TERM PROJECT, FALL SEMESTER 2012

elevator

C++11 Enhancements for Eclipse CDT

IFS INSTITUTE FOR SOFTWARE

Authors
André Fröhlich
Christian Mollekopf
Advisor
Peter Sommerlad

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Task

We want to develop two Eclipse-Plugins to help developers writing and fixing code according to the C++11 standard.

The first plugin concentrates on initializers and should detect both uninitialized declarations, as well as non C++11 initializations and allow to easily fix or convert to C++11 initializations. For instance would an uninitialized declaration, such as:

```
int x;
```

be marked in the code and a quickfix for the issue would ask the developer for an initialization value and fix the initialization:

```
int x{};
```

The second plugin should allow to extract lambda expressions into functions or functors to allow for code reuse. This includes identifying local variables which need to be passed to the generated function/functor, identifying if the variables should be passed by reference or by value, and asking for a suitable name for the function/functor.

For instance could a lambda expression like this:

```
[](int x, int y) { return x + y; }
```

be transformed into a reusable function like this:

```
int add(int x, int y) { return x + y; }
```

Or a lambda expression from:

```
auto a = [](int x, int y) { return x + y; }
```

to:

```
struct Addition {
    int operator()(int x, int y) { return x + y; }
};
```

A dialog box may prompt for a name, similar to the existing Ëxtract Function...r̈efactoring.

Both Plugins should be developed together with suitable documentation, and a fully automated and installable build.

Abstract

The aim of this project was to develop an Eclipse plug-in to help highlighting and fixing initialization issues in respect to the new C++11 brace-initializer syntax. The C++11 brace-initializer provides a uniform syntax for initialization, and is preferred in C++11 code over pre-C++11 initialization mechanisms such as copy-initialization or direct-initialization.

The developed Eclipse plug-in detects initialization issues and provides an automated quickfix, to make it as easy as possible to fix initialization issues. Besides detecting issues, the plug-in also ensures that no semantic changes are introduced by a quickfix, so the resulting code is equivalent.

The plug-in is written in Java, using Eclipse CDT's Codan, which is a static code analysis framework for C++, providing an abstract syntax tree, that we traverse and modify to detect and fix the issues. To realize the Eclipse plug-in the existing quickfix-plug-in framework was used to ensure that the plug-in integrates well into eclipse and behaves as any other quickfix plug-in.

To enable efficient fixing of larger code-bases to C++11-initializations, an additional mass-refactoring has been implemented. This functionality can be triggered from the usual refactoring menus, to fix either complete source files or even full projects.

Management Summary

This chapter shows you the goals of the Elevator project, gives you an overview how the Elevator plug-in can help you converting existing C++source code to the new C++11 syntax and ends with a look at what can be improved even more.

Prior to C++11 there was no uniform way to initialize variables. In addition uninitialized variables are a common issue. C++11 introduced with brace initialization a uniform way to initialize variables. Take a look at this simple example.

```
int a();
int b = 4;
int c(2);
int d;
```

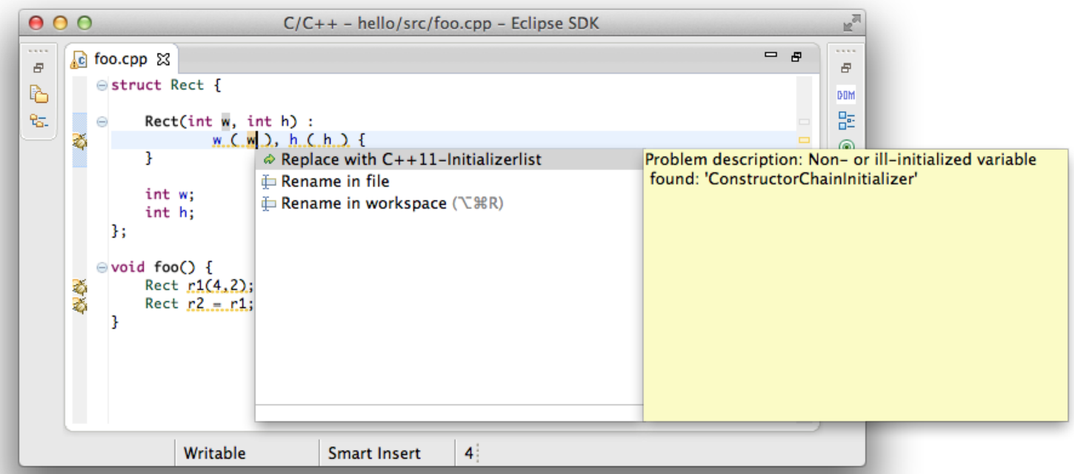
Notice that the last variable is not initialized. The other variables show common ways to initialize variables. In C++11 we can uses braces {} for initialization.

```
int a{};
int b{4};
int c{2};
int d{};
```

This new syntax can also be used like an initalizer list to initialize sequences of values. It also prevents narrowing issues.

During the project we have developed an Eclipse CDT plug-in that helps to convert C++source code to the new C++11 syntax. After installating the plug-in it highlights any occurrence of a variable initialization not using the new brace syntax. To resolve a issue a developer can select it and use the provided Eclipse quickfix to resolve it. In addition it is possible to convert all occurrences in the active editor or even in the project.

To achieve this the plug-in uses the static code analysis frame of Eclipse CDT. It can parse and alter the abstract syntax tree und find issues and convert them.


C/C++ – hello/src/foo.cpp – Eclipse SDK
foo.cpp
Replace with C++11-Initializerlist
Rename in file
Rename in workspace (⌥⌘R)
Writable
Smart Insert
4

```
struct Rect {
    Rect(int w, int h) :
            w ( w ), h ( h ) {
    }

    int w;
    int h;
};

void foo() {
    Rect r1(4,2);
    Rect r2 = r1;
}
```

Problem description: Non- or ill-initialized variable found: 'ConstructorChainInitializer'

The Elevator Eclipse CDT plug-in in action

Even the plug-in is useful to convert legacy code, it would be nice to be able to convert the code back to not use the new brace syntax. This could be useful when one wants to use new code in a project where the new C++11 syntax is not permitted. Unfortunately we had not enough time to implement this feature. It would also be conceivable to support more C++11 features.

Contents

1. Introduction

Everyone who writes C++code, occasionaly forgets to initialize variables. The code compiles and often the programm seems to work as expected. Unfortunatly this is error-prone and can lead to undefined behaviour.

With the introduction of the C++11 standard there is a new, simple and consistent way to initialize variables using curly braces.

It would be convient if the integrated development enviroment could provide some assistance for such cases. In the Eclipse CDT there are already markers avaible for various kinds of hints, and some even provide quickfixes as needed.

The idea is to develop a new Eclipse CDT plugin which provides a marker for the introduced problem, as well as a quickfix which allows to fix the problem automatically.

The plug-in should for example detect and mark the following uninitialized variable:

```
int i;
```

An invocation of the corresponding quickfix, would then result in the variable being default initialized:

```
int i{};
```

Similarly non-C++11 code should be refactored to use the C++11 initializer syntax:

```
int i = 1;
int y(1);
```

would result in:

```
int i{1};
int y{1};
```

2. Analysis

The following relevant cases of initialization exist:

No.	Name	Example
1.	No initialization for POD types	int x;
2.	Constructor default initialization	T x;
3.	Direct initialization	T x(3);
4.	Copy initialization	T x = 3;
5.	Braced-init-list default initialization	T x{};
6.	Braced-init-list initialization	T x{3};

Table 2.1.: Initializations

The default strategy for conversion looks the follwing:

Description	Detected	Converted
POD default initialization	1. int x;	5. int x{};
Default initialization	2. T x;	5. T x{};
Direct initialization	3. T x(3);	6. T x{3};
Copy initialization	4. T x = 3;	6. T x{3};
Reverse default initialization	5. T x{};	2. T x = T();
Reverse direct initialization	6. T x{3};	3. T x(3);

Table 2.2.: Default Conversions

Example:

```
T x; //=> T x{};
```

```
T x(3); //=> T x{3};
```

```
T x(3, 2); //=> T x{3, 2};
```

```
T x(3, func()); //=> T x{3, func()};
```

2.1. Special Cases

The following section lines out the existing special cases where the default strategy does not apply.

2.1.1. Container using an initializer-list-constructor

If a type has an initializer-list constructor, we cannot convert the constructor call to the initializer-list syntax without introducing potential ambiguity.

Example:

```
vector<int>(4,2) //=> vector<int> x{2,2,2,2}, not vector<int>x{4,2}.
```

Strategy:
If the constructor call contains arguments and the type contains an initializer-list-constructor, the call must not be converted to the braced-init-list.

2.1.2. Use of implicit type conversion when using direct- or copy-initialization

Direct-initialization using the Constructor-initializer and copy-initialization allow the use implicit type conversion, but the direct-initialization using the brace-initializer does not. Therefore conversions of an initialization which relies on implicit type conversion must not be carried out automatically. The user may decide to use explicitconversion, but this is not safe to do automatically.

Example:

```
    int i = 3.2; // => int i{3.2}; //Invalid!
    int i = 3; // => int i{3};
```

Strategy:
Only convert automatically if types are matching exactly and no implicit conversion is happening. This can be checked by looking for an appropriate constructor in the target type.

2.1.3. boost::assign

Initializations using boost::assign have to be skipped as we cannot convert them.

Example:

```
//works
boost::unordered_map<int, std::string> map = boost::assign::map_list_of
(1, string("Air"))
(2, string("Dark"));

//doesn't work
boost::unordered_map<int, std::string> map {boost::assign::map_list_of
(1, string("Air"))
(2, string("Dark"))};
```

The second example doesn't work because the unordered_map has no constructor which takes a generic_list (the return type of map_list_of), and the type must match exactly when using the brace-initializer. When using the equal-initializer on the other hand, the type must only be convertible to the target type. The conversion would thus break valid code.

Non initialization boost::assign uses can be safely ignored though, as we can default-initialize the vector as usual.

```
vector<int> x; //=> vector<int> x{};
x += 1, 2, repeat(8, 5), 7;
```

Strategy:
If boost::assign is detected within a declaration, the declaration should be skipped.

2.1.4. Reference initialization

Uninitialized references should be ignored by the plugin as they're illegal anyways and cannot be default initialized. Equal-Initialized references should be converted to the brace-initializer as usual, but as there is a bug in the current standard, this feature should be configurable and disabled by default.

Example:

```
int x;
int &y = x; //=> int &y{x}; //Illegal in current standard but valid in future
    revision
```

Strategy:
Ignore uninitialized references. Configurable conversion for copy-initialized references.

Note:
While the used draft of the C++11-standard[1] does not support initializing references using the brace-initializer syntax, the feature is being worked on and future versions of the standard should support this..

[1] N3337, *Working Draft, Standard for Programming Language C++.*

2.1.5. Declaration Lists

Multiple uninitialized declarations should be handled properly.

Example:

```
int x, y=3, z(4); //=> int x{}, y=3, z{4}
```

Strategy:
Iterate over declarations and handle each individually.

2.1.6. Function declarations

Function declarations have the same syntax as a constructor call but must be ignored.

Example:

```
int x();
```

Strategy:
Ignore function declarations.

2.1.7. Pointers

Pointers should be default initialized just as any other variable resulting in a nullptr.

Example:

```
int *x; => int *x{};
```

Strategy:
Initialize as usual.

2.1.8. Member variables

Member variables should be ignored, as this can lead to a change of semantics and the initialization is covered upon instantiation.

Example:

```
struct A {
    int x;
};
```

Strategy:
Ignore member variables.

2.1.9. Constructor init-list

Constructor init-lists should be converted to C++11-initialization as well.

Example:

```
C(): a(1), b(2){}; // => C(): a{1}, b{2}{};
```

Strategy:
Initialize as usual.

2.2. Features

The plugin is implemented as a Marker with a Quickfix to correct the problem. Additionally a refactoring will be provided to enable mass conversion.

2.2.1. Marker

The plugin should detect missing, invalid and non-C++11 initializations and set a marker accordingly. This marker is then visible in the editor window and helps the developer to spot initialization errors and allows to trigger a matching quickfix.

2.2.2. Quickfix

The quickfix should allow for automatic correction of the detected problems. It can be invoked as usual in Eclipse by clicking on the marker or using the quickfix-shortcut.

The quickfix should honor the current text-selection, so if i.e. only a single variable is marked within a declaration list only that one should be refactored.

2.2.3. Reverse-Quickfix

To revert the quickfix operation a Quick-Assist should be provided to revert the operation. The reverse operation should not be implemented as quickfix as this would result in always a problem being detected (either the quickfix or the reverse-quickfix).

2.2.4. Mass-Refactoring

The mass-refactoring allows to automatically fix all initialization problems in a file or project, to allow efficiently fixing larger codebases.

2.2.5. Reverse-Refactoring

The reverse-refactoring allows to convert either individual or batch convert all C++11 initializations to non-C++11 initializations.

3. Implementation

3.1. Logical Structure

To implement the desired functionality, we use the project structure as described in the appendix. It consists of several Eclipse/Maven projects:

- One eclipse project contains the core functionality
- One eclipse project contains the unit tests
- An eclipse parent project to reference the other two projects

The core functionality is split up in one class for the checker and one for the quickfix.

3.2. Implementation Details

The plugin was implemented using the CDT-plugin framework for quickfixes and uses the Abstract Syntax Tree (AST)[1] provided by Codan[2], the static code analysis framework. Further information about CDT-plugin development can be found in A.2.1

3.2.1. Control Flow

The basic cotrol-flow goes the following:

1. The Checker traverses the AST of reports any problems found using `reportProblem()`
2. CDT places a marker in the editor for every reported problem, and allows the user to trigger the quickfix if `QuickFix.isApplicable()` returns true
3. If the quickfix gets triggered `QuickFix.modifyAst()` get's called which applies the quickfix by replacing the offending statement using `ASTRewrite.replace()`

[1]Wikipedia, *Abstract Syntax Tree — Wikipedia, The Free Encyclopedia*.
[2]wiki.eclipse.org, *Codan Static Analysis*.

3.2.2. Checker

Since we're primarily interested in declarators, the primary entry point for the checker is:

```
int visit(IASTDeclarator element)
```

The function consists of all cases which we explicitly want to avoid and should be skipped, and in the end the detection of the initializers which we want to detect and report a problem for.

According to the problem analysis the following declarations are explicitly ignored:

- function declarations
- parameter declarations
- already correctly initialized variables
- class members
- types with initializer-list constructors
- initializations using boost::assign
- initializations using implicit type conversions

Constructor init-lists aren't part of a declarator and are therefore individually handled in:

```
int visit(IASTInitializer initializer)
```

Detected problems for which a quickfix should be provided are reported using the `reportProblem()` function which takes the following arguments:

- String id: The problem id which is used in the plugin.xml to connect the quickfix to the problem.
- IASTNode node: The node where the problem was found, which is used for the marker position.
- Object [] args: Arguments which can be used to transport further information to the quickfix.

3.2.3. Quickfix

Before the quickfix is executed, the framework checks if the quickfix is applicable to the problem using:

```
bool isApplicable(IMarker marker)
```

This function should return true if the quickfix is applicable to the problem, or false if not.

The following function is called if a quickfix is executed:

```
void modifyAST(IIndex index, IMarker marker)
```

The index is used to retrieve the AST and the marker contains the position where a problem was detected.

Since the marker only contains the absolute position in the file, the target statement first needs to be retreived from the AST:

```
IASTNode targetStatement = getEnclosingDeclarator(astName);
```

Once the traget statement is identified, the quickfix can be applied to generate a replacement statement:

```
IASTNode newStatement = resolveConstructorInitializer(ast.getASTNodeFactory(),
    targetStatement);
```

While generating the replacement statement, it is important to copy as much as possible from the original statement, to avoid changing any unnecessary aspects of the original statement. This is generally achieved by copying the full declarator and then replacing the initializer only.

Finally the function applies the quickfix and modifies the AST using:

```
ASTRewrite r = ASTRewrite.create(ast);
r.replace(targetStatement, newStatement, null);
Change c = r.rewriteAST();
c.perform(new NullProgressMonitor());
```

This results in the code being effectively modified in the editor.

3.2.4. Mass conversion

The mass conversion features triggers all the markers found. This works similar to the CheckTestCase class which also has to invoke the quickfixes from markers in an editor.

First the markers as well as an instance of the quickfix are required.

```
IMarker[] markers = projectResource.findMarkers(
    IProblemReporter.GENERIC_CODE_ANALYSIS_MARKER_TYPE,
    true,
    2);
AbstractCodanCMarkerResolution quickFix = new QuickFix();
```

It is now possible to iterate over the array of markers and invoking the quickfix for each marker.

```
for (IMarker marker : markers) {
    quickFix.run(marker);
}
```

The difference between doing a mass refactoring for the current editor and the current project is the resource we need to obtain the markers from.

The current editor is accessible with the `getActivePage()` Method from the current workbench window. The editor also provides a way to get the project.

```
IWorkbench workbench = PlatformUI.getWorkbench();
IWorkbenchWindow workbenchWindow = workbench.getActiveWorkbenchWindow();
IWorkbenchPage workbenchPage = workbenchWindow.getActivePage();
IEditorPart editor = workbenchPage.getActiveEditor();
IEditorInput editorInput = editor.getEditorInput();
IResource editorResource =  ResourceUtil.getResource(editorInput);
IResource projectResource =  editorResource.getProject();
```

All dirty editors have to be saved:

```
editor.doSave(null);
PlatformUI.getWorkbench().saveAllEditors(false);
```

3.3. Unit Tests

There are two test cases. The first simply tests for placed markers to investigate the behavior of the Checker.java class. The other one focuses on testing all functionalty covered by the Quickfix.java class.

Care has been taken that each testcase only tests a single requirement and that the tests are a simple as possible.

The checker testcases cover all default conversions as well as well as all exceptions which are explicitly ignored as listed in 3.2.2. Further tests have been added for likely false-positives. Full path coverage is not targeted, there are as few tests as possible and as many as necessary.

More information about the unit-test framework can be found in A.2.5.

3.4. Review

The implementation can sometimes be a bit of a drag as much of the work consists of figuring out how to find a particular statement in the AST, which basically is a huge type hierarchy, so one usually has to cast the way through all the interfaces to finally get the statement one is looking for. Also since there exists such a mass of types in the tree it is often very difficult to figure out which statement actually conveys the information one is looking for. Therefore a lot of work has to be done using a testcase and the debugger to get what one wants, using lots of internal api and usually overly complex processes, which can then be cleaned-up in a second round.

3.4.1. Checker

While the code should be fairly easy to understand and extend, it is also very easy to break one of the special cases as they are all implemented in the same function as conditional return's, all cases affect each other directly. The test-driven development approach is therefore essential with all basic cases covered by the testcases.

Most parts are implemented cleanly, but the following implementations should be revised:

- Checker.getTargetTypeConstructors: Uses internal API due to an exception being thrown when using the interface.
- Checker.hasInitializerListConstructor: Relies on string comparison to detect the initializer_list type (which i.e. doesn't work with typedefs).

3.4.2. Quickfix

The quickfix implementation has to detect the problem case again (although the checker already did that once), which means there is some duplication. For this reason the ch.hsr.ifs.cdt.elevator.initializerlist.Problems class was meant as a way to bundle the problem information, so it could be used from both the checker and the quickfix parts. Since the duplication is marginal, the Problems class wasn't really used in the end though. This could be refactored to either use the Problems class properly, or remove it entierly.

3.4.3. Problems

- It is not possible to select multiple variables in the editor and apply the quickfix to those only, as this is not supported by the framework.

3.4.4. Reverse Quickfix

Due to the limited time available it was not possible to implement the reverse quickfix functionality. As there exists no prepared solution for Quickassists using CDT, this would have to be developed as well, and that would have been too much effort within the available time.

4. Conclusion

4.1. Project Review

While the problem we wanted to solve seemed fairly trivial at first, we quickly started to discover more and more cases where the default method of simply replacing any existing initializer with the brace-initializer does not work. This in turn required a deeper analysis of the C++code to be able to judge if a quickfix can be safely applied or not. Of course, as we tried to understand the C++code better, that opened new problems again; i.e. as you try to understand if two types are compatible you have to start inspecting the constructors of the target type, you have to deal with reference types, you have to deal with builtin types (which might or might not have a non-narrowing conversion), you have to deal with template types (which you first have to fully expand due to the use of `using namespace` and `namespace X = Y`, and so on. So in that respect the problem became a lot complexer than we expected initially, but at the same time also more interesting of course.

From a project management side, the project went fairly smooth. The regular meetings forced us to concentrate on individual tasks which we could accomplish, which in turn ensured we could put all our effort in accomplishing those tasks and not in finding out what there is to do. We could have used the bugtracker more rigorously, which would have improved the visibility of the current project status. Also, it wasn't ideal how we initially setup the bugtracker, with the traget versions misused as timeboxed milestones, and with a strong focus on the gantt-chart, which resulted in duplicating bugs, just for the sake of having it under every target milestone in the gantt-chart. That is of course not helpful for establishing the current state of the project and results in a lot of effort for maintaining the bugtracker. Once we settled for using just tickets with sub-tickets, that effort was reduced a lot, and helped us to get meaningful information from the bugtracker (e.g. what there is to do).

Overall the project was managed fairly efficiently, and we learned some points where we can improve in future projects.

The resulting plugin should do it's job well, and since it was developed with care that no valid code is broken, it should also not get in the way of the developer, which is of course very important for a plugin that aims to support. The plugin may still have some rough spots which would need to be ironed out, simply because it didn't receive any real world testing so far, and will first need to prove itself by being used productively.

Unfortunately we didn't manage to implement all the features we planned for originally, such the reverse refactoring, and the extract-lambda plugin.

4.2. Outlook

Further development could tackle the following problems:

Reverse refactoring To get C++11 code to compile with a non-C++11 compiler, it would be desirable to have a reverse refactoring, which transforms C++11 initializations to non-C++11-initializations. This refactoring should be applicable to single statements, as reverse operation of the quickfix, but also as mass-refactoring to fix larger code-bases.

Support for further C++11 features The plugin could provide support for more C++11 features, which are not limited to the new initializerlist syntax. The aim should be to help developers leveraging the new features of C++11 and to make it easier to modernize existing code-bases using this plugin.

Extract-lambda expressions plug-in The extract-lambda expression refactoring could be provided as a separate plugin or as part of elevator plugin.

4.3. Planning

The following figures show the planning as initially estimated, after a first revision in week 45, and how it ended up to be at the end of the project.

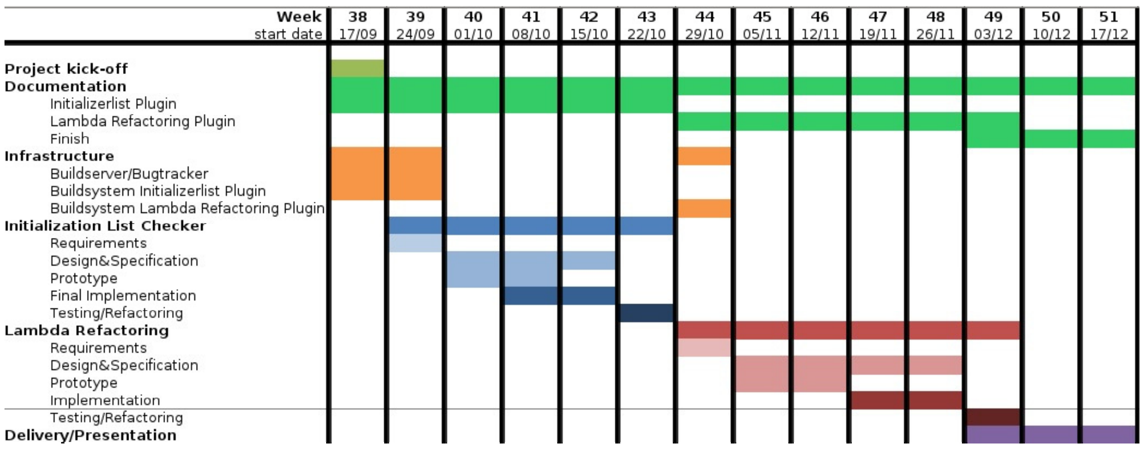


Figure 4.1.: Initial Planning

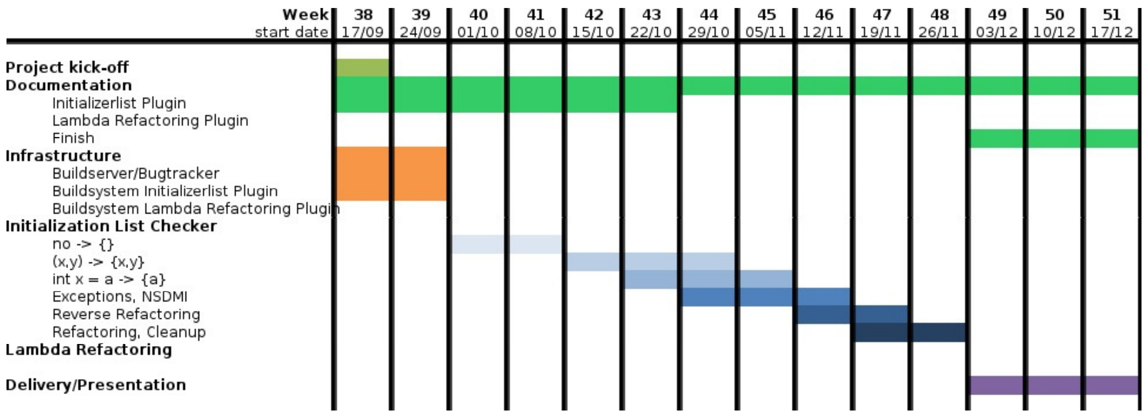


Figure 4.2.: First Iteration

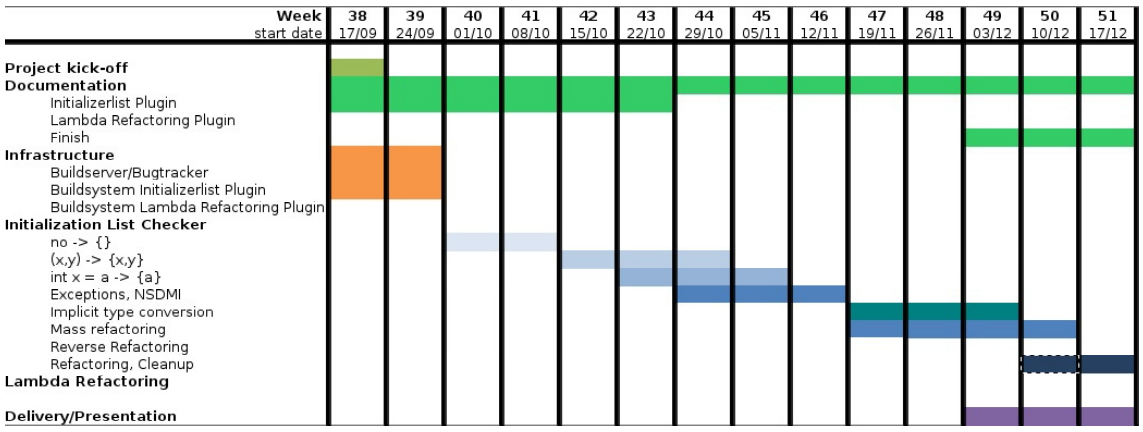


Figure 4.3.: Final Planning

The planning changed quite a bit over time as we found new requirements or removed planned features, and had to be adjusted accordingly.

4.3.1. Spent Time

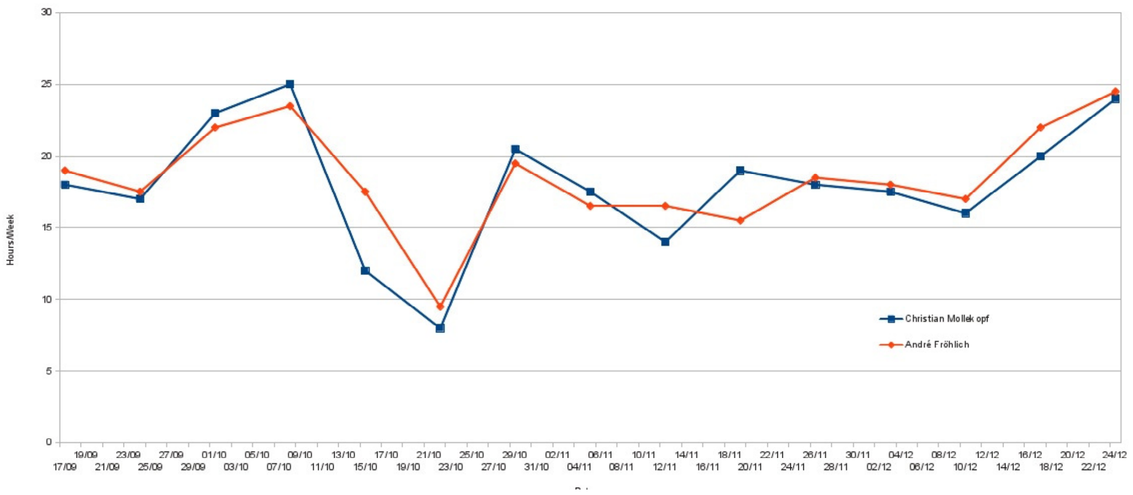


Figure 4.4.: Spent Time

The time we spent on the project fluctuated a bit around the 17.5h per week which was the expected effort. The initial peak was during the setup of the buildsystem, which caused some trouble, followed by a large drop caused by private responisbilities and followed by illness. The curves stabilized afterwards until the end of the project, where we had to give another boost to finish the documentation. The curves look very similar since we usually worked together on Monday and Tuesday.

4.4. André Fröhlich

Since I had not much experience with C++, I wanted to do this project. I was also a good way to refresh my knowledge from the C++module (Programmieren 3). It was also important to me to work on something that seemed to be useful in the end.

My professional background was to be a web-developer, so many things were new to me - at least the application of them in such a big project.

I had a little experience with latex and git and I was able to learn much using these tools so extensive during the project. I never used a continuous integration server, but it was pretty easy. However, as we were only two this would not have been necessary.

Redmine as a project management tool was not bad, but in hindsight I would have preferred something simpler and more agile. Maybe a little more like Github. The time tracking and project planning was pretty time consuming with Redmine.

Although I am convinced my Java programming experience is nod bad, I still had some respect for the Eclipse framework. At the same time it was a joy to be able to learn something new.

However, the framework was also a little tedious. It's incredibly large and therefore requires a lot of time in order to find your way. That is also the reason why the project has not progressed as quickly as I desired. Many problems that I thought they were difficult to solve were actually quite simple. On the other hand the framework complicated problems.

We often had to search on the internet to solve a particular problem or find out how something works. We also had to read the source to understand how to use the API.

Altough it was interesting to explore the framework, so it cost us too much time. I would have preferred to implement more features instead.

Either way, it was a very good experience and preparation for the bachelor thesis, and I enjoyed working with Christian.

4.5. Christian Mollekopf

Having worked most of my professional, private and academic life with C/C++ and even from that part a large part was spent with the Qt-Framework and tools like git and cmake, this project took me out of my comfort zone to face some new technologies.

This was my second real Java-based project, and the first time I worked with maven, the eclipse-framework, jenkins, and latex.

Maven proved to be somewhat of an entry barrier, mostly due to the lack of documentation how to build eclipse plugins using it. I appreciate the declarative aspect of the pom.xml files though and find it an interesting approach compared to the traditional Makefiles/Ant scripts. Another interesting aspect is that maven wants to provide more than just building, and even goes some steps to direct publishing, right from the same tool. All in all, a refreshing new take at buildsystems.

It was also nice to finally get to know jenkins a little closer, which does indeed a nice job as an automated buildsystem. I personally didn't use it a lot, as I anways run all the tests locally regularly, and with a two man team the gain isn't really there.

Latex provided about what I expected; a steep learning curve until we had figured out all the necessary quirks, but it was a breeze working with it afterwards. Especially due to the excellent interoperability with git because of the plaintext files. Also, the produced documentation really does looks fantastic compared to any word document or alike.

I wasn't overly impressed by the eclipse framework. I'm sure there are reasons why it is what it is, but I just don't like how eclipse seems to try to develop everything there is to an IDE in one gigantic framework, instead of using existing or creating new self-contained, reusable libraries. At least that was my first impression. The parts we've been working with worked well enough, but didn't really give the feeling of an overall concept and a good design, but rather as if they were just hacked together to get some job done. That feeling was probably also fueled by a lack of documentation and thus, also understanding.

Since a large part of the work consisted of searching through API's for the right function to call, to get the needed information, and not solving generally hard problems or writing a lot of code, I learned to appreciate some debugging tools eclipse and java have to offer. We also succeeded in using the test driven appraoch much more rigorously than I ever managed before, and I think that this really payed off.

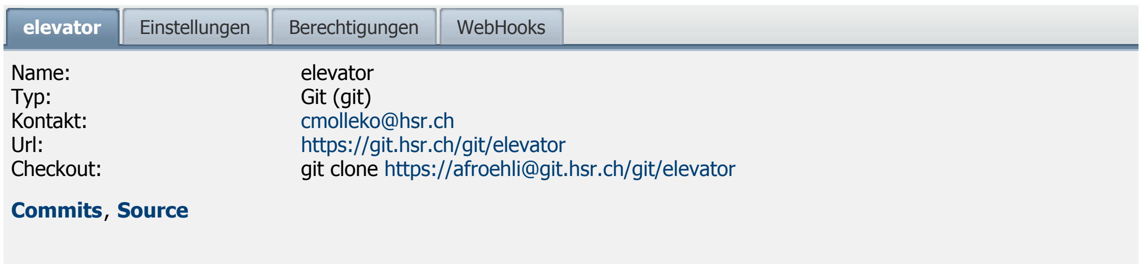
Overall, I think I learned quite a lot of things, even if not primarily in programming, but more in the realm surrounding it.

A. Appendix

A.1. Environment

A.1.1. Version Management

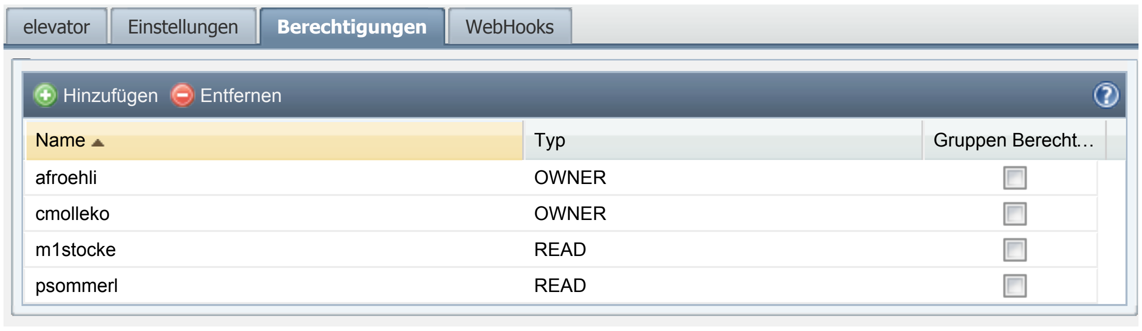
We decided to use Git as our verison control system. Although an installation on the provided server (sinv-56051.edu.hsr.ch) would be an easy task, using Ubuntu's apt-get command, we used the Git service on git.hsr.ch. In case this services turns out to be unreliable it is possible to migrate to an own Git installation with a minimum of effort.



Name:	elevator
Typ:	Git (git)
Kontakt:	cmolleko@hsr.ch
Url:	https://git.hsr.ch/git/elevator
Checkout:	git clone https://afroehli@git.hsr.ch/git/elevator

Commits, **Source**

Figure A.1.: Git Configuration Settings



Name ▲	Typ	Gruppen Berecht...
afroehli	OWNER	☐
cmolleko	OWNER	☐
m1stocke	READ	☐
psommerl	READ	☐

Figure A.2.: Access Configuration

We added read-only access for our advisors in addition to obvious access rules for us.

Although we are both familiar with Git and it now belongs to the school curriculum, we want to point out that there is an excellent free book[1] written by Scott Chacon available as a reference.

A.1.2. Build System

The buildsystem is based on maven and tycho.

[1]Chacon, *Pro Git*.

Maven

Apache Maven[2] is not really a traditional buildsystem, but rather a system to specify the project structure, in a way that enables automated builds. Unlike traditional Makefiles known from C/C++ development, or ANT files, which are just scripts executing a bunch of commands, maven pom-files specify projects modules, dependency repositories, and various plugins which serve a certain purpose, such as executing automated tests. That means the specification is much more declarative, in constrast to the imperative nature of traditional buildsystems.

Due to the high level of abstraction, it is fairly complex to understand what is happening and how to get started, but the resulting system is very powerful. The buildsystem we set-up, installs the complete build environment, including all required dependencies, runs all automated tests, creates an eclipse repository and builds the latex based documentation.

That means, all that is required to build the project on a new system is:

- Installing Git and Maven
- Checking out the git repository of the project
- Running "maven clean verify latex:latex" will clean any existing leftovers from previous builds, build both the Eclipse plug-in and the documentation (including downloading all required dependencies), and run all the unit-tests of the project.

This ensures the project can be built on any machine with a minimum of effort, and that a the builds are fully reproducible.

Tycho Tycho[3] is a maven plug-in providing integration for Eclipse plug-in projects. It extracts information from the existing eclipse spec-files (i.e. plugin.xml), and provides that meta-information to maven, so no duplication is required although two differen buildsystems are used.

Further information on setting up tycho can be found here[4].

To execute unittests, the tycho-surefire-plugin was used.

Documentation Since the documentation is written in latex, it is built as well using maven. To build the documentation, the maven-latex-plugin from www.codehaus.org[5] was used.

The documentation can be built using the maven "latex:latex" target.

[2]Apache-Foundation, *Apache Maven*.
[3]eclipse.org, *Tycho*.
[4]Vogel, *Tycho tutorial*.
[5]codehaus.org, *Latex Maven Mojo*.

Eclipse-integration While the maven-eclipse integration works well based on the tycho plugin, the other way is barely existing. While there are maven integration plugins, eclipse doesn't really internally use maven to build the project, but uses it's own build-tools anyways. All the maven integration tools do, is allowing to schedule a maven build from eclipse, but since the result isn't used by eclipse afterwards, that functionality is hardly useful.

A.1.3. Project Management

For collaboration we decided to use Redmine. Redmine is a free and open source, web-based project management and bug-tracking tool[6].

The installation process was fairly simple. Using the apt-get command we installed the redmine as well as the apache2 and the libapache2-mod-passenger packages. The later is an apache module for supporting Ruby on Rails on which Redmine is built.

```
sudo apt-get install redmine libapache2-mod-passenger
sudo ln -s /usr/share/redmine/public/ /var/www/redmine
sudo nano -w /etc/apache2/mods-available/passenger.conf
```

```
PassengerDefaultUser www-data
```

```
sudo nano -w /etc/apache2/sites-available/default
```

```
<Directory /var/www/redmine>
    RailsBaseURI /redmine
    PassengerResolveSymlinksInDocumentRoot on
</Directory>
```

```
sudo service apache2 restart
```

A.1.4. Continuous Integration

For continuous integration we decided to use Jenkins, a popular and extendible server. Jenkins only needs a Java Servlet container to run on.

For our project we set up two jobs. One for building the plug-in and the second for building the documentation.

Both jobs are configured to grep the source code from our git repository and build it with maven. The jobs are triggered to be run once an hour if a change in the repository was made.

[6]Wikipedia, *Redmine — Wikipedia, The Free Encyclopedia*.

With maven it was possible to build as well as test the plug-in using Jenkins.

To get the source from the git repository we used the Git Jenkins plug-in. Because the tests require an X server for running Eclipse the Xvfb was also used. To build the documentation with LaTeX no special Jenkins plug-in was needed as we used maven to build the documentation.

A.1.5. Documentation

The documentation was written using LaTeX. It is recommended to use the texlive LaTeX distribution, which was used in the version 2012.

The following wikibook[7] contains a lot of helpful tipps for working with latex.

A.2. Eclipse Framework

To implement the C++ refactoring plugin, "Codan", the static analysis framework of CDT, has been used.

For providing Quickfixes, two major parts are required. First detecting problems and setting an according marker with a provided quickfix option, and second applying the actual quickfix.

A.2.1. CDT Plugin interface

A CDT plugin has the following required parts:

- plugin.xml: Describes and defines the used extension points.
- Activator implementing the AbstractUIPlugin interface: Defines a PLUGIN_ID. The Activator is referenced in the MANIFEST.MF as Bundle-Activator, and controls the plugin lifecycle.

Used Extension Points as defined in plugin.xml:

- org.eclipse.cdt.codan.core.checkers: Defines a checker which can be used to detect problems and place a marker. The extension point is also used to define specific problems.
- org.eclipse.cdt.codan.ui.codanMarkerResolution: Defines a resolution for a problem (Quickfix).
- org.eclipse.cdt.codan.ui.codanProblemDetails: Allows to supply additional information about the found problem.
- org.eclipse.core.resources.markers: Allows to define custom markers.

Further information on eclipse plugin development can be found in the tutorials here[8].

[7]WikiBooks, *LaTeX — Wikibooks, Open books for an open world*.
[8]Vogel, *Eclipse tutorials*.

A.2.2. AST

The following is a list of the most important AST nodes:

- IASTDeclarator: A Declarator node (a single variable with a type and initializer).
- IASTInitializer: An initializer node.
- IASTExpression: Contains type information.
- ICPPConstructor: A constructor with arguments.
- ICPPClassType: Allows to retrieve constructors of a type (including implicit constructors).
- IType: Represents a type, which can be either a builtin type or also a custom type.
- IASTName: Represents a name in the program such as a variable name.
- IBinding: The binding represents the semantics of a name. It can i.e. be used to get the fully expanded type of a name, which uses typedefs for the type specification.

While there are many more, these are the most important ones used throughout the code, so it is vital to understand all of them. Most of them have several specialized subclasses, so it is always a good idea to have a look at the type hierarchy should a new class appear while writing/debugging code.

A.2.3. AbstractIndexAstChecker

The org.eclipse.cdt.codan.core.cxx.model.AbstractIndexAstChecker is a visitor[9], which traverses the Abstract Syntax Tree (AST), everytime CDT's parser detects a modfication. The tree is traversed top-down, node by node, using the processAst() member function. The return value of processAST() allows the implementation to decide if it wants to go any deeper in the tree or stop at the current node. Using the reportProblem() function, found problems can be reported (using the problem ID as specified in plugin.xml), which results in a marker being placed in the editor.

A.2.4. AbstractAstRewriteQuickfix

The org.eclipse.cdt.codan.ui.AbstractAstRewriteQuickFix interface requires essentially two functions to be implemented for a quickfix:

- isApplicable() gets a marker as argument and should return true if the quickfix is applicable to the reported problem.
- modifyAST() receives a marker and the current project index, to apply the actual quickfix by modifying the AST.

Since problems are reported per declarator, the quickfix also only operates on single declarators (and not complete declarations). This ensures that i.e. variables in declaration-lists are processed individually.

[9]Wikipedia, *Visitor Pattern — Wikipedia, The Free Encyclopedia*.

A.2.5. Testing

The unittests are in a separate project to avoid dependencies from the plugin to test-libraries. The test is implemented as a separate plugin, which is loaded into an eclipse instance and runs all the tests. For checker test-cases there is the org.eclipse.cdt.codan.core.test.CheckerTestCa interface, for QuickfixTests there exists no official equivalent though. Therefore the QuickFixTestCase class was copied from the internal API of codan.

The InitializerCheckerTest contains all tests for the checker part, while the InitializerQuickfixTest contains the tests for the quckfix. Both require all tests to be prefixed with "test" and execute all tests in a random order.

Checker Tests Checker Tests allow the specification of a codefragment in a comment above the testfunction, which is then loaded into the editor when the test is executed. One can then assert in which line an error should be detected.

An example test looks like this:

```
// class T;
// T x;
public void testFindUninitializedType() {
    loadCodeAndRunCpp(getAboveComment());
    checkErrorLine(2);
}
```

This test ensures that an error is detected on the second line of the comment, upon `T x;`, which is an uninitialized variable.

Similarly it is possible to check that no error has occured:

```
// int foo();
public void testIgnoreFunctionDecl() {
    loadCodeAndRunCpp(getAboveComment());
    checkNoErrorsOfKind("elevator.initializerlist");
}
```

By using `checkNoErrorsOfKind("elevator.initializerlist");` instead of `checkNoErrors();`, one can circumvent problems due to some built-in plugins reporting code-issues, such as an error being reported regarding a declared but never defined function foo() here.

Quickfix Tests The Quickfix test also loads a codefragment from the comment above the testfunction, and then runs the Quickfix on that codefragment. The assert checks if the expected transformation is present in the result string.

An example test looks like this:

```
// class T{};
// T x;
public void testUninitializedClass() throws Exception {
    loadcode(getAboveComment());
    String result = runQuickFixOneFile();
```

```
        assertContainedIn("T x { };", result);
    }
```

This test ensures that the result contains the string `T x ;`.

Running the Tests

The tests should be executed as ”JUnit Plug-in Test” using JUnit4. It is possible to either execute the whole testproject, but also individual test-classes or even individual test-functions.

Problems

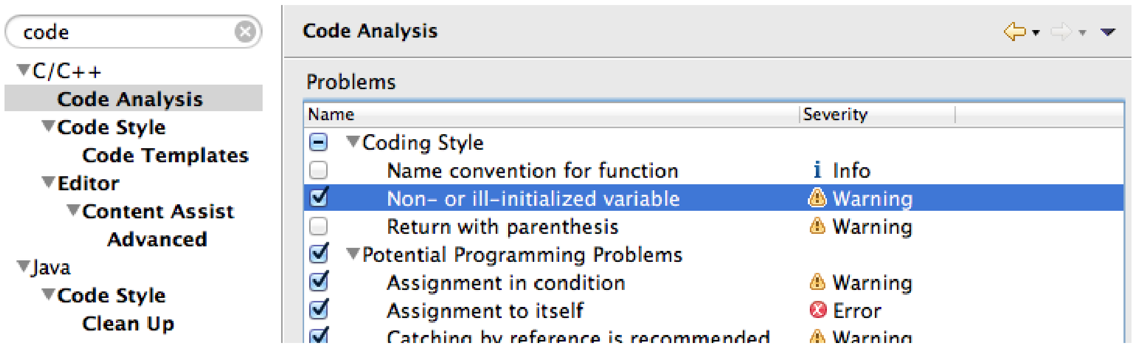
- It is not always guaranteed that a unittest behaves exactly the same as if an eclipse instance was started and a certain feature tested manually. This is because some parts are not covered by the unittests, such as the isApplicable function of the Quickfix.
- Note that the Quickfix tests depend on the formatting setting in eclipse as it is a simple string comparison including whitespace.
- The Quickfix-test depends on a startup delay to ensure the plugin is loaded. Depending on the hostcomputer this delay can be too small, resulting in failing tests. The delay can be adjusted in QuickFixTestCase.runQuickFixOneFile (i.e. to 1500).

A.3. User Manual

The Elevator plug-in for Eclipse CDT gives you the ability to detect and convert uninitialized and initialized varbiables to the new C++11 style brace initialization syntax.

A.3.1. Installation

This manual asumes that you are familar with Eclipse and Eclipse plug-in installation. To install the Elevator plug-in simply add the update site and install the plug-in. After the installation was succesful, you'll find under "Code Analysis" in the Eclipse preferences window a new problem item called "Non- or ill-initialized variable".

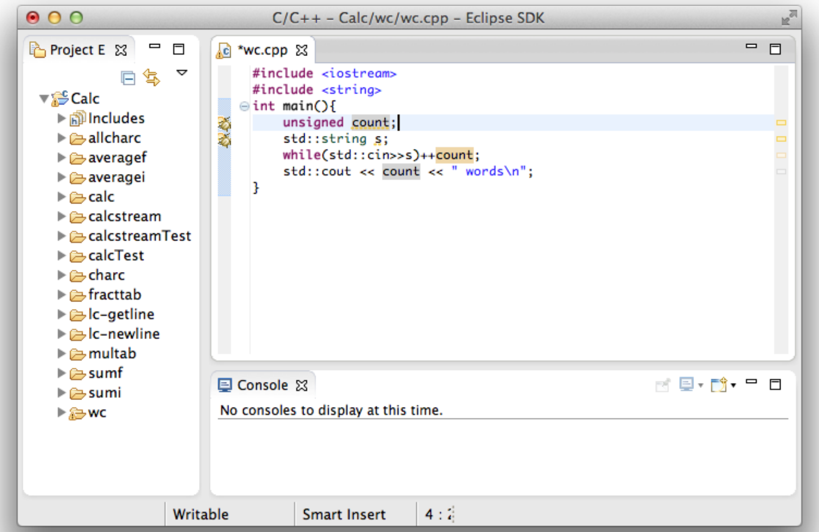


Problems

	Name	Severity
☐	Coding Style	
☐	Name convention for function	Info
✓	Non- or ill-initialized variable	Warning
☐	Return with parenthesis	Warning
✓	Potential Programming Problems	
✓	Assignment in condition	Warning
✓	Assignment to itself	Error
✓	Catching by reference is recommended	Warning

Figure A.3.: The code analysis preference page

A.3.2. Detection



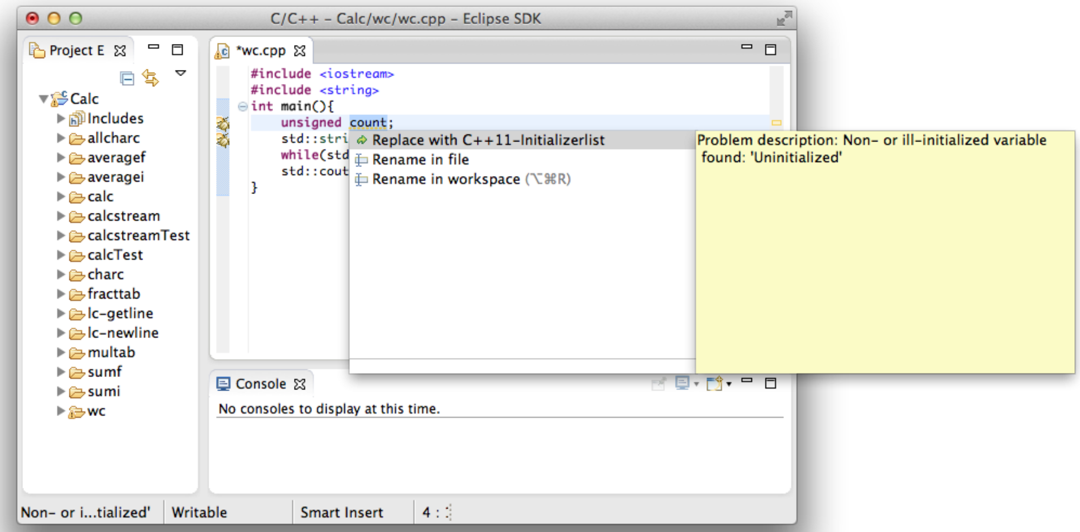
```
#include <iostream>
#include <string>
int main(){
    unsigned count;
    std::string s;
    while(std::cin>>s)++count;
    std::cout << count << " words\n";
}
```

No consoles to display at this time.

Figure A.4.: An open file with detected issues

The plug-in automaticaly detects initialization issues and marks them in the editor.

A.3.3. Conversion



```
#include <iostream>
#include <string>
int main(){
    unsigned count;
    std::stri
    while(std
    std::cout
}
```

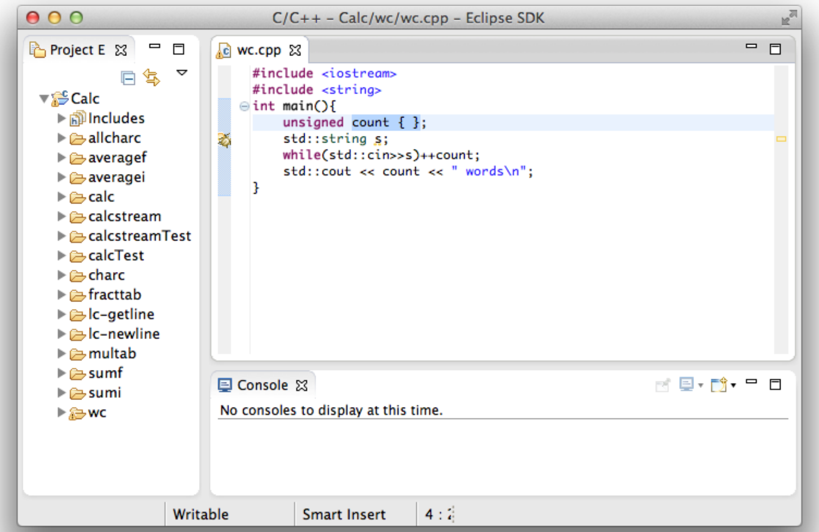
Problem description: Non- or ill-initialized variable found: 'Uninitialized'

No consoles to display at this time.

Non- or i...tialized'

Figure A.5.: Invoking a quickfix

To convert a detected issue, you can click on the icon on the side and chose the provided quickfix to convert the issue.



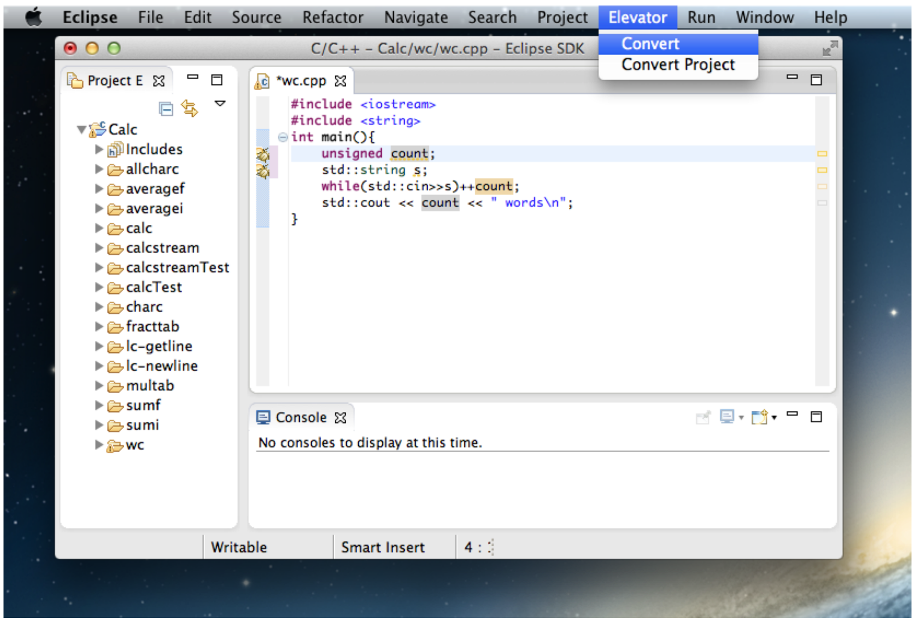
```
#include <iostream>
#include <string>
int main(){
    unsigned count { };
    std::string s;
    while(std::cin>>s)++count;
    std::cout << count << " words\n";
}
```

No consoles to display at this time.

Figure A.6.: The file after the conversion of the first issue

A.3.4. Mass Conversion

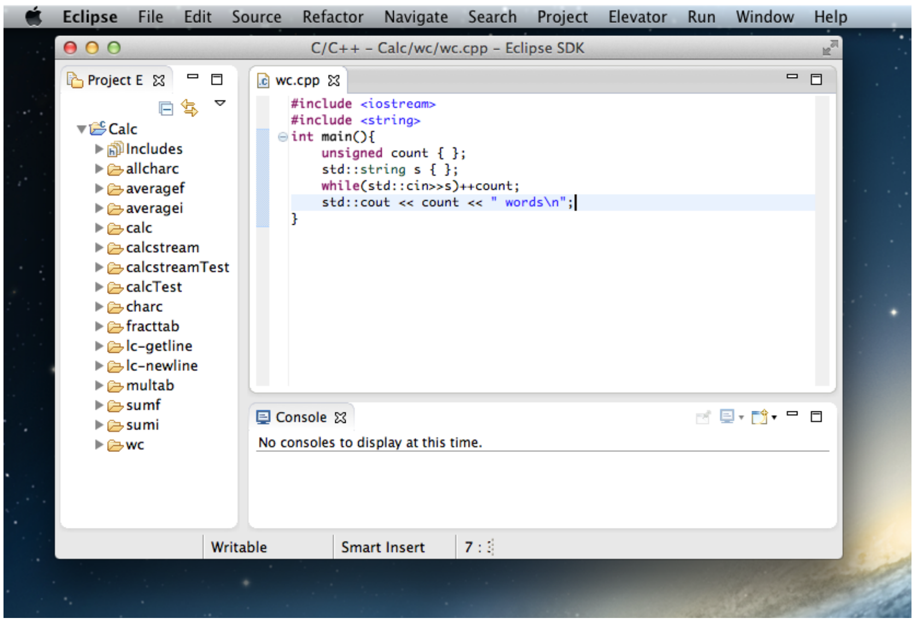
It is possible to convert all issues at once. To accomplish this simply choose "Convert" from the "Elevator" menu.



```
#include <iostream>
#include <string>
int main(){
    unsigned count;
    std::string s;
    while(std::cin>>s)++count;
    std::cout << count << " words\n";
}
```

No consoles to display at this time.

Figure A.7.: Convert all issues of the current editor



```
#include <iostream>
#include <string>
int main(){
    unsigned count { };
    std::string s { };
    while(std::cin>>s)++count;
    std::cout << count << " words\n";
}
```

No consoles to display at this time.

Figure A.8.: All initialization issues are gone

To convert all issues of the current project, choose "Convert Project" from the "Elevator" menu.

B. Glossary

Name	Description
Declaration	A specification how a name should be interpreted: int f(int x)
Declarator	A single variable, function or type within a declaration: int x
Initializer	A specification of the initial value of a declaration
initializer-list-constructor	Constructor using the std::initializer_list template
nullptr	A Null-Pointer
target type	Left-hand side type in assignment
source type	Left-hand side type in assignment
narrowing conversion	conversion with precision loss (int x = 3.2; //=¿ x = 3)
Conversion constructor	Constructor taking single argument of source type, allowing a non-narrowing conversion
AST	Abstract Syntax Tree[1]

Table B.1.: C++

Name	Description
Not Initialized	int x;
List-initialization	Initialization from a braced-init-list ({3,4,3})
Constructor-Initializer (direct-initialization)	int x(3);
Equal-Initializer (copy-initialization)	int x = 3;
Brace-Initializer	int x{3};
Constructor init-list	C(): a(1), b(2), c3;

Table B.2.: Initializations

Bibliography

Apache-Foundation. *Apache Maven.* [Online; accessed 13-11-2012]. 2012. URL: http://maven.apache.org.

Chacon, Scott. *Pro Git.* 1st. Berkely, CA, USA: Apress, 2009. ISBN: 978-1430218333. URL: http://git-scm.com/book/.

codehaus.org. *Latex Maven Mojo.* [Online; accessed 13-11-2012]. 2012. URL: http://mojo.codehaus.org/latex-maven-plugin/.

eclipse.org. *Tycho.* [Online; accessed 13-11-2012]. 2012. URL: http://eclipse.org/tycho/.

N3337, ISO/IEC. *Working Draft, Standard for Programming Language C++.* Geneva, Switzerland: ISO/IEC, 2012. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf.

Vogel, Lars. *Eclipse tutorials.* [Online; accessed 13-11-2012]. 2012. URL: http://www.vogella.com/eclipse.html.

— *Tycho tutorial.* [Online; accessed 13-11-2012]. 2012. URL: http://www.vogella.com/articles/EclipseTycho/article.html.

WikiBooks. *LaTeX — Wikibooks, Open books for an open world.* [Online; accessed 13-11-2012]. 2012. URL: http://en.wikibooks.org/wiki/LaTeX.

wiki.eclipse.org. *Codan Static Analysis.* [Online; accessed 13-11-2012]. 2012. URL: http://wiki.eclipse.org/CDT/designs/StaticAnalysis.

Wikipedia. *Abstract Syntax Tree — Wikipedia, The Free Encyclopedia.* [Online; accessed 13-11-2012]. 2012. URL: http://en.wikipedia.org/wiki/Abstract_syntax_tree.

— *Redmine — Wikipedia, The Free Encyclopedia.* [Online; accessed 13-11-2012]. 2012. URL: http://en.wikipedia.org/w/index.php?title=Redmine&oldid=521828609.

— *Visitor Pattern — Wikipedia, The Free Encyclopedia.* [Online; accessed 13-11-2012]. 2012. URL: http://en.wikipedia.org/wiki/Visitor_pattern.