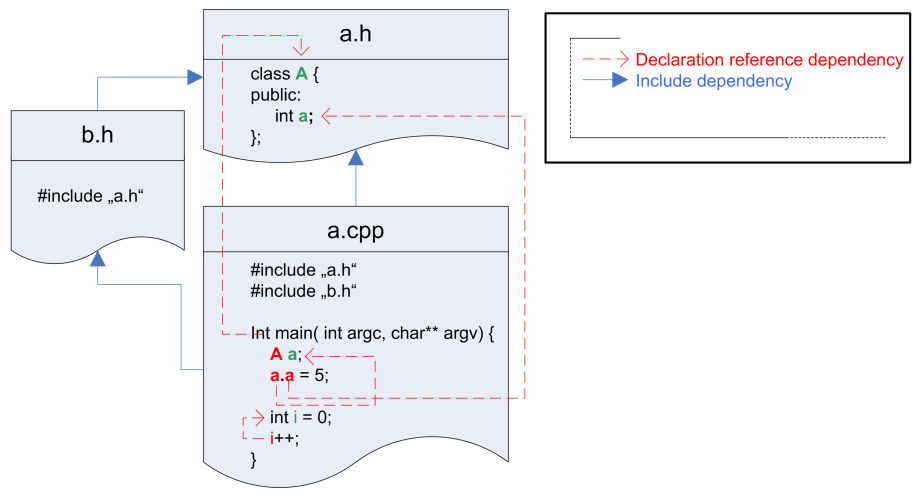


Figure 1.1: Declaration Reference Illustration

Note that text highlighted in green are *declarations* whereas red text represents a *declaration reference*. Red dashed lines implies a dependency between a *declaration reference* and a *declaration* which, from now on, will be called a *declaration reference dependency*.

Here, I would also like to highlight that in C++, a *definition* is always also a *declaration*. Hence, when the term *declaration* is used, if not specified otherwise, it refers to either a *definition* or a *declaration*.

An important term, which also needs clarification here, is the term *include path*. Normally, people think of include paths as an argument passed to a C++ compiler, which then is used to locate required header files. As an example, *GCC* accepts an include path as value of the option `-I`. Actually, the argument passed is the path of a directory, which would make the term *include directory path* more accurate.

The term *include path*, in the scope of this documentation and also the ReDHead plugin, refers not to these compiler include paths but rather to an ordered set of *include dependencies* which forms a path from one C++ code file to a target C++ header file. An *include dependency* in Figure 1.1 is depicted as a blue arrow. As an example, in Figure 1.1, one can see two different *include paths* from file *a.cpp* to file *a.h*. The first one is just “`-> a.h`”, the second one is “`-> b.h -> a.h`”. So the ordered set of the first path contains only one element, the one of the second *include path* contains two.

## 1.5 Static Analysis in General

One should be aware that optimizing the *include structure* of a project is only a small part of possible features that can be implemented with the help of *static code analysis*. As already mentioned, *Eclipse JDT* is a very noteworthy example which contains a wide range of such features which make the life of a software engineer much more comfortable. These features help a software engineer to procure high quality code in the shortest possible time period. Here, an (incomplete) list of features being provided by *Eclipse JDT* that rely on *static code analysis*:

- Refactoring (see [Fow99])
- Open declaration
- Auto-completion of code while typing
- Quickfixes for errors like:
  - Implement method
  - Add field
  - Create class
  - Surround with try catch
  - Change type of variable
  - Change method signature



- Organize imports
- Open type hierarchy
- Open call hierarchy
- Outline view
- Show syntax error while typing
- etc.

## 1.6 Feasibility Study

In the term project, which preceded this master thesis, there was a feasibility study done on the topic *static include analysis for C++*. Here, I would like to refer to the documentation [Fel09] of the term project, Chapter 2, where the documentation of the feasibility study is found. As can be guessed, since the project was extended to this master thesis, the result of the study was that in general a *static include analysis for C++* is implementable. The task, however is far from trivial and there are still some minor problems that needs to be approached.

## 1.7 Document Overview

Before starting the implementation of ReDHead, there was a lot of thinking about what basic structure is required to even be able to run *static include analysis* on. The outcome of this can be found in Chapter 2. Chapter 3 describes what choices I made about how to engage the development of the ReDHead plugin. Chapter 4 describes the *static include analysis* algorithms, which will help a software engineer when looking at a C++ project's include structure. Note that not all the introduced algorithms have been implemented yet. The basic data structure provided by ReDHead, which is used by the *static include analysis* algorithms, is described in detail in Chapter 5. The following Chapter 6 introduces details about the ReDHead plugin's implementation and its components. Chapter 7 describes the ReDHead features in a user centered aspect. A list of several other tools, which also cope with C++ *static analysis* is available in Chapter 8. Information about tasks that were especially hard can be found in Chapter 9. In Chapter 10 there is information about plugin features which could not yet be implemented. Appendix A contains information about the *continuous integration* server setup, followed by useful *Eclipse* plugin example code in Appendix B. Organizational information about the ReDHead project can be found in Appendix C.

## 2 Abstraction Concept for C++ Source Code

Obviously, it is not enough to just textually parse source code to do static analysis. It is clear, that to effectively perform static analysis, an abstract representation of the elements which are contained in the C / C++ programming language is required. Since I am not the first person which is in need of such an abstract code representation, I had a look at existing concepts which could be helpful in performing *static include analysis*.

### 2.1 AST

The first abstract code representation I want to talk about, which is used often for code analysis, is the *Abstract Syntax Tree* (short AST). The AST is a tree data structure which contains *nodes* that represent C / C++ language constructs. This can for example be *statements*, *class declarations*, *variable names* etc.

An AST is a very capable construct to find certain node types. This is one thing that is needed when performing static analysis. However, this is only the first step to perform *static include analysis*. Imagine that when one encounters the following statement in an AST.

```
1 MyClass *myClass;
```

The instantiation contains the name `MyClass`, which refers to a class *declaration* that is located somewhere else. So, `MyClass` is a *declaration reference*. To perform detailed static analysis, it is required to find the *declaration* of exactly that class. Finding this *declaration reference dependency*, however, goes beyond the possibilities of an AST. In the following Section 2.2, we will be looking for a solution of that problem.

## 2.2 System Dependence Graph

To fill the gap revealed in the previous section, one can use the code representation of the *system dependence graph* (short SDG) which will be introduced in this section.

The *system dependence graph* was initially introduced in 1990 by Horwitz et al. [HRB90]. It defines an abstract graph representation for a procedural programming language which supports function calls, but neither *object orientation*, *macros* nor other language constructs of C++. The difference here to an AST is, that an AST is confined to a single code file, whereas a SDG is not.

*System dependence graphs* were introduced as a instrument to help in the process of *slicing*. Slicing is a graph traversal process which, given a program point  $p$  and a variable  $x$ , finds all parts of a program which affect and are affected by  $x$  at point  $p$ . This slicing process can be helpful to perform debugging [ADS93], automatic parallelization [BW88, Wei83] or to automatically integrate program variants [HPR89].

In 1996, Larsen and Harrold introduced a SDG for *object oriented* languages while using the language C++ for examples [LH96]. In Figure 2.1 one can find an example of an SDG which does not yet uses *object orientation*. So Figure 2.1 is a SDG as defined by Horwitz et al. [HRB90]. In the following example images, the C++ source code given in Figure 2.1 will be adapted so it uses *object orientation* and also *polymorphism*.

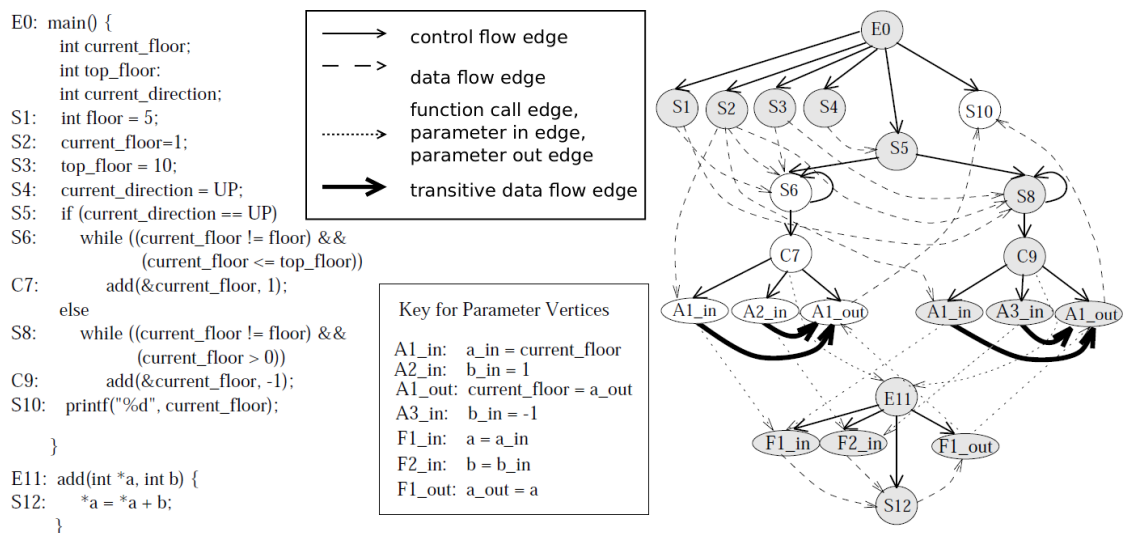


Figure 2.1: Elevator Example System Dependence Graph with Function Calls [LH96]

To basically understand the graph, it is important to map vertices to code statements by using the labels on the left side of the source code (e.g. E0 on the first code line). Solid edges in the graph depict possible program execution control flow. Here, I would like to specially mention the two *function call vertices* C7 and C9 which are connected by a *function call edge* to the *function definition vertex* E11. Dashed graph edges represent a program's data flow. S1, for example, is connected by a *data flow edge* to S6 because

the value of the variable `floor` that is defined in `S1` is used in `S6`.

To understand the data flow in and out of a function call, *actual-in* (e.g. `A1_in`), *actual-out* (e.g. `A1_out`), *formal-in* (e.g. `F1_in`) and *formal-out* (e.g. `F1_out`) vertices have been added to the graph. These can be mapped to source code with the help of the *Key for Parameter Vertices* agenda.

For a precise description of the graph, I refer to [LH96] and [HRB90].

The following Figure 2.2 introduces the class `Elevator`, which splits the beforehand introduced elevator program into member functions. Note that there is not yet a `main` program available in the figure. It will follow in Figure 2.4.

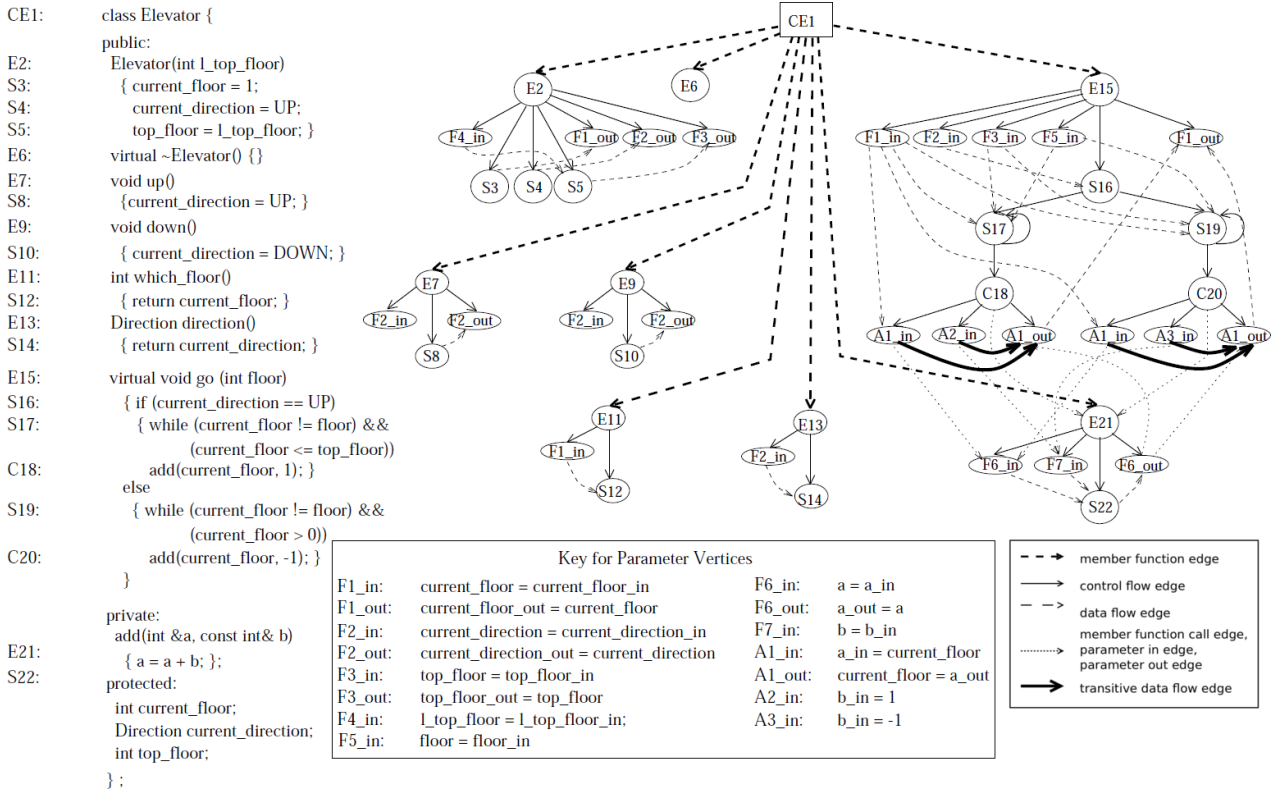


Figure 2.2: Object Oriented System Dependence Graph [LH96]

Note that the next two graphs contain part of their preceding graphs, which means that one must also take into account the code which accompanies these preceding graphs and their the codes line labels to understand the meaning of the current graph.

Compared to the graph in Figure 2.1, this graph also contains *member function edges*, which are represented as bold dashed edges which connect the *class vertex* `CE1` to its *member function vertices* (e.g. `E7`).

Shown in Figure 2.3 is the class `AlarmElevator` which inherits from `Elevator`.

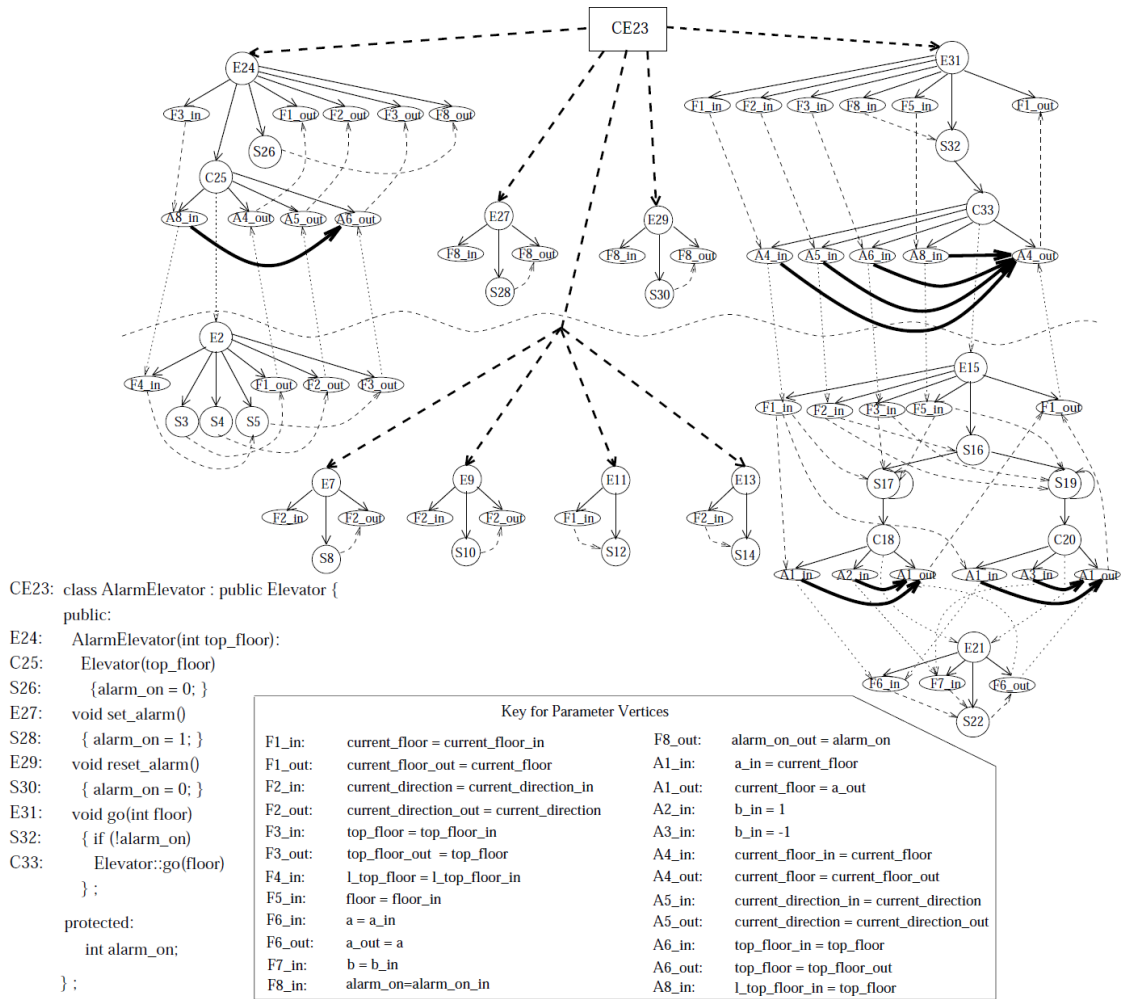


Figure 2.3: Object Oriented Polymorphic System Dependence Graph [LH96]

The `AlarmElevator`'s elements are placed in the graph above the horizontal, dashed and waved line, whereas the base class' elements are located below. This line is crossed in two cases. The first is by *member function edges* since the deriving class inherits these member functions. The second case is by *member function call edges* in the case of overloaded member functions when calling their super implementation.

In the following Figure 2.4, the `main` program is shown, which uses either the `AlarmElevator` or the `Elevator` class. Note the vertex `C38`, which calls the overloaded member function `go()`, connects to the artificial vertex `P1`, which again leads to both the implementations of the `go()` member function. This is due to the fact, that the evaluation of which member function is really called is only possible at runtime.

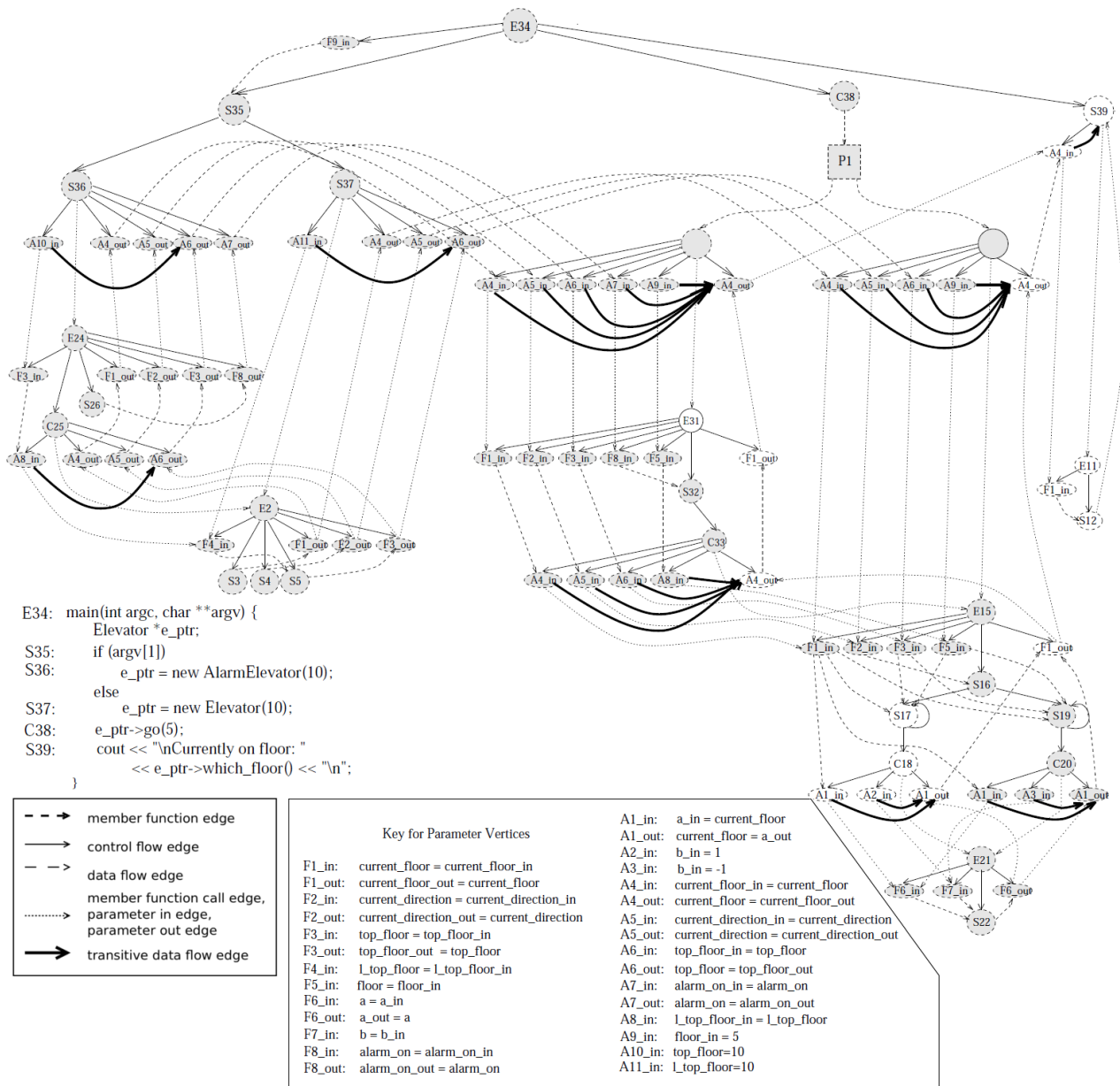


Figure 2.4: Main Function [LH96]

When taking into account the introduced graphs above, we now have a SDG for *object oriented* C++, which allows us to solve the problem of resolving *declaration reference dependencies* by traversal. However, there are many points which mark the C++ SDG as suboptimal.

**Unneeded edges** Several edge types given in the C++ SDG are not important for *static include analysis*. This includes the *control flow*, the *data flow*, the *parameter-in* and the *parameter-out* edges.

**Costly graph traversals** it will be very time-consuming to traverse a whole graph for only trying to resolve one *declaration reference dependency*.

**Vertices are not distinct** A single vertex in the C++ SDG can contain several *declaration references*.

**Missing Physical Design Elements** To perform static include analysis, it is required that the vertices for *physical design* elements, namely source and header files, are also represented in the graph. This allows then to also introduce graph edges which represent include directives which are also required. The C++ SDG only contains *logical design* elements at the moment. A more detailed description on *logical* and *physical design* can be found in Section 5.1.

As can be deduced from the list above, also the C++ SDG is not the optimal abstract code representation to perform *static include analysis*. The following Section 2.3 introduces a graph, which is, in my point of view, optimal to perform *static include analysis*.

## 2.3 ReDHead Graph

This section will introduce the *ReDHead graph*, which is designed to hold all the information that is required to provide the ability to perform *static include analysis* in a fast and intuitive way.

The introduction to the *ReDHead graph* will be accompanied with example graphs that represent the same C++ code that was already present in the SDGs in Section 2.2.

A *ReDHead graph* basically consists of 5 different elements, whereof three are vertices and two are directed edges. A black rectangle represents a source or header file vertex. These can be connected with blue solid edges, which depict include directives. Green rectangles represent *declaration* vertices. *Declaration reference* are represented by vertices which are red ovals. And lastly, red dashed edges depict *declaration reference dependencies*.

The label of a vertex can be interpreted as following: The first part is the name of the correlating *declaration* or *declaration reference*. In the case of many *declaration references* with long names, the names are replaced with DR which is simply short for *Declaration Reference*. The number following the name of the vertex represents the

code line number in which the vertex originates. The code segments' lines are numbered in green on the left-hand side of the source code. In the case of several *declaration references* on the same line, the line number is followed by a “\_” and an index number which indicates the position of the *declaration reference* on that line. As an example, let us assume there is a document with the code `i = j;` on line 15 and a graph containing the vertex labeled DR15.2. Then, the vertex depicts the 2nd *declaration reference* which originates from code line 15, which is `j`.

Note that a *file-contains* edge has been omitted since one can plainly see that all the *declaration* and *declaration reference* vertices would have one, leading to the *file* vertex that contains them.

Also note that each *declaration* and *declaration reference* vertex correlates exactly to one *node* of an AST. This is very helpful because this will allow to continue the traversal of the *ReDHead graph* into the AST and also the other way around.

Figure 2.5 shows the *ReDHead graph* that correlates to the SDG from Figure 2.1, which is also contained in the Figure 2.5.

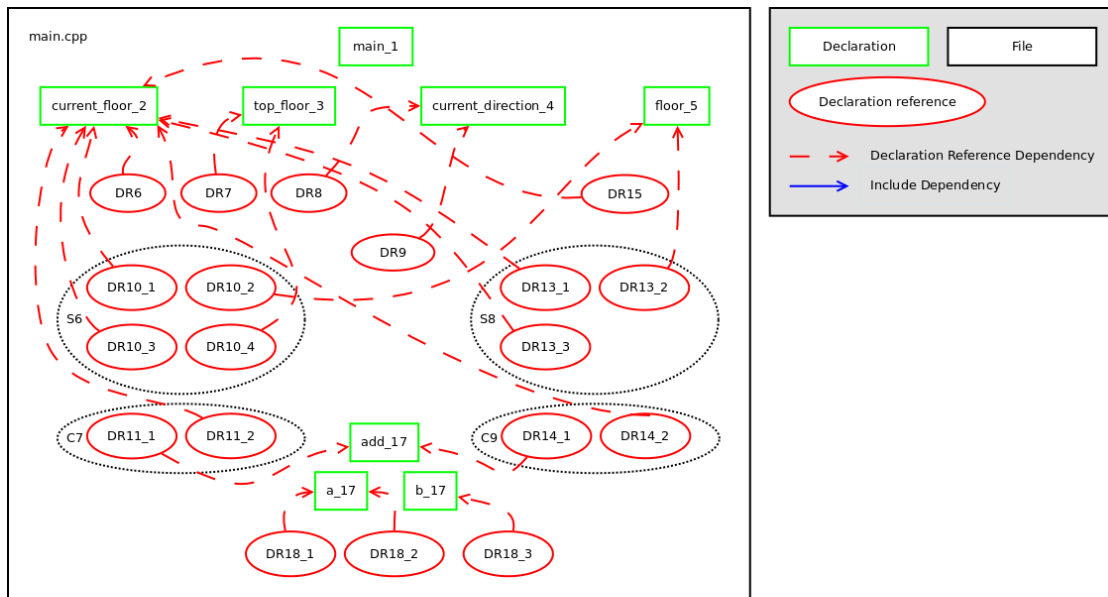
To let the graphs be compared easily, the vertices in the *ReDHead graphs* are arranged as similar as possible to the arrangement of the SDG. Obviously, in the SDG there are no vertices for variable definition, whereas in the *ReDHead graph* there are. Some nodes in the SDG are represented in the *ReDHead graph* as several *declaration reference*. These *declaration reference* are enveloped by a black, dotted oval which is labeled with the name of the correlating vertex name in the SDG.

The graph shown in Figure 2.6 depicts the object oriented SDG which was shown in Figure 2.4. In the figure, one can also see the *include dependencies* between different source and header files. The program's source code and the SDG of Figure 2.4 including additional green line numbers necessary to comprehend the *ReDHead graph* in Figure 2.6, can be found in Figure 2.7.

## 2.4 Conclusion

As was explained, the *Abstract Syntax Tree*, the *System Dependence Graph* and also the *ReDHead graph* each are very useful to perform *static code analysis*. When one focuses on *static include code analysis* though, the *ReDHead graph*, combined with additional information from ASTs, is the optimal provider for the required information.





```

1. E0: main() {
2.     int current_floor;
3.     int top_floor;
4.     int current_direction;
5. S1:   int floor = 5;
6. S2:   current_floor=1;
7. S3:   top_floor = 10;
8. S4:   current_direction = UP;
9. S5:   if (current_direction == UP)
10. S6:   while ((current_floor != floor) &&
              (current_floor <= top_floor))
11. C7:   add(&current_floor, 1);
12.     else
13. S8:   while ((current_floor != floor) &&
              (current_floor > 0))
14. C9:   add(&current_floor, -1);
15. S10:  printf("%d", current_floor);
16.     }
17. E11: add(int *a, int b) {
18. S12:  *a = *a + b;
19.     }

```

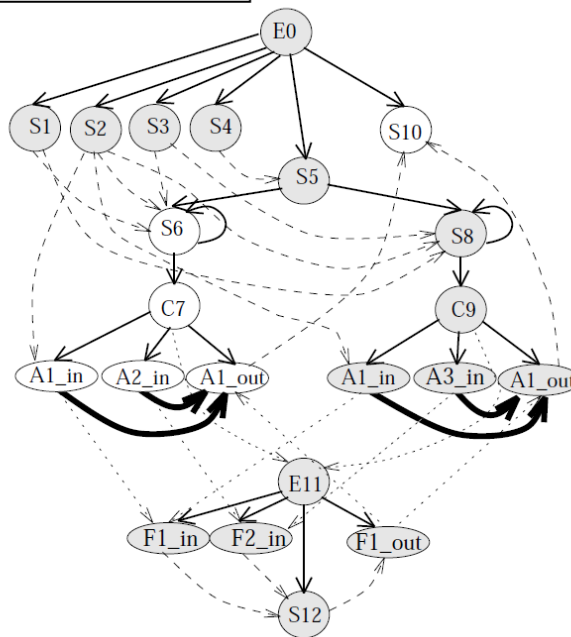


Figure 2.5: Simple Example for a RedHead Graph

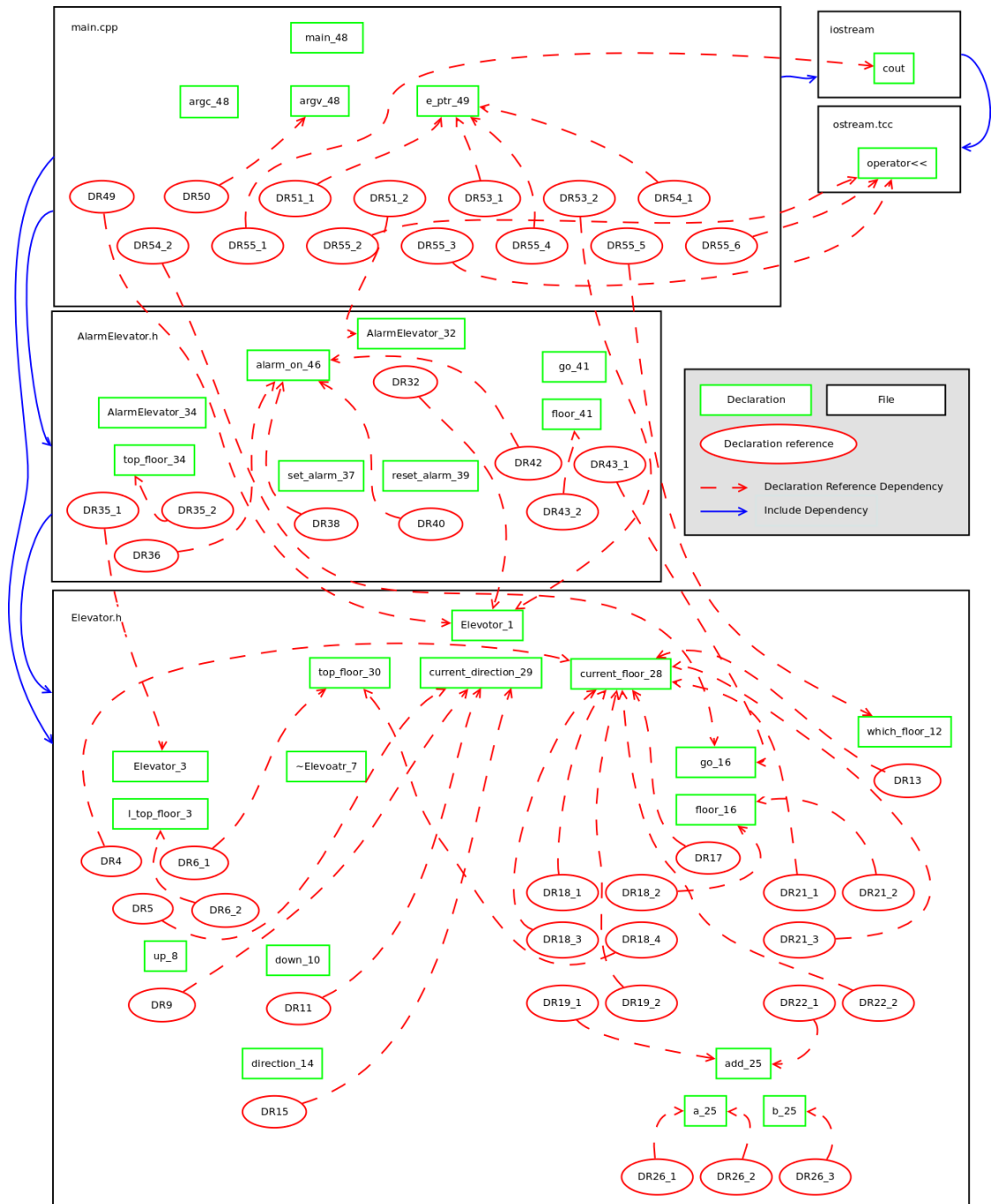
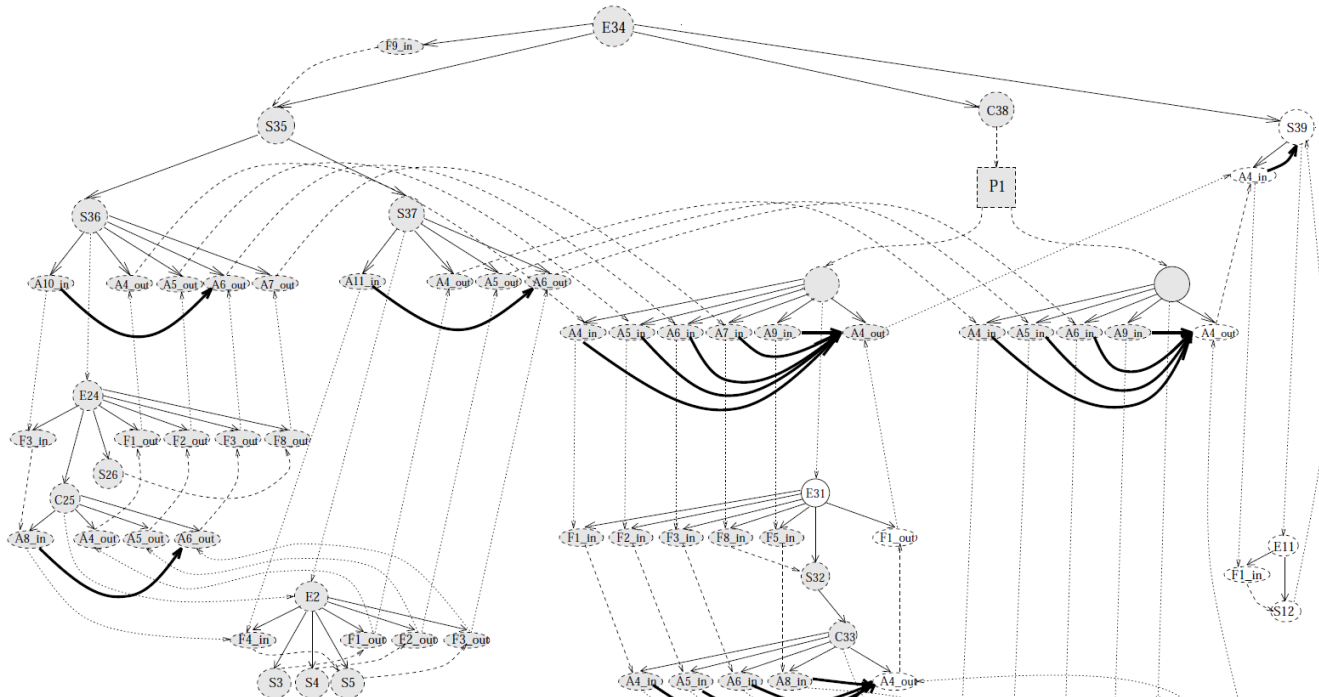


Figure 2.6: Complex Example for a RedHead Graph



```

1. CE1:   class Elevator {
2.       public:
3.         Elevator(int l_top_floor)
4.         S3:   { current_floor = 1;
5.         S4:   current_direction = UP;
6.         S5:   top_floor = l_top_floor; }
7.         virtual ~Elevator() {}
8.         E7:   void up()
9.         S8:   { current_direction = UP; }
10.        E9:   void down()
11.        S10:  { current_direction = DOWN; }
12.        E11:  int which_floor()
13.        S12:  { return current_floor; }
14.        E13:  Direction direction()
15.        S14:  { return current_direction; }
16.        E15:  virtual void go (int floor)
17.        S16:  { if (current_direction == UP)
18.        S17:  { while (current_floor != floor) &&
19.                (current_floor <= top_floor)
20.                add(current_floor, 1); }
21.        S19:  { while (current_floor != floor) &&
22.                (current_floor > 0))
23.                add(current_floor, -1); }
24.        private:
25.        E21:  add(int &a, const int& b)
26.        S22:  { a = a + b; };
27.        protected:
28.        int current_floor;
29.        Direction current_direction;
30.        int top_floor;
31.    };

32. CE23: class AlarmElevator : public Elevator {
33.        public:
34.        E24:  AlarmElevator(int top_floor):
35.        C25:   Elevator(top_floor)
36.        S26:   { alarm_on = 0; }
37.        E27:   void set_alarm()
38.        S28:   { alarm_on = 1; }
39.        E29:   void reset_alarm()
40.        S30:   { alarm_on = 0; }
41.        E31:   void go(int floor)
42.        S32:   { if (lalarm_on)
43.        C33:   Elevator::go(floor)
44.        };
45.        protected:
46.        int alarm_on;
47.    };

48. E34:  main(int argc, char **argv) {
49.        Elevator *e_ptr;
50.        if (argv[1])
51.        S36:   e_ptr = new AlarmElevator(10);
52.        else
53.        S37:   e_ptr = new Elevator(10);
54.        C38:   e_ptr->go(5);
55.        S39:   cout << "\nCurrently on floor: "
56.                << e_ptr->which_floor() << "\n";
57.    }

```

Figure 2.7: Comparison SDG and Code for Figure 2.6

## 3 Used CDT and Eclipse Components

This chapter describes the environment of the ReDHead plugin in detail. There are reasons given for the choice of the IDE and the used software components, on which ReDHead will be built upon.

### 3.1 Eclipse CDT

The plugin that will be developed in the scope of the ReDHead project will be an Eclipse plugin to be used together with C++ IDE *Eclipse CDT*. CDT contains a, yet still almost empty, static analysis framework called Codan, which the ReDHead plugin can hook itself into. Also, CDT provides access to AST structures for C++ code. Furthermore, there is the CDT indexer which provides indispensable help when resolving *declaration references*.

#### 3.1.1 Compile Configurations

When developing C++ software, one normally uses a *build system* like *make*, *cmake*, *qmake* etc. These build systems are used to assemble the numerous arguments which will be passed to the compiler that is used to compile the C++ software. These arguments, coming from the *build system*, combined with a compilers own arguments, result in a so called *compile configuration*, which is mainly used by the compiler itself. However, a key feature to successfully run static software analysis is such a *compile configuration*. Without the *compile configuration*, one could not reliably resolve all dependencies contained in the C++ software since a lot of ambiguous cases would arise.

Acquiring such a *compile configuration* is a very complex process since there are many different C / C++ compilers which normally comes along with one of several different *build system*. How costly it can be to obtain such *compile configurations* is in detail described in [BBC<sup>+</sup>10].

Seen from this point of view, it becomes clear, that to effectively develop *static include analysis* for C++ code, a powerful IDE which already provides a suitable *compile configuration* is required. Since CDT does this, I can save a lot of time in the development process of the ReDHead plugin.

#### 3.1.2 AST and Indexer

CDT's AST and Indexer functionality already provides probably all the basic information which is needed to statically analyze C++ code. To effectively do so, this information

will be used to set up a powerful data structure on which *static include analysis* will be performed.

Each C++ project in Eclipse CDT already provides what was before referred to as *compile configuration*. In the project properties the user can set *preprocessor symbols* manually if he wants to do so. Since this information is, by default, used to build CDT ASTs, this CDT *compile configuration* is, in my point of view, a very good point to start.

The include analysis of the file or project could also be done on the basis of a completely user defined *compile configuration*, which can be set in a ReDHead properties page. Concretely, this would mean that the user can define if a *preprocessor symbol* is set or not set, what include directories shall be considered, and so on. ReDHead would then analyze only these code parts, which are *active* in means of the *preprocessor symbols* in the *compile configuration*.

Such a properties page would mean additional effort and does something very similar to what CDT already provides. Hence, I will not implement such a properties page.

## 3.2 UI Elements

Invoking a static include optimizer is achieved through menu entries. As an alternative, for certain optimizers, also the automatic invocation while the user is typing is a promising feature used by ReDHead. This alternative can be achieved, by using the CDT Codan framework [cod09].

The best point to start with *static include analysis* results in the perspective of visualization is to add markers to CDT editor instances for optimization suggestions and let the user choose how to go on from there. The user then can decide if he wants to solve the highlighted problem by applying a proposed quickfix. In Eclipse, these proposals can be activated by the user when pressing *ctrl+1*.

In certain cases, only adding markers will not be a favorable solution. For such cases, an other presentation possibility was found. In the ReDHead plugin, this is a dialog which lets the user decide on what to do with optimization suggestions.

## 4 Dependency Optimization Algorithms

In this section I describe *static include analysis* algorithms which are considered useful for a C++ software engineer. These algorithms are the main features of the ReDHead plugin. They are based on the *ReDHead data structure* described in Chapter 5, which provides them with the basic functionality to perform *static include analysis*.

The following sections contain a pseudo-code implementation. These pseudo-code listings give a good overview of the real algorithm implementations. They are not a complete representation of the functionality since this would bloat the pseudo-code to much. But they should give a good idea on how an algorithm basically works.

Figure 4.1 shows a simplified version of the *ReDHead graph* as described in Chapter 2. These kind of figures will be used to support the introduction of the algorithms described in the following sections. Compared to the other graphs that are present in this paper, these graphs also show *include paths*.

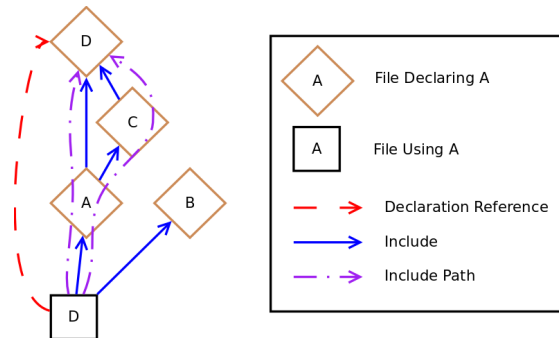


Figure 4.1: Example Include Path Graph

Black rectangle vertices depict a file which contains a *declaration reference*. The letter inside of the rectangle is the name of this *declaration reference* and not the name of the file. A brown diamond-shaped vertex illustrates a *declaration*. Also here, the contained name describes the name of the *declaration* and not the one of the file. Blue, solid edges visualize includes, whereas red, dashed edges depict *declaration reference dependencies*. And lastly, the violet dashed-dotted edges implicate *include paths*. Note that *include paths* are, as already mentioned in Chapter 1, a set of includes. So they always go along one or several blue edges. In the case of an *include path* with a length of two or more, it also goes on through files, as can be seen in the file vertices A and C.

The last part in this chapter, Section 4.8, describes how the best position to insert an include is found. This part is used by all the algorithms which propose the insertion of include directives.

## 4.1 Finding Unused Includes

*Eclipse JDT* automatically marks all unnecessary Java *import statements* which are contained in a Java source file. The algorithm described here is the C++ equivalent to the one of Java. One should be aware that Java *imports* are not transitive whereas C++ *includes* are, which makes the task much more difficult than the one in Java.

This algorithm is file based, which means that its starting position is one given C++ source or header file. The algorithm is also available for a whole project. In this case the algorithm is run for each C++ file that is contained in the given project.

To give an overview, the algorithm is first described in pseudo-code:

```

includeList := findListOfIncludes(presentFile)
2
declarationRefList := findAllDeclarationReferences(presentFile)
availableIncludePathList := findListOfIncludePaths(declarationRefList)
5
pickedIncludeList := filterAvailableIncludePaths(availableIncludePathList)
unusedIncludeList := removeAllRequiredIncludes(includeList, pickedIncludeList)
8
markIncludesAsUnused(unusedIncludeList)

```

To help understanding the textual description which follows, the example in Figure 4.2 was added here. The active file on which the algorithm is run is the one represented as black box labeled D, H, G in the bottom of the graph.

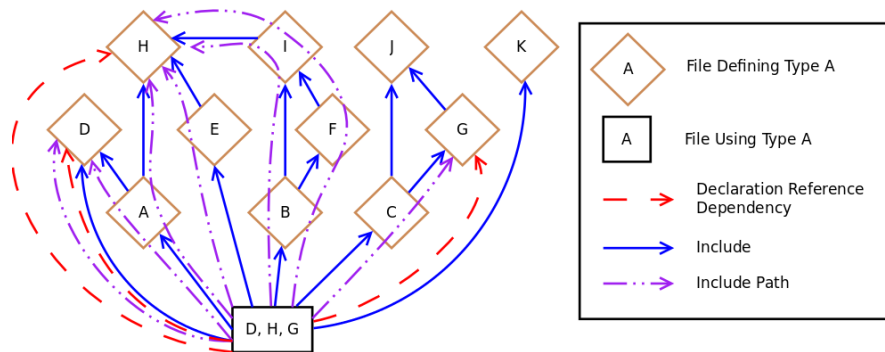


Figure 4.2: Find Unused Includes Example

The first step of the algorithm gathers all present *include directives* in the file (pseudo code line 1). This information can be retrieved by the the active file. In the example figure, these includes are the blue arrows which lead to A, B, C, D, E and K. In a second step, the algorithm collects all *declaration references* contained in the given file (pseudo code line 3). In the example, these are the *declaration references* D, H and G. For each

*declaration reference* it finds the correlating *declaration* and the *include paths* which leads from the active file to the file containing the *declaration* (pseudo code line 4). In the example graph, these are all the *include paths* in the graph. Note here that sometimes it is possible that there is more than one *include path* that leads to the file containing the *declaration* as can be seen in the example graph for *declaration H*. Also note that the resulting list `availableIncludePathsList` in most cases contains more *include paths* than required. So proposing only the include to K to be removed in the example graph would not be accurate.

The next step is to find all required includes by filtering the list `availableIncludePathList` (line 6) or rather their first path elements. This filtering process should only select all these includes in the list `availableIncludePathList`, so that all files which contain used *declarations* are still reachable at least once. This filtering process is rather complex and is thus described in detail in the following subsection 4.1.1. The following step is to remove all required includes from the list `includesList` which yields a list of remaining includes which are all these that are unneeded (line 7).

The component which decides which of all available include paths contained in the active file (list `includePathList`) is best to be left in the document, the routine `filterAvailableIncludePaths()`, are described in the following section.

#### 4.1.1 Optimal Include Path Selection

The reason for the presence of this section is, that a *declaration* which is required through a *declaration reference* in the current file can be included through several *include paths*.

If there are several possible paths available to navigate through *include directives* to a *declaration* and these paths have different starting include elements, one needs to decide which one of them is the best one because, seen from the given *declaration reference*, only one of these starting includes is required to be included in the current document. To make a good decision, however, one does not only need to look at the in *include paths* leading from one *declaration reference* to one *declaration* but at all of these relations. This means that, given all the *include paths* leading out of the current document, one can remove some paths until all the paths are necessary so that it is still possible to reach all the *declarations* which are used through *declaration references* in the current document.

```

filterAvailableIncludePaths(availableIncludePathList) {
    pickedIncludeList := EmptyIncludeList()
3
    pickMandatoryIncludes(pickedIncludeList)
    while hasIncludeChoice(availableIncludePathList) {
6
        removeWorstChoice(availableIncludePathList)
        pickMandatoryIncludes(pickedIncludeList, availableIncludePathList)
    }
9
    return pickedIncludeList
}

```

The process of finding all the required includes is achieved by picking includes until all the includes which are required are in the list `pickedIncludesList`. Picking includes



happens in `pickMandatoryIncludes()`, in pseudo code line 4 and 7. For each path of which the target file cannot be reached through any other path, the first element in the path is thus mandatory to satisfy some *declaration reference dependencies* in the active file. In `pickMandatoryIncludes()`, all these first elements are added to the list `pickedIncludesList`. When an include is picked, all the *include paths* which start with the picked include are not needed anymore because their target file is now reachable through the picked include. This means, that not only the *include paths* itself, but also all others which have the same targets can be removed from the list `availableIncludePathsList`.

Now, one checks if the list `availableIncludePathsList` is not empty. If it is, there is nothing to do anymore and the list of picked includes can be returned.

At the moment after all the mandatory includes have been picked, of all the first elements of the remaining *include paths* in `availableIncludePathsList`, must at least one be unnecessary. So next, in `removeWorstChoice()`, of all remaining first elements of the *include paths* in `availableIncludePathsList`, the worst one is selected. Then, all the paths in `availableIncludePathsList` which start with that element are removed. It is now possible, that some of the remaining first elements became mandatory. So the process of `pickMandatoryIncludes()` and `removeWorstChoice()` is repeated until the `availableIncludePathsList` is empty.

The tricky part here, which makes the *find unused includes* algorithm interesting, is the decision of determining which one of the remaining includes is the worst one. The question that arises here, is the one of what makes includes good or bad. To measure goodness or badness, I use two characteristics of an include.

The first characteristic is the amount of target files that are reachable through a given include. In the example Figure 4.2 the include to **A** can reach both **D** and **H** whereas through the include to **E**, only **H** can be reached.

The second characteristic that is used is the amount of all recursively included files. The include to **E** includes two files (**E**, **H**), whereas the one to **B** includes four (**B**, **F**, **I**, **H**). Using the two characteristics in the following formula, a penalty is calculated for each remaining include. The include with the biggest penalty is the one that is selected for removal in `removeWorstChoice()`.

$$penalty = (2 + amountRecusivelyIncludedFiles)/amountTargetFilesReachable^2$$

Important to see is that the selection of the include to remove is a heuristic approach. The `penalty` calculation above is not a perfect metric to find the worst include. This means that in very special cases it can happen, that the *find unused includes* algorithm proposes not as many includes for removal as possible. The heuristic approach here was chosen, so that the algorithm is able to deliver suggestions in the shortest possible period of time. An optimal calculation would, with big C++ files, take minutes if not hours to complete.

## Example Run

To help understand the process of the filtering of the available *include paths* which was described in 4.1.1, the process will be illustrated here with an example. Figure 4.3 is a repetition of Figure 4.2 additionally containing numbers indexing the *include paths*, to make the description steps below easier. Also, the pseudo code given in Section 4.1.1 is repeated again so one does not have to leaf back and forward continuously.

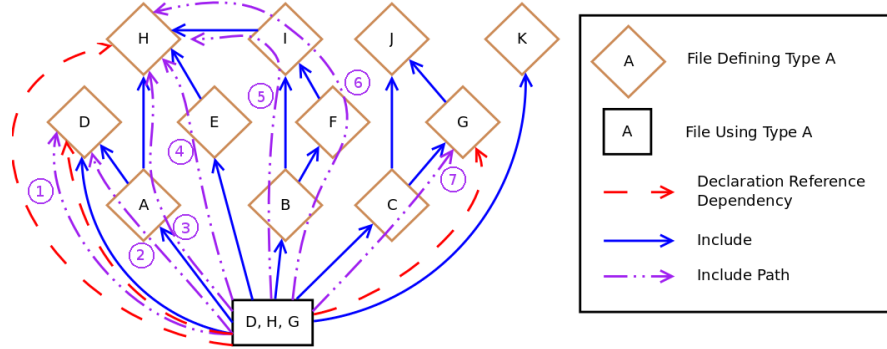


Figure 4.3: Find Unused Includes Example with Numbered *Include Paths*

```

2 filterAvailableIncludePaths(availableIncludePathList) {
  pickedIncludeList := EmptyIncludeList()

  pickMandatoryIncludes(pickedIncludeList, availableIncludePathList)
5 while hasIncludeChoice(availableIncludePathList) {
  removeWorstChoice(availableIncludePathList)
  pickMandatoryIncludes(pickedIncludeList, availableIncludePathList)
8 }
  return pickedIncludesList
}

```

Execution Point	availableIncludePath-List	pickedIncludeList	Additional Information
line 4	{1,2,3,4,5,6,7}	{ }	In <code>pickMandatoryIncludes</code> , include C gets selected since the target file of <i>include path 6</i> (G) can only be reached through C. <i>include path 7</i> thus gets removed from <code>availableIncludePathList</code>
line 6, iteration 1	{1,2,3,4,5,6}	{C}	Penalties: $A = (2 + 3)/2^2 = 1.25$ , $B = (2 + 4)/1^2 = 6$ , $D = (2 + 1)/1^2 = 3$ , $E = (2 + 2)/1^2 = 4$ . B is worst, so the <i>include paths 5</i> and <i>6</i> get removed.
line 7, iteration 1	{1,2,3,4}	{C}	None of the includes A, B, D or E is mandatory required so nothing is picked.
line 6, iteration 2	{1,2,3,4}	{C}	Penalties: $A = (2 + 3)/2^2 = 1.25$ , $D = (2 + 1)/1^2 = 3$ , $E = (2 + 2)/1^2 = 4$ . E is worst, so the <i>include path 4</i> get removed.
line 7, iteration 2	{1,2,3}	{C}	In <code>pickMandatoryIncludes</code> include A gets selected since the target file of <i>include path 3</i> (H) can only be reached through A. <i>include paths 1, 2</i> and <i>3</i> gets removed from <code>availableIncludePathList</code> because all their target files (D, D and H) are reachable through A.
line 5, iteration 3	{ }	{C, A}	The iteration is stopped here since <code>availableIncludePathList</code> is empty.

### 4.1.2 Algorithm Enhancements

When running this algorithm on a real C++ source file, the file contains quite a lot of *declaration references* which thus also often yield a lot of duplicate *include paths*. The number of *declaration references* and *include paths* handled by the *find unused includes* algorithm can easily go into the thousands. Thus, grouping both of them already on construction, so that there are no duplicates, decreases their number a lot. Besides a huge performance gain, the algorithm is not affected at all through this change.

## 4.2 Organize Includes

Eclipse JDT contains an *organize imports* feature that can be used to automatically import all needed classes into a Java class. The target of the *Organize Includes* algorithm is to do the same thing in C++ code with includes.

*Organizing includes* basically does two different things. The first thing is to find includes that are unused and to propose them for removal. This part is already covered by the *finding unused includes* algorithm described in 4.1. A small difference here is that *organizing includes* only makes sense in the scope of a single file and not in the scope of a whole project. The second thing that *organize includes* does, is to add missing includes, which are required to successfully compile the active file. To achieve this goal, the ReDHead plugin needs a possibility to resolve *declaration references* to any *declarations* in a project's scope. This includes for example also the *declarations* of the *C++ Standard Library* or any other headers which are available in the project.

The following pseudo code implementation gives a good overview of the task that is fulfilled by the *organize includes* algorithm.

```
2 declarationRefList := findAllDeclarationReferences(presentFile)
3 filesToIncludeList := EmptyIncludeList()
4
5 for declarationRef in declarationRefList {
6     declarationRefDependency := findDeclarationRefDependency(declarationRef)
7     declaration := findDeclaration(declarationRefDependency)
8     includePathList := findIncludePathsToDeclaration(declarationRef, declaration)
9
10    if isEmpty(includePathList) {
11        addToList(filesToIncludeList, fileOf(declaration))
12    }
13 }
14 proposeToInclude(filesToIncludeList)
runFindUnusedIncludesAlgorithm(presentFile)
```

The pseudo code implementation should be understandable easy enough. First all *declaration references* are collected. Then, for each of these *declaration references*, first the *declaration reference dependency* and then the *declaration* is found and the *include paths* are retrieved that lead from the *declaration reference* to the *declaration*. Should the list of retrieved *include paths* be empty, an include to the file that contains the *declaration*

is suggested to be added. The last pseudo code line triggers the *find unused includes* algorithm, which is, as already mentioned, part of the *organize includes* algorithm.

The graph in Figure 4.4 illustrates a simple example to organize includes.

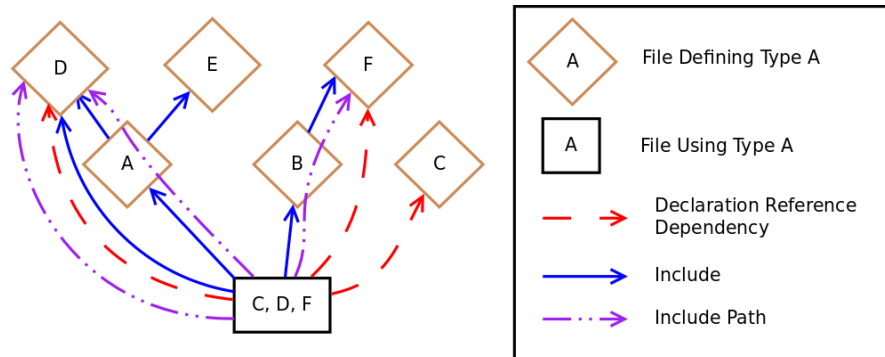


Figure 4.4: Organize Includes Example Graph

The *declaration references* which are gathered in pseudo code line 1 are for the example: C, D and F. When finding *include paths* in pseudo code line 7, the retrieved list for C is empty and a suggestion for an include to the file containing the *declaration* of C is added in pseudo code line 10. Why the include to A is proposed for removal in code line 15, can in detail be read in the description of the *find unused includes* algorithm in Section 4.1.

### 4.3 Directly Include Referenced Declarations

The *directly include referenced declarations* algorithms is based on the idea one should not rely on included header files to again include others. To clarify this, an example is given. In the graph on the left hand side in Figure 4.5 one sees, that both the files labeled A,B,C and A use the *declarations* B and C. If, for any reason, the file labeled A now loses the dependency to C and the include to C gets thus removed (graph in Figure 4.5 on the right-hand side), there will be no available *include path* to C, which results in compiler error in the file labeled A,B,C. This happens, even if the file labeled C has not been changed at all.

In Figure 4.6 the same situation is given again, but with additional includes from the file labeled A,B,C to B and C. Here, the problem at hand will never arise when the include from A to C gets removed. So by *directly including the referenced declarations* B and C, the (transitive include) coupling to A can be reduced. The C++ code at hand will get less error-prone to changes in the include structure.

This idea which was introduced above can also be found described in [Lak96] on page 113.

So the current algorithm allows the user to automatically suggest the inclusion to *declarations* which are used in a given file. This strategy is closely related with Java's

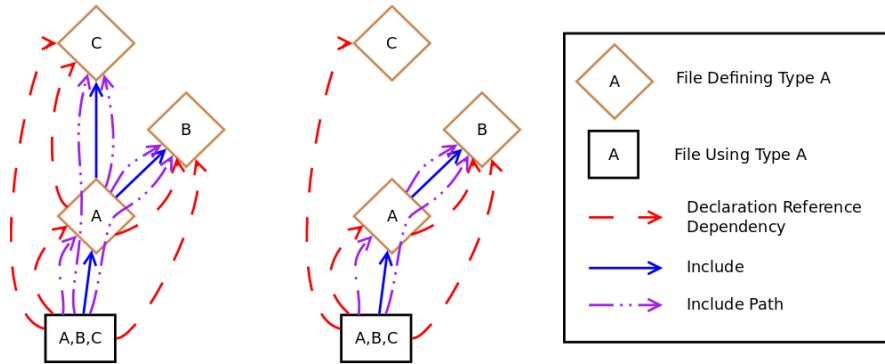


Figure 4.5: Problematic Example Graph

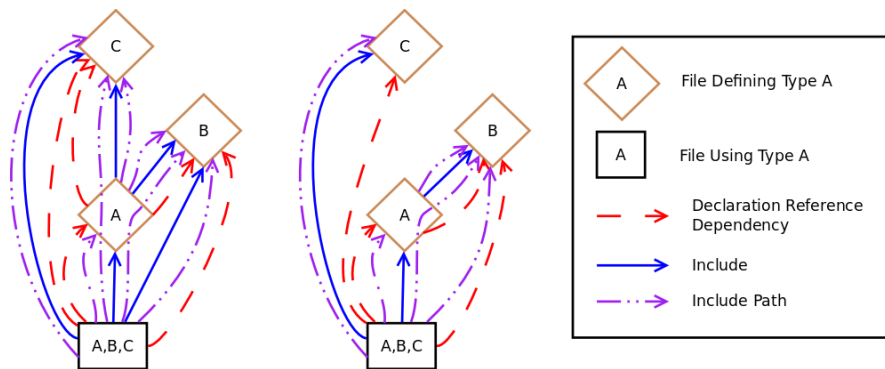


Figure 4.6: Unproblematic Example Graph

way of importing classes, since every class that is used needs to be imported directly in Java's code files. To refactor towards this goal, on the one hand one needs to add *include directives* for every not yet directly included *declaration*. On the other hand, one needs to remove *include directives* which do not include any used *declarations* directly.

The following listing shows the algorithm's pseudo code:

```

includeList := findListOfIncludes(presentFile)
includesToRemoveList := includeList
3
if existsFileNameCorrelatingHeaderFile(presentFile) {
    fileNameCorrelatingHeaderFile := getFileNameCorrelatingHeaderFile(presentFile)
6    addToList(includeList, findListOfIncludes(fileNameCorrelatingHeaderFile))
}
declarationRefList := findAllDeclarationReferences(presentFile)
9 filesToIncludeList := emptyList()

for declarationRef in declarationRefList {
12    declaration := findeDeclaration(declarationRef)
    referencedFile := getFileOfDeclaration(declaration)
    if not containedInList(includeList, referencedFile) {
15        addToList(filesToIncludeList, referencedFile)
    }
    removeFromList(includesToRemoveList, referencedFile)
18 }
proposeToInclude(filesToIncludeList)
proposeToRemove(includesToRemoveList)

```

The algorithm's pseudo code actually covers a special case which arises if the present file is an implementation file of a C++ class (e.g. `Something.cpp`) that is defined in the header file which correlates by file name to the present file (e.g. `Something.h`). In this case, all the files which are included in this header are treated as if they would be contained in the present file (pseudo code line 4-7).

Besides the effect of decreased coupling which was already described above, the algorithm has the positive effect that, even if the include list might grow significantly, there are no more unused *declarations* included than possible. This might result in a slightly optimized compile time. One should also be aware that this algorithm sometimes stands in conflict with the *find unused includes* algorithm. When including all used *declarations* directly, it might happen, as can be seen in the example in Figure 4.6, that some of them are, in the resulting include structure, included indirectly through several *include paths* which then will cause the *find unused includes* algorithm to propose the removal of some includes.

## 4.4 Find Unused Files

*Find unused files* is an algorithm that informs the user about files in his project which contain dead code. In a big C++ project this might come in very handy because the task would be very time-consuming and error-prone when executed manually. This algorithm is intended to be run on a full project and not on a single file. When it is triggered in

the context of a single file, the *find unused files* algorithm is triggered on the enclosing project instead.

The following pseudo code implementation introduces the algorithm:

```

1  unusedFileList := findAllFileInProject(presentProject)
4  for sourceFile in findAllSourceFilesIn(presentProject) {
   4  removeFromList(unusedFileList, sourceFile)
   7  for include in recursivelyGetIncludes(sourceFile) {
     7  includedFile := findIncludedFile(include)
     7  removeFromList(unusedFileList, includedFile)
   }
10 }
markFilesAsUnused(unusedFilesList)

```

Note that the *find unused files* algorithm, as it is described above, is also capable of finding several headers which are unused but which include each other. This can also be seen on the example run demonstrated bellow on the example project in Figure 4.7. Note that all the node labels in this graph do not represent *declaration* and *declaration reference* names as usual. Here they just represent the name of the file.

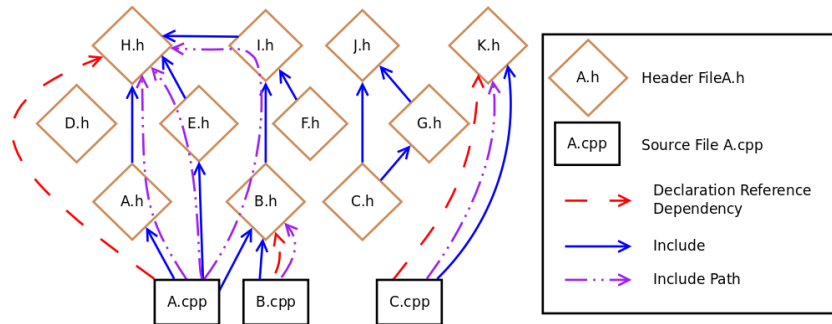


Figure 4.7: Example for Finding Unused Files

Each line in the following table represents one iteration of the outer loop in the pseudo code implementation (line 3).

Execution Point	unusedFilesList	sourceFile	Additional Information
Iteration 1	{A.cpp, B.cpp, C.cpp, A.h, B.h, C.h, D.h, E.h, F.h, G.h, H.h, I.h, J.h, K.h}	A.cpp	In the current iteration, the files {A.cpp, A.h, B.h, E.h, H.h, I.h} get removed from unusedFilesList
Iteration 2	{ B.cpp, C.cpp, C.h, D.h, F.h, G.h, J.h, K.h}	B.cpp	In the current iteration, the files {B.cpp, B.h, I.h, H.h} get removed from unusedFilesList
Iteration 3	{C.cpp, C.h, D.h, F.h, G.h, J.h, K.h}	C.cpp	In this iteration, the files C.cpp, K.h} get removed from unusedFilesList

The remaining files in the list `unusedFilesList` (C.h, D.h, F.h, G.h, J.h) are marked as unused.

## 4.5 Static Code Coverage

As already described in Section 4.3, there are often *declarations* included which are not all used.

The aim of the *static code coverage* algorithm is to highlight which parts of a project's code is used and which ones are unused. To achieve this goal, one searches all *declarations* in a project which are active. Then, one finds all used *declarations* by traversing the *ReDHead graph* along all the *declaration reference dependency* edges, starting with the previously found active *declarations*.

Note that such a traversal of the *ReDHead graph* is very similar to the slicing process. A slicing process, as introduced in detail in [HRB90], is performed on a graph similar to the SDG introduced in Section 2.2. The process performed here, in the *static code coverage* algorithm is, as mentioned, only similar.

A normal program slice is defined as parts of a program with respect to program point  $p$  and variable  $x$ . It consists of all statements and predicates of the program that might affect the value of  $x$  at point  $p$ . To produce a slice, a SDG is traversed along its *control flow* and *data flow* edges.

The *static code coverage* algorithm starts to slice with a given set of program point that are considered as active. Here, the focus is not on the *data flow* and the *control flow* edges, but rather only on the *function call* edges as in Figure 2.1. As already shown in the Section 2.3, what is a *call* edge in an SDG is represented in the *ReDHead graph* as a *declaration reference dependency*. So one might say that the *ReDHead static code coverage* algorithm is a more coarse grained slicing algorithm.

To find all active program points, one must first see if the project under consideration produces an executable, which implies that it has a `main` function, or if it does not, which makes the project, as it will be called hence, a *library project*.

In the case that there is a `main` function, this function combined, with all the public variables and static member variables, make the collection of active program points.

In the case of a *library project*, all *declarations* contained in any source, but not header files, are taken as active.



Here the algorithm in pseudo-code:

```

1  unusedDclarationList := recursiveFindAllIncludedDeclarations(presentProject)
   activeDeclarationList := findActiveDeclarations(presentProject)
   usedDeclarationList := emptyList()
4
   removeAll(unusedDclarationList, activeDeclarationList)
   addAll(usedDeclarationList, activeDeclarationList)
7
   for declaration in activeDeclarationList {
     containedDeclarationRefList := findContainedDeclarationRefs(declaration)
10    for declarationRef in containedDeclarationRefList {
      newDeclaration := findeDeclaration(declarationRef)
      appendToList(activeDeclarationList, newDeclaration)
13      addToList(usedDeclarationList, newDeclaration)
      removeFromList(unusedDclarationList, newDeclaration)
    }
16  }
   markUsed(usedDeclarationList)
   markUnused(unusedDclarationList)

```

Note that in pseudo code line 12, a *declaration* is appended to the list `activeDeclarationList`. The loop in line 8 is at that moment iterating over exactly this list. I assume here that (1) this does not break the execution of the pseudo code and (2) that the appended element is included into the iteration on line 8. This rather strange behavior is here used to keep the pseudo code as small and as simple as possible.

Figure 4.8 will demonstrate the algorithm in the given sample run.

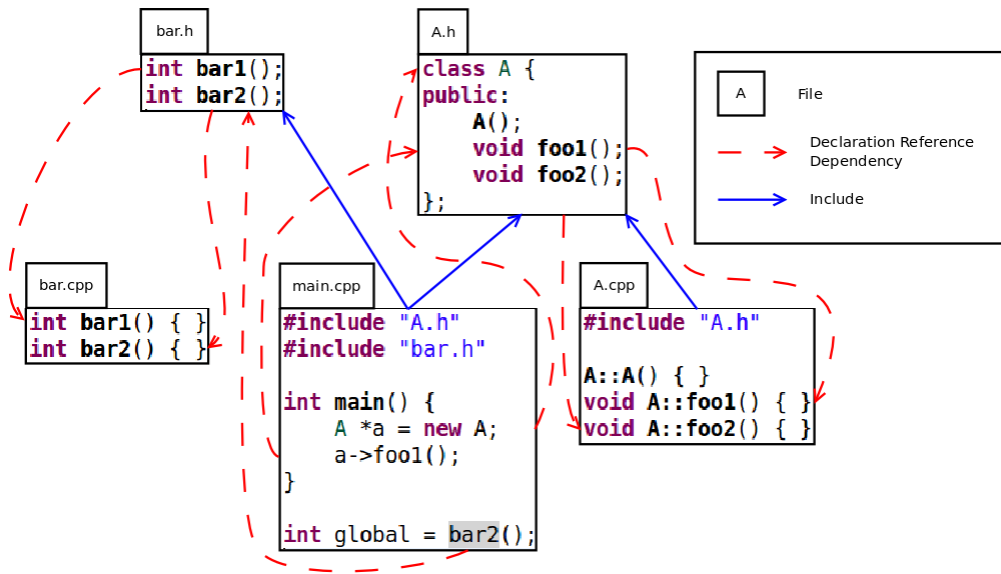


Figure 4.8: Static Code Coverage Example

In the sample run table which follows, the notation `global:main.cpp` can be read as *declaration of global in file main.cpp*. Note that each table entry represents an iteration

of the outer loop in pseudo code line 8. The `declaration` variable in line 8 is always the first element shown in column `activeDeclarationList`. This means that all the *declarations* that have already been iterated over, are not shown in the column anymore.

Execution Point	activeDeclarationList	unusedDclarationList	usedDeclarationList	Additional Information
Iteration 1	{main:main.cpp, global:main.cpp}	{A:A.h, foo1:A.h, foo2:A.h, A:A.cpp, foo1:A.cpp, foo2:A.cpp, bar1:bar.h, bar2:bar.h, bar1:bar.cpp, bar2:bar.cpp}	{main:main.cpp, global:main.cpp}	Function <code>main</code> uses <code>A:A.h</code> , <code>foo1:A.h</code> , <code>A:A.cpp</code> . These are removed from <code>unusedDclarationList</code> and added to <code>activeDeclarationList</code> and <code>usedDeclarationList</code>
Iteration 2	{global:main.cpp, A:A.h, foo1:A.h, A:A.cpp}	{foo2:A.h, foo1:A.cpp, foo2:A.cpp, bar1:bar.h, bar2:bar.h, bar1:bar.cpp, bar2:bar.cpp}	{main:main.cpp, global:main.cpp, A:A.h, foo1:A.h, A:A.cpp}	Definition <code>global</code> uses <code>bar2:bar.h</code> which is removed from <code>unusedDclarationList</code> and added to <code>activeDeclarationList</code> and <code>usedDeclarationList</code>
Iteration 3	{A:A.h, foo1:A.h, A:A.cpp, bar2:bar.h}	{foo2:A.h, foo1:A.cpp, foo2:A.cpp, bar1:bar.h, bar1:bar.cpp, bar2:bar.cpp}	{main:main.cpp, global:main.cpp, A:A.h, foo1:A.h, A:A.cpp, bar2:bar.h}	Class declaration <code>A</code> does not refers anything so nothing is added or deleted.
Iteration 4	{foo1:A.h, A:A.cpp, bar2:bar.h}	{foo2:A.h, foo1:A.cpp, foo2:A.cpp, bar1:bar.h, bar1:bar.cpp, bar2:bar.cpp}	{main:main.cpp, global:main.cpp, A:A.h, foo1:A.h, A:A.cpp, bar2:bar.h}	Member function declaration <code>foo1</code> refers <code>foo1:A.cpp</code> which is removed from <code>unusedDclarationList</code> and added to <code>activeDeclarationList</code> and <code>usedDeclarationList</code>
Iteration 5	{A:A.cpp, bar2:bar.h, foo1:A.cpp}	{foo2:A.h, foo2:A.cpp, bar1:bar.h, bar1:bar.cpp, bar2:bar.cpp}	{main:main.cpp, global:main.cpp, A:A.h, foo1:A.h, A:A.cpp, bar2:bar.h, foo1:A.cpp}	Constructor definition <code>A</code> does not refers anything so nothing is added or deleted.
Iteration 6	{bar2:bar.h, foo1:A.cpp}	{foo2:A.h, foo2:A.cpp, bar1:bar.h, bar1:bar.cpp, bar2:bar.cpp}	{main:main.cpp, global:main.cpp, A:A.h, foo1:A.h, A:A.cpp, bar2:bar.h, foo1:A.cpp}	Function declaration <code>bar2</code> refers <code>bar2:bar.cpp</code> which is removed from <code>unusedDclarationList</code> and added to <code>activeDeclarationList</code> and <code>usedDeclarationList</code>
Iteration 7	{foo1:A.cpp, bar2:bar.cpp}	{foo2:A.h, foo2:A.cpp, bar1:bar.h, bar1:bar.cpp}	{main:main.cpp, global:main.cpp, A:A.h, foo1:A.h, A:A.cpp, bar2:bar.h, foo1:A.cpp, bar2:bar.cpp}	Member function definition <code>foo1</code> does not refers anything so nothing is added or deleted.
Iteration 8	{bar2:bar.cpp}	{foo2:A.h, foo2:A.cpp, bar1:bar.h, bar1:bar.cpp}	{main:main.cpp, global:main.cpp, A:A.h, foo1:A.h, A:A.cpp, bar2:bar.h, foo1:A.cpp, bar2:bar.cpp}	Function definition <code>bar2</code> does not refers anything so nothing is added or deleted.
Iteration 9	{}	{foo2:A.h, foo2:A.cpp, bar1:bar.h, bar1:bar.cpp}	{main:main.cpp, global:main.cpp, A:A.h, foo1:A.h, A:A.cpp, bar2:bar.h, foo1:A.cpp, bar2:bar.cpp}	Loop iteration ends here since <code>activeDeclarationList</code> is empty.

The result of the algorhtim run is, that the *declaration* and *definition* for both `foo2` and `bar1` are unused and the rest of the project is in use.

## 4.6 Replace Includes with Forward Declarations

If a type gets used in a file and is only used as pointer or reference type, there is no requirement to actually include the full *declaration* of that type. Instead, a *forward declaration* is enough.

```
#include "A.h" // Example with an include directive
3 class B {
  public:
    void foo(const &A a);
6 };
```

```
class A; // Example with a forward declaration
3 class B {
  public:
    void foo(const &A a);
6};
```

The example given above demonstrates this. The advantage of the second listing over the first one is that the coupling between the two involved files in the first listing is removed and that the compile time is diminished.

The idea behind the *replace include with forward declaration* algorithm is to find scenarios similar to the first listing and refactor them so they become like the second listing.

Note that *forward declarations* are always possible where an *incomplete type* is sufficient. More information about *incomplete types* can be found in the C++ Standard [Ins03].

The following listing shows a pseudo code implementation of the algorithm:

```
declarationRefList := findAllDeclarationRefs(presentFile)
declarationSet := findAllReferencedDeclarations(declarationRefList)
3
4 for declaration in declarationSet {
5   refsToDeclarationList := findRefsToDeclaration(declarationRefList, declaration)
6   if canReplaceWithForwardDeclaration(declaration, refsToDeclarationList) {
7     includesToRemove := findRemovableIncludesTo(declaration) //normally only 1
8     proposeForRemoval(includesToRemove)
9     proposeToAddForwardDeclaration(declaration)
10  }
11 }
```

Note that the *declaration* which are found in pseudo code line 2 are added to a *set*, which implies that, even if there are several different *declaration references* to the same *declaration*, only one of them will be contained in the *set*. This is enough to represent the relation to the *declaration*.

The function `canReplaceWithForwardDeclaration()` checks several conditions which needs to be satisfied so that an include can be replaced with a *forward declaration*. These conditions are introduced in the following listing.

```
1  canReplaceWithForwardDeclaration(refsToDeclarationList, declaration) {
    if not areUsedInOnlyDeclaration(refsToDeclarationList) and not ←
        areOnyPointerAndRefs(refsToDeclarationList) {
2      return false
3    }
4    includePathList := findPathsTo(declaration) //normally only 1 path
    includeList := firstElementsOf(includePathList)
7    if not areAllIncludesOnlyUsedToResolve(includeList, refsToDeclarationList) {
        return false
8    }
9    return true
10 }
```

Note here that the `areUsedInOnlyDeclaration()` function tests, if the `refsToDeclarationList` are only used in *declarations* which are not themselves *definitions*.

#### 4.6.1 Refactor Towards *iosfwd*

The standard C++ library contains a file named `iosfwd` which contains *forward declarations* for common input and output stream types. As an algorithm extension to the one which replaces includes with *forward declarations*, one could in the case of streams, add an *include directive* to `iosfwd` instead of *forward declaring* the used stream types.

## 4.7 Introduce Redundant Include Guards

Every C++ developer knows that *include guards* are necessary to prevent redefinition of symbols (mostly types). *Include guards* achieve that each file is only included exactly once. So during compilation, the same include files is opened and closed in numbers, only to find that there is nothing to do with it because it was already included earlier. In big software project, this results in a considerable amount of time that is wasted by opening and again closing files. The open and close operations executed by the compiler, however, can be prevented by introducing *redundant include guards* in the file which includes a header file. A more detailed description of *redundant include guards* can be found in [Lak96] on page 82. Here a dummy example:

```
1  //main.cpp
    #include "header.h"
    #include "header.h"
4  #include "header.h"
    #include "header.h"
```

```

1 //header.h
  #ifndef HEADER_H
  #define HEADER_H
4
  class A { };
7 #endif /* HEADER_H */

```

When compiling the code from the previous listings, the file *header.h* is opened and closed four times. The operation, however, was only required exactly once. By introducing *redundant include guards* into *main.cpp*, as can be seen in the following listing, results in faster compile time.

```

2 //main.cpp
  #ifndef HEADER_H
  #include "header.h"
  #endif /* HEADER_H */
5 #ifndef HEADER_H
  #include "header.h"
  #endif /* HEADER_H */
8 #ifndef HEADER_H
  #include "header.h"
  #endif /* HEADER_H */
11 #ifndef HEADER_H
  #include "header.h"
  #endif /* HEADER_H */
14 #ifndef HEADER_H
  #include "header.h"
  #endif /* HEADER_H */

```

Adding *redundant include guards*, as demonstrated above, is an annoying, time-consuming and also error-prone task. Thus, adding automated support in form of a ReD-Head algorithm that adds *redundant include guards* will be a very valuable feature when utilizing *redundant include guards*. The following listing demonstrates the algorithm:

```

allIncludesList := findAllIncludes(currentFile)
2
  for include in allIncludesList {
    includedFile := includedFileOf(include)
5    if not alreadySurroundedByIncludeGuards(include) and containsIncludeGuards(←
      includedFile) {
      includeGuardName := findIncludeGuardNameIn(includedFile)
      proposeToSurroundWithIncludeGuard(include, includeGuardName)
8    }
  }

```

## 4.8 Finding Optimal Insert Positions

Some of the *static include analysis* algorithms described above propose that new include directives should be added to a code file. If a software engineer decides to apply the proposed changes, the questions arises where the optimal position is to insert the proposed include directives.

This section describes the ReDHead component which calculates insert offset for include directives for a given source or header file.

The simplest answer to this position question is to just insert them at the beginning of the document. There are two problems with this solution. The first one is that, when inserting include directives into a header file, the include guards are bypassed. The second one is that, when inserting into a source file, one is strongly disadvised to insert additional include directives in front of the one that includes the header file which correlates by name to the source file's name. The reason for this is given on page 110 in [Lak96].

So a better place to insert includes is after the last of the already contained include guards. However, this is tricky because one should never insert in the scope of a conditional compilation block.

```
3 #include "Onething.h"
  #ifdef WIN32
  #include "Anotherthing.h"
  #endif
  /* ... */
```

For the code given in the listing above, the correct insert position is not after the second include directive but after the `#endif` statement. However, that one should never insert inside of a conditional compilation block is not true in a single case. This is the case with *include guards*.

```
1 #ifndef A_FILE_H
  #define A_FILE_H
  #include "Onething.h"
  #include "Anotherthing.h"
  class AClass { };
  #endif
```

Inserting after the `#endif` here is wrong because it belongs to the *include guard*. The listing above brings us to the next restriction. One of course should never insert after *declaration* (here `AClass`). So the point to insert in the listing above would be is obvious place after the last include directive and before the `AClass` declaration.

In the case that there is an *include guard* but no include directives, then the point to insert further includes is directly after the definition of the *include guard's* symbol.

A sample of the conflicting case can be found in the following listing:

```
3 #ifndef A_FILE_H
  #define SOMEOTHER THING
  #include "Onething.h"
  #include "Anotherthing.h"
  class AClass { };
  #endif
```

In this case there is no valid *include guard* which means that there is only a conditional compilation block. Firstly, we want to insert after that conditional compilation block.

But secondly, we cannot insert after the first *declaration* in the code. The tradeoff which is taken here is to insert at the beginning of the document.

## 4.9 Include Substitution

In the case of certain external library header files, the process of algorithms which propose to add includes is sometimes to explicit. I will demonstrate the problem with an example from the *C++ Standard Library*. Imagine the following C++ code:

```
3 int main() {  
    std::vector<int> v;  
    return 0;  
}
```

When running *find unused includes*, adding `#include <bits/stl_vector.h>` would be suggested. This suggestion is not wrong at all since the class `vector` is declared in exactly that file. But still, a software engineer would be surprised by the suggestion since one expects the proposal of `#include <vector>`.

To solve the problem at hand, ReDHead algorithms do, before suggesting to insert includes, check a given list of *include substitutions* which in cases as the one mentioned above yields the correct include proposal.

This list was not created by hand, but is rather generated automatically from the given C++ standard library header directory.

## 5 ReDHead Data Structure

The algorithms described in the previous Chapter 4 are, as already mentioned, based on the the *ReDHead data structure*, of which the functionality is describe in the current chapter in detail. The *ReDHead data structure* is an implementation of the *ReDHead graph*, that was described in Section 2.3. Basically, the *ReDHead data structure* gives access to the vertices and edges that were described with the ReDHead graph. Also, this chapter will contain *ReDHead graph* images which are used to describe special code scenarios.

Note that this chapter gives no internal implementation details of the *ReDHead data structure*. More information about internal implementation details can be found in Section 6.2 in the following Chapter 6.

The following class diagram in Figure 5.1 shows an overview of the of the important classes that are contained in the *ReDHead data structure*.

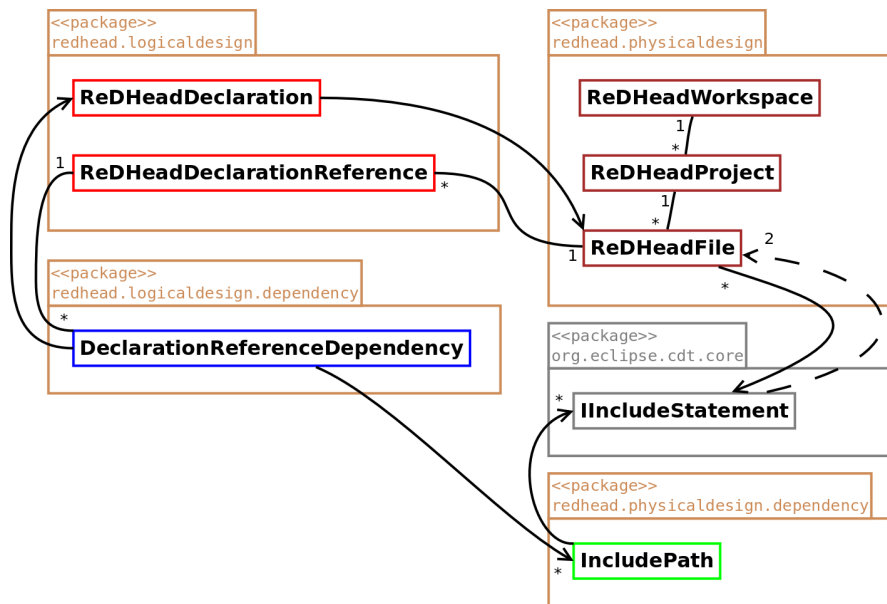


Figure 5.1: ReDHead Data Structure

Each of the components in the `redhead.logicaldesign` package grant access to the component's name as well as to location information which consists of file offset, line number, line offset and length. The components in the `redhead.physicaldesign` package can each yield information about its location, meaning a file or folder path.



How the *ReDHead data structure* can be used to traverse the graphs, which were introduced in Section 2.3 and Chapter 4, can be seen in Figure 5.1 by following the diagram's edges. As can be seen in Figure 5.1, a `ReDHeadDeclarationReference` can yield several `DeclarationReferenceDependencies`. Actually, there are several methods available to retrieve a part of these available dependencies. In the Section 5.3 a detailed description is given on what these methods are and why they are necessary.

The following sections in this chapter describe noteworthy details about the *ReDHead data structure* that were important to implement the algorithms described in Chapter 4.

## 5.1 Logical and Physical Design

The terms *logical* and *physical design* have already briefly been mentioned in Section 2.2. Here I will give a more detailed explanation on the two design approaches.

In the diagram in Figure 5.1, one can see the *logical design* components on the left hand side and the *physical design* components on the right hand side. Note that, compared to the *ReDHead graph*, the *physical design* components were extended by the `ReDHeadProject` and the `RedHeadWorkspace`.

*Logical design* is what most software engineer understand just as design of a software product. Logical design components are for example *functions*, *classes*, *structs* and *namespaces*. They can be used to organize the functionality of a software product so it gets a useful, understandable and does not contain redundant code. Some of the mentioned components use others. This creates coupling between these components which is represented in the *ReDHead data structure* with the `DeclarationReferenceDependency` class instances..

*Physical design*, on the other hand, is something very similar in the task of designing. But one could say it is a second dimension of the design process. Here, the question is how to organize a system's physical units. These units are files and directories, and the relation between them, the include directives. It also belongs to the tasks of the *physical design* to put the components of the *logical design* into the physical units in a smart manner.

Obviously, the two ways to design are not completely independent of each other since the coupling which, exists in the *logical design*, influence the include structure of a project which belongs to the *physical design*. And the *logical design's* components are distributed in the *physical design's* units, the files. To develop a good software product, both the task of *logical* and *physical design* should get enough attention. If one is neglected, the software products code quality is decreased.

## 5.2 Declaration References

`RedHeadDeclarationReferences` can be obtained from any `RedHeadFile`. Often, many `RedHeadDeclarationReferences` contained in one file refer exactly to the same `RedHeadDeclaration`. To prevent the repeated execution of the same operation on similar `RedHeadDeclarationReferences`, only one of these similar `RedHeadDeclarationReferences` is returned by the class `RedHeadFile`. So the set returned contains only as many `RedHeadDeclarationReferences` as there are different `RedHeadDeclarations` referenced. This optimizes the runtime of the algorithms which use the *RedHead data structure*

## 5.3 Declaration Reference Dependencies

A `RedHeadDeclarationReference` is able to return several `DeclarationReferenceDependency` instances.

Note here that, to improve the readability, this was neglected in the pseudo code implementations of the `RedHead` algorithms in Chapter 4.

A `RedHeadDeclarationReference` is also able to filter this collection of dependencies according to certain criteria. The following example *RedHead graphs* in this section introduce these filtering functions of the `RedHeadDeclarationReference` in detail.

Figure 5.2 contains four code files, one contains the definition of the class `X`, two others each contain a forward declaration of the class `X`. The last file contains a `RedHeadDeclarationReference` to class `X`. The class `RedHeadDeclarationReference` contains the method `getAllDependencies()`, which returns the complete list of available `DeclarationReferenceDependencies`, which here are all the red, dashed edges. Note that the list also include a dependency to the forward declaration which is not included in the file *main.cpp*.

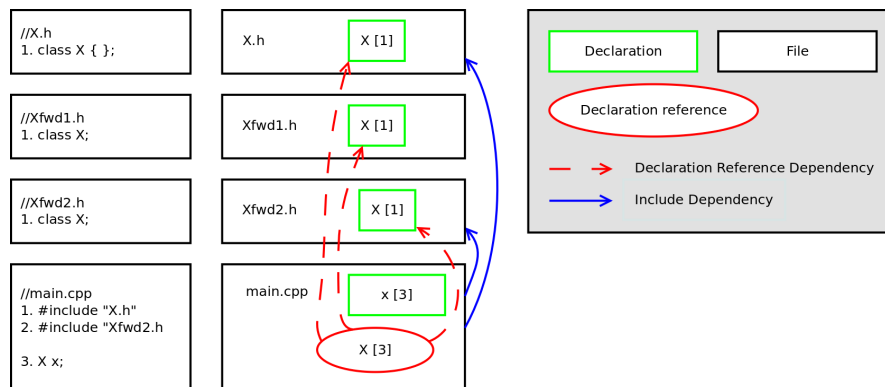


Figure 5.2: All Available `DeclarationReferenceDependencies`

The graphs shown in Figure 5.3 shows all the `DeclarationReferenceDependencies` which are returned by the method `getIncludedDependencies()` of the class `RedHead-`

**DeclarationReference.** The dependency to the file *Xfwd1.h* which is not included, was grayed out and is not contained in the collection returned by `getIncludedDependencies()`.

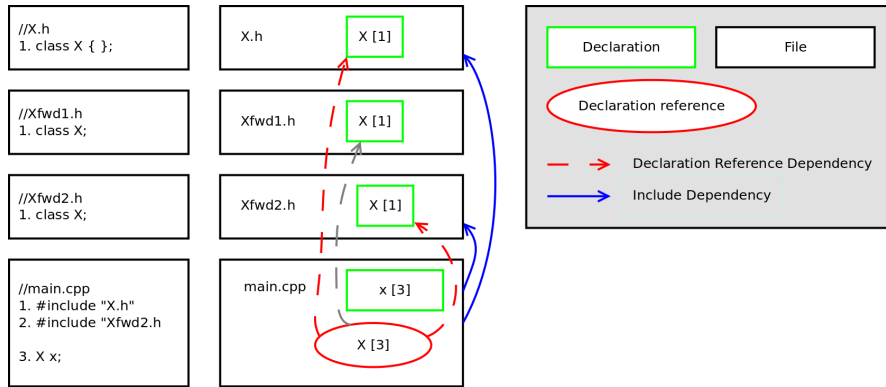


Figure 5.3: Only Included DeclarationReferenceDependencies

Method `getRequiredDependencies()` returns only these *DeclarationReferenceDependencies* that are required, which is here the one to the *declaration* of class *X* in file *X.h*. To find out which of all the available *DeclarationReferenceDependencies* are required, the *ReDHead* data structure evaluates if a forward declaration of class *X* is sufficient or not. In Figure 5.4, `x` in the file *main.cpp* is neither a pointer nor reference-variable, so the *definition* of *X* is required.

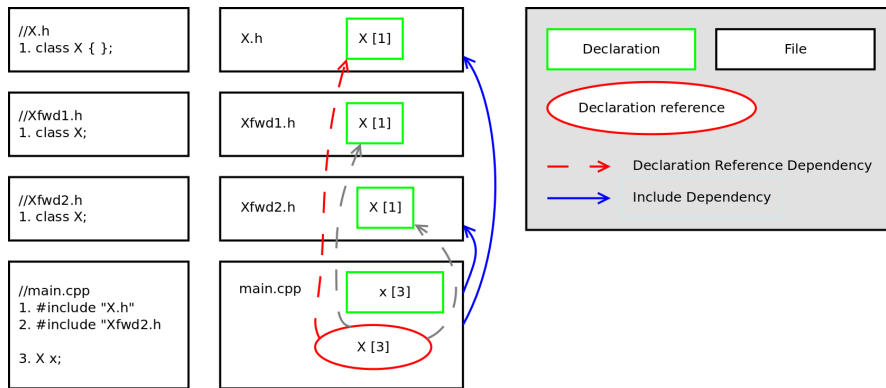


Figure 5.4: Only Required DeclarationReferenceDependencies

In Figure 5.5, the class *X* is in *main.cpp* only used as a pointer-variable. So a forward declaration is enough. This results that in this case, `getRequiredDependencies` of the class `ReDHeadDeclarationReference` will return the dependency to the forward declaration in the file *Xfwd2.h*. Note here, that this dependency to the forward declaration is preferred over the one in the file *Xfwd1.h*, because *Xfwd1.h* is not included into *main.cpp*.

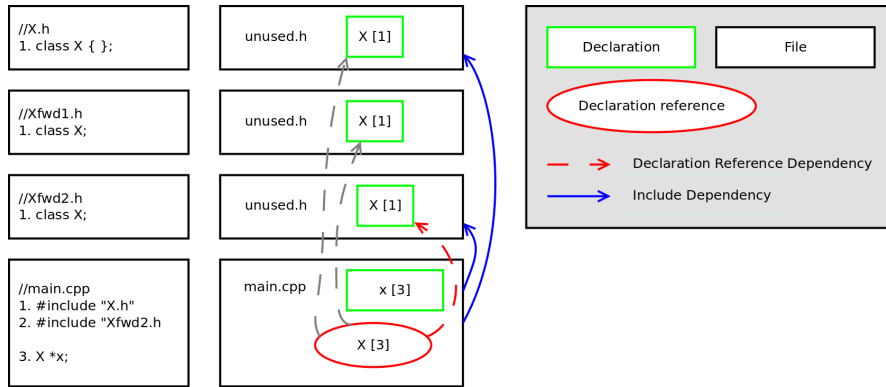


Figure 5.5: Only Required DeclarationReferenceDependencies

Namespaces in C++ are a somewhat special construct because they can be defined several times. Initially, this leads to the problem that the algorithms *find unused includes* proposed to include all the files which contain a definition of a used namespace, which of course is wrong. The solution of the problem in the *ReDHead data structure* is, that only these definitions are referenced, which are also used. What *used* here means can be seen in Figure 5.6. The one dependency to file *unused.h* is not returned because it is also not used.

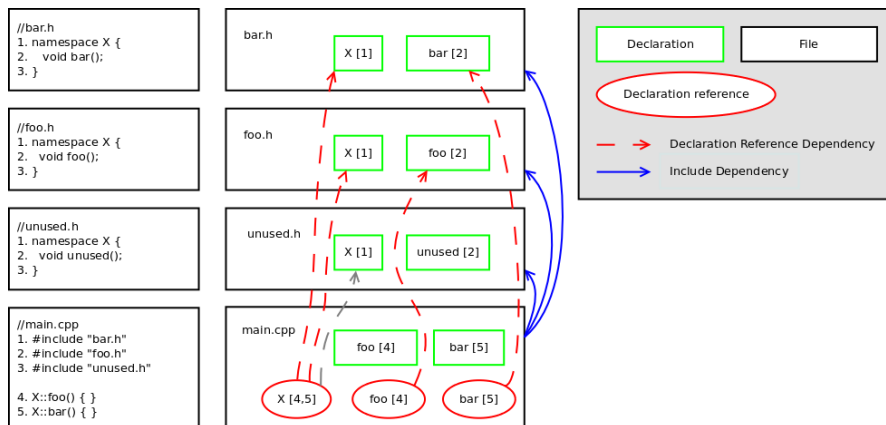


Figure 5.6: Only Required Namespace DeclarationReferenceDependencies

The *ReDHead graph* does not represent the *ReDHead data structure* in every detail. The reason for this is, that the code which represents a `RedHeadDeclaration` of course contains the name which is declared. So the code that yields `RedHeadDeclaration` actually also always yields a `RedHeadDeclarationReference`. This means that for each *declaration* contained in a *ReDHead graph*, there should also be a `DeclarationReferenceDependency` edge that leads to itself. The reason, not to show these edges in the *ReDHead graph*, is, that this would only be redundant information which would bloat the graph.

In the *ReDHead data structure* however, this so called *self dependency* is useful since otherwise the resolution of a `DeclarationReference` that originates from the code of a *definition* would sometimes not yield any `DeclarationReferenceDependencies`. Yielding one `DeclarationReferenceDependency` to itself is a more dynamic solution.

As shown in Figures 5.1, one `ReDHeadDeclarationReference` in some cases return not only one, but rather several, `DeclarationReferenceDependencies`. The following subsections, besides the examples given above, show some additional special cases which should be considered to understand the *ReDHead data structure*.

### 5.3.1 Preprocessor Symbols

Normally, the *ReDHead data structure* contains `DeclarationReferenceDependencies` independent of whether the file containing the `ReDHeadDeclarationReference` is included by the one that contains the `ReDHeadDeclarationReference`. In the case of preprocessor symbols, as shown in Figure 5.7, this is not the case. In the case in which such a `DeclarationReferenceDependency` would be available, the algorithm *find unused includes* would falsely propose to include file *other.h*, which would then practically destroy the effect of conditional compilation, which of course we do not want.

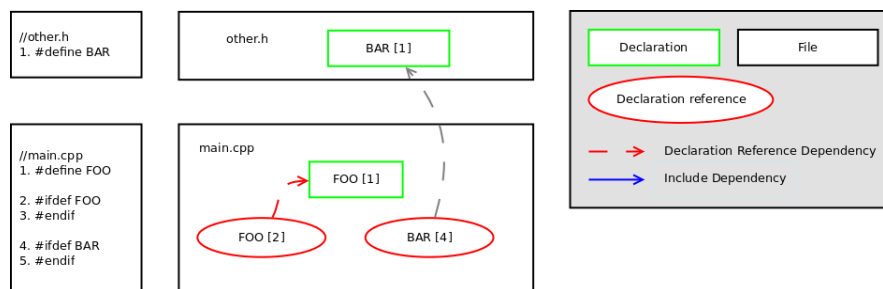


Figure 5.7: Preprocessor Symbols

### 5.3.2 Function-like Macros

The example Figure 5.8 shows the `DeclarationReferenceDependencies` that are produced for function-like macros. Important here to understand is, that, since macro expansion is textual replacement, the call to the `AB()` macro in file *DIRECT\_CALL.h* does not create a `DeclarationReferenceDependency` to the macro's definition which also originates from *DIRECT\_CALL.h*. Instead, these `DeclarationReferenceDependencies` originate in files containing calls to that macros, which is the file *main.cpp* in the example.

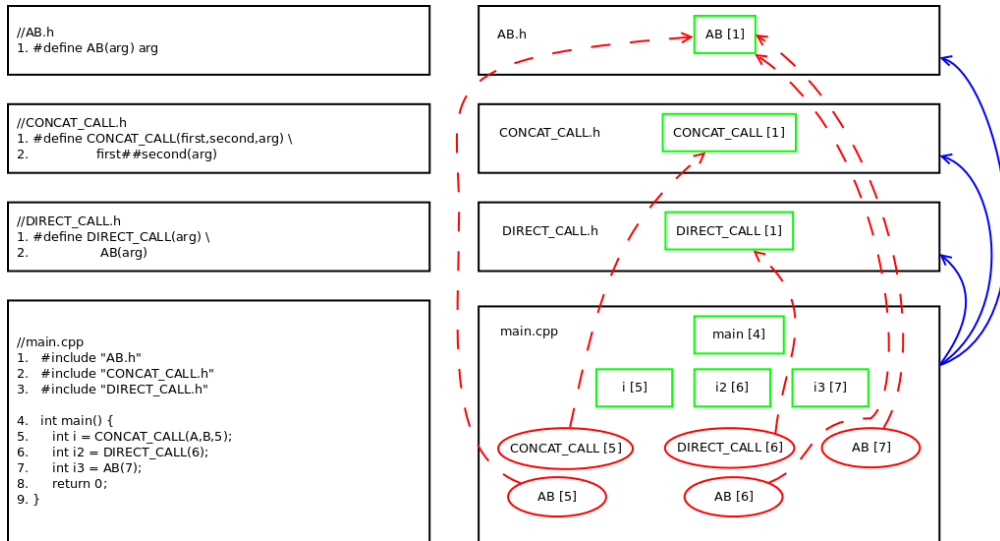


Figure 5.8: Function Style Macros

### 5.3.3 Templates

Figure 5.9 demonstrates, that `RedHeadDeclarationReferences` which originate from template functions or template classes, often yield several `DeclarationReferenceDependencies` to any `RedHeadDeclaration` which might match. Important here is, that there is no determination if all these `RedHeadDeclarations` are really required eventually. In certain cases, this could result in false positives in proposition made by some `RedHead` algorithms.

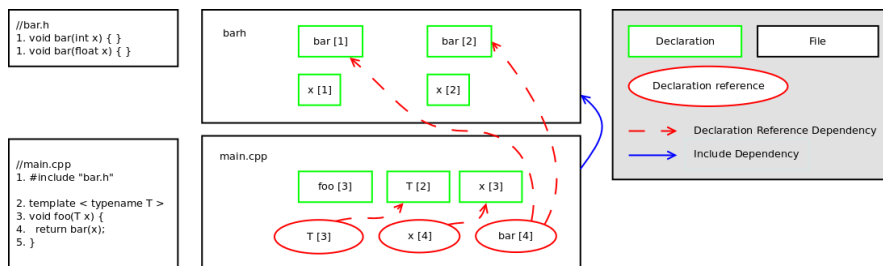


Figure 5.9: C++ Templates Example

## 5.4 Includes

Instead of introducing an own class that represents include relations, `RedHead` uses the existing class of `CDT` to represent *include relations*, since the required information about the include relation can be acquired there. The class in Figure 5.1 which is labeled `IIncludeStatement` does not exist with that name. The reason to call it like this, is,

that there are actually two classes which are used to represent include directives in CDT, which can both be used for my cause. Sadly, they do not share any common interface or super class.

Also, the Figure 5.1 is not completely correct here because it makes the reader believe that the `IIncludeStatement` gives access to the included `ReDHeadFile` instance (dashed arrow edge), which of course is not true since CDT has no relations to `ReDHead`. Instead of directly giving access to the included `ReDHeadFile`, a helper class' method takes care of that task.

## 5.5 Include Paths

*Include paths* are, as first mentioned in 1.4, an ordered set of include relations. Very important here is the fact, that in the *ReDHead data structure*, an *include path* is defined as the path that leads out of any file to a target file. The originating file, however is no part of the path. So the `IncludePath` instances in Figure 5.10, are all considered equal. This allows for example to easily compare include paths independent of the file which contains them.

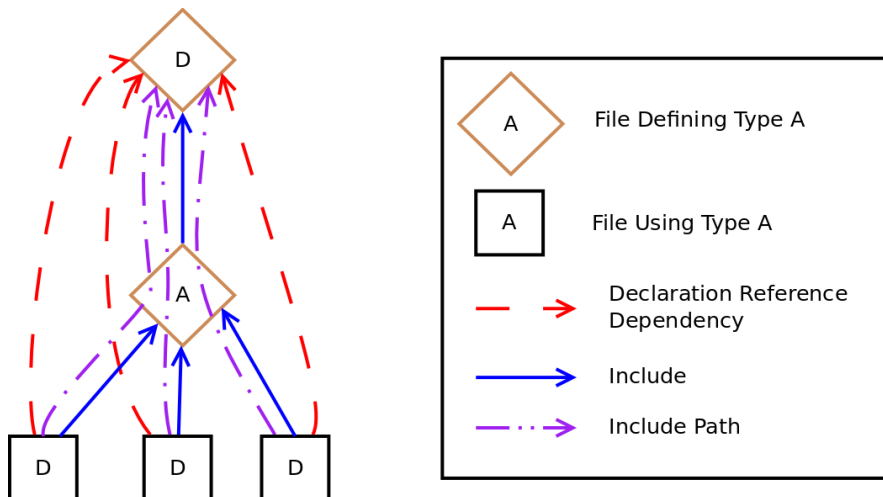


Figure 5.10: Equal Include Paths Example

## 6 Implementation

This chapter describes implementation details which are worth mentioning. To give an overview of the whole plugin, the class diagram of the important classes is shown in Figure 6.1. For detailed description of the classes shown in the diagram, read the following sections.

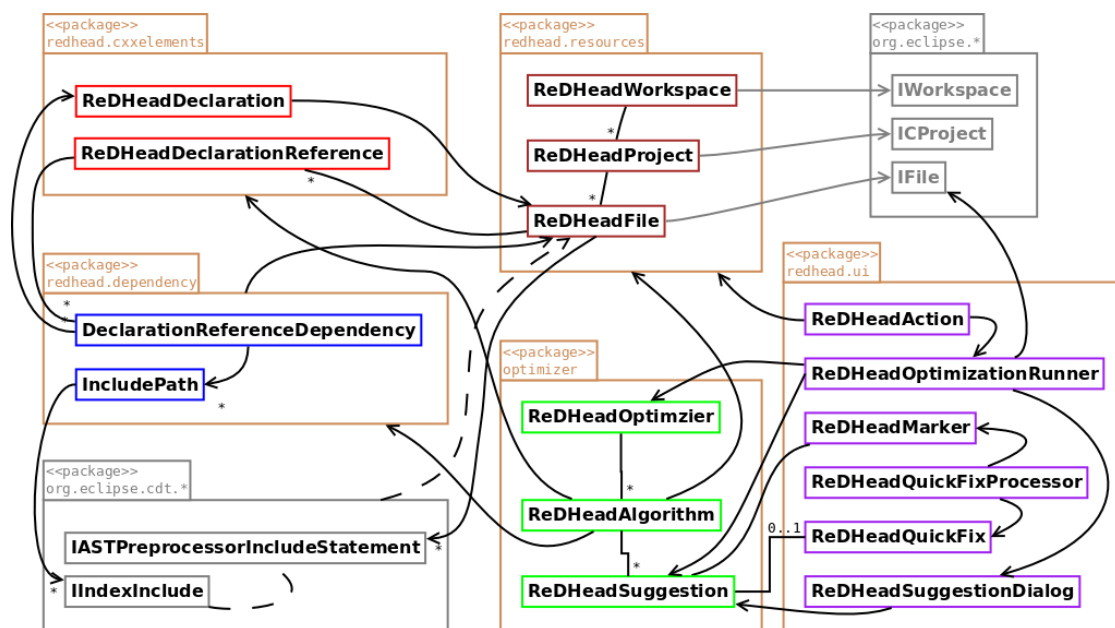


Figure 6.1: ReDHead Class Diagram

### 6.1 Used CDT Functionality

The *ReDHead data structure* which was introduced in Chapter 5 gets constructed with information which is delivered from CDT components. The important part of these components are introduced here.

#### 6.1.1 AST

*Abstract Syntax Trees* in CDT are instances of `IASTTranslationUnit`. These grant access to `IASTNode` that represent *declarations*, *includes*, *macro definitions* and so on. To find certain node type one can use *node visitors* to traverse AST's. Instances of



`IASTTranslationUnit` can be constructed with the help of a file-name and the CDT helper class `CoreModelUtil`.

### 6.1.2 Indexer

The CDT indexer is a very important CDT component, since it provides the *ReDHead data structure* with the functionality to find the dependencies between *declaration references* to *declarations*. Concretely, it can resolve `IBindings` which for example can be acquired by the AST nodes of type `IASTName` as described in 6.1.3.

Also, the CDT indexer can produce lists of all transitively included files that are included into one given file.

`IIndex` instances can be acquired through the CDT plugin class `CCorePlugin`.

### 6.1.3 INames

Instances of CDT's `IName` represent *C++ identifiers* in C++ code. In CDT, there are two basic subtypes of `IName`. These are `IASTName` and `IIndexName`. The first one can be found by traversing ASTs, the second one by querying the CDT indexer. `ReDHead` uses `INames` when producing the *ReDHead data structure*.

## 6.2 ReDHead Data Structure

This section will describe internal implementation details of the *ReDHead data structure*. For a detailed description of the functionality provided by the *ReDHead data structure* I refer to Chapter 5.

Figure 6.2 shows the *ReDHead data structure*'s starting components for the `ReDHead` algorithms.

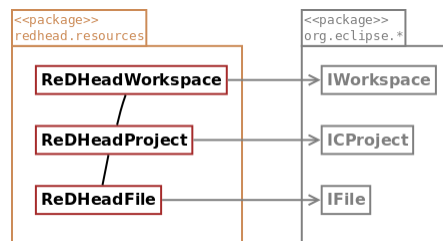


Figure 6.2: Basic `ReDHead` Data Structure

I decided to implement my own file, project and workspace classes to avoid a lot of helpers and wrappers. The reason for this decision are experiences of several other Eclipse plugin projects I worked on that also used existing classes which could not be adapted or extended directly. These are namely *Ruby Refactoring* [CFS06] and *CDT C++ Refactoring* [Ins].

Because I work with Eclipse and CDT, the ReDHead data structure is based on the resource classes provided from Eclipse and CDT as shown in Figure 6.2. The ReDHead resource classes add additional functionality which is used by the ReDHead algorithms.

In order to be able to optimize *static include analysis*, additional information is needed. An overview over the components which provide this information is given in Figure 6.3.

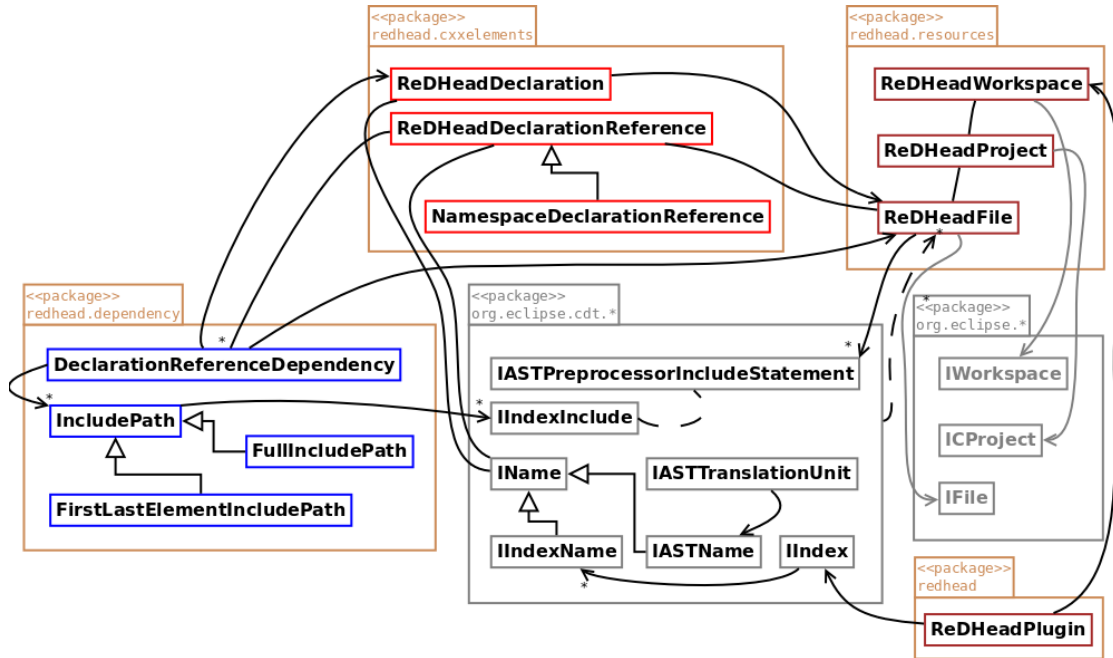


Figure 6.3: Complete ReDHead Data Structure

Note that the dashed diagram relations require a helper class method in order to find the related component.

Also note that package names all begin with *redhead*. This is short for *ch.ifs.hsr.redhead*, which is the base name of all ReDHead packages.

The following paragraphs describe the components contained in the class diagram. The description contains explanation on what other components are required to construct a component and where ReDHead gets access to it.

## Resource Classes

Classes contained in the resource package *redhead.resources* represent files or folders used in the ReDHead plugin. All these classes are initiated with their correlating Eclipse equivalent, which can be accessed through static methods of the Eclipse class *ResourcesPlugin*.

Access to *ReDHeadFiles* is granted by *ReDHeadProject* instances which themselves can be found with the help of the *ReDHeadWorkspace*. The *ReDHeadWorkspace* is ac-

cessible through the *Singleton* instance of the RedHead plugin (see Singleton pattern in [GHJV97]).

Each `RedHeadFile` contains the AST that represents the file. This AST is used either by the file itself to construct `RedHeadDeclarationReference` and to return includes or for individual analysis by any RedHead algorithm.

## C++ Element Classes

The classes in the package `redhead.cxxelements` encapsulates a certain CDT AST or index element. Information that is required by RedHead algorithms considering these classes can be accessed there.

### RedHeadDeclarationReference

A `RedHeadDeclarationReference` can be seen as an abstract AST node representation. It represents a C/C++ identifier that can be resolved to a `RedHeadDeclaration`, which makes the AST node that it contains an instance of `IASTName`. `RedHeadDeclarationReferences` are accessible through a `RedHeadFile` which creates `RedHeadDeclarationReference` instances by traversing the file's AST for certain *IASTName* instances. The `RedHeadDeclarationReference` to a collection of `DeclarationReferenceDependencies`.

A `RedHeadDeclarationReference`, which represents a C++ namespace name, is instantiated as the subclass `NamespaceDeclarationReference`, which uses an additional filtering criteria in the overloaded method `getRequiredDependency()`. This is required to achieve the behavior described in Section 5.3 (Figure 5.6).

### RedHeadDeclaration

A `RedHeadDeclaration` encapsulates an `IName`. It is the representation of that AST node to which a `RedHeadDeclarationReference` resolves to. The `IIndexName` instance, which is necessary to create a `RedHeadDeclaration`, is returned when resolving the `IBinding` of the `IASTName` contained in a given `RedHeadDeclarationReference`.

Resolving and acquiring the `IBinding` is somehow complicated because there are several different approaches which need to be combined to find the correct `IName` instances.

## Dependency Classes

Dependency classes represent dependencies between other RedHead classes. There is always a origin and a target file involved.

### DeclarationReferenceDependency

The `DeclarationReferenceDependency` class represents a link between a `RedHeadDeclarationReference` and a `RedHeadDeclaration`. For its creation, an instance of one

of the linked classes, which are described above, is required. The `DeclarationReferenceDependency` gives access to `IncludePaths`. The `DeclarationReferenceDependency` is the right class for this, because it can get access to both the originating and target `ReDHeadFile` of the `SourcePaths`.

## IncludePath

An `IncludePath` represents one path of include edges from an include `ReDHeadFile` to a target `ReDHeadFile`. This can be seen as the *physical design* equivalent of a `DeclarationReferenceDependency`, which itself a *logical design* element (see 5.1 for information about the two designs). The small difference here is, that, as described in 5.5, `IncludePath` does not contain the originating `ReDHeadFile` as starting element but starts with the file that is included.

As can be seen in the Figure 6.3, the elements contained in `IncludePaths` are not files, but rather `IIndexInclude`. These can be used to find both the including and the included file. Important to know here is, that `ReDHead` only makes use of the included file. This gives the advantage that `IncludePath` instances can be reused and compared, independent of the file that contains the first include. So as an example, each `IncludePath` generated by the code `#include <vector>`, independent of the file that contains it, is considered equals.

Note also here, that, instead of `IIndexIncludes` one could also use `IASTPeprocessorIncludeStatements`. They can basically provide the same information as the `IIndexInclude`. In spite of the same functionality, as already mentioned in Chapter 5, they do not share any common interface or base class.

`IncludePaths` are always constructed as a collection since there might always exists several paths from a first included file to a target file. To explain the construction of `IncludPath` instances, I must first explain the two subclasses of `IncluePath`.

Up to now, what was referred to as a *include path* is actually an instance of `FullIncludePath`. The `FullIncludePath` contains all the path elements starting with the first included file. To construct a `FullIncludePath` the whole include structure, starting on the originating file, is recursively traversed to find the target file. The traversal functionality is provided by the CDT indexer.

To minimize construction time of a `FullIncludePath`, the recursive traversal to find the target file, in the case of the target file is part of the current project and the traversal leaves the scope of the current project, skips further traversing outside of the project. This is done under the assumption that header files, which are not part of the project, also do not again include headers that themselves are inside of the project.

The second subclass of `IncludePath` is the `FirstLastElementIncludePath`. The clue here is that it only contains the `IIndexInclude` to the first included file and the one to the last included file. Most of `ReDHead`'s algorithm do not care at all about the elements in between the first and the last one. The CDT indexer can, without any recursive traversal, deliver all the files which are transitively included through a given

originating file. It is clear that path construction here takes much less time in the case of `FirstLastElementIncludePath`.

## Lazy Loading

For all the above described classes, contained fields which can be retrieved through *getter* methods are lazy loaded. So fields are not initialized on object construction but on first access which prevents the evaluation of unnecessary elements and ensures that evaluation is only done once.

### 6.2.1 Data Stores

During the phase of optimizing the *ReDHead data structure* as described in 9.3 several store classes were created to prevent the unnecessary recreation of *data structure* elements. These were created to store:

- **ReDHeadDeclarations**
- **IncludePaths**
- **IncludeSubstitutions** as introduced in Section 4.9
- **IIndexIncludes**
- **ReDHeadSuggestions**

### 6.2.2 Clean Up

When setting up data structures which allocate a lot of memory, one needs to make sure that as much of the occupied memory is released as fast as possible again. In Java, releasing memory is of course achieved by the garbage collector, however this is only done with objects that are not referenced anymore. So an important task, after the ReDHead plugin performed static include analysis, is for the *ReDHead data structure* to be cleaned up completely. This implies that all information which is to outlast the cleanup process cannot contain parts of the *ReDHead data structure*. This means, that `ReDHeadSuggestion` instances cannot keep members such as *ReDHeadFiles*, AST nodes, or `ReDHeadDeclarationReferences`. But still they must be capable to create *ReDHeadQuickFixes* that propose solutions for the suggested problems. So what now remains in *ReDHeadSuggestions* are names of files and offset locations of nodes which are required to generate *ReDHeadQuickFixes*.

## 6.3 Optimization Algorithms

Figure 6.4 shows the class diagram that demonstrates how the `ReDHeadOptimizer` works. The `ReDHeadOptimizer` uses one or several `ReDHeadAlgorithms`, which's task is to collect

**ReDHeadSuggestions**. These suggestions each give a hint on anything that is not optimal when looking at the *include structure* of the given C/C++ code.

As a sample the class diagram contains the *find unused includes algorithm*.

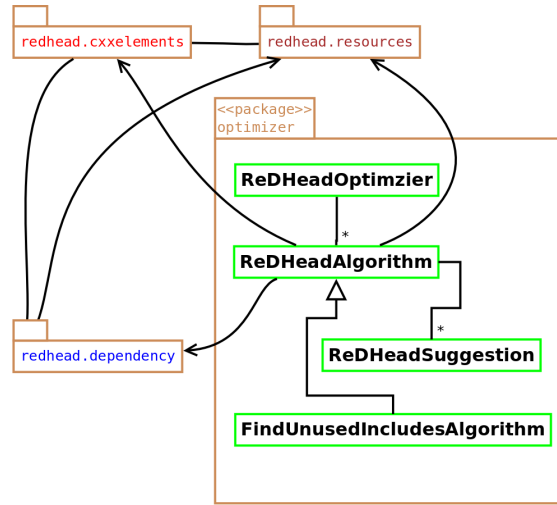


Figure 6.4: Optimization Algorithms Class Diagram

Each of the **ReDHeadAlgorithms** looks at the *ReDHead data structure* and tries to locate potential points that could be optimized. The **ReDHeadOptimizer** gathers all these points in the form of **ReDHeadSuggestions**. The UI component which makes use of the gathered suggestions is the **ReDHeadOptimizationRunner** which is described in the following section. Often, **ReDHeadSuggestions** also bring along a solution proposal in the form of a **ReDHeadQuickFix** that can fix the problem which the suggestion points out. The following list shows the algorithms which have been implemented during the ReD-Head project. There will not be any further description on them since their functionality was already described in Chapter 4 in detail.

- Finding Unused Includes
- Organize Includes
- Directly Include Referenced Declarations
- Find Unused Files
- Static Code Coverage

## 6.4 UI Integration into Eclipse

In this section the classes contained in the package *redhead.ui* of Figure 6.1 are introduced. For a user centered description of the ReDHead interface components read Section 7.2.

To add user interface components such as menu entries and markers to Eclipse, one uses Eclipse's extension points by adding XML extension tags to the *plugin.xml* of the ReDHead plugin.

ReDHead uses the following extension points:

- *org.eclipse.help.toc* to add the ReDHead user manual to the Eclipse help.
- *org.eclipse.cdt.codan.core.checkers* to use the CDT's Codan framework.
- *org.eclipse.ui.actionSets* to add menu entries to the main menu bar of Eclipse.
- *org.eclipse.ui.bindings* and *org.eclipse.ui.commands* to bind the shortcut *ctrl+shift+o* to the *auto organize includes* algorithm menu entry.
- *org.eclipse.ui.perspectiveExtensions* to bind the *ReDhead Static Analysis* main menu bar additions to the CDT perspective.
- *org.eclipse.ui.popupMenus* to add pop-up sub-menus to the *C++ Project*, *Project Explorer* and *Navigator* view of Eclipse.
- *org.eclipse.ui.editors.annotationTypes* and *org.eclipse.ui.editors.markerAnnotationSpecification* To give the *ReDheadMarkers* a custom appearance.
- *org.eclipse.cdt.ui.quickFixProcessors* to register the *ReDHeadQuickFixProcessor* which links *ReDHeadQuickFixes* to the *ReDHeadMarkers*.
- *org.eclipse.core.resources.markers* to define own ReDHead marker types.

Concrete examples on how these extension points are used can be found in Appendix B.

In the previous section, the *ReDHeadOptimizer* was introduced. This component is used by the *ReDHeadOptimizationRunner* which is invoked through a *ReDHeadAction*. The *ReDHeadAction* is the element which was registered by the extension points *org.eclipse.ui.actionSets* and *org.eclipse.ui.popupMenus* and is thus bound to the registered menu items which are also registered through these extension points. The *ReDHeadAction* is responsible to find the starting point for the *ReDHeadAlgorithms* to run. This starting point is either a *ReDHeadProject* or a *ReDHeadFile*.

Whenever the *ReDHeadOptimizationRunner* is triggered through a click on a menu entry and thus a *ReDHeadAction*, it shows the *ReDHeadSuggestions* produced by the ReDHead algorithms in Eclipse. This is either done by displaying the *ReDHeadSuggestionDialog* or by adding markers to the CDT editor. The *ReDHeadSuggestionDialog* is introduced in 7.2. If the *ReDHeadSuggestionDialog* is used or not can be configured by the user. To find out more about this read 7.2.

When the user requests quickfixes for any *ReDHeadMarker*, the *ReDHeadQuickFixProcessor* is called, which returns the *ReDHeadQuickFix* that can solve the suggestion indicated by a *ReDHeadMarker*.

Note that the *ReDHead annotations*, which are introduced in 7.2 and are produced by the *static code coverage* algorithm, are in real normal Eclipse markers. The only difference to others is that annotations have a more customized appearance. So instead of adding marker icons they just highlight text.

### 6.4.1 Codan

Above, the extension point *org.eclipse.cdt.codan.core.checkers* was mentioned. This extension point is used to register the *organize includes* algorithm as a Codan checker. This causes the algorithm to also be triggered while the user is typing.

### 6.4.2 Problem Feedback

Sometimes, ReDHead algorithms encounter problems during execution. Such problems are logged, and also presented to the user by using the logging functionality of Eclipse. To log such problems, *IStatus* instances are created which are passed to this logging functionality.

These statuses are then (1) presented to the user after the *ReDHeadOptimizationRunner* stopped running and (2) added to the *Error Log* view of Eclipse.

## 6.5 Testing

In modern software engineering, one uses automated tests which are run repeatedly when adding or changing code. During development, the tests are expanded so they also cover newly implemented functionality. In this section I describe the ReDHead testing framework and show how a sample test looks like.

A ReDHead test normally runs the following steps:

1. Setup a C++ project to test on
2. Add some testing source and header files
3. Run a method under test, which might be:
  - Run an algorithm
  - Access elements of the ReDHead data structure
  - Access ReDHead dependency classes
4. Check the results on correctness
5. Clean up the project that was tested on



Luckily, Eclipse CDT already contains refactoring tests which proceed in a very similar way as described in the list above. It provides an elegant solution which covers points 1, 2 and 5 from the list above. The ReDHead testing framework thus is an adaptation of the refactoring testing framework.

To write a new test, one has to provide the source code that will be added to the project to test on. Additionally, a test class is needed that runs the method under test and validates the results.

The source code of the testing project is provided in one file in a special file format which is then interpreted by the testing framework. Here, the file *CoverageTest2ManyFiles.rts* as an example:

```

2  //!CoverageTest2ManyFiles
  //ch.hsr.ifs.redheadtests.coveragetests.CoverageTest2ManyFiles
  //@A.cpp
  #include "Unused.h"
5  #include "Used.h"

  int main(int argc, char** argv) {
8    X x;
  }
  //@Unused.h
11 class Unused { };
  //@Used.h
  class X { };

```

Note that the example file contains code segments which represent three small C++ files. The comment *//@A.cpp* for example signals the start of file *A.cpp*. The first line of the listing is the test's name and the second one denotes the test class which will be instantiated.

The test class *CoverageTest2ManyFiles.java* looks like this:

```

2  public class CoverageTest2ManyFiles extends ReDHeadTest {
  public CoverageTest2ManyFiles(final String name, final Vector<TestSourceFile> files) {
5    super(name, files);
  }

  @Override
8  protected void runTest() throws Throwable {
    StaticCoverageAnalysisAlgorithm algorithm = new StaticCoverageAnalysisAlgorithm();
    List<ReDHeadSuggestion> suggestions = runAlgorithms(algorithm);

11    assertEquals(3, suggestions.size());
    String expectedTextInUse = "This declaration is in use through the file A.cpp.";
14    String expectedTextNotInUse = "This declaration is not in use through the file A.cpp.";
    assertSuggestion(suggestions.get(0), "A.cpp", expectedTextInUse, 39, 41);
    assertSuggestion(suggestions.get(1), "Used.h", expectedTextInUse, 0, 12);
17    assertSuggestion(suggestions.get(2), "Unused.h", expectedTextNotInUse, 0, 17);
  }
}

```

The shown test first instantiates the algorithm *StaticCoverageAnalysisAlgorithm* which it wants to test. Then it runs the algorithm and retrieves the proposed *ReDHeadSuggestion*. In the second part of the test, attributes of these suggestions are tested on their correctness.

## 6.5.1 External Include Directories

Per default, testing projects do not contain any references to external include directories. This also includes the C++ standard library. This makes sense because the testing project would become machine dependent since there might be different library version which are installed in different locations.

Nevertheless, tests are required which can validate the correct behavior of the ReDHead plugin concerning these external include directories. The refactoring test framework, on which the ReDHead testing framework is based on, however this did not provide the possibility to specify any such external include directories. After spending a lot of time to figure out how the `CProject` instance that was created in the setup phase of the test could get added such directory paths, I found the correct way to achieve what I desired. However, CDT's approach to solve this problem is quiet strange.

```
2 private void addExternalPaths() {
    String[] paths = new String[] {"/path/to/first/dir", "/path/to/second/dir"};
    TestScannerProvider.sIncludes = array;
}
5
@Override
8 protected void tearDown() throws Exception {
    TestScannerProvider.clear();
}
```

To make the testing indexer also consider the external directories I want to add, they do not have to be added to the created instance of `ICProject`, but rather to the globally accessible field `TestScannerProvider.sIncludes`. As can be seen in the listing, the `TestScannerProvider` has to be cleared after the test was run in the `tearDown` method. I think do I not have to go into detail here to show that this is not a very nice approach to solve the problem at hand.

# 7 User Manual

A detailed description of the features provided by the ReDHead plugins is given here. Whereas in Chapter 6 the focus was on the technical implementation details, it is now on the user centered features. Note that this chapter (1) is readable independently of the rest of this documentation, and (2) is also contained in the ReDHead plugin itself as user manual which can be reached under *Help -> Help Contents -> ReDHead Static Include Analysis*.

The ReDHead plugin is a static source code analysis tool for C++ projects with the focus on include structure optimization. This help provides information about how the plugin is used, about its functionality and how results are presented to the C++ developer.

The ReDHead Help contains the following sections:

- Introduction 7.1
- Usage 7.2
- ReDHead Code Analysis Algorithms 7.3
- How ReDHead Analysis Works 7.4

## 7.1 ReDHead Introduction

The ReDHead plugin is a static source code analysis tool for C++ projects with the focus on include structure optimization. The name *ReDHead* means **Refactor Dependencies of Header Files**.

Target is to support a C++ developer in the task to design the include structure in a given project. The programmer is provided with suggestions which will help him in an easy way to improve the include structure of his project.

Improving the include structure of a big C++ project is a very tedious and also delicate task which consumes a lot of time. The process of improving and cleaning up of the include structure is crucial to maintain the overview in a project and to minimize compile time. A very time consuming task is, for example, finding unused includes in source and header files. In complex systems, it is practically impossible to detect if an include directive is still in use or not.

The aim of ReDHead is to assist the user in this challenge and heavily reduce time consumption.

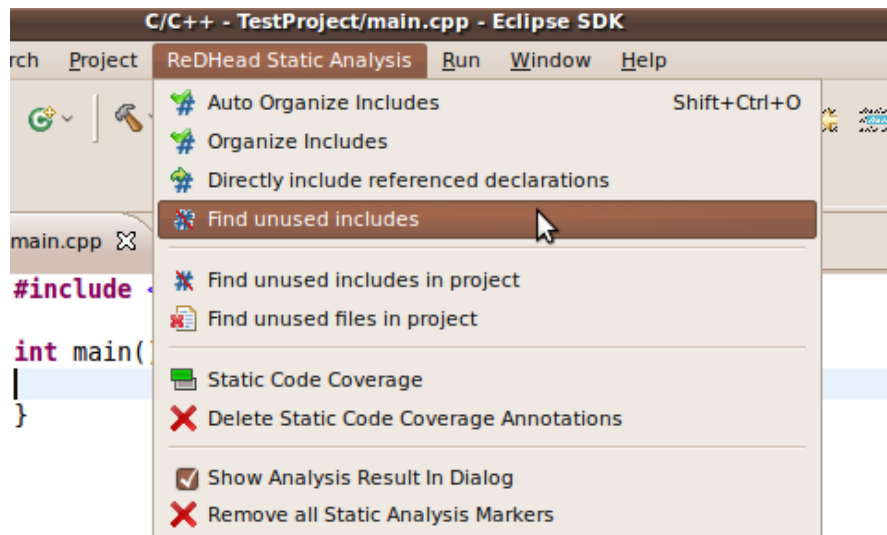
The help section *Usage* 7.2 describes how the ReDHead plugin can and should be used and how that is achieved. The section *ReDHead Code Analysis Algorithms* 7.3 gives a detailed description about the available algorithms and the suggestions they propose. The last section *How ReDHead Analysis Works* 7.4 gives a general insight into what the ReDHead Plugin does and how static include code analysis is achieved in Eclipse CDT.

## 7.2 Usage

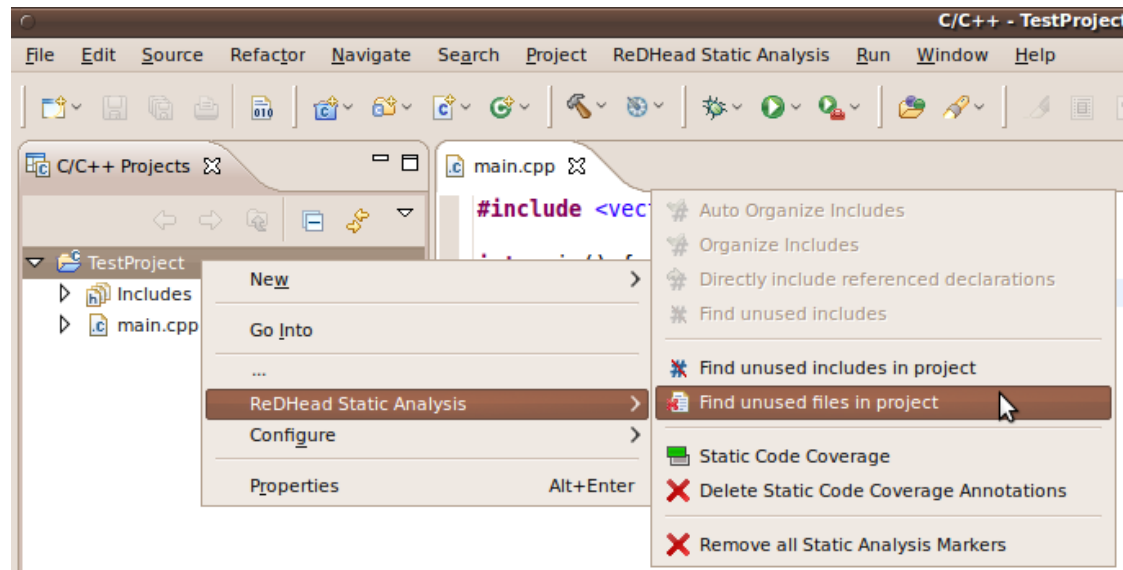
This section of the ReDHead help shows how a programmer can make use of the include structure optimization algorithms provided by the ReDHead plugin. The description includes information on how to trigger a ReDHead algorithm and on how the outcome is presented to the user.

### Running a ReDHead Algorithm

A ReDHead algorithm can be started in two different places. Either through the main menu-bar entry, which is visible when the *C++ perspective* is active.



Or through the pop-up menu accompanying a source or header file or a C++ project in either the *C++ Explorer*, the *Project Explorer* or the *Navigator* Eclipse view.



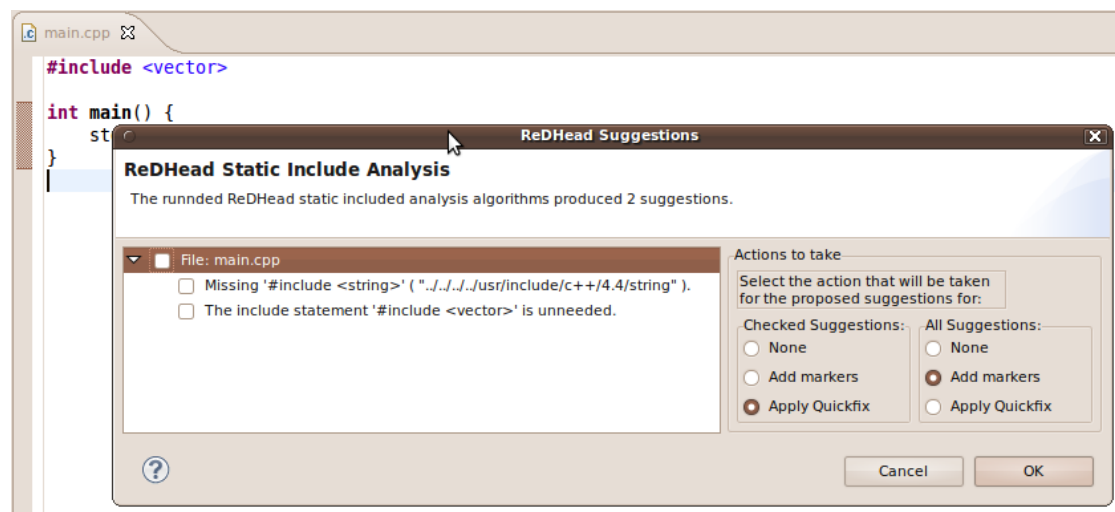
As can be seen in the sample screenshot, some of the algorithms can not be started on a project. These can only be triggered if the focus is either on a source or header file in the *C++ Explorer*, *Project Explorer* or *Navigator*, or inside of an open C++ editor.

## Visualization of ReDHead Suggestions

As already mentioned, ReDHead algorithms produce suggestions that provide information on the given code. These suggestions will be visualized either in the *ReDHead suggestion dialog* or they will be shown in the C++ editor in the form of markers or annotations.

### ReDHead Suggestion Dialog

The *ReDHead suggestion dialog* will show up with almost all algorithms, but only if the *Show Analysis Result in Dialog* checkbox menu entry is enabled. Note that this checkbox menu entry is only available through the main menu-bar entry (first screenshot above).

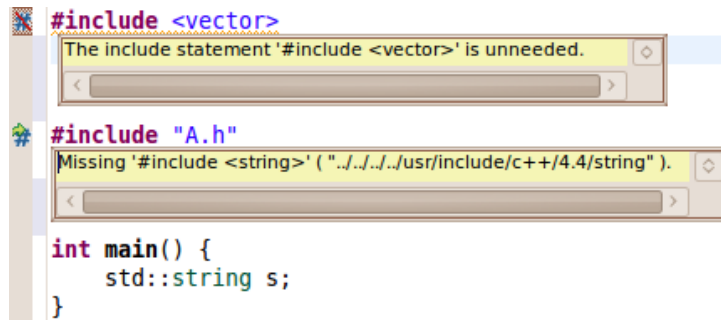


The *ReDHead suggestion dialog* will not show up when running *Static Code Coverage* or *Auto Organize Includes*.

In the suggestion dialog one can choose how to proceed. There are options to apply to all the suggestions or only to these that are checked in the suggestion tree on the left side.

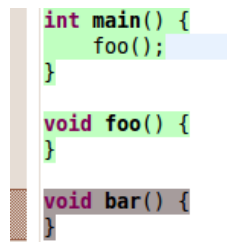
## ReDHead Suggestion Markers

*ReDHead suggestion markers* show up in the editor as normal markers with one of ReDHead's icons.



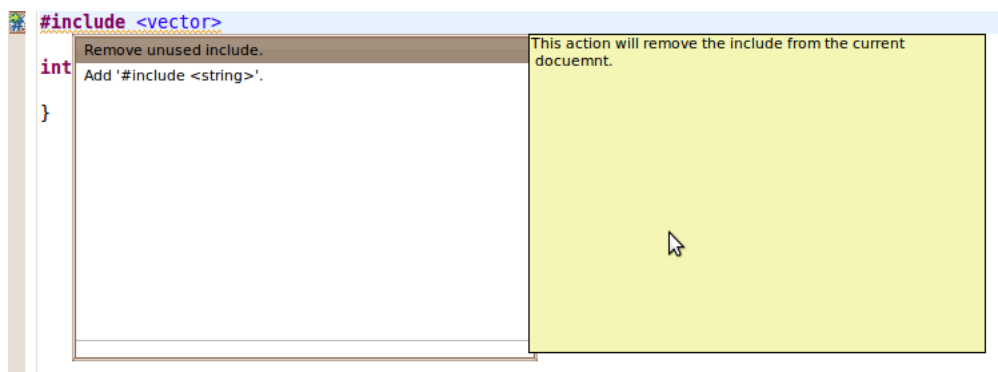
## ReDHead Annotations

Annotations are added to the editor when running *Static Code Coverage* on a C++ project.



## Applying Proposed Suggestion Solutions

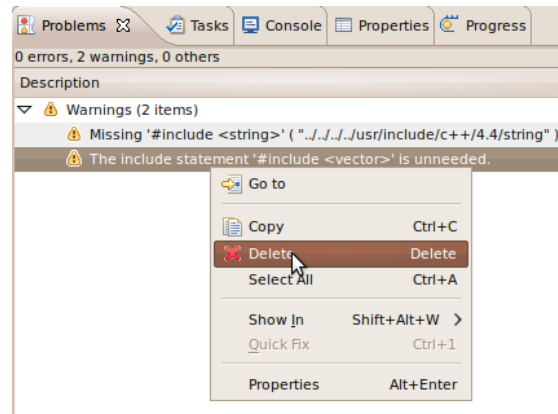
Almost all of the ReDHead suggestions bring along solution proposals (called QuickFix) which can be applied on a given document to resolve the issue. These solution proposals will be shown by putting the cursor on a line containing a marker and pressing *ctrl+1*.



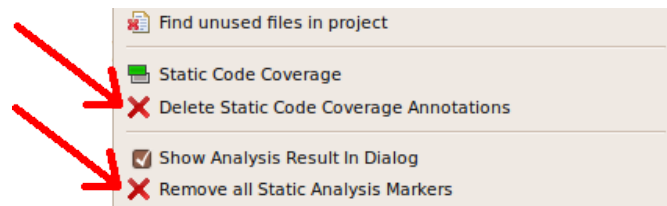
After applying a quickfix on a document, the document will be adapted according to the current suggestion and the marker will be removed.

## Deleting ReDHead Markers

There are two ways to get rid of ReDHead editor markers (without applying them). To delete one or several specific markers one can use the the *Problems* view of Eclipse.



To delete all ReDHead markers, choose *Remove all Static Analysis Markers* from the ReDHead menu. In addition it is also possible to remove all static code coverage annotations from a project.



## Codan Integration

In the subsection *CDT Codan Integration 7.2.1* you can find a description on what the Codan framework is, on the integration of ReDHead into the Codan framework and how the integration can be configured.

### 7.2.1 CDT Codan Integration

CDT Codan [cod09], which stands for **code analysis**, is the CDT framework for static code analysis. It provides other Eclipse plugins such as ReDHead with the possibility to analyze C++ code while the user is typing. The Codan framework [cod09] allows to analyze C++ code while the user is typing. ReDHead brings along an *Organize Includes* implementation for the Codan framework. So as long as the ReDHead *Organize Includes*



Codan implementation is active, markers which propose to add and remove include directives will appear automatically when C++ programmer is typing.

### Configuring Codan

The Codan Organize Includes can be enabled and disabled under *Window -> Preferences -> C++ -> Code Analysis -> ReDHead Organize Includes*.

## 7.3 ReDHead Code Analysis Algorithms

This help sections describes each ReDHead algorithm provided in a subsection. To find out how to use ReDHead algorithms read section *Usage 7.2*.

This ReDHead Help section contains the following subsections:

- Find Unused Includes 7.3.1
- Organize Includes 7.3.2
- Auto Organize Includes 7.3.3
- Directly Include Referenced Declarations 7.3.4
- Find Unused Files 7.3.5
- Static Code Coverage 7.3.6

### 7.3.1 Finding Unused Includes

The *Find Unused Includes* algorithm analyzes either a whole C++ project or a single source file to find all *#include*-statements which are not necessarily required. Consider the following example:

```
3 #include <vector>
4 #include <string>
5
6 int main() {
7     std::string s;
8 }
```

Clearly the first include directive is not needed here. When running the *Find Unused Includes* algorithm on the sample code it will suggest the removal of the first include as shown in the following screenshot.

```
main.cpp
#include <vector>
#include <string>

int main() {
    std::string s;
}
```

### 7.3.2 Organize Includes

The *Organize Includes* algorithm analyzes a single file and tries to make all necessary suggestion so that the file will then compile without giving any warning of the type "XY was not declared in this scope.". In addition it will suggest the removal of all unnecessary includes. Consider the following example:

```
#include <vector>
3 int main() {
    std::string s;
}
```

When running the *Organize Includes* algorithm on the sample code it will suggest the removal of the currently existing include and the insertion of an include directive to <string>).

```
main.cpp
Multiple markers at this line
- Missing '#include <string>' ( \".././././usr/include/c++/4.4/string\" ),
- The include statement '#include <vector>' is unneeded.
int main() {
    std::string s;
}
```

### 7.3.3 Auto Organize Includes

The *Auto Organize Imports* proposes the same suggestion as the *Organize Imports* 7.3.2 algorithm. But it does not add markers or shows the *RedHead Suggestion Dialog*. Instead of that, it automatically applies all the solution proposals to the generated suggestions. To each automatically applied change an info marker is added so the user can track the applied changes and also revert them if he likes.

To use this feature as fast as possible, it is also bound to the shortcut *ctrl+shift+o*.

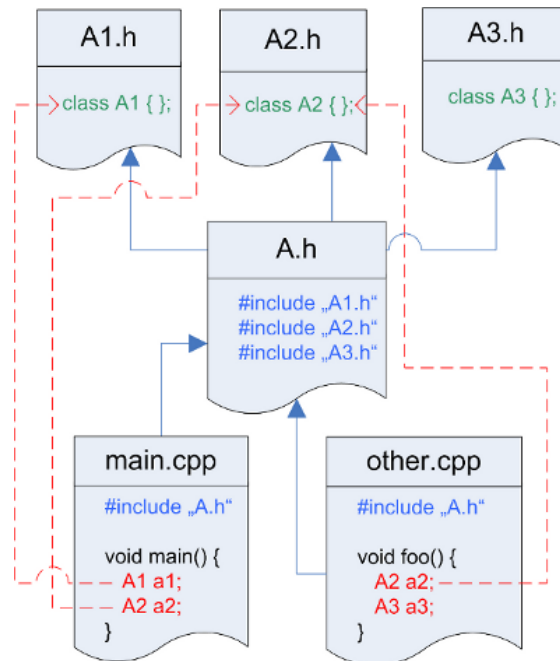
### 7.3.4 Directly Include Referenced Declarations

Directly including referenced files will suggest to add an include directive to each header file that defines a type that is used in a file under consideration.

This will alter the include structure of a project in favor of the reduction of include dependencies, coupling and compile time.

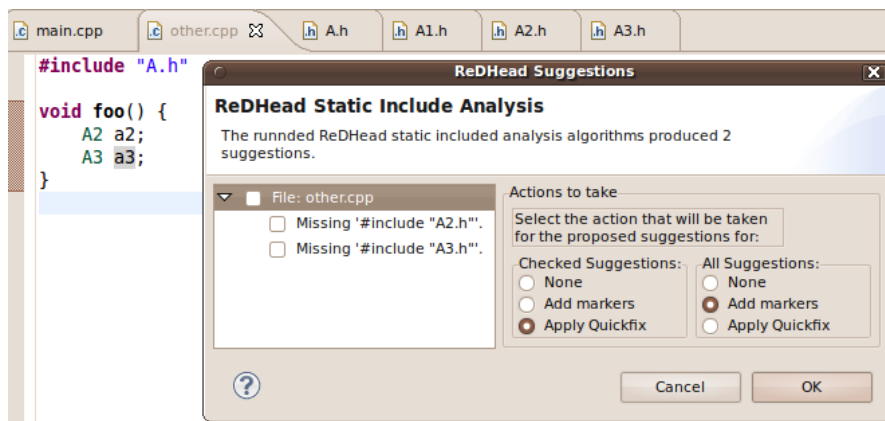
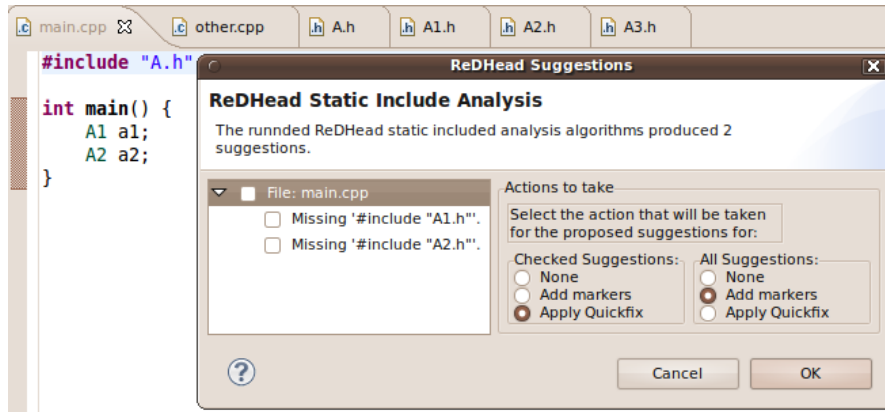
A detailed description why directly including referenced files is useful can be found in the book *Large-Scale C++ Software Design* [Lak96] by John Lakos on page 113.

As an example, consider the following C++ project graph:



Note that the blue relations (arrows) between the files (boxes) represent include directives in the originating file which includes the target file. Red dashed arrows represent dependencies to declared types in header files.

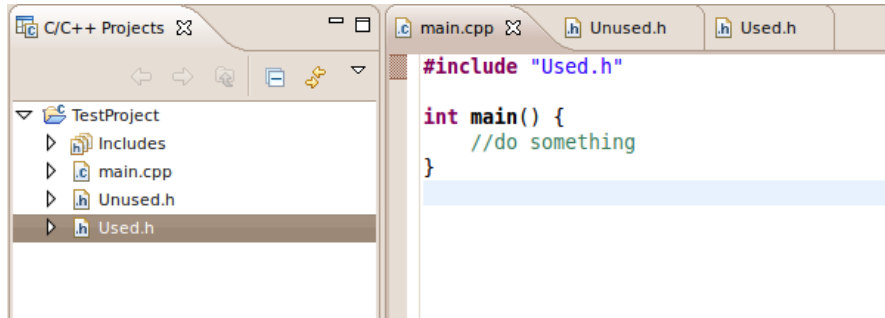
The result, when running the algorithm on the file *main.cpp* and then on the file *other.cpp* is shown in the two following figures.



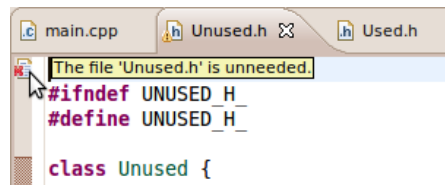
### 7.3.5 Finding Unused Files

The *Find Unused Files* algorithm always analyzes a whole project for files that are not used at all. Files that are not in use can either directly be removed with the help of the *ReDHead Suggestion Dialog* or marked as unused in the corresponding files.

As an example, consider the following C++ project:



When running the *Find Unused Files* algorithm, the file *Unused.h* will be proposed to be removed as can be seen in the screenshot.



### 7.3.6 Static Code Coverage

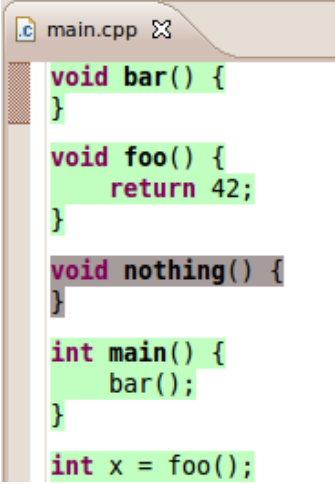
The Static Code Coverage algorithm analyzes a whole C++ project and helps the programmer gain insight into which parts of his code are effectively used and which are not. Compared with other *code coverage* tools, ReDHead does not need to run the code itself to find unused parts.

The *Static Code Coverage* algorithm can run in two different modes. In the *main-mode*, where a *main()* function is present in the project, only the main function and global variables are used as starting point. In the *library-mode*, where a *main()* function is not available, all declarations in source files are used as starting point.

The *Static Code Coverage* algorithm can decide for itself in which mode it will run, depending on the presence of a *main()* function. Consider the following example code:

```
1 void bar() { }
4 int foo() {
  return 42;
}
7 void nothing() { }
10 int main() {
  bar();
}
13 int x = foo();
```

Since a *main* function is available here, the algorithm will run in the *main-mode*. This means the starting point consists of the *main* function and the definition of the *int x*. The functions *foo* and *bar* are reachable through the starting point, whereas *nothing* is not. So the result, as can be seen below, marks all the declarations except the *nothing* function as used.

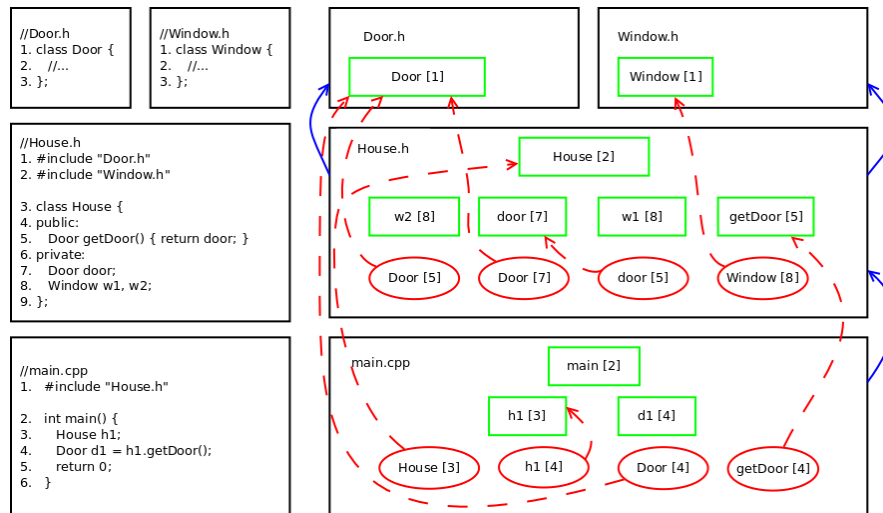


```
.c main.cpp ⌵
void bar() {
}
void foo() {
  return 42;
}
void nothing() {
}
int main() {
  bar();
}
int x = foo();
```

## 7.4 How ReDHead Include Analysis Works

All the ReDHead static include analysis algorithms are based on the *ReDHead data structure* which is constructed with code information obtained from the CDT AST (Abstract Syntax Tree) and the CDT Indexer. The *ReDHead data structure* contains several types of elements which are introduced in the following subsections.

Some of the element types' descriptions will make use of the graph in the following image on the right hand side. On the left hand side, the C++ source code that produced the graph is given. Element labels in the graph that end with square brackets which contain a number ( e.g. [7] ) indicate that the element belongs to this source code line.



### File

A *file* represents the physical file that is present on the file system. In the graph image above it is represented as a box. The file elements provide access to the *include dependency* and *declaration reference* elements.

### Include Dependency

An *include dependency* represents an include directive that is a link between an including file and an included file. It is represented in the graph as a solid blue arrow. The element grants access to both the including and the included file and to the *include path* element which is described below.

### Declaration

A *declaration* element represents a normal C++ declaration (class forward declaration, variable declaration, typedef, class declaration, etc.). Note that a C++ definition is always also a declaration. In the graph, a *declaration* is represented as a green box.

## Declaration Reference

A *declaration reference* represents any C++ element that references a *declaration*. This can for example be the name of a *class*, a *variable name* or a *function name*. A *declaration reference* grants access to the *declaration reference dependency* element. In certain cases, a *declaration reference* gives access to several *declaration reference dependencies*. In the graph, a *declaration reference* is represented as a red circle.

## Declaration Reference Dependency

A *declaration reference dependency* is the link between a *declaration reference* and a *declaration*. Through the *declaration reference dependency* ReDHead algorithms can gain access to *include path* elements. Sometimes there are several such elements returned by a *declaration reference dependency*. The *declaration reference dependency* element is represented in the graph as a red, dashed arrow.

## Include Paths

An *include path* is a list of include directives which lead from a *declaration reference* to a *declaration*. *Include paths* are not directly visible in the graph above. But they can be seen as a combination of several blue arrows.

Note that the term *include path* does not refer to what is known as include path that is passed to a C++ compiler and is actually the name of a directory that contains C++ header files.

## Example Algorithm Run

A ReDHead algorithm starts in a given point, which is either a C++ file or project and scans the *ReDHead data structure* for certain parts of information it is looking for. It could for example look for *declaration reference dependencies* which do not yield any *include paths* (which means that we are missing an include directive).

In the end, the algorithm assembles a collection of suggestions which are then presented to the user. The necessary information on where an Eclipse editor marker should be placed or where a solution should be inserted can be obtained from the collected data structure elements.



## 8 Market Analysis

This chapter gives an overview of the market of all *C++ static code analysis* tools. Note that only a part of these tools mentioned in the following table compete with the Eclipse ReDHead plugin. Most of them do not provide *static include analysis* features but rather other static analysis features. The list ranges from academical to commercial, open source, etc.

	<i>Static Code Analysis (non-include)</i>	<i>Resolve Dependencies</i>	<i>Visualize Dependencies</i>	<i>Find Unneeded Includes</i>	<i>Organize Includes</i>	<i>Code Coverage</i>	<i>Only C</i>	<i>Platform</i>	<i>IDE Integration</i>	<i>Out-Dated</i>	<i>Commercial</i>
Cppcheck	✓	×	×	×	×	×	×	W & L	✓	×	×
Dehydra	×	×	×	×	×	×	×	L	×	×	×
Pork	×	×	×	×	×	×	×	L	×	×	×
Treehydra	×	×	×	×	×	×	×	L	×	×	×
Mygcc	✓	×	×	×	×	×	×	L	×	×	×
CodeSonar	✓	×	×	×	×	×	×	W & L	×	×	✓
Coverity Static Analysis	✓	✓	×	×	×	✓	×	W & L	✓	×	✓
Klocwork Insight	✓	✓	×	×	×	×	×	W & L	✓	×	✓
PolySpace	✓	×	×	×	×	×	×	W & L	✓	×	✓
PC-Lint / FlexeLint	✓	✓	×	✓	×	×	×	W & L	✓	×	✓
C++lint	✓	✓	✓	×	×	×	×	W & L	✓	×	✓
QA · C++	✓	×	×	×	×	×	×	W & L	✓	×	✓
Safer C	✓	×	×	×	×	×	×	W & L	×	✓	×
Goanna	✓	×	×	×	×	×	×	W & L	✓	×	✓
Splint	✓	×	×	×	×	×	×	L	✓	×	×
AbsInt	✓	✓	✓	×	×	×	×	W & L	×	×	✓
Uno	✓	×	×	×	×	×	×	?	×	✓	×
LDRA Testbed	✓	✓	×	×	×	✓	×	W & L	×	×	✓
Parasoft C/C++ Quality Solution	✓	×	×	×	×	×	×	W & L	✓	×	✓

	<i>Static Code Analysis (non-include)</i>	<i>Resolve Dependencies</i>	<i>Visualize Dependencies</i>	<i>Find Unneeded Includes</i>	<i>Organize Includes</i>	<i>Code Coverage</i>	<i>Only C</i>	<i>Platform</i>	<i>IDE Integration</i>	<i>Out-Dated</i>	<i>Commercial</i>
PVS-Studio	✓	×	×	×	×	×	×	W	✓	×	✓
Cantata++	✓	✓	×	×	×	✓	×	W & L	✓	×	✓
Logiscope	✓	✓	×	×	×	✓	×	W & L	×	×	✓
Visual Assist X	✓	✓	×	×	×	×	×	W	✓	×	✓
IncludeManager	✓	✓	only includes	×	×	×	×	W	✓	×	×
CScout Refactoring Browser	✓	✓	×	×	×	×	✓	W & L	×	✓	×
Eclipse CDT	✓	✓	includes as tree	×	×	×	✓	W & L	✓	×	×

Note that in the content of the column *Platform*, *W* refers to Windows and *L* refers to Linux and Unix like platforms.

Also note that there are projects lists which can be used to perform static analysis, but do only provide directly usable features. This means that, before being able to use, the user has to provide some kind of script, which then is used to perform customized static analysis. Because of that, such features do not have a check mark in the the column *Static Code Analysis (non-include)*.

The list given above can also be found in the ReDHead wiki <http://redhead.ifs.hsr.ch/wiki/SimilarTools>. The list given there also contains links to the tool's home-pages.

For a list of tools which directly compete with the features of ReDHead, I refer to the Chapter 5 in [Fel09].

## 8.1 Similar CDT Features

Also CDT itself contains some limited *static code analysis* itself. These are, on the one side, the build output which is parsed and added to the CDT editor in form of markers. Additionally to this, the newest version of CDT also brings along some rather simple Codan checkers. These are listed in the following list:

- StatementHasNoEffectChecker
- CatchByReferenceChecker
- NamingConventionFunctionChecker

- NonVirtualDestructor
- SuggestedParenthesisChecker
- AssignmentInConditionChecker

## 9 Challenges

During the development of the `ReDHeadPlugin`, some tasks were especially hard to complete. The most noteworthy of these tasks are described in this chapter.

### 9.1 Adapting the CDT Index

Normally, the CDT indexer is aware of all the C++ code which is somehow reachable through any of the source files which are contained in a project. This means that all the headers which are included into any source file are represented in the CDT index. However, headers which are not included in any way are not represented in the indexer. This is even true for header files that are contained in the current CDT project itself.

For the `ReDHead` plugin, this brings forth the problem that when using a type in any file, even if a type is declared inside of a header file in a given project, there will be no suggestion generated to include this header file because the relation from the usage of the type to its declaration is not possible due to the indexer's ignorance of the header file.

For header files present in the current CDT project, this issue can be solved easily because the indexer can be configured to also index all header files. However, this does not remove the whole problem since the `ReDHead` plugin is also supposed to be able to propose includes to external header files. This can be header files of the C++ standard library or any other header which is contained in one of the include directories that is linked to the CDT project.

Also, the remaining problem can be solved thanks to the indexers capability of indexing certain specified file up-front. Files to parse up-front can be configured as a property of a project's indexer. So the solution to make all types known to the indexer that are not located inside of the project itself is, to add all the external header files to this list. The list of course grows very long since only the C++ standard library (version 4.4) already contains about 4'600 header files. Often, there are also several other libraries included which lets the list grow again. Parsing all these files takes a lot of time of course. The good thing here is, that the adaptation of the index is persistent so that it is only necessary one for a project.

So the `ReDHead` plugin adapts the index of a project by letting the indexer parse all the files, which are contained in any include directory or sub-directory that are known to a CDT project. An instance of `ICProject` is capable of returning all these directories as instances of `IIncludeReferences`. By recursively traversing each of these directories, a

list of all contained files is created and provided to the projects index as parse up-front list. Then the indexer needs to be re-indexed once.

This adaptation of the index does, up to now, happen implicitly when any ReDHead algorithm is run. Re-indexing the whole indexer each time an algorithm is run is of course suboptimal. Because of that, before the index is adapted, it is checked if the adaptation is possible at all.

## 9.2 Synchronism of CDT Index and AST

The CDT Codan framework triggers implemented checkers while the user is typing. The AST which is passed to the checker to be examined is created with the current content of the open editor. This means that the contained node is up-to-date, even if the file that is edited has not yet been saved. The problem which here arises, is that changes on files which have not been saved, are also not yet represented in the CDT indexer. This results in the fact, that whenever a codan checker makes use of the CDT indexer, the information that it contains might be outdated. The state of the AST and the indexer thus is not necessarily the same during the Codan checker's work. This can, for example, result in a list of includes returned by the indexer which is bigger or smaller than the list which is returned when accessing the same list through the AST.

The result of this effect is, that static analysis which is done by also using the index instead of only the AST, can yield results which are totally wrong. To approach this issue, I reported the problem as a bug of CDT. Sadly, the bug ticket has already been closed as *WONTFIX* with hint that one must consider the possibility that node position returned by the index might not be up-to-date and thus the node positions of the AST should be used because they are up to date. The possibility of added or removed nodes has somehow skipped the CDT developers' attention.

## 9.3 Algorithm Performance

When I first implemented the *ReDHead data structure* and some of the ReDHead algorithms, my main focus was to develop well designed code that produces correct results on the various, but small code samples which form the ReDHead test suite. The tests represent the various features and possibilities well. The only thing that these small artificial code fragments can not do is representing a real world C++ project which comprises a large amount of code.

To test the result of the ReDHead algorithms, I used a real C++ project which is under development at the *Institute for Software* in Rapperswil. The first few runs were a bit disillusioning because the algorithms, even when initiated only on a single file, performed very badly both in means of memory and execution time. A run the *find unused includes* algorithm on one code file took more than five minutes and used more than one gigabyte of memory. When the algorithm was triggered on the whole project, after running for

more than half an hour, the algorithm was interrupted by an `OutOfMemoryError` because the two gigabyte memory which were available for the application were exhausted.

Clearly, the RedHead plugin was in dire need of some performance optimization. During the following work to make RedHead perform better, memory usage and execution time were regularly measured to judge the achieved advancement. Before presenting these measured numbers, following some numbers on the previously mentioned C++ project. Measurements were performed either on one specific single file or on the whole project. The file contains 200 line of C++ code (while skipping comments and empty lines). The whole project consists of about 190 source and header files, totaling to about 30'000 lines of C++ code.

The following tree charts give an overview on the performed measurements. The first two Figures 9.1 and 9.2 show the execution, first of the single file and then on the whole project. The third Figure 9.3 shows the memory usage of both the file and project runs. Each dot on a chart line represents a measurement. Sometimes, measurements were only taken for either file or project. In this case some dots are missing. These remaining spaces on the line are necessary, so that the chronological order between file and project measurement is still evident.

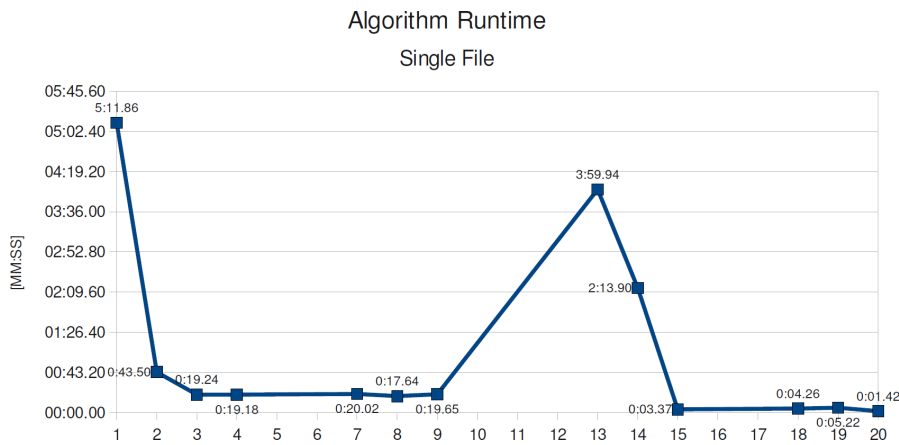


Figure 9.1: Execution Time Single File

The peak which arises from measurement 13 and 14 was the adaptation toward *first-last-element include parts* based on `IIndexs` which was introduced in 6.2. This adaptation, including adequate caching mechanisms, was finished at measurement 15. When comparing measurement 9 and 15, it becomes obvious that the adaptation was prosperous.

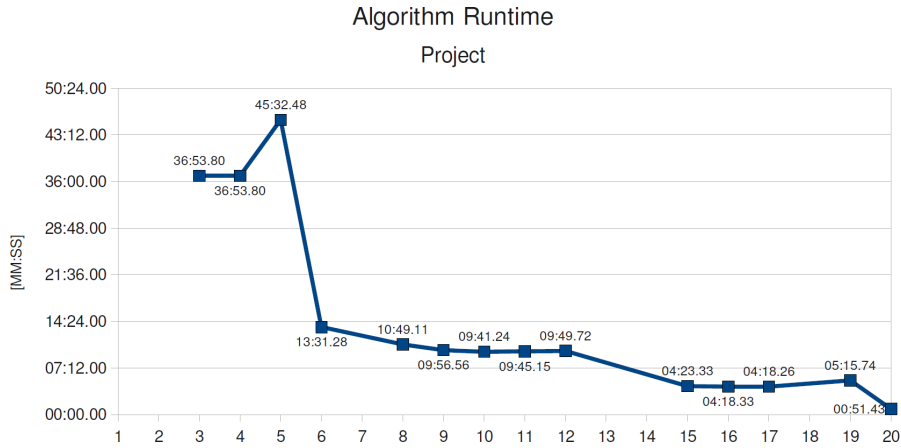


Figure 9.2: Execution Time Project

The measurements on the whole project do only start at index 3. The reason for this is that before, the running of the *find unused includes* algorithm did not terminate successfully due to the already mentioned `OutOfMemoryError`.

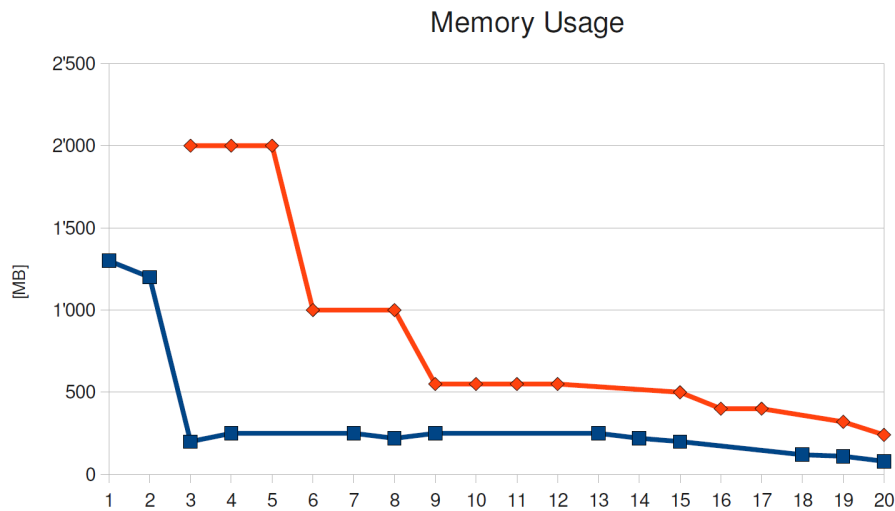


Figure 9.3: Memory Usage

All the memory measurement shown in Figure 9.3 do also cover the memory consumption of the testing Eclipse instance as well. The consumption of the Eclipse application is roughly 50 to 80 megabytes.

As can be seen in the charts, the performance gain is fortunately very big. To makes this possible, several changes have been introduced. Firstly, a lot of caching was added so that none of the `ReDHead` object which can be seen in Figure 5.1 are instantiated more than necessary. To show how important this is, the following list contains some

information on this.

- Instantiations of `ReDHeadDeclarationReferences` was reduced to 37%.
- Instantiations of `IncludePaths` was reduced to less than 1%.
- Instantiations of `ReDHeadDeclarations` was reduced to less than 30%.

Further optimal improvements have been achieved by optimizing the iterations during the creation and traversal of the *ReDhead data structure*. To give an example, `ReDHead` now often uses iterators of Java collection which allow to remove the current element instead of collecting elements to remove in order to prevent a `ConcurrentModificationException`.

Also, a lot of bugs have been fixed among which the most severe one caused the memory consumption to raise exponentially.

Also noteworthy here is that all the measurements have been taken directly after the the testing Eclipse instance was started. Due to caching in the CDT indexer, if one re-runs an algorithm after a first launch, the execution time is lowered again by 35 to 50 percent.

## 9.4 Preprocessor Problems

While creating `ReDHeadDeclarationReferences` of a given file, `ReDHead` needs to locate all `IASTName` instances which can be found in the AST of the file. This process proved to be more complex than I initially thought of, because it is quite complicated to get all these `IASTNames`. Most names can be found easily enough by using a visitor to traverse the AST.

When it comes to preprocessor nodes, which also are represented inside of the AST the visiting behavior of CDT becomes a bit strange because preprocessor-related names, for example macro function calls or preprocessor symbols, are not visited. Some instances, like for example `IASTPreprocessorFunctionStyleMacroDefinition` or `IASTPreprocessorIfdefStatement`, contain a *getter* method for its `IASTName`, the `accept()` method which would be responsible to initiate a visitor to visit the `visitName()` method with that name is not even overloaded.

It gets even more complicated when it comes to instances of `IASTPreprocessorIfStatement` and `IASTPreprocessorElifStatement`. These instances are not able to return any contained `IASTNames` directly. The only way I could figure out to get all required `IASTNames` out of these instances is shown in the following code listing.



```

private static void addAllChildNamesHack(IASTPreprocessorStatement node) {
2   IASTNodeSelector selector = node.getTranslationUnit().getNodeSelector(null);
   int curOffset = node.getFileLocation().getNodeOffset();
   int endOffset = curOffset + node.getFileLocation().getNodeLength();
5   IASTName name = null;
   do {
8     name = selector.findFirstContainedName(curOffset, endOffset - curOffset);
     if (name != null) {
       foundName(name);
       if (name.getParent() instanceof IASTPreprocessorMacroExpansion) {
11        IASTPreprocessorMacroExpansion expansion = (←
           IASTPreprocessorMacroExpansion) name.getParent();
           for (IASTName nestedName : expansion.getNestedMacroReferences()) {
14             foundName(nestedName);
           }
       }
       curOffset = name.getFileLocation().getNodeOffset() + name.getFileLocation←
           ().getNodeLength();
17   } while (name != null);
}

```

I think it is obvious that this is a very strange way to offset-crawl a given `IASTPreprocessorStatement` to reach the contained `IASTNames`. Noteworthy here is, that the `IASTNames` returned by `getNestedMacroReferences()` in listing line 11 are not found by the `selector` object and thus need to be considered as shown in the listing.

## Vast Amount of CDT and Eclipse Code

To find certain information, it could not be avoided that I had to browse the vast amount of CDT's and Eclipse's implementation. In general this code is well designed and also easy to read and understand. The crux however lies in its extent. There is so much code that one can spend hours trying to find a certain thing. Often one then discovers that, had one looked at "the other" code section first, it would only have taken some minutes to find. Such incidences can probably not be avoided, but can still be very unnerving.

## Resolving Declaration References

Resolving a *declaration reference* to its *declaration* means in the CDT code to resolve one `IASTName` to another `IASTName`. To do so in the normal cases is very simple. I however soon discovered that there are many different ways to do so and that for many different scenarios another approach is required to resolve the dependency. After inspecting a lot of CDT code, I found that the method `runOnAST` in the class `OpenDeclarationsJob` was the right spot to begin searching. What I discovered there, was that the steps performed to find the target `IASTName` is a huge mess of different ways with scores of loops, if-then-else statements, casts and much more to get only one other `IASTNode`. After spending a lot of time to find out all of this, I was able to do the resolution I required to fulfill my test's requirement. It was not necessary to re-implement all of these ways that I found. This can be positive and also negative. Either some of the CDT code is old and was

not yet removed because no one was sure if it really can be removed, or I will find other scenarios in the future which will force me to re-implement also all or some of these remaining ways.

# 10 Outlook

In the future, the development of the ReDHead Eclipse plugin will go on. There is a good chance that the plugin might get a commercial product distributed by a partner company.

However, the ReDHead plugin is far from finished. The following sections in this chapter list tasks that have not yet been achieved in the ReDHead master thesis. The tasks are grouped into features that can be improved and other ones that are still completely missing.

## 10.1 Improvements

### Improve accuracy of static code coverage

The *static code coverage* algorithm is not precise enough when it comes to the instantiation of types. Regard the following example code listing.

```
2 struct X {  
    X();  
    X(int i);  
};  
5  
X x(42);  
X *pX = new X(7*6);
```

Running the *static code coverage* algorithm on the sample given above, it will mark all the constructors and also their implementation as used. Of course this is not completely correct here.

The reason for this is that the the CDT AST does not contain a `IASTName` node which refers to the constructor calls. One could argue that such a `IASTName` is obviously not required because the name of the second constructor is not used in the term `x(42)`. Nevertheless, without this `IASTName`, there is no way to let the CDT indexer resolve the relation to the second constructor which clearly exists in the mentioned term. The `IASTName` instance which is created in the initialization `new X(7*6)` however, refers to the correct constructor.

The *static code coverage* algorithm now just treats all the constructors, and also the destructor, if one exists, as used because otherwise all of these would be marked as unused which in almost every case leads to wrong results. This approach is for certain scenarios not accurate enough. The listing given above without the last code line is such a case.

I see three possible solutions for this problem. The first is to file a CDT bug and hope that they will introduce the additional mentioned `IASTName` to the CDT AST. The second solution is to extend the *ReDHead data structure* that it manually finds such problems when constructing `DeclarationReferenceDependencies`. The last solution would be to add special checking functions to the *static code coverage* algorithm which are aware of this problem and navigate manually to the required constructor to mark it as used.

Both latter solutions of course are suboptimal and would slow the ReDhead algorithms' execution.

### **Make `ReDHeadSuggestionStore` persistent**

Suggestions that are created by a ReDhead algorithm get stored in the `ReDHeadSuggestionStore`. At the current time, these stored suggestions get lost when Eclipse is shut down. A useful enhancement thus would be to make these suggestions persistent so that possible suggested solutions could still be applied after Eclipse was restarted.

### **Check for Includes to add when proposing to remove one**

The problem at hand here was already explained in the the description of the algorithm *directly include referenced includes* 4.3. Suppose a code file *A* relies upon *B* to include *C*. In the case that the *remove unused includes* algorithm proposes the include to *C* to be removed because it is not required anymore there, *A* will not have an *include path* to *C* anymore. So an improvement for the *find unused includes* algorithm would be to also propose to add an include to *C* in file *A* alongside of the proposal to remove the include to *C* from *B*.

### **Complete *directly include referenced declarations***

The *directly include referenced declarations* algorithms proposes to add include directives to all the files which contain referenced declarations. When applying all these suggestions, the situation might arise, that some of the already existing includes in the file under consideration are not needed anymore. So, as already described in the pseudo code in Section 4.3, the *directly include referenced declarations* algorithms should propose includes, which become useless after applying all suggestion solutions, as removable.

### **Make *find unused includes* compatible to *directly include referenced declarations***

The problem here is demonstrated on the graphs in Figure 10.1

When applying the *directly include referenced declarations* on the graph in the left-hand side of the figure, one is suggested to add an include to the file labeled *C*. After applying the suggestions solution one gets the graph in the right-hand side of the figure. When now running the *find unused includes* algorithm on the graph on the right-hand side it proposes to remove the just added include to the file labeled *C* again.

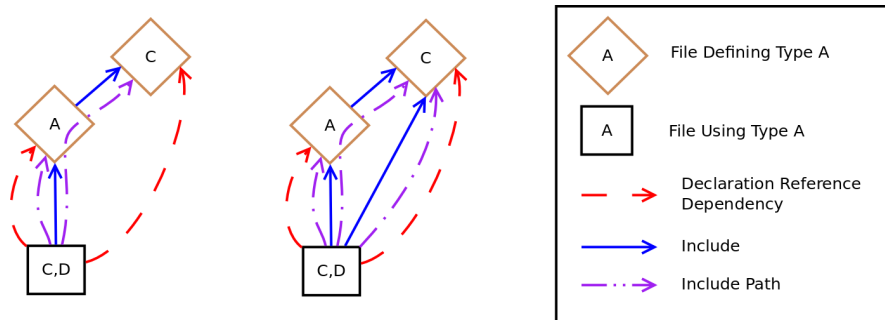


Figure 10.1: Compatibility Problem

Both the algorithms work perfectly right. Thus both the suggestions to once remove and once add the include are no false positives when only considering the task of each of the applied algorithms. Nevertheless, *find unused includes* should be configurable to not propose includes which directly include a referenced type to be removed.

### **Organize includes should also propose forward declarations**

The *organize includes* algorithm suggests to add additionally required include directives. Whenever it is enough, it suggests to include a file that only contains a forward declaration of a referenced type. Instead of suggesting this, it would be even better to just propose to add the forward declaration itself instead of including one. It is obvious that this would not reduce the coupling of the file current file.

### **Organize includes should also propose the removal of forward declarations**

The last section suggested to improve the *organize includes* algorithm by the capability to propose the insertion of forward declarations of types. Furthermore, it would also be useful if it could propose forward declarations to be removed in the case when it is not used anymore. Once it does, one should consider to change the algorithms name, because it does not exactly organize only includes anymore.

### **Not adapt the CDT indexer automatically**

During the testing phases of the ReDHead plugin, it became clear that adapting the CDT indexer automatically as described in 9.1 when a ReDHead algorithm is run, is not the optimal solution. The machine on which I developed ReDHead has a solid state hard drive (SSD) which is about tree time faster than a normal hard drive. Due to this, I underestimated the time which is required to (1) rebuild the index and (2) check if a index adaptation is even necessary. When testing on an older notebook with projects that link to include directories of several external libraries, it came clear that adapting the index takes far more than a minute. Also testing if an adaptation is necessary alone took almost up to a minute.

So a better way of adapting the index would be to trigger the process by the user manually with the help of a menu entry.

### **Prevent index adaptation for each project**

Often, a CDT Eclipse workspace contains many projects which themselves depend on each other. Each of these projects often depend on the same external libraries. During the adaptation of a projects indexer as described in 9.1, each project belonging to one project indexes all these external library header, independently if they have already been indexed by the indexer of other projects. When indexing the whole standard C++ and boost library's headers more than ten times, a lot of unnecessary time will be wasted when waiting for the re-indexing task to complete.

A good thing here would be to have a common indexer which can be shared among projects, which contains information about all these external headers which are contained in every newly created C++ project. The CDT `IIndex` instances already work in composite way. This means that one can, by using the static methods of `CCorePlugin`, request one `IIndex` instance which actually encloses the indexes for several projects. If such a common indexer, as I suggested, exists, one could easily achieve that when requesting the index for one project, a composite indexer that combines both the indexer of the project and the common one.

### **Make index adaptation configurable**

Instead of letting the indexer index every include directory known to a project, the directories, of which all files are added to the indexer, should be configurable by the user. This would help shorten the adaptation time since the header file, for example of the C++ standard library, are often included several time in different version, whereof of course only the newest is used.

### **Markers should be clickable**

As described several times before, the user needs to press `ctrl+1` to show available quickfixes for a ReDHead marker. It is not possible to just click with the left mouse button to see the available quickfixes. Since for makers in the Java editor this is possible, it should also be available for ReDHead markers.

### **#pragma, #define and #undef directives**

`#pragma`, `#define` and `#undef` statements in file *A* can in certain situations have an impact on all include files which are included after such a statement. The bad thing here is, this can not only be headers which are included after the statement under consideration in file *A* itself, but also file *B* which contain an include to *A* or files that are included into *B* after the include to *A*.

The problem that arises here, is that it is practically impossible to decide, if adding or removing an include to such a file *A* has any impact on other files or not. A valuable

enhancement for ReDHead would be to inform the user about this possible impacts by pointing out that there might be unwanted side effects when adding or removing a certain include directive as proposed by a current suggestion.

### Type incomplete type names should be resolvable

When opening an empty C++ file and typing `mytype t;`, the *organize includes* algorithm is not capable to help because the name of the type `mytype` is incomplete. only the code in one of the following listings can make use of the *organize includes* algorithm.

```
2 using namespace mynamespace;  
  mytype t;
```

```
1 mynamespace2::mytype t;
```

It would be very nice if ReDHead was capable of proposing one out of several includes to one out of all types which map the incomplete name `mytype`. The crux here is, that the CDT indexer will probably help in solving this problem.

### User problem feedback

During traversal of the *ReDHead data structure* by any ReDHead algorithm, there are some kind expected problems which might occur. This can for example be includes or *declaration references* that cannot be resolved to the target file or the *declaration*. Such problems are at the moment reported to the user with the help of the `IStatus` logging support of Eclipse. In the case of such problems, after the algorithm finished running, a pop-up dialog is presented by Eclipse which contains problem messages which were constructed by ReDHead.

In the case when algorithm results are presented in the `ReDHeadSuggestionDialog` this Eclipse pop-up dialog is only shown after the suggestion dialog has been closed again. This is of course suboptimal since the user should get notified about these potential problem before he decides to apply all quickfixes directly in the suggestion dialog. A helpful enhancement would be to integrate the created problem messages into the `ReDHeadSuggestionDialog`.

### Configurability of include substitution

The *include substitution* feature which is described in 4.9, right now only works for the standard C++ library's headers. Since this feature would also be useful for the headers of other libraries, it should be adaptable by the user in a ReDHead properties page.

### Add support for polymorphic member function calls to *static code coverage*

The ReDHead data structure does not yield any *declaration reference dependencies* when resolving a *declaration reference* in file *F1* of a member function *M1* which is declared

*virtual* in class *C1* to an overridden member function in a deriving class *C2*. On first sight, this looks like a bug. It is not though, since when for example determining which header files need to be included into *F1*, it would be wrong to propose an include to the file containing *C2*.

This now yields the problem, that the *static code coverage* algorithm does not mark all overriding member function of *M* as used, but rather only the member function *M* of the base class.

I see two approaches to solve this problem. The first is to make the *ReDhead data structure* configurable so it can handle both required situations by yielding *declaration reference dependencies* to overloading member function for the *static code coverage* algorithm, but by yielding only one dependency to the base class for all other ReDHead algorithms. The second approach is to make the *static code coverage* algorithm aware of the problem and have it construct additional dependencies to overloading member function.

## 10.2 Unimplemented Features

### 10.2.1 ReDHead include tag cloud

A tag cloud is a cloud of words or terms, where each term has a custom font size depending the amount of use of the given file. These different sized terms are arranged to form a cloud of words. When looking at such a cloud, one can get a limited, but still nice overview on the focal point of the cloud. So an *include tag cloud* would visualize all the includes which are contained in a project as such a cloud, which helps the user to realize within seconds the mainly used files in his project. It can also help to detect coupling in *physical design*.

### 10.2.2 ReDHead graph view

A tree view which visualizes *declaration reference dependencies* or *include dependencies* is in my point of view a very valuable feature. I was not able to spend any time on this feature yet and thus added this point to the Chapter *Outlook 10*.

### 10.2.3 Implement further algorithms

In this project, not all of the algorithms proposed in Chapter 4 were implemented. One of the major goals in a future work will be to implement further algorithms and enhance the existing ones. The following list shows the algorithms that are yet to be implemented:

- Replace includes with forward declarations 4.6
- Introduce redundant include guards 4.7 which is described in detail also in [Lak96] on page 85.



#### **10.2.4 Combine *compile configuration* results**

The rather vague idea that is described here is to, instead of running static analysis with only one *compile configuration* (see 3.1.1), run it with several ones. *Compile configuration-specific ReDHead data structures* could then be compared to gain a more detailed insight into the include structure of a given project.

# A Continuous Integration Setup

In this section you can find a detailed description how the ReDHead continuous integration environment is set up.

## A.1 Continuous Integration Introduction

When changing existing software one should always make sure that one does not break any existing features while adding or changing others. To not put ones luck to the test using *continuous integration* [Fow] is crucial. In continuous integration, the whole project is built periodically and automated tests are run on the server to make sure that a software project is running properly.

This task is mastered by the build server Hudson [Sun] which observes the Git repository and builds the ReDHead plugins and this documentation automatically when the repository was updated. Then it triggers the automated ReDHead project tests. The result is published on the ReDhead webpage <http://redhead.ifs.hsr.ch/hudson/>. I myself am notified about success or failure through my RSS feed-reader.

## A.2 ReDHead Project Server

The build server used in the ReDHead project is a virtual server provided by HSR. The following sections describe the software component which are running on the server and provide some notes on the setup.

### A.2.1 Git

To maintain the ReDHead repository, Git [Git] is running on the server. Setting up Git on the remote side is easy.

```
2 $ sudo apt-get install git-core
  $ mkdir ReDHead.git
  $ cd ReDHead.git
  $ git --bare init
```

To initially check out the repository on the client one does:

```
2 $ sudo apt-get install git-core
  $ git clone <username>@<servername>:ReDHead.git
```

*username* is the username of the user on the server in which's home directory the repository was created and *servername* is the ip-address or the DNS name of the server.

## A.2.2 Hudson

To automate the build of my plugins, my documentation and running the tests on the server, I use the Hudson continuous integration server. Hudson is configurable on its webpage, which makes its use very simple. It supports automated Git checkout and running of Ant and shell scripts. These scripts that are called are responsible to build the ReDHead plugin, to run its tests and to deliver the results back to Hudson.

Installing Hudson is simple:

```
1 $ sudo echo "\ndeb http://hudson-ci.org/debian binary/" >> /etc/apt/sources.list
   list
$ sudo apt-get update
$ sudo apt-get install hudson
4 $ sudo /etc/init.d/hudson start
```

Now you can connect to <http://localhost:8080/> and configure Hudson from there. To run an Ant or shell script automatically just add a job, let Hudson check out the Git repository if it was updated and run the Ant or shell script.

## A.2.3 Trac

The project management system *Trac* 0.11 is used as ticketing system and to present the ReDHeadProject to the public. After the installation, one is able to browse the Git repository source online and the Hudson build status is shown in the Trac timeline.

Trac is running as an Apache module. So the first setup step is to install Apache and Trac.

```
2 $ sudo apt-get install apache2
$ sudo easy_install Trac
$ apt-get install libapache2-mod-python libapache2-mod-python-doc
$ a2enmod mod_python
```

Now we create the ReDHead Trac project.

```
2 $ sudo mkdir /var/lib/trac/ReDHead/
$ sudo trac-admin /var/lib/trac/ReDHead/ initenv
$ sudo chown -R www-data /var/lib/trac/ReDHead/
```

To install the Trac plugins which enable Git and Hudson support download the zipped sources from <http://trac-hacks.org/wiki/GitPlugin> and <http://trac-hacks.org/wiki/HudsonTracPlugin>, unzip them and run each ones *setup.py* script.

The last thing to do is to adapt the Trac configuration which is located at */var/lib/trac/ReDHead/conf/trac.ini*. I will only post the lines that I have added or modified.

```

[components]
3 hudsontrac.hudsontracplugin.hudsontracplugin = enabled
  tracext.git.* = enabled

[git]
6 cached_repository = true
  git_bin = /usr/bin/git
  persistent_cache = true
9 shortrev_len = 6

[header.logo]
12 alt = IFS Logo
  height = -1
  link = http://redhead.ifs.hsr.ch
15 src = site/LOGO_IFS_TRANSPARENT_RIGHT_COLOR_ENGLISH.png
  width = -1

[HUDSON]
18 alternate_success_icon = true
  feed_url = http://localhost/hudson/rssAll
21 main_page = http://redhead.ifs.hsr.ch/hudson

[trac]
24 # ...
  repository_dir = /home/felu/ReDHead.git
  repository_type = git
27 # ...

```

Note that *site* refers to the path */var/lib/trac/ReDHead/htdocs*.

#### A.2.4 Apache Configuration

The Apache web-server is reachable on port 80 from the internet. Apache redirects requests to Trac, Hudson and delivers ReDHead resources. The following listings show the apache-config located in */etc/apache2/sites-enabled/trac*.

The first part of the config shows the typical Trac apache config.

```
<VirtualHost *>
2   ServerAdmin lfelber@hsr.ch
   ServerName redhead.ifs.hsr.ch
   DocumentRoot /var/www
5   ErrorLog /var/log/apache2/error.trac.log
   CustomLog /var/log/apache2/access.trac.log combined

8   <Location />
       SetHandler mod_python
       PythonInterpreter main_interpreter
11      PythonHandler trac.web.modpython_frontend
       PythonOption TracEnv /var/lib/trac/RedHead
       PythonOption TracUriRoot /
14      PythonOption PYTHON_EGG_CACHE /tmp
   </Location>

17  <Location /login>
       AuthType Basic
       AuthName "trac"
20      AuthUserFile /etc/apache2/trac.passwd
       Require valid-user
   </Location>
```

The rest of the config shown here allows access to the resource files which are published to `/var/www/RedHeadFiles` by the automated build.

```
   <Location "/RedHeadFiles">
2     SetHandler file
   </Location>
</VirtualHost>
```

Since Hudson runs an independent application it needs its own port. My Hudson runs on port 8080 which is not reachable from the outside. To make Hudson accessible from the outside, I use Apache's `mod_proxy` as a reverse http proxy. The following listing shows the `mod_proxy` configuration located in `/etc/apache2/mods-enabled/proxy.conf`.

```
<IfModule mod_proxy.c>
2 ProxyPass /hudson http://localhost:8080/hudson
  ProxyPassReverse /hudson http://localhost:8080/hudson
  ProxyRequests Off
5 ProxyPreserveHost On

<Proxy *>
8 Order deny,allow
  Allow from all
</Proxy>
```

The crux here is to get Hudson itself running not on `/` but on `/hudson/`. To accomplish this the argument `-prefix=/hudson` needs to be added to the hudson script located in `/etc/default/hudson`.

```
...  
HUDSON_ARGS="--webroot=/var/run/hudson/war --prefix=/hudson"
```

### A.3 Automated Building of the ReDHead Plugin and Its Tests

To build and test my plugins on the ReDHead server automatically with Hudson, a script is needed which is capable of first building the ReDHead plugins and then executing its tests. The crux here is, that my plugins contain dependencies to Eclipse and Eclipse CDT components.

Running tests in Eclipse is easy and can be done through “Run AS → JUnit Plugin Test”. This sets up a complete new Eclipse instance which includes all the plugins in the current workspace, runs the Eclipse instance and starts the tests through the Eclipse-Ant-Runner in that instance. The results of the test run is shown in the JUnit view of the invoking Eclipse instance.

To mimic exactly this behavior on the server requires several steps:

1. Starting a virtual display (which is required to launch the tests)
2. Set up a complete new Eclipse instance which contains all required CDT plugins.
3. Build the ReDHead plugins while resolving the plugins dependencies against the Eclipse instance set up in the previous step.
4. Install the ReDHead plugins into the Eclipse instance.
5. Launch the ReDHead tests
6. Stop the virtual display

After the execution of all of these steps, the result of the launched tests can be recycled by Hudson and presented on the Hudson build server page.

### A.3.1 Build Scripts

The ReDHead build environment is fully contained in the ReDHead repository in the folder *build* which is shown in Figure A.1. This brings the big advantage, that, after the repository has been checked out on any computer, the automated build can be launched without any additional configuration.

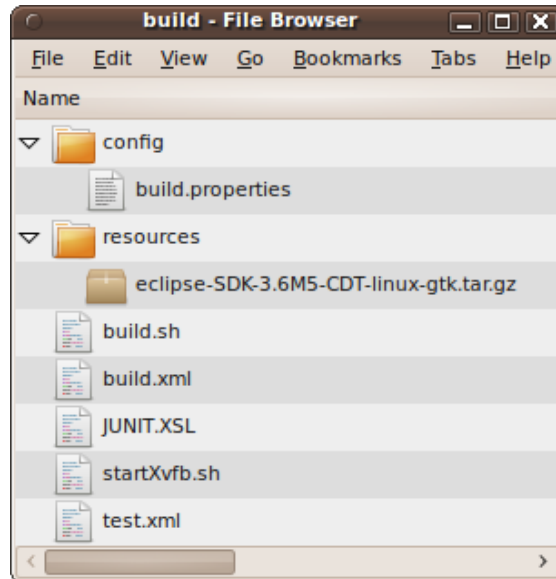


Figure A.1: Build directory overview

The following sections describe how the steps listed in A.3 are accomplished. To trigger the whole build process, the following command can be executed in the repository's *build* directory from a command shell. This is exactly what the Hudson build server does after having automatically checked out the ReDHead Git repository.

```
1 ./build.sh
```

The first two and the last step are taken care of by the ReDHead *build.sh* script which can be seen in the following listing:

```
# The ReDHead build script sets up the environment to run the build.xml ant ↵
  script
2
# make sure we're in the current dir
  cd `dirname $0`
5
if [ -f /usr/bin/Xvfb ]
then
8 #export display
  export DISPLAY=:2
#start virtual X display (on :2)
11 ./startXvfb.sh start
else
  echo "running without virtual display"
14 fi

#setup
17 echo "cleaning directory results"
  rm -rf results
  mkdir results -p
20

#unzipping
  echo "unzipping eclipse"
23 tar xzf resources/eclipse-SDK-3.6M5-CDT-linux-gtk.tar.gz -C results/

# Let's go!
26 java -jar results/eclipse/plugins/org.eclipse.equinox.launcher`1.1.0.v20100118.↵
  jar \
    -ws gtk -arch x86 -os linux -application org.eclipse.ant.core.antRunner

29 #stop virtual X display
if [ -f /usr/bin/Xvfb ]
then
32 ./startXvfb.sh stop
fi
```

The important parts of the script are lines 11, 23, 26, 31. On line 11 the virtual display *Xvfb* is started. Line 23 extracts the Eclipse instance, which already contains all required CDT plugins. On line 26 the Equinox Ant runner which is contained in the previously extracted Eclipse instance is launched. The Ant runner will run the *build.xml* which takes care of the remaining build tasks. Here, launching a normal Ant runner would not be enough since the Equinox Ant runner provides parameters and tasks which are required to build the ReDHead plugins and to run its tests. Line 31 shuts down the virtual display again.



The script which is responsible to start and stop the virtual display *Xvfb* is shown in the following listing:

```
#!/bin/bash
start() {
3 Xvfb :2 -nolisten tcp -shmem >>/tmp/Xvfb.out 2>&1&
  RETVAL=$?
  echo
6 return $RETVAL
}

9 stop() {
  killall Xvfb
  return 0
12 }

case "$1" in
15 start)
  start
  ;;
18 stop)
  stop
  ;;
21 esac
```

The build process which is launched through the *build.xml* is an Eclipse PDE build process that is capable of building plugins and running tests with minimal configuration effort. Since Eclipse products are always distributed as *features*, which are a set of Eclipse Plugins, there is also a ReDHead testing feature which contains the ReDHead, the ReDHead Codan and the ReDHead test plugins. The PDE build process is configured in the file *config/build.properties*. The values I had to set can be seen in the following listing:

```
topLevelElementId = ch.hsr.ifs.redhead.tests.feature
buildDirectory=results
3 baseLocation=results/eclipse
```

The original *build.properties* file is a template which can be found in the extracted Eclipse instance in the *eclipse/plugins/org.eclipse.pde.build\_\*/templates/headless-build/* directory. It is quite big but setting the three properties shown above is sufficient.

The following listing shows a shortened version of the *build.xml* file which is responsible for building the ReDHead plugins, installing them into Eclipse and running the tests.

```

<project default="build">
3  <target name="build" depends="init, setup">
    <antcall target="install-redhead-source"/>
    <antcall target="zips"/>
6    <antcall target="tests"/>
    <antcall target="publish-test-results"/>
    </target>
9
    <target name="init">
        <!-- sets many path / file properties -->
12    </target>
        <!-- ... -->
15
    <target name="install-redhead-source">
        <copy todir="${buildDirectory}/features/ch.hsr.ifs.redhead.feature">
18            <fileset dir="${basedir}/../Software/ch.hsr.ifs.redhead.feature"/>
        </copy>
        <copy todir="${buildDirectory}/plugins/">
21            <fileset dir="${basedir}/../Software/">
                <include name="ch.hsr.ifs.redhead/**"/>
                <include name="ch.hsr.ifs.redhead.tests/**"/>
                <include name="ch.hsr.ifs.redhead.codan/**"/>
24            </fileset>
        </copy>
    </target>
27
    <target name="zips" depends="init,init-eclipse-props">
        <ant antfile="build.xml" dir="${pde.build.scripts}">
30            <property name="builder" value="${basedir}/config" />
        </ant>
    </target>
33
    <target name="tests" depends="init" unless="hasErrors">
        <record name="${testReportDir}/testsLog.txt" action="start" loglevel="←
36            verbose"/>
        <unzip src="${redhead.featureZip}" dest="${buildDirectory}" />
39
        <ant antfile="test.xml">
            <property name="eclipse-home" value="${eclipse.home}"/>
            <property name="library-file" value="${eclipse.home}/plugins/org.eclipse.←
                test`3.3.0/library.xml"/>
            <property name="os" value="${baseos}"/>
            <property name="ws" value="${basews}"/>
            <property name="arch" value="${basearch}"/>
42        </ant>
45
        <xslt style="${basedir}/JUNIT.XSL" in="${testReportDir}/ch.hsr.ifs.redhead.←
            tests.RedHeadTestSuiteAll.xml" out="${testReportDir}/junits.html" />
        <record name="${testReportDir}/testsLog.txt" action="stop" />
48    </target>
51
    <target name="publish-test-results">
        <copy file="${testReportDir}/junits.html" tofile="/var/www/ReDHeadFiles/←
            junits.html" failonerror="false"/>
    </target>
</project>

```

Building the ReDHead plugins first requires to copy the source code of the plugins to the expected location. This is done by the Ant-target `install-redhead-source`. Building the ReDHead feature itself is achieved in the Ant-target `zips`. Note that the property `builder` which is passed there refers to the `config` directory which contains the `build.properties` file which is listed above.

The Ant-task `tests` extracts the previously built ReDHead feature into the Eclipse instance and launches the Ant file `test.xml` which is listed below. Executing the tests results in the file `ch.hsr.ifs.redhead.tests.ReDHeadTestSuiteAll.xml` which contains the test results. This file is used by the Hudson build server to display as result. The file is then also XSLT-transformed into a HTML file which can be found on the ReDhead web-site.

The following listing shows the `test.xml` file that is responsible to start the testing instance of Eclipse and also run the ReDHead tests.

```
1 <project name="ReDHead Automated Tests" default="redhead" basedir=". ">
4   <property name="redhead-loc" value="${eclipse-home}/redhead`tests`data`dir"/>
4   <target name="redhead">
7     <ant target="ui-test" antfile="${library-file}" dir="${eclipse-home}">
7       <property name="data-dir" value="${redhead-loc}"/>
7       <property name="plugin-name" value="ch.hsr.ifs.redhead.tests"/>
7       <property name="classname" value="ch.hsr.ifs.redhead.tests.↵
          ReDHeadTestSuiteAll"/>
10    </ant>
10    </target>
13 </project>
```

Important to see here is that the name of the base ReDHead test suite is passed in listing line 12 to the Ant task `ui-test` which is started in line 9.

## A.4 Automated Build of the Documentation

On the ReDHead Hudson server, there is also an Ant task which is automatically run to build the project documentation which is published automatically to `/var/www/ReDHeadFiles`. The documentation is then directly accessible from the Trac Wiki, where you can find a link to it.

The Ant build file to build the documentation requires the Ant task `latex` which comes in a jar that can be downloaded from the Ant webpage from <http://Ant.apache.org/external.html>. The build file `build.xml`, which can be found in the `Documentation` directory of the repository, is short compared to others:

```
2 <project default="all">
  <taskdef name="latex"
    classname="de.dokutransdata.antlatex.LaTeX"
    classpath="/usr/share/ant/lib/ant-latex.jar"/>
5
  <target name="doLaTeX">
    <latex
8      latexfile="ReDHead.tex"
      verbose="on"
      clean="on"
11     pdftex="on"
      workingDir="."
    />
14 </target>

  <target name="CopyDoc">
17     <copy file="ReDHead.pdf"
          tofile="/var/www/ReDHeadFiles/ReDHead.pdf"/>
    </target>
20 <target name="all" depends="doLaTeX, CopyDoc" />
</project>
```

## B Eclipse Plugin Samples

Here you can get insight into how I adapted the Eclipse user interface to my needs. Each Eclipse plugin is configured in its *plugin.xml* file which, together with the *MANIFEST.MF*, is the entry point to integrate itself into Eclipse.

### B.1 UI Menu Integration

To integrate the ReDHead functionality into the Eclipse menu, one uses several extension points to register *action* classes which are instantiated when the menu entry is clicked on. Then the run method of the *action* is executed. Depending on what kind of menu is extended, the interface for the action class is either *IWorkbenchWindowActionDelegate* or *IObjectActionDelegate*.

#### B.1.1 Extending the Main Menu-Bar

The following listing which belongs to the *plugin.xml* demonstrates how to extend the main menu-bar of Eclipse with an example menu bar entry. The extension point that is used here is `org.eclipse.ui.actionSets`.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
3 <plugin>
  <extension point="org.eclipse.ui.actionSets">
6   <actionSet label="RedHead Action Set" visible="false" id="RedHeadActionSet">
     <menu label="RedHead Static Analysis" id="staticAnalysisMenu">
9       <separator name="StaticAnalysisSeparator"/>
     </menu>
     <action
12       class="ch.hsr.ifs.redhead.ui.actions.FindUnusedIncludesAction"
        icon="icons/unusedInclude.gif"
        id="ch.hsr.ifs.redhead.actions.FindUnusedIncludesAction"
        label="Find unused includes"
15       menubarPath="staticAnalysisMenu/StaticAnalysisSeparator"
        tooltip="Finds unused includes and marks them.">
     <enablement>
18       <or>
         <objectClass name="org.eclipse.cdt.core.model.ITranslationUnit"/>
         <objectClass name="org.eclipse.jface.text.ITextSelection"/>
21       <and>
         <objectClass name="org.eclipse.core.resources.IResource"/>
     </enablement>
  </actionSet>
  <!-- ... -->
```

```

1  <!-- ... -->
      <or>
          <objectState name="extension" value="cpp"/>
          <objectState name="extension" value="c"/>
          <objectState name="extension" value="h"/>
          <objectState name="extension" value="hpp"/>
7      </or>
      </and>
    </or>
  </enablement>
</action>
</actionSet>
13 </extension>

<extension point="org.eclipse.ui.perspectiveExtensions">
16   <perspectiveExtension targetID="org.eclipse.cdt.ui.CPerspective">
      <actionSet id="RedHeadActionSet"/>
    </perspectiveExtension>
19 </extension>

</plugin>

```

The sample action which is added here will show up in the menu as can be seen in Figure B.1.

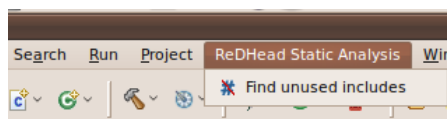


Figure B.1: Example Menu Entry

The visible label of the menu-bar entry is set in listing line 7. The one of the menu entry itself can be found in line 12. The action class which will be instantiated and executed when the menu entry is clicked is specified in line 11. The menubar path defined in line 15 binds the menu entry to the menu which is defined in line 7 and 8. The `enablement` which starts in line 17 represents a binary expression and causes the menu entry to be disabled if the current selection should not be in one of *C Editor* or on any C++ file or project.

The second extension which can be seen on line 36 causes the menu to only be shown when the C++ perspective is active.

### B.1.2 Extending the Navigator's Pop-up Menu

The XML extension point, which adds a menu to the pop-up which shows up when right-clicking on items, takes effect on all of the following Eclipse views.

- C++ Explorer
- Project Explorer
- Navigator

Demonstrated in the following listing is the extension point `org.eclipse.ui.popupMenus`

```
3 <?xml version="1.0" encoding="UTF-8"?>
  <?eclipse version="3.4"?>
  <plugin>
6     <extension point="org.eclipse.ui.popupMenus">
7       <objectContribution
8         id="ch.hsr.ifs.redhead.ui.popupMenu"
9         objectClass="java.lang.Object">
10        <menu
11          id="popupReDheadMenu"
12          label="RedHead Static Analysis"
13          path="additions">
14          <separator name="StaticAnalysisSeparator"/>
15        </menu>
16        <action
17          class="ch.hsr.ifs.redhead.ui.actions.FindUnusedIncludesAction"
18          enablesFor="1"
19          icon="icons/unusedInclude.gif"
20          id="ch.hsr.ifs.redhead.actions.FindUnusedIncludesAction"
21          label="Find unused includes"
22          menubarPath="popupReDheadMenu/StaticAnalysisSeparator"
23          tooltip="Finds unused includes and marks them.">
24        </action>
25        <visibility>
26          <or>
27            <objectClass name="org.eclipse.cdt.core.model.ITranslationUnit"/>
28            <and>
29              <objectClass name="org.eclipse.core.resources.IProject"/>
30              <objectState name="nature" value="org.eclipse.cdt.core.cnature"/>
31            </and>
32            <objectClass name="org.eclipse.cdt.core.model.ICProject"/>
33            <and>
34              <objectClass name="org.eclipse.core.resources.IResource"/>
35              <or>
36                <objectState name="extension" value="cpp"/>
37                <objectState name="extension" value="c"/>
38                <objectState name="extension" value="h"/>
39                <objectState name="extension" value="hpp"/>
40              </or>
41            </and>
42          </or>
43        </visibility>
44      </objectContribution>
45    </extension>
  </plugin>
```

The sample action which is added here will show up in the pop-up menu as can be seen in Figure B.2.

The label of the menu that is added can be found in code line 11 together with the menu that is defined in line 9. The menu is added to the separator group `extentions`, which by default already exists. The label of the added menu entry, together with the icon definition and the class which will be instantiated and run when the menu entry is clicked on, can be found inside of the `action` definition starting on line 15. In line 21,

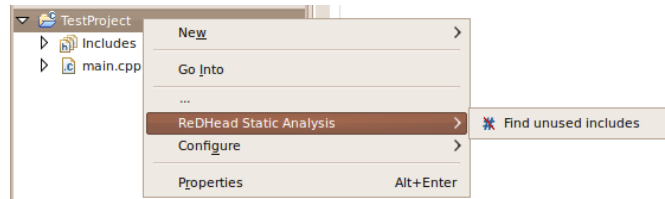


Figure B.2: Example Pop-up Menu Entry

the action is coupled to the menu defined in line 9 and its contained separator (line 13).

Note that the `visibility` which starts in line 25 defines a boolean expression. This expression is used to determine if the added menu is visible for a given selected item.

## B.2 Example Problem Marker

The example shown here creates and removes a dummy marker into the file opened in the active CDT editor.

To create a marker put the following code into the `run` method of an `action` class .See Section B.1 to find out how to add `actions` to Eclipse menus.

```

1  public void run(final IAction action) {
    CEditor ceditor = (CEditor) PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage().getActiveEditor();
    IResource resource = ceditor.getInputCElement().getResource();
4   try {
        IMarker marker = resource.createMarker(ICModelMarker.C_MODEL_PROBLEM_MARKER);
        marker.setAttribute(IMarker.MESSAGE, "myDescription");
7       marker.setAttribute(IMarker.SEVERITY, IMarker.SEVERITY_ERROR);
        marker.setAttribute(IMarker.CHAR_START, 3);
        marker.setAttribute(IMarker.CHAR_END, 5);
10    } catch (CoreException e) {
        System.err.println("Ups, exception with sample marker");
        e.printStackTrace();
13    }
    }

```

Removing all existing markers can be achieved with the following code (also in the `run` method of an `Action` class)

```

1  public void run(final IAction action) {
    CEditor ceditor = (CEditor) PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage().getActiveEditor();
    IResource resource = ceditor.getInputCElement().getResource();
4   try {
        resource.deleteMarkers(null, true, IResource.DEPTH_INFINITE);
    } catch (CoreException e1) {
7       System.err.println("deleting markers failed");
        e1.printStackTrace();
10    }
    }

```



## B.2.1 Customized Markers

In certain cases, one wants to give a marker, which is added, custom properties. This can for example be to add an own marker icon, to define a background color etc. The following listing shows how such a marker definition looks. It defines a new marker type which, instead of showing the normal problem icon, shows a customized RedHead icon.

```
2 <?xml version="1.0" encoding="UTF-8"?>
  <?eclipse version="3.4"?>
  <plugin>
5     <extension
      point="org.eclipse.core.resources.markers"
      id="ch.hsr.ifs.redhead.unusedincludemarker">
8       <super type="org.eclipse.core.resources.problemmarker" />
      <persistent value="true" />
    </extension>
11
    <extension point="org.eclipse.ui.editors.markerAnnotationSpecification">
      <specification
14         colorPreferenceValue="254,155,0"
          annotationType="ch.hsr.ifs.redhead.unusedincludeannotation"
          verticalRulerPreferenceValue="true"
17         colorPreferenceKey="indexResultIndicationColor"
          contributesToHeader="false"
          overviewRulerPreferenceValue="true"
20         presentationLayer="3"
          textStylePreferenceValue="NONE"
          symbolicIcon="warning"
23         icon="icons/unusedInclude.gif"
          label="Unused Include Directive"
          textPreferenceValue="true"
26         textPreferenceKey="indexResultIndication"
          verticalRulerPreferenceKey="indexResultIndicationInVerticalRuler"
          overviewRulerPreferenceKey="indexResultIndicationInOverviewRuler"
29         showInNextPrevDropdownToolBarActionKey="↔
            isIndexResultInNextPrevDropdownToolBarAction"
          showInNextPrevDropdownToolBarAction="true"
          isGoToNextNavigationTargetKey="isIndexResultGoToNextNavigationTarget↔
            "
32         isGoToNextNavigationTarget="false"
          isGoToPreviousNavigationTargetKey="↔
            isIndexResultGoToPreviousNavigationTarget"
          isGoToPreviousNavigationTarget="false">
35      </specification>
    </extension>
38
    <extension point="org.eclipse.ui.editors.annotationTypes">
      <type
41         markerType="ch.hsr.ifs.redhead.unusedincludemarker"
          name="ch.hsr.ifs.redhead.unusedincludeannotation">
      </type>
    </extension>
44
  </plugin>
```

Creating a customized marker requires three things. The first is the definition of a new marker (listing line 5). The second thing is the definition of the marker's look

(line 12). The last thing is to link the new marker look to the new marker definition (line 38). The new icon which is shown with our new maker can be found on line 23.

In the marker definition above, there is a marker id defined for the new marker on listing line 7. To create a maker of the newly defined type, one can follow the example shown in the previous Section B.2, expect that, instead of the term `ICModelMarker.C_MODEL_PROBLEM_MARKER` which also refers to a marker id, one can use the new id `"ch.hsr.ifs.-redhead.unusedincludemarker"`. The resulting maker can be seen in Figure B.3.

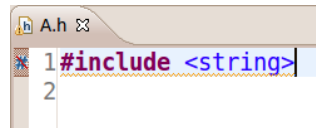


Figure B.3: Example Customized Marker

### B.3 Example Quickfix

In Eclipse, quickfixes are attached to problem markers. They are shown in Eclipse when pressing `ctrl+1`. To get the `RedHeadQuickFixes` into Eclipse, I needed to add the following to the `plugin.xml`. This registers the `RedHeadQuickFixProcessor` so it gets asked for quickfixes by Eclipse.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
3 <plugin>
  <extension point="org.eclipse.cdt.ui.quickFixProcessors">
    <quickFixProcessor id="RedHeadQuickFixProcessorsExtension" name="RedHead ↵
      Quickfix Processor"
6    class="redhead.ui.RedHeadQuickFixProcessor">
    <handledMarkerTypes>
      <markerType id="org.eclipse.myplugin.audits"/>
9    </handledMarkerTypes>
    <enablement>
      <with variable="projectNatures">
12      <iterate operator="or">
        <equals value="org.eclipse.cdt.core.cnature"/>
      </iterate>
15    </with>
    </enablement>
    </quickFixProcessor>
18 </extension>
</plugin>
```

The class attribute of the quickFixProcessor defines the name of the QuickFix-Processor class which is shown here:

```
1 public class RedHeadQuickFixProcessor implements IQuickFixProcessor {
    public ICompletionProposal[] getCorrections(final IInvocationContext context, ←
        final IProblemLocation[] locations) throws CoreException {
        ArrayList<ICompletionProposal> proposalList = new ArrayList<←
            ICompletionProposal>();
4        proposalList.add(new RedHeadDummyCompletionProposal());
        return proposalList.toArray(new ICompletionProposal[0]);
    }
7
    public boolean hasCorrections(final ITranslationUnit unit, final int problemId←
        ) { return true; }
}
```

Here the RedHead ICompletionProposal implementation is used. You can see a dummy implementation of this class here:

```
public class RedHeadDummyCompletionProposal implements ICompletionProposal {
3
    public String getIdString() {return "myDummyID";}
    public int getRelevance() {return 0;}
    public void apply(final IDocument document) {/* apply changes here */}
6    public String getAdditionalProposalInfo() {return "My accurate description.";}
    public String getDisplayString() {return "RedHead dummy QuickFix";}
9
    public IContextInformation getContextInformation() {return null;}
    public Image getImage() {return null;}
    public Point getSelection(final IDocument document) {return null;}
12 }
```

I should probably mention here that the quickfix in CDT is only shown when invoked through *ctrl+1* and not by clicking on the problem marker's icon. However it seems that this is also true for all the other CDT problem markers. I will not investigate further on this issue, but this should maybe be looked into.

## B.4 Example Codan Checker

The CDT Codan framework allows to check the content of documents also while typing. The components that can be added to the Codan framework are called *Codan checkers*. When a file gets changed in a C++ editor, each Codan checker is called to analyze the document and report problems. These problems will then be shown as markers in the editor. Given some useful checkers, this is a very nice feature that helps to produce C++ code even more easy. The following listing shows the additions, which have to be added to *plugin.xml* to implement a Codan checker.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
3 <plugin>
  <extension point="org.eclipse.cdt.codan.core.checkers"
    id="org.eclipse.cdt.codan.core.internal.checkers">
6     <checker
      class="ch.hsr.ifs.redhead.codan.checkers.ExampleChecker"
      id="ch.hsr.ifs.redhead.codan.checkers.ExampleChecker"
9      name="Find Unused Includes Checker">
      <problem
12         category="org.eclipse.cdt.codan.core.categories.ProgrammingProblems"
         defaultSeverity="Warning"
         id="ch.hsr.ifs.redhead.codan.problems.ExampleProblem"
         messagePattern="{0}"
15         name="Unused Include"/>
      </checker>
    </extension>
18 </plugin>
```

In the listing above, the Codan checker class is registered (listing line 7). Its implementation is shown in the following listing. Beside the checker itself, additionally, a problem type is registered (code line 11).

```
public class ExampleChecker extends AbstractIndexAstChecker {
  private static final String ER_ID = "ch.hsr.ifs.redhead.codan.problems.↵
    ExampleProblem";
3
  @Override
  public void processAst(IASTTranslationUnit ast) {
6    reportProblem(ER_ID, ast.getDeclarations()[0], "This declaration is a ↵
      problem.");
    reportProblem(ER_ID, ast.getDeclarations()[2], "This one as well.");
9  }
}
```

The example checker shown above calls the `reportProblem` method twice, each time reporting a given AST node as a problem. This will cause Codan to mark these two nodes in the editor.

## B.5 Undo-Redo Operations

When ReDHead applies a *quickfix*, normally, `TextChangees` are applied on `IDocument` instances. These operation automatically add `IUndoableOperations` to Eclipse, which allows the user to undo and redo the applied text change with the help of the *Edit* menu. If one wants to add such undo and redo behavior for any custom operation, this can be done as shown in the following example listing. The custom operation, which shall be undoable and redoable in the listing could for example be a change which deletes an other file. In this case, the `Change` argument passed into `addUndoRedoOperation` of in the listing would be a `DeleteFileChange` instance.

```
private void addUndoRedoOperation(Change change) {  
    IUndoableOperation operation = new ChangeOperation(change);  
3    IWorkbench workbench = PlatformUI.getWorkbench().getActiveWorkbenchWindow().getWorkbench();  
    IOperationHistory operationHistory = workbench.getOperationSupport().getOperationHistory();  
    operation.addContext(IOperationHistory.GLOBAL_UNDO_CONTEXT);  
6    try {  
        operationHistory.execute(operation, null, null);  
    } catch (ExecutionException e) {  
9        throw new RedHeadException(e);  
    }  
}
```

# C Organizational

In this chapter you can find organizational information about the ReDHead project. You will find a list of tools I used, followed by some information about the project plan and the time schedule. The last section contains my personal impression.

## C.1 Project Environment

The environment consists of two parts. This is on the one hand my notebook where I develop and document on and on the other hand the project server which is used to present the project to the public, to maintain the project repository and to build and test my code and documentation.

### C.1.1 Development Environment

The following list describes all the components which were used to develop the ReDHead plugin and to write this documentation.

**Ubuntu** As development platform I used an Ubuntu 9.10, code-name Karmic Koala [Ubu].

**Git** Git [Git] version 1.6.3.3 serves as the repository for ReDHead so all changes that are done are logged and trackable.

**Eclipse** As development editor I used the well known Eclipse Platform [Eclb].

**CDT refactoring-test-editor** To write down automated testing C/C++ code I used the refactoring-test-editor which was developed by Emanuel Graf.

**TeXlipse Eclipse Plugin** As  $\text{\LaTeX}$  editor, the Eclipse plugin TeXlipse [Tex] was used.

**texlive** The TexLive Ubuntu packages are required to compile this  $\text{\LaTeX}$ -documentation.

**Visio** To draw a part of the code diagrams, I used Microsoft Visio [Micb].

**Dia** All class diagrams were drawn with the open source tool Dia [Dia] as well as some other of the diagrams contained in this documentation.

## C.1.2 ReDHead Build Server

The ReDHead project server can be found at <http://redhead.ifs.hsr.ch/>. The following components are running on the server.

**Ubuntu** The server operating system is a Ubuntu 8.04.3 LTS [Ubu].

**Git** Git [Git] version 1.6.4.3 runs on server side to store the repository.

**Apache** Apache version 2.2 is used as web server [Apab].

**Trac** As a project management environment, I use Trac version 0.11 [Edg].

**texlive** The TexLive packages are required to compile the L<sup>A</sup>T<sub>E</sub>X-documentation [Tex].

**Hudson** The build server in use is Hudson version 1.323 [Sun].

**Ant** Building of the ReDHead plugin is done with the help of Ant tasks [Amaa].

## C.2 Project Plan

In this section you can find two versions of the ReDHead project plan of this master thesis. Figure C.1 shows the initial project plan that was created in the first week of the master thesis.

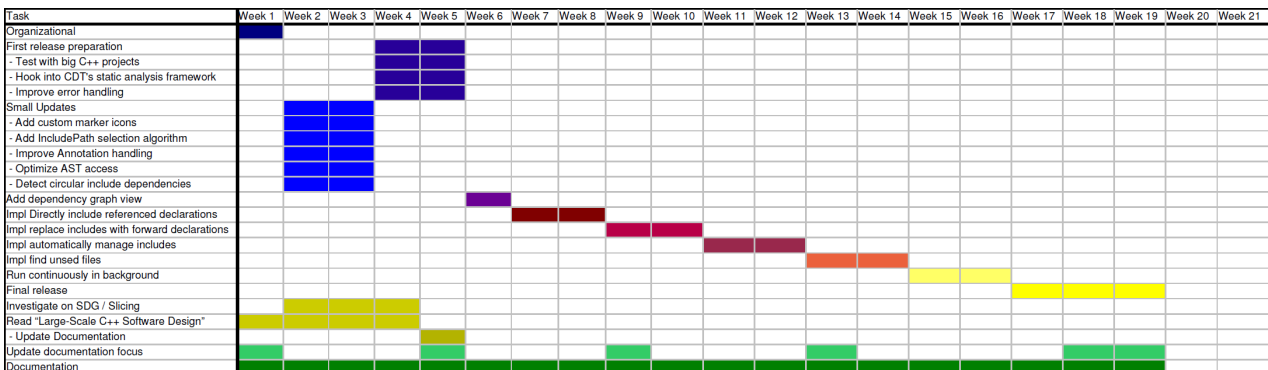


Figure C.1: Initial Project Plan

Note that the investigation on the task *Run continuously in background* yielded that keeping the *ReDHead data structure* up-to-date on changes in ASTs and indexer would be very complex to achieve and also is not necessary because the *ReDHead data structure* as it is available now performs very well so that even reconstructing it while a programmer is typing each time works well enough.

The task *Investigate on SDG / Slicing* resulted in the Chapter 2 of this documentation.

During the first half of the master thesis, the project plan was refined. There were several tasks that were added which were necessary so the *RedHead data structure* worked precisely and also fast enough. Note also that, due to some of these added tasks, others have been postponed to the end of the project. The decision to postpone these tasks was taken either with the approval or even because of the advice of my supervisor.

The resulting project plan can be seen in Figure C.2.

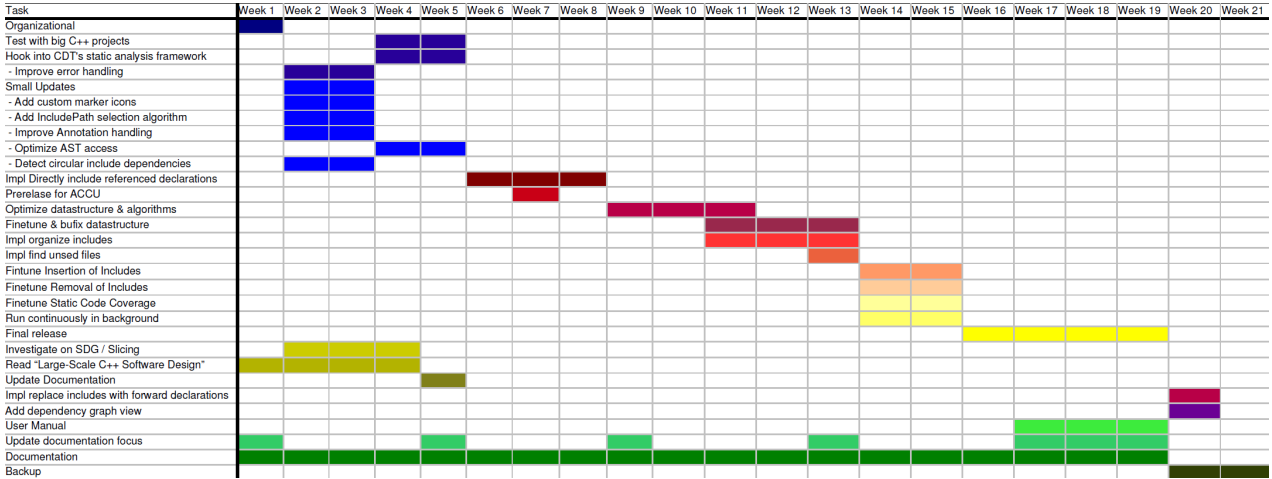


Figure C.2: More Detailed Project Plan



### C.3 Time Schedule

In Figure C.3 the blue bars show how many hours I worked each week. The yellow line shows the weekly target time of the project. In green you can see the average work-time of the the past weeks.

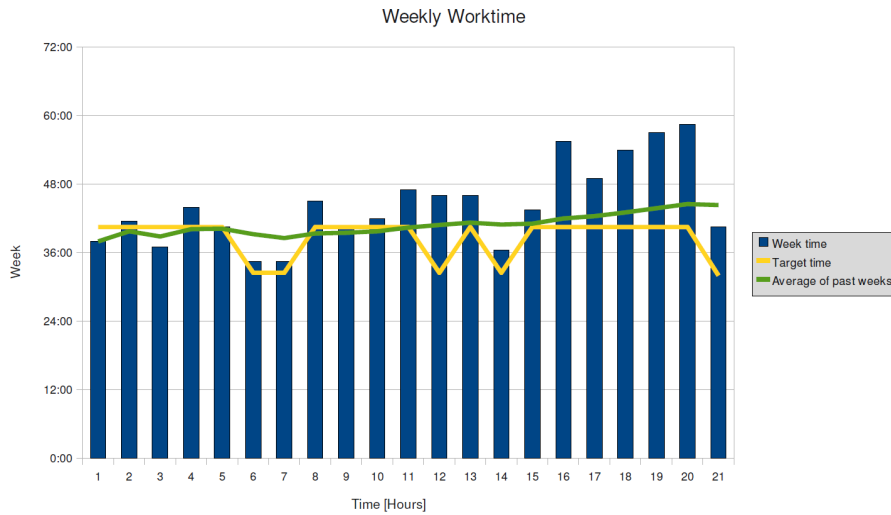


Figure C.3: Weekly working hours

Figure C.4 shows the accumulated working hours over the whole project period. In orange you see the time I totally worked, in blue the target time.

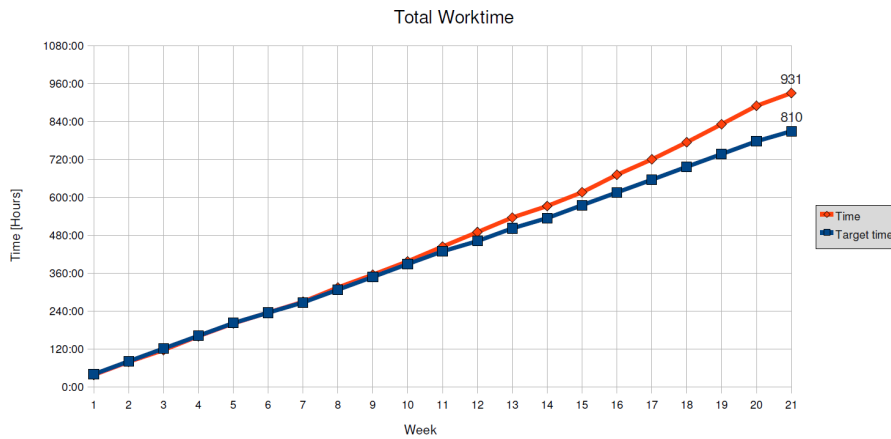


Figure C.4: Total working hours

## C.4 CDT Bug Tickets

The following list shows all the CDT bug tickets that were created during the ReDHead master thesis. Most of the bugs deal with very special indexer scenarios. Indexer bug tickets were processed by the CDT developer Markus Schorn who reacted very quickly and also very competently.

**[https://bugs.eclipse.org/bugs/show\\_bug.cgi?id=295424](https://bugs.eclipse.org/bugs/show_bug.cgi?id=295424)** Calling the quick-fix menu in an editor and double clicking on *No suggestions available*, the document was deleted from offset 0 to the offset of the caret.

**[https://bugs.eclipse.org/bugs/show\\_bug.cgi?id=296906](https://bugs.eclipse.org/bugs/show_bug.cgi?id=296906)** Sometimes, it was not possible to resolve the *operator <<* for the indexer. This bug reported the problem.

**[https://bugs.eclipse.org/bugs/show\\_bug.cgi?id=306819](https://bugs.eclipse.org/bugs/show_bug.cgi?id=306819)** The problem describe here is to find all *IASTNames* also inside of *IASTPreprocessorStatements* which was already described in 9.4. Note that this problem was and will not be fixed.

**[https://bugs.eclipse.org/bugs/show\\_bug.cgi?id=312399](https://bugs.eclipse.org/bugs/show_bug.cgi?id=312399)** This bug was reported because when clicking on a name after an `#undef` statement, a `NullPointerException` was thrown.

**[https://bugs.eclipse.org/bugs/show\\_bug.cgi?id=316931](https://bugs.eclipse.org/bugs/show_bug.cgi?id=316931)** Assumed there is a function that takes an `int` argument. The function is first declared where the argument's name is `j` and then defined where the name is `i`. When asking the indexer to resolve the declaration's name `j`, the result was that it returned a reference to `i` in the definition, but not to `j` itself, as it normally would.

**[https://bugs.eclipse.org/bugs/show\\_bug.cgi?id=316836](https://bugs.eclipse.org/bugs/show_bug.cgi?id=316836)** This bug is the one that has already been discussed in 9.2 that concerns the synchronism of AST and indexer in the case of modified, not yet saved, files. This problem has not been solved.

## C.5 Personal Impression

My overall personal impression on the ReDHead project is very positive. I was able to gain even more experience on how to work in software projects. The big difference between this project and predecessor project like my diploma thesis [CFS06] was that I was completely on my own. The following two lists show positive and negative points of these projects.

Positive Points:

**Only my own code** All the code which was written in this project is completely my own. This raises the overview I have over the project code. Like this I can make sure there is no unknown code duplication.

**Effectively implemented design** Based on the improved overview of the code I can design (and also re-design) more effectively because I have all the knowledge that is needed for this task.

**Supervisor** When working alone, the person in the supervising role gets much more important because he is the sole person you can go to with problems.

**Early documentation** I started documenting very early in the term project. This is a very good thing to do because one does not need to reconstruct precise knowledge one had at a past time.

**Extensive Performance Testing** Even due to the fact that I used a bigger, real C++ project to test how the ReDHead plugin performs, I think that I should have put more focus on this issue. The fact that I found out that parts of the developed components do not perform fast enough to be used in a release highlights this fact.

Negative Points:

**Early documentation** Even though early documentation is, as mentioned, a good thing to write down detailed knowledge at a given time, one has to re-process these parts again and again because the circumstances change while the project proceeds. When software components change during refactoring, or maybe even are removed, the documentation changes along all over. This results in a big loss of time which could have been spent on other tasks.

The optimal way to document during a software project thus is, in my point of view, to document early and continuously, but rather than writing full text passages which can be considered finished at the time of writing, while only updating the outline and writing text in a sketchy style. Like this, all the facts which are important to end up in the documentation will be available, while the loss of time when re-processing the documentation will be small, because the real documentation text will only be written during the ending phase of the project and thus only once.

**Discipline and Motivation** When one works completely alone, except for the weekly meetings with the supervisor, work discipline is bound to falter from time to time. I think I could master this difficulty quite well, but if I could choose, I would definitely prefer a team project because the obligation towards and the help from other team-members makes this barrier much smaller.

Overall, I think this master thesis was a great experience to add to my growing collection of software engineering projects that I worked on.

At this point I want to thank my advisor, Prof. Peter Sommerlad, for all the guidance I received and for the motivation he helped to raise.

## C.6 Changes Since Term Project

Some section of figures in this documentation, which, for the integrity of the documentation, could be set aside, are either completely, or in a reformulated, also contained here. The following two list identifies such sections.

- Adapted: Paragraph describing Figure 1.1 and the figure itself.
- Unchanged: JDT feature list in Section 1.5
- Adapted: Section 3.1
- Adapted and extended: Section 4.6
- Unchanged: First three paragraphs of Section 6.2
- Adapted: Section 6.3
- Unchanged: Section 6.5, extended with Subsection 6.5.1
- Unchanged: Section *Vast Amount of CDT and Eclipse Code* and *Resolving Declaration References* in Chapter 9
- Adapted: Section 10.2.2
- Unchanged: Appendix Section A.1 and A.2 and A.4
- Adapted: Appendix Section A.3
- Unchanged: Appendix Section B.2 and B.3
- Unchanged: Appendix Section C.1

## D Nomenclature

**Annotation** An annotation, as we use the expression here, is highlighted text within an editor. Note that there are also *Java Annotation*. What I refer to as *annotations* here are not *Java Annotations*. An *Annotation* can not only highlight text with a background-color, but also with an icon or by underlining the text with sinuous lines.

**AST** Abstract Syntax Tree

**CDT** C++ Development Tooling [Ecla]

**Completion proposal** Other term for *quickfix*.

**Declaration** The term *declaration* refers to C++ type, function, or variable declarations and so on.

**Declaration Reference** A *declaration reference* denotes for example be a *function call*, *type* or *variable name*.

**IDE** Integrated Development Environment

**Include Guard** This term denotes conditional inclusion like *#ifdef*, *#ifndef* that guard *#include* directives

**JDT** Java Development Tooling [Eclb]

**Quickfix** A *quickfix* is the Eclipse construct that comes along with a *problem marker* and can be applied to quickly solve a given problem denoted by the *problem marker* at hand.

**Type** *Type* in C++ means a *class*, *struct*, *template* or *fundamental type*.

**VS** Visual Studio [Mica] statements.

**SDG** System Dependence Graph [HRB90]

**PDG** Program Dependence Graph [HRB90]

**Graph** A graphic depicting the relationship between vertices

**Edge** A relation visualization on a graph

**Vertex** A item contained in a graph

**Physical Design** see page 12 in [Lak96]

**Logical Design** see page 12 in [Lak96]

**GCC** Gnu Compiler Collection

**Eclipse PDE** Eclipse Plugin Development Environment

## Bibliography

- [ADS93] Hiralal Agrawal, Richard A. Demillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23:589–616, 1993.
- [Aaaa] The Apache Ant Project, <http://ant.apache.org/>. *Apache Ant Homepage*.
- [Apab] The Apache Software Foundation, <http://httpd.apache.org/>. *Learning the Java Language*.
- [BBC<sup>+</sup>10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.
- [BW88] Lee Badger and Mark Weiser. Minimizing communication for synchronizing parallel dataflow programs. In *ICPP (2)*, pages 122–126, 1988.
- [CFS06] Thomas Corbat, Lukas Felber, and Mirko Stocker. *Refactoring Support for the Eclipse Ruby Development Tools*. <http://r2.ifs.hsr.ch/trac>, 2006.
- [cod09] *Codan - CDT Code Analysis*. <http://wiki.eclipse.org/CDT/designs/StaticAnalysis>, 2009.
- [Dia] Dia Community, <http://projects.gnome.org/dia/>. *Dia Homepage*.
- [Ecla] Eclipse Foundation, <http://www.eclipse.org/cdt/>. *Eclipse CDT*.
- [Eclb] Eclipse Foundation, <http://www.eclipse.org/>. *Eclipse Homepage*.
- [Edg] Edgewall Software, <http://trac.edgewall.org/>. *Trac Homepage*.
- [Fel09] L. Felber. *ReDHead - Refactor Dependencies of C/C++ Header Files*. [http://redhead.ifs.hsr.ch/ReDHeadFiles/ReDHead\\_EndOfTermProject.pdf](http://redhead.ifs.hsr.ch/ReDHeadFiles/ReDHead_EndOfTermProject.pdf), 2009.
- [Fow] Martin Fowler. *Continuous Integration*. Martin Fowler.
- [Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.
- [GHJV97] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, July 1997.

- [Git] The Git Project, <http://git-scm.com/>. *Learning the Java Language*.
- [HPR89] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11:345–387, 1989.
- [HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12:26–60, 1990.
- [Ins] Institute for Software, HSR University of Applied Sciences, <http://r2.ifs.hsr.ch/cdtrefactoring>. *CDT C++ Refactoring Homepage*.
- [Ins03] British Standards Institute. *The C++ Standard: Incorporating Technical Corrigendum No. 1*. John Wiley & Sons, Hoboken, NJ, USA, 2003.
- [Lak96] John Lakos. *Large-scale C++ software design*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.
- [LH96] L. Larsen and M.J. Harrold. Slicing object-oriented software. *Software Engineering, International Conference on*, 0:495, 1996.
- [Mica] Microsoft Corporation, <http://www.microsoft.com/visualstudio/en-us/>. *Microsoft Visual Studio Homepage*.
- [Micb] Microsoft Corporation, <http://office.microsoft.com/en-us/visio/>. *Visio Homepage*.
- [Str97] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
- [Sun] Sun Microsystems, Inc., <http://hudson-ci.org/>. *Hudson Homepage*.
- [Tex] Texlipse.sf.net community, <http://texlipse.sourceforge.net/>. *TeXlipse Homepage*.
- [Ubu] Ubuntu Community, <http://www.ubuntu.com/>. *Ubuntu Homepage*.
- [Wei83] Mark Weiser. Reconstructing sequential behavior from parallel behavior projections. *Inf. Process. Lett.*, 17(3):129–135, 1983.