HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

bachelor thesis

Test Driven Development for Eclipse CDT

Thomas Kallenberg, Martin Schwab

June 17, 2011

Supervisor: Prof. Peter Sommerlad

In 2006, Prof. Peter Sommerlad introduced the CUTE unit testing framework for C++ to promote unit testing in C++ [ISO11]. The objective of this bachelor thesis is to ease practice of Test-Driven Development (TDD) by providing code generation in Eclipse CDT when using CUTE. Referring to an undefined entity when writing code results in a parse error. Based on such an error, a resolution should be provided that generates the missing code defining the entity. Code generation features three advantages: It speeds up the TDD workflow, allows focus on the actual program logic and is useful for C++ programmers using CDT even when not practicing TDD.

Throughout this bachelor thesis, the Eclipse development environment for C++ has been extended by a plug-in that provides automated code generators. Twelve easy to use quick fixes have been developed, along with a mechanism to extract definitions to separate files. In addition, to provide dynamic labels and more specific problem markers, static code checkers have been extended. Until recently, CDT's problem marker have only been used to indicate errors in source code. Combined with the developed plug-in, they may be used to generate missing code blocks, too, making Eclipse CDT and unit testing for C++ more attractive.

As the C++ language has a lot of features, more special cases have probably to be regarded to make acceptance into CDT or CUTE possible. A further step for improving TDD support in CDT is to develop a *Move Refactoring* which allows to move generated code to another file.

Executive Summary

Problem

Test-Driven Development aims to lead to better software. First write a test, then add the actual functionality, refactor and integrate. This process benefits of at least two advantages. First, new code is not implemented speculatively and second, the written software is tested right away.

Modern programming languages and their Integrated Development Environments (IDE) grew up with these principles. However, long established languages like C++ and most of their IDEs are not designed to provide support for TDD.

In 2006 Peter Sommerlad introduced a Unit testing framework for C++ called CUTE. Together with the integration in Eclipse CDT, an IDE for C++, one big gap towards fundamental support of TDD for C++ was closed.

```
#include "cute.h"

void testSquareArea() {
  ASSERT_EQUALS(16, MySquare(4).getArea());
}
```

Listing 1: Testing C++ code with CUTE

However, most code still has to be written manually – a possible source of errors.

Now if the parser is able to complain about missing code, why not just complete it? This could be dangerous as there are decision to be taken deliberately by a programmer. Such a decision could include which constructor should be chosen if there are multiple ones or do not take basic types as arguments. However, partial automation is possible without affecting correctness.

```
int testDivide() {
  int i = divide(10, 2);
}
```

Listing 2: The interface of `divide()` is defined by calling the function

In Eclipse CDT it is possible to automatically complete function calls with autocomplete if the function is declared. However, generating code such as a function signature matching a function call is not yet possible.

If such code could be generated in a fast way, a test could be written and all missing parts could be generated. The programmer could fully concentrate on the program logic.

```
#include "cute.h"

int divide(int, int) {
  return int();
}

void testDivide() {
  ASSERT_EQUALS(5, divide(10, 2));
}
```

Listing 3: Generated function `divide()` with default return statement

Solution

To fix missing code in a fast and correct manner, the plug-in relies on source code problem markers Quick Fixes and linked mode editing.

As shown in figure 0.1 markers are hints displayed on the left side of the source code editor.

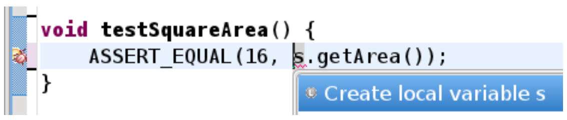
Quick Fixes are actions bound to a marker which makes corrections to the source code without a dialog and the changes are displayed in the same source editor providing the possibility to go on immediately.

Linked mode, as shown in figure 0.6 is a feature of Eclipse to edit multiple positions simultaneously. Editing the return type of a function will produce the same text after the return statement in the function body.

The source code is analyzed instantly by the Eclipse CDT parser while typing and different kinds of problems are detected. Such problems are displayed as markers. At any time, the programmer may hit a shortcut for additional information about the problem of such a marker and various actions to correct them in a fast and simple manner are presented. These corrections, called Quick Fixes, start refactorings. They analyze source code around the marker and generate code to make the marker and the problem disappear. In addition, linked mode is started to edit names and options after the change has been completed. On the following pages the modifications with examples are presented which support TDD in the plug-in.

Create Local Variable

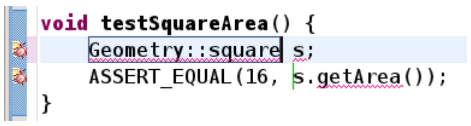
In a first step, a local variable is created. Analyzing the code does not return an informative result since it cannot be decided what kind of type variable `s` is.



```
void testSquareArea() {
    ASSERT_EQUAL(16, s.getArea());
}
```

Figure 0.1.: Local variable `s` is missing

After creation of the local variable the type of `s` can be changed to `Geometry::square`.

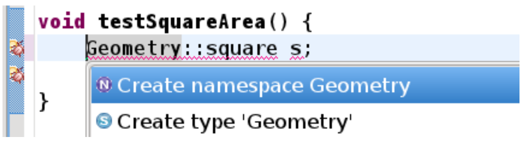


```
void testSquareArea() {
    Geometry::square s;
    ASSERT_EQUAL(16, s.getArea());
}
```

Figure 0.2.: Local variable `s` has been declared

Creating Namespace

The first marker indicates that the namespace `Geometry` could not be found. To resolve this, the Quick Fix to create a namespace is used.

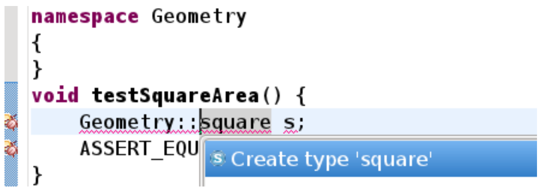


```
void testSquareArea() {
    Geometry::square s;
}
```

Figure 0.3.: Namespace `Geometry` needs to be created

Creating Types

The next marker indicates that the type `square` does not yet exist. Using the *Create new Type* Quick Fix will create type `Square` in namespace `Geometry`.

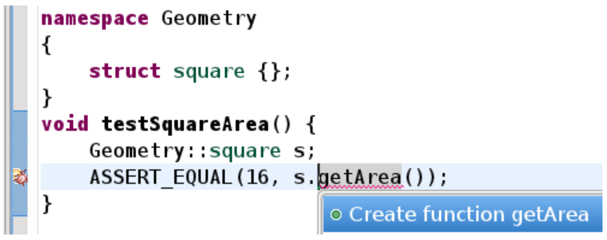


```
namespace Geometry
{
}
void testSquareArea() {
    Geometry::square s;
    ASSERT_EQU
}
```

Figure 0.4.: Type `square` in namespace `Geometry` needs to be created

Creating (Member) Functions

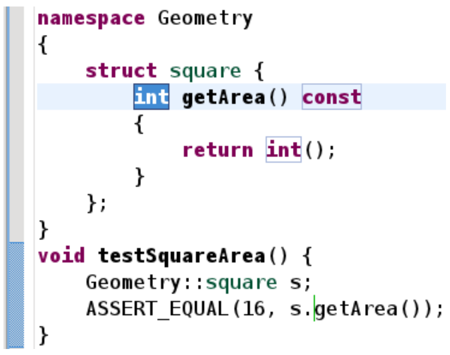
The member function `getArea()` is still missing. Quick Fix *Create Member Function* will create this function type and will return the default constructed type of the return type. To determine the necessary return type, the called `ASSERT` macro is analyzed and the appropriate type is extracted.



```
namespace Geometry
{
    struct square {};
}
void testSquareArea() {
    Geometry::square s;
    ASSERT_EQUAL(16, s.getArea());
}
```

Figure 0.5.: Member function `getArea()` needs to be created

After creating the function, various options such as `const` or the return type can be edited with linked mode.

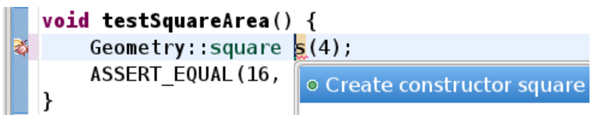


```
namespace Geometry
{
    struct square {
        int getArea() const
        {
            return int();
        }
    };
}
void testSquareArea() {
    Geometry::square s;
    ASSERT_EQUAL(16, s.getArea());
}
```

Figure 0.6.: Editing function with linked mode

Creating Constructors

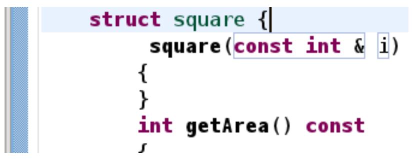
After `square` has been inserted, more code is still missing. Preferably, the length of a side should be passed in via a constructor. Changing the default constructor call and adding an argument for the side length is done manually.



```
void testSquareArea() {
    Geometry::square s(4);
    ASSERT_EQUAL(16,
}
```

Figure 0.7.: Changing the constructor call

A new marker is displayed informing the user that there is a call to a constructor that does not exist. Using the *Create Constructor* Quick Fix will generate the needed constructor.

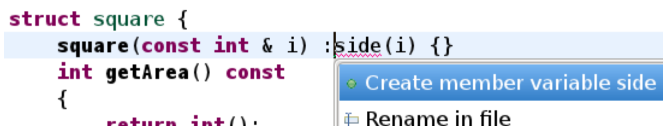


```
    struct square {
        square(const int & i)
        {
        }
        int getArea() const
```

Figure 0.8.: Newly created constructor call

Creating Member Variables

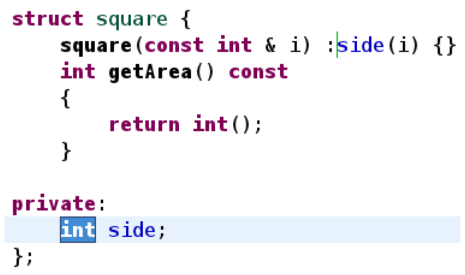
The passed in variable `i` needs to be assigned to a member variable. Adding an initializer list will trigger a new marker indicating that the name `side` does not exist.



```
struct square {
    square(const int & i) :side(i) {}
    int getArea() const
    {
```

Figure 0.9.: Create missing member variable

The *Create Member Variable* solves this problem by creating a privately declared member variable in the struct.

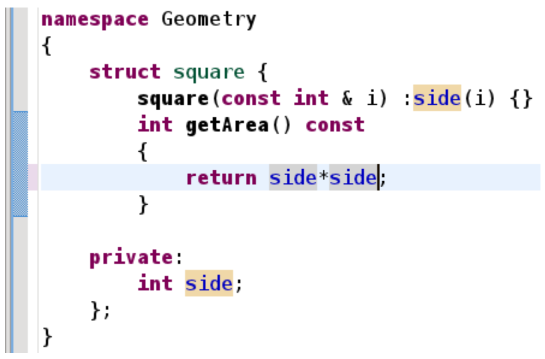


```
struct square {
    square(const int & i) :side(i) {}
    int getArea() const
    {
        return int();
    }

private:
    int side;
};
```

Figure 0.10.: Editing the type of `side` with linked mode

Implementing the Functionality

The actual program logic is implemented manually: `getArea()` must return the square of `side`.



```
namespace Geometry
{
    struct square {
        square(const int & i) :side(i) {}
        int getArea() const
        {
            return side*side;
        }

    private:
        int side;
    };
}
```

Figure 0.11.: Implementing the functionality

How to Use

All these features are provided as one installable plug-in together with CUTE.

Limitations and Outlook

The developed features are mainly designed for use with creation of unit tests. For other purposes, it is possible that support for more language constructs has to be added. Furthermore, static code analysis is a tradeoff between complexity and speed. As checkers become more complex, code analysis will become slower. In future, for large projects, less useful checkers will have to be turned off to ensure responsiveness of error markers.

In a next step, this plug-in could be bundled with CUTE which is simple because of the plug-in architecture. At a later time, the new features may be proposed to the CDT community. Anyway, Quick Fixes for code generation are a hot topic and new features will be added to CDT in the near future.

Thanks

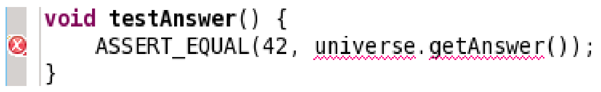
We would like to thank our "Eclipse team" at the Institute For Software IFS for supporting us with the development of this plug-in. First thanks goes to Prof. Peter Sommerlad for supervising this project. Emanuel Graf integrated our semester thesis into CDT and helped with architectural issues. Thanks also to Lukas Felber for having a solution for almost every Eclipse issue, especially in case of tests. Other team members helped with feedback on documentation and a lot of humor.

Another big thanks goes to our families and friends who also reviewed documentation and were missed out a little bit during our bachelor thesis.

Contents

1. Introduction

For this bachelor thesis, a plug-in is developed that supports the development cycle of *Test-Driven Development*: *Red – Green – Refactor* [Bec03]. Basically, tests are written before the actual code exists. Following the parse errors, code is inserted to make the code compile. Yet the unit tests stay red.



```
void testAnswer() {
    ASSERT_EQUAL(42, universe.getAnswer());
}
```

Figure 1.1.: example test with two parse errors about missing code

Now if the parser is able to complain about missing code, why does it not complete it? This could be dangerous as there are decisions to be taken deliberately by a programmer. For example choosing one of multiple constructors or constructors with a non-basic type as argument. However, partial automation is possible without affecting correctness.

Automation is the subject of this thesis. It is objected to provide a tool to C++ programmers which helps generating such missing code. The tool should integrate into Eclipse[Ecl11a] CDT [Bau10], an extensible and industrial strength development environment for C++.

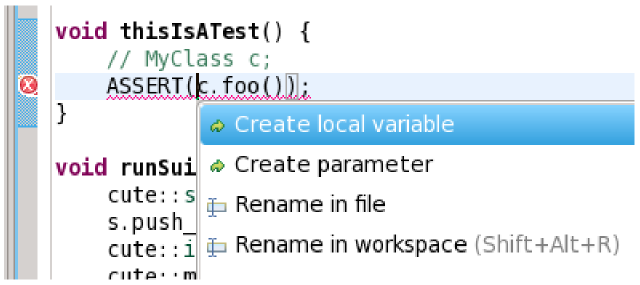
Readers are guided through this documentation by sample code written for the CUTE[Som11b] unit testing framework. Throughout the project it is used as reference test environment. Nevertheless, the plug-in that is being developed should also work with other C++ test frameworks.

1.1. Current Situation

Various C++ unit testing frameworks exist for Eclipse CDT, providing mechanisms to generate new test suites and test cases. However, to get really into the TDD workflow, a lot of code still has to be written by hand.

The good news is that missing definitions are already detected [Ecl10a] by code analysis (codan[Ecl11g]). Codan checks the source code for errors during typing. The product of Codan are markers on the left site of the editor which indicate possible errors. Such a marker may be associated with a so called Quick Fix [Eclb] which provides the user with a solution to solve the particular problem.

Although Codan existing for a while, there are not yet many Quick Fixes to solve problems. In most cases no help is offered to generate missing code. In summer 2010, functionality for creating a local variable[Ecl10b] has been added to the CDT repository[1], so a first step has already been taken.



```
void thisIsATest() {
    // MyClass c;
    ASSERT(c.foo());
}

void runSui
    cute::s
    s.push_
    cute::i
```

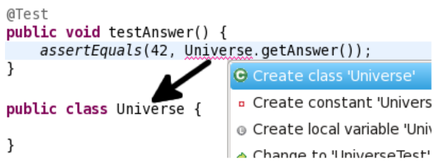
Figure 1.2.: generating a local variable in CDT

1.1.1. Peeking inside Eclipse JDT

Eclipse JDT offers a set of tools for TDD. Figure 1.3 visualizes how code automation helps writing a JUnit [JUN11] test method in Java.

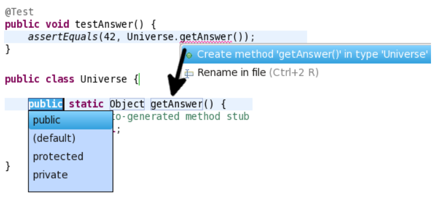
However, in the authors' views, JDT has some usability flaws such as overloaded user dialogs. In addition there is no guarantee that a solution in Java is also a good way to solve problems in

[1]since v201008041021



```
@Test
public void testAnswer() {
    assertEquals(42, Universe.getAnswer());
}

public class Universe {

}
```

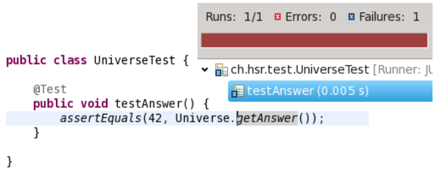
(a) Parser complains about missing code. Generate new class `Universe`.



```
@Test
public void testAnswer() {
    assertEquals(42, Universe.getAnswer());
}

public class Universe {

    public static Object getAnswer() {
            o-generated method stub
            ;

}
```

Create method 'getAnswer()' in type 'Universe'
Rename in file (Ctrl+2 R)
public
(default)
protected
private

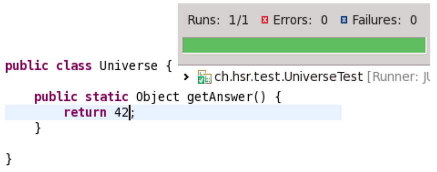
(b) Generate empty method `getAnswer()` which returns zero.



```
public class UniverseTest {

    @Test
    public void testAnswer() {
        assertEquals(42, Universe.getAnswer());
    }

}
```

```
Runs: 1/1   Errors: 0   Failures: 1
ch.hsr.test.UniverseTest [Runner: JU
testAnswer (0.005 s)
```

(c) Test fails because the created method returns zero.



```
public class Universe {

    public static Object getAnswer() {
        return 42;
    }

}
```

```
Runs: 1/1   Errors: 0   Failures: 0
ch.hsr.test.UniverseTest [Runner: JU
```

(d) Tests go green after changing the default return value to `42`.

Figure 1.3.: TDD workflow realized in Eclipse JDT (Java)

C++. Furthermore, introducing the Java way of thinking should be avoided for CDT features. It is intended to implement similar functionality as JDT with best possible usability.

1.2. Motivation

Automating frequent programming steps is helpful. Programmers should be able to focus on their task instead of typing code all the time. There are several reasons why TDD should have better support in Eclipse. First, it makes TDD, Eclipse CDT and CUTE tests more attractive. Second, the planned functionality is helpful for other tasks too, so every C++ programmer who uses CDT may profit. And in the end, working on a large project like Eclipse is an interesting challenge.

1.3. Basic Workflow Example

At the beginning of TDD there is a test suite. After creating a suite, a new test function is added and filled with an `ASSERT` statement. It specifies the expected result of the future code.

Listing 1.1 shows an example of a CUTE test function which does not compile as the symbol `square` is undeclared.

```cpp
1  #include "cute.h"
2  
3  void testSimpleSquare() {
4      ASSERT_EQUAL(4, square(2));
5  }
6  
7  void runSuite(){
8          cute::suite s;
9          s.push_back(CUTE(testSimpleSquare));
10         cute::ide_listener lis;
11         cute::makeRunner(lis)(s, "The Suite");
12 }
13 
14 int main(){
15     runSuite();
16     return 0;
17 }
```

Listing 1.1: Symbol `square` inside `testSimpleSquare` is missing

In subsequent code listings, repeating test framework code (lines 1-3 and 7-17 in listing 1.1) will be omitted. Starting at the `testSimpleSquare` test function, examples are illustrating the convenience of code generation during test writing.

1.3.1. Generate a Free Function

First, a test is written and the expected result is defined.

```cpp
void thisIsATest() {
    ASSERT_EQUAL(4, square(2));
}
```

Listing 1.2: defining the designated functionality

The missing function is generated by Eclipse above the test function.

```cpp
int square(int i) {
  return int();
}
```

```

void thisIsATest() {
    ASSERT_EQUAL(4, square(2));
}
```

Listing 1.3: generated function `square` with default return value

The compiler now accepts the code and the tests turn red caused by the default return value of *square()*. It may now be replaced by the square of the parameter.

Make the Tests go Green

The last example leaves the programmer with compiling code and the tests go red because of *square()* returning the default value of zero. Remember the TDD mantra[Bec03]: Red, Green, Refactor. The next step is to make the test go green. So far there is a default return value `int()` which corresponds to the needed return type. In this case the function body is `return i*i;` which causes the test to pass. Listing 1.4 shows the code that leads to a green bar.

```
int square(int i) {
  return i*i;
}

void thisIsATest() {
    ASSERT_EQUAL(4, square(2));
}
```

Listing 1.4: replaced return value to make tests green

Move the Implementation

As soon as the tests show green, the final step is to move the newly implemented code to the place where it belongs – the refactoring step of the TDD lifecycle. For this purpose the *Move Function Refactoring* may be used. In the code of listing 1.4 the function `square()` should be moved to its own header file.

The *Move Function Refactoring* will move the function `square` to a newly created header file called `square.h` and leaves the test code and the implementation separated and clean.

```cpp
#include "square.h"

void thisIsATest() {
    ASSERT_EQUAL(4, square(2));
}
```

Listing 1.5: Test source file: `test.cpp`

```cpp
#ifndef SQUARE_H_
#define SQUARE_H_

int square(int i) {
  return i*i;
}

#endif /* _SQUARE_H */
```

Listing 1.6: Separated header file: `square.h`

1.3.2. Add a Missing Type

In listing 1.7, the type `Universe` and its member are missing and need to be created.

```cpp
void thisIsATest() {
    Universe universe;
    ASSERT_EQUAL(42, universe.getAnswer());
}
```

Listing 1.7: The type universe is missing

The missing type is indicated by an error marker. A Quick Fix is registered to the error marker. As the Quick Fix is applied by the programmer, a simple `struct` is generated in front of `thisIsATest()`. Listing 1.8 demonstrates the situation after the missing type has been created.

```cpp
struct Universe {
};

void thisIsATest() {
    Universe universe;
    ASSERT_EQUAL(42, universe.getAnswer());
}
```

Listing 1.8: A new struct has been created

Since several keywords are available to define a class, such as `class`, `struct` or `enum`, the desired keyword can be chosen from an editable proposal menu. See figure 1.4 for an example of linked mode editing with a proposal menu.

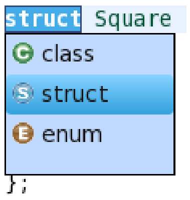


Figure 1.4.: Proposal menu offered by the Eclipse framework.

1.3.3. Create a Member Function

The newly created type `Universe` is still missing its member `getAnswer()`. `getAnswer()` is still producing a parse error as it is an unknown member function for type `Universe`. To resolve this parse error, another Quick Fix is offered to create an appropriate member function. Having applied the latter, the code should be in a compiling state as in listing 1.9.

```
struct Universe {
  int getAnswer(int i) {
    return int();
  }
};

void thisIsATest() {
    Universe universe;
    ASSERT_EQUAL(42, universe.getAnswer());
}

void runSuite(){
[...]
```

Listing 1.9: Newly created member function `getAnswer()`

Here, linked mode editing may also be used to quickly edit the return type after the member function has been created.

1.4. Expected Result

In the end there should be no problems in creating new free or member functions, new types or namespaces. Visibility on member functions can be changed fast. Adding or removing types from functions should not be a problem anymore. Moving functions is supported in a basic way. Classes may be moved to their own files and an `#include` statement is inserted where needed.

The result should be easy to install from Eclipse CDT, either as a standalone plug-in or bundled with the CUTE unit testing plug-in. It may be also be integrated into the CDT project.

A user manual is created, explaining how to effectively use the plug-in and to be fully aware of all TDD features offered by the plug-in. For developers, there is an extension manual explaining how to easily create new Quick Fixes and refactorings for code editing.

2. Analysis

TDD for CDT is a broad topic and a look at Eclipse JDT for Java reveals that the number of possible code generation features is immense. The scope of this analysis is reduced to features that can be considered helpful for Test-Driven Development.

This chapter discusses which features are helpful for TDD and for which reason, where generated code should be put and lists special cases that should be kept in mind during implementation.

2.1. Where to Put New Code?

This topic was discussed controversially at the beginning of the project. The question where to put new code is difficult to answer. Usually, tests are placed into a separate project. Which effective project is the project under test? If there is more than one, which should be preferred?

Standing in front of a hard problem with many solutions, the idea to use a wizard was close. However, many wizards in CDT are too overloaded because they try to do too much (look at the *New Class* wizard for an example). Therefore, a simpler solution was looked for.

In a meeting, the idea came up to put the new code above the current code block. During the design of a new unit test, this has the following advantages:

- The code compiles because the declaration is above the calling code.
- Switching between tests and implementation is faster because the new code is close to the test function and can be reviewed immediately.
- No user dialog is required, which shortens the process of writing tests.

There are some disadvantages to this solution too. First, if a function is called from multiple points inside the code, it possible that the new code requires a separate forward declaration which is not favorable. However, this is not critical to TDD.

Second, the generated code has to be moved separately. This can be solved by providing a comfortable feature to move code to the project under test. A full-featured *Move Refactoring* would support such a behavior.

2.2. Generating Types

Each member function corresponds to a certain type. For object-oriented programming, types are fundamental, which is why their creation is indispensable for this plug-in. Before a parser checker is able to complain about a missing member function, the corresponding type has to be resolved correctly.

There is a temptation to offer the overloaded *Create Class* wizard here. However, the programmer is able to continue much faster if the class is just inserted above the current test function. Unlike Java, in C++ there is no one-public-class-per-file [Mic05] limitation.

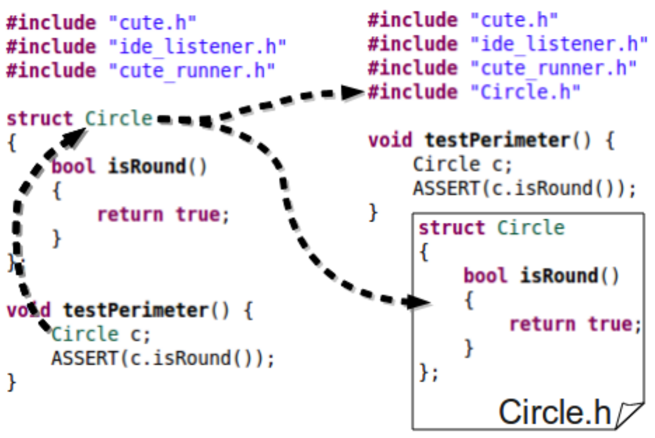
Later, after having implemented the member functions, the new type may be moved to another source file using the feature to extract types described in 2.7.2. The type creation and extraction process is illustrated in figure 2.2.

2.2.1. Generating Classes

Listing 2.1 shows a basic situation with a missing type. This is the simplest case of code generation that needs to be implemented. Code generation needs to create a new type named `MyClass` above the test function `testNewClass()`.

```cpp
void testNewClass() {
  MyClass c;
  ASSERT_EQUAL(c.member(), true);
}
```

Listing 2.1: Creating a simple new type.



```
#include "cute.h"
#include "ide_listener.h"
#include "cute_runner.h"

struct Circle
{
    bool isRound()
    {
        return true;
    }
};

void testPerimeter() {
    Circle c;
    ASSERT(c.isRound());
}
```

```
#include "cute.h"
#include "ide_listener.h"
#include "cute_runner.h"
#include "Circle.h"

void testPerimeter() {
    Circle c;
    ASSERT(c.isRound());
}
```

```
struct Circle
{
    bool isRound()
    {
        return true;
    }
};
```

Figure 2.1.: Create a class, implement it and move it

To make the plug-in usable for other purposes, too, classes should be created out of other situations, too.

For test fixtures as supported by CUTE[Som08] a test is specified inside a struct as in listing 2.2.

```
struct fix {
  void test() {
    klass k;
    ASSERT(k.member());
  }
};
```

Listing 2.2: Test fixture as supported by CUTE

The class under test, in this case `klass`, should then be created outside of the fixture class as shown in listing 2.3.

```
struct klass {
  boolean member() {
        return true;
  }
};

struct fix {
  void test() {
    klass k;
    ASSERT(k.member());
  }
};
```

Listing 2.3: Test fixture as supported by CUTE

2.2.2. Nested Classes and Namespaces

A class may be placed inside a specific namespace as shown in listing 2.4 and 2.5.

```
void testFoo() {
  N::MyClass c;
  ASSERT_EQUAL(c.foo(), true);
}
```

Listing 2.4: Call to an undefined namespace.

After applying a Quick Fix for creating namespaces the code looks like in listing 2.5

```
namespace N {
  struct MyClass {};
}
void testFoo() {
  N::MyClass c;
  ASSERT_EQUAL(c.foo(), true);
}
```

Listing 2.5: Creating a class inside a namespace.

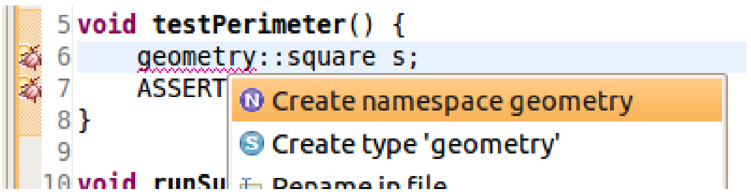
If scope of the current selection is inside a class, a newly created class should be nested to reduce name conflicts.

2.2.3. Namespace or Type

Given the source code in listing 2.6. If X does not exist, it is not possible to decide whether X should be created as a namespace or as a type. To solve this dilemma, two quick fixes have to be offered, one to create a namespace and one that creates a new type.

```
void testNamespaceOrType() {
  X::MyClass c;
  ASSERT_EQUAL(c.foo(), true);
}
```

Listing 2.6: Should `X` be created as namespace or as type?



```
void testPerimeter() {
    geometry::square s;
    ASSERT
}

void runSu
```

Figure 2.2.: Both, namespace and type creation is offered

2.2.4. Classes, Structs, Enums and Unions

C++ offers multiple keywords for declaring types. Is is up to the programmer to choose between the keywords `class`, `struct`, `enum` and `union`. To reduce visibility issues, `struct` could be used as default, as its members are public by default. However, this is a matter of coding style. An option is to provide a selection of keywords from which the user may choose from. No support needs to be implemented for `union` since it is not necessary in C++. In C++, type members are already non-virtual by default which supersedes the use of the `union` keyword. [Mic11]

2.2.5. Detecting Interfaces

It is theoretically possible to gain information about the interface a new class has to provide. In listing 2.7, a situation is shown where a class should be created which satisfies a certain interface.

```
void foo(Listener x) { /* ... */ }
void testSomeFunction() {
  MyClass c;
  ASSERT_EQUAL(foo(c), true);
}
```

Listing 2.7: Interfaces

2.2.6. Templated classes

The creation of templated classes should be allowed by using a class with it's actual template parameters.

```cpp
void function() {
  A<int> a;
}
```

Listing 2.8: Use of a templated class which does not exist

`A` is not defined yet so it should be complained about the missing type. Creating the class `A` will result in the following code seen in 2.9.

```cpp
template<typename T>
struct A {
};

void function() {
  A<int> a;
}
```

Listing 2.9: Templated class was created

This should also work for multiple template argumentss as for non-typed template parameter seen in 2.10

```cpp
template<typename T, int i>
struct A {
};

void function() {
  A<int, 5> a;
}
```

Listing 2.10: Class `A` with non-typed template parameter `i`

2.3. Generating Functions

During (member) function creation, several aspects have to be considered. A new function needs a correct signature, has to be inserted with an appropriate visibility and const correctness[Sut09] should be regarded. A set of such constraints for new functions are discussed in the following sections.

2.3.1. Function Return Type

To be most supportive, suitable return types should be detected automatically. Hence, before the function is created, the function call is analyzed to gather information about the return type. The chosen return type may be edited in linked mode right after the function was inserted.

Procedures

In case of procedures, `void` is assumed as return type. A function is considered a procedure if no calculation is done with its return value. Listing 2.11 shows an example of a missing procedure.

```
void test() {
  procedure();
  ASSERT(procedure_was_successful);
}
```

Listing 2.11: A procedure which should have a `void` return type

Basic Data Types

In case of basic types, it is always legal to return an integer. However, if for example a function is called inside an `if(...)` statement, it is more intuitive to create a function returning a `bool` instead. In case of other basic data types, loss of precision is possible. In addition, `std::string` should be preferred over `const char[]` as return type.

Binary Expressions

In case of binary operators, the type of the sibling operand can be used. There may be other possible types, especially in case of inheritance, but the primary objective is to produce compiling code. A selection of possible types could be presented in a proposal menu after function creation. For simplicity, only the closest type should be used while linked mode could give the option to choose another type manually.

An example of a binary operator is the assignment operator in listing 2.12.

```
int x = function();
```

Listing 2.12: Simple assignment

This leaves only the option of an **int** as return type – the type of the left-hand side of the assignment.

If the sibling operand cannot be resolved to a type too, the correct return type can be searched in the expression's parent. The type is looked up from inside out until there is an assignment or another expression which reveals the desired data type.

Another case could be a cast like this:

```
int x = static_cast<int>(function());
```

Listing 2.13: Determining function return type in case of a cast

Since this does not indicate a direct return type of `function()`, it could return anything that could be converted to an `int`.

Unary Expressions

Unary expressions are similar to binary expressions. However, because the operator itself does not make clear what data type is used, the parent node has to be searched for the correct return type.

2.3.2. Determining Parameter Type

Similar to the function return type, parameter types have to be determined too. Parameter types may be looked up based on the unresolved function call. In listing 2.14, based on the argument 42, a parameter of type `int` is created.

```
void foo(int i) {
}

void testAnswer() {
  foo(42);
}
```

Listing 2.14: Determining function parameter types

As in the above example, determining the parameter type for literal nodes is straightforward. In case of a function call, the corresponding return type of the function has to be copied. For example, to create the function `myFunction()` in listing 2.15, the signature of `mulitply` needs to be inspected to determine the second parameter type.

```
double multiply(double first, double second);

void testMyFunction() {
  myFunction(1, mulitiply());
}
```

Listing 2.15: Looking up the type of a function

Parameter Names

When a function with parameters is created, each parameter should have an intuitive and unique name to ensure the code compiles. For basic data types, the first letter of the type name, e.g. `i` for `int`, is taken. If multiple parameters are created with the same type, the next free letter is chosen as shown in listing 2.16.

```
void foo(int const & i, int const & j,
         int const & k, double const & d) {
}

void testAnswer() {
  foo(42, 43, 44, 45.0);
}
```

Listing 2.16: letters used for basic type parameters

If a function is called with a symbol as argument (like function calls, variable names), the name of the symbol in lowercase is taken. In case of name collisions, either an incrementing number is appended to the symbol name, or the next free letter is chosen in case of basic types (like `i` for `int`, then `j`, `k`, `l`, etc...). Shadowing of variables should be avoided and the next number should be chosen that does not conflict with another symbol.

Listing 2.17 shows an example that covers all discussed parameter names and collisions.

```
void foo(double const & d, double const & e,
         int & e1, int & delta, int & delta1) {
}

void testAnswer() {
  int e, delta;
  foo(42.0, 43.0, e, delta, delta);
}
```

Listing 2.17: avoiding name conflicts for parameter names

2.3.3. Adding a Free Function

A new function definition shall be created with parameters that match the data type of the calling functions arguments as discussed in sections 2.3.2 and 2.3.2. The insertion point of the new function is one of the following:

1. Above the parent function definition if the current scope is not inside a type definition. Example:

```
void freefunction() {
}
void test() {
  freefunction();
}
```

Listing 2.18: free function created above a function call

2. Above the outermost type definition if current scope is located inside a type definition. Example:

```
namespace N {
void freefunction() {
}
class A {
  struct B {
    void test() {
      freefunction();
    }
  };
};
}
```

Listing 2.19: free function created above type definitions

3. Above the current expression statement else. Example:

```
int freefunction() {
  return int();
}
int x = freefunction();
```

Listing 2.20: free function created above an assignment

Inside a type definition, both options, *Create Free Function* and *Create Member Function*, must be available.

2.3.4. Adding a Member Function

After a class has been created, member functions may be added to it. The system must deal with the following cases:

- One or more (see [ISO11] 3.2, §5: *One Definition Rule*) class definitions exist: add the new member function to all definitions. Example:

```cpp
struct MyClass;
struct MyClass {

};
struct MyClass {

};
void testMember(){
  MyClass m;
  m.member();
}
```

Listing 2.21: before

```cpp
struct MyClass;
struct MyClass {
  void member() {}
};
struct MyClass {
  void member() {}
};
void testMember(){
  MyClass m;
  m.member();
}
```

Listing 2.22: after

- Only a class declaration exists: replace the declaration with a class definition containing the new member function. Example:

```cpp
struct MyClass;


void testMember(){
  MyClass m;
  m.member();
}
```

Listing 2.23: before

```cpp
struct MyClass {
  void member() {}
};
void testMember(){
  MyClass m;
  m.member();
}
```

Listing 2.24: after

In case of multiple declarations, one of the occurences is replaced.

- No declaration exists: do not offer a resolution in this case. The class first needs to be declared properly to be able to add members. Even if it is theoretically possible to create both at a time, class definition and the requested member inside, do not offer this. A resolution should only do one thing at a time. Example:

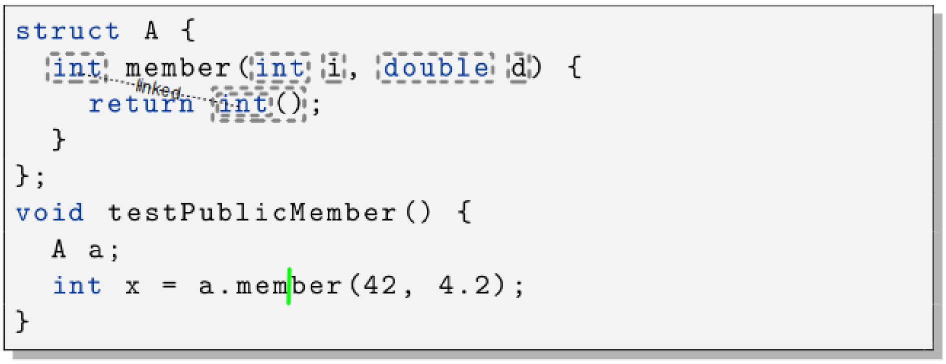
```
void testMemberFunction() {
  MyClass m;
  m.member();
}
```

Listing 2.25: `MyClass` needs to be created before members may be added

The system should be able to handle static member function calls as well.

Editing Assistance

After creation of the new member, its declaration specifier, parameter types, parameter names and if available its return type may be edited in linked mode. Assistance for editing the function name may be left away because it is definitly non-ambiguous. An example is shown in figure 2.3.



```
struct A {
  int member(int i, double d) {
    return int();
  }
};
void testPublicMember() {
  A a;
  int x = a.member(42, 4.2);
}
```

Figure 2.3.: Assisted editing in linked mode after function has been created. Green: the exit position

Visibility

Members of a class are private by default and members of a struct are public by default[ISO11]. If the member is created based on a function call outside of the scope of the target class, it must have public visibility. Otherwise it cannot not be accessed after creation. Listing 2.26 describes an example of a generated member function that needs public visibility to be accessible.

```
class A {
  void member() {
  }
};
void testPublicMember() {
  A a;
  a.member();
}
```

Listing 2.26: A member that needs public visibility

Else, if a member is created from within the class it should be private. An example is shown in listings 2.27 and 2.28. Existing visibility labels should be reused if available.

```
class ATest {
public:
  void testMember() {
    member();
  }



};
```

Listing 2.27: before

```
class ATest {
public:
  void testMember() {
    member();
  }
private:
  void member() {
  }
};
```

Listing 2.28: after

2.4. Changing Arguments

Whenever a function is called with the wrong set of arguments, a resolution may be offered to fix the problem. This situation occurs if the arguments of a function call do not match the parameters of an existing function with the same name. An example of a function call with wrong arguments is described in listing 2.29.

```
void badArgs() { ... }
void testBadArgs() {
  badArgs(42);
}
```

Listing 2.29: a function call with non-matching arguments

The situations of non-matching function arguments may be categorized as follows:

- Calling function has more arguments than the declared function (see listing 2.29 above)
- Calling function has less arguments than the required declared function:

```
void badArgs(int i) { ... }
void testBadArgs() {
  badArgs();
}
```

Listing 2.30: less arguments than parameters

- The number of arguments and parameters is equal but types do not match:

```
void badArgs(int i) { ... }
void testBadArgs() {
  badArgs("noIntegerArgument");
}
```

Listing 2.31: wrong arguments

In this case it is important to be aware that fundamental types (`int`, `char`, `double`, etc... as described in [ISO11] 3.9.1) are not considered a type mismatch, so the following code is compilable:

```
void badArgs(int i) { ... }
void testBadArgs() {
  badArgs(4.2);
}
```

Listing 2.32: wrong arguments

Depending on the situation, arguments need to be added, removed or changed. For all cases, the option to create the missing function and hence overload the existing function should always be available.

In case of function overloading, it is possible that the system needs to handle more than one candidate function. In this case, multiple resolutions may be displayed. The number of displayed resolutions should be limited to a number of overviewable items.

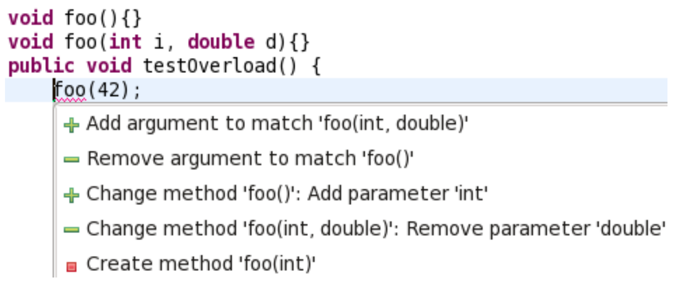
To avoid a combinatorial explosion, to reduce the complexity and to make the offered resolutions more foreseeable, combinations of the above three cases, listings 2.30, 2.31 and 2.32 should not result in a resolution offered to the programmer. Take listing 2.33 as an example where no resolution to append arguments should be offered. The only Quick Fix offered here is the create member function.

```
void foo(int i, int j) { ... }

void testResolve() {
  foo("string");
}
```

Listing 2.33: *Add Argument* plus *Change Argument* needed at the same time

At last, a decision has to be made whether the arguments of the calling function or the parameters of the function declaration should be changed. On the one hand, changing parameters would better support TDD because during TDD a system is designed. On the other hand, changing parameters potentially breaks the code and needs to be done carefully. Based on these arguments, it was decided to only offer changing arguments.

If both changing parameters and arguments are implemented, the resolutions offered to the user may look like in figure 2.4, a screenshot taken from Eclipse JDT.



```
void foo(){}
void foo(int i, double d){}
public void testOverload() {
    foo(42);
```

Add argument to match 'foo(int, double)'
Remove argument to match 'foo()'
Change method 'foo()': Add parameter 'int'
Change method 'foo(int, double)': Remove parameter 'double'
Create method 'foo(int)'

Figure 2.4.: Parameter Quick Fix examples from Eclipse JDT

2.4.1. Removing Arguments

Whenever removing arguments can lead to a well-defined function call, an appropriate resolution may be offered to do so. An example is shown in listing 2.34.

```
void foo(int i) { ... }

void testResolve() {
  foo(42, 43.0);
}
```

Listing 2.34: remove `43.0` to match `foo(int)`

What if the second argument were an integer too? It is not clear which one to remove. In this case, remove the argument at the end, assuming that backmost arguments are of less importance. An example is described in listings 2.35 and 2.36.

```
void foo(int i) {
}
void test() {
  foo(1, 2, 3);
}
```

Listing 2.35: before

```
void foo(int i) {
}
void test() {
  foo(1);
}
```

Listing 2.36: after

A resolution is offered for each overload that matches at least one argument and may be reached by removing an arbitrary number of arguments. Listing 2.37 shows an example where no resolution should be displayed because cropping arguments will never lead to a correct function call.

```
void foo(int i) { ... }

void test() {
  foo("noInt", "noInt", "noInt");
}
```

Listing 2.37: at least one matching argument needed

2.4.2. Appending Arguments

Default arguments may be added to the function call to match an overload of a function:

```
void foo(int i) { ... }

void testResolve() {
  foo();
}
```

Listing 2.38: add a default argument `0` to the call to match `foo(int)`

To reduce the number of combinations, *Append Argument* is only offered if current arguments match the first parameters of an overload. In this case, a valid function call may result by adding new arguments.

Adding default arguments may be difficult in case of complex constructors or when using pointers. However, in this case it would be already useful to insert a variable name at the right place. Based on this name, a local variable may be created in a further step. An example of a nested constructor which results in a complex default argument is shown in listings 2.39 and 2.40.

```
struct A { A(int); };
struct B { B(A);   };
struct C { C(B);   };

void testNest() {
  C var;
}
```

Listing 2.39: missing argument

```
struct A { A(int); };
struct B { B(A);   };
struct C { C(B);   };

void testNest() {
  C var(B(A(0.0)));
}
```

Listing 2.40: added argument

2.4.3. Swapping Arguments

If a called function cannot be resolved but has an overload candidate which accepts the same number of parameters but in the wrong order, reordering arguments is an option. The result of a resolution on 2.41 shall look like 2.42.

```
void foo(A a, B b) {
}
void testSwap() {
  A a;
  B b;
  foo(b, a);
}
```

Listing 2.41: before swap

```


void testSwap() {
  A a;
  B b;
  foo(a, b);
}
```

Listing 2.42: after swap

If the order of the arguments differs in more than two places, the answer for correct ordering can be ambiguous. In ambiguous situations, no resolution should be offered. See listing 2.43 for an example.

```
void func(A, B, B, A);
void testSwap() {
  A a1, a2;
  B b1, b2;
  foo(b1, a1, a2, b2);
}
```

Listing 2.43: ambiguous argument swapping

It is arguable whether *Swap Argument* really supports TDD as the calling code is adapted according to the declaration instead of the other way around. Its implementation is optional.

2.4.4. Changing Types

Given an unresolved function call with correct number of arguments, wrong types and swapping not possible, a resolution may be offered that changes the function call to an appropriate overload.

```
void foo(Foo f, Bar b) { ... }

void testResolve() {
  foo(1, 2.3);
}
```

Listing 2.44: Wrong argument order for `foo()`

A resolution should fix the code to the function `foo` with types `Foo` and `Bar`. The resulting code should look like in 2.45.

```cpp
void foo(Foo f, Bar b) { ... }

void testResolve() {
  foo(Foo(), Bar());
}
```

Listing 2.45: Correct arguments to `foo()`

For this functionality too, it is questionable whether it is helpful for TDD since it changes the caller instead of the callee. However, it would be comfortable for updating tests in other situations.

2.5. Generating Variables

In this context, variables may be looked at as if they were functions without parameters. They have to be visible if they are called outside of their defining class and they need a certain type.

Local variables are a special case with no visibility issues. Create Local Variable, proposed by Tomasz Wesolowski[Wes11], was the first Quick Fix in CDT that was using the Codan infrastructure. However, it lacks macro support which is essential for working with the CUTE framework. In addition, basic types are only created as integers which is annoying if it were obvious that a more precise data type could have been chosen.

Member variables called from outside of the class, need to be publicly accessible as shown in listing 2.46.

```cpp
class Type {
public:
  int memberVariable;
};

void test() {
  Type t;
  t.memberVariable = 42;
}
```

Listing 2.46: Creation of a public member variable

If the calling function is defined inside the same class, the new member variable should be private (”shy code” [Tho05]) as shown in listing 2.47.

```cpp
struct Type {
  void internalMember() {
    this->privateMemberVariable = 42;
  }
private:
  int privateMemberVariable;
};
```

Listing 2.47: Creation of a private member variable

2.6. Changing Visibility

The code in listing 2.48 does call a not visible member function in a type.

```cpp
class Type {
  void foo() {
  }
};

void test() {
  Type t;
  t.foo();
}
```

Listing 2.48: Member function `foo()` is not visible

This should produce an error and a Quick Fix should be displayed for changing the visibility of `foo` to public resulting in listing 2.49.

```cpp
class Type {
public:
  void foo() {
  }
};

void test() {
  Type t;
  t.foo();
}
```

Listing 2.49: Member function `foo()` was changed to public

2.7. Move Refactorings

In most cases, new code is placed right above the test function definition. After the code has been implemented, it should be moved to an appropriate source file.

The type to be extracted is chosen by the position of the cursor. Corresponding to the scope where the cursor is placed the type is extracted and the header file is named accordingly.

In the case of namespaces the namespace is not moved but the namespace is created around the moved class in the newly created header file shown in listing 2.52.

```

namespace N {
klass k {
  | //<- cursor
};
klass j {
};
}
```

Listing 2.50: before extract

```
#include "klass.h"
namespace N {
//klass has been
//moved to klass.h

klass j {
};
}
```

Listing 2.51: after extract

```
namespace N {
klass k {
};
}
```

Listing 2.52: extracted `klass.h` (without include guards)

However in nested types the class definition which is most close to the cursor is extracted as listings 2.53 and 2.54 show.

```cpp
class a {
 class b {
  class c {
  }; | //<- cursor
 };
};
```

Listing 2.53: before extract

```cpp
#include "c.h"
class a {
 class b {
 //moved to c.h
 };
};
```

Listing 2.54: after extract

2.7.1. Moving Functions

Newly created code inside a test file sooner or later needs to be moved to a place where it belongs inside the project being tested. This is why a *Move Function Refactoring* is needed that offers moving the created code to another source file.

If code is moved, it is important to introduce an `include` preprocessor statement in the test file if needed.

2.7.2. Extracting Types to Separate Source Files

This feature basically takes the code of an type implementation and moves it into a new source file. The file name equals the type name. A problem is that the target project has to be chosen by someone. A new class file along the test source files is not very convenient since implementation project and and test project are separated and the implementation would be in the wrong place.

3. Implementation and Solution

This chapter describes how the analyzed features were implemented in Eclipse CDT.

A separate Eclipse plug-in was chosen in favor of changing and patching CDT directly. This has the advantage that the plug-in may be installed via an update site even if it is not integrated into CDT. To reduce integration effort, code was developed against a recent CVS tag of Eclipse CDT.

3.1. Approach

First features were added by registering Quick Fixes at the codanMarkerResolution extension point. All names inside the AST that resolve to a `IProblemBinding` were already discovered and reported by Codan checkers. This enabled a fast start for implementing *Create Type*, *Create Local Variable* and *Create Free Function*.

Later, the logic that produces the problem markers, the Codan checkers, were extended to report more information needed inside the Quick Fixes. For Quick Fixes with a dynamic number of possible resolutions, like for *Appending Arguments* in case of overloading, other solutions had to be implemented.

To be able to provide a larger set of Quick Fixes, multiple features were developed iteratively and in parallel. This way, not every possible special case is covered but it is more realistic to provide a complete set of code generation Quick Fixes for TDD.

3.2. Prerequisites

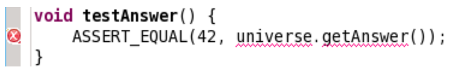
What techniques are needed to implement the discussed features? First, there has to be an infrastructure that allows to analyze the code in form of an abstract syntax tree. Second, some functionality is needed that scans this structure for potential errors. Those errors have to be made visible to the programmer in some way. Last, functionality is needed to offer a resolution at the error location.

3.2.1. Source Inspection

The C++ parser of Eclipse CDT provides an abstract syntax tree (AST) that can be traversed using the visitor pattern. The nodes inside this tree must not be edited directly. Instead, a rewriter is provided to alter code based on the nodes of an AST.

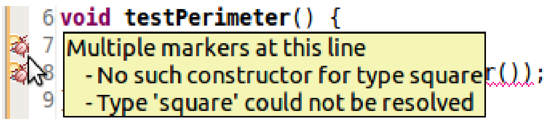
3.2.2. Problem Detection

Eclipse CDT already contains an infrastructure to scan the AST for nodes that are in some error state. The AST itself provides information about names that could not be resolved by trying to match names to their underlying definition and declaration. The Codan subproject has built-in checkers which collect errors and produce a warning inside the source code by underlining the problematic text passages. Additionally red markers were placed at the affected line of code to indicate to the user that there is a problem in this section.



```
void testAnswer() {
    ASSERT_EQUAL(42, universe.getAnswer());
}
```

Figure 3.1.: A checker reports multiple problems

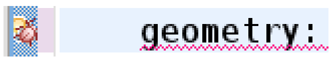


```
void testPerimeter() {
Multiple markers at this line
  - No such constructor for type square
  - Type 'square' could not be resolved
r());
```

Figure 3.2.: A checker reports multiple problems

3.2.3. Resolutions / Quick Fix

If a problem has been found and there is a solution for it Codan displays a yellow light behind the red marker to indicate that there is a problem.

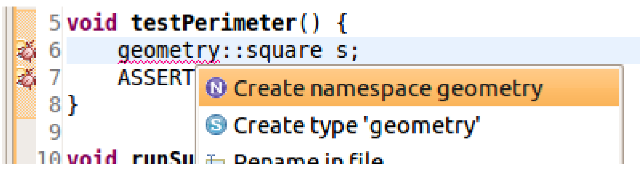


```
geometry:
```

Figure 3.3.: A light indicates that there is a solution to a found problem

The terms `Resolution` and Quick Fix describe the same idea. Resolution is the term used inside CDT source code and Quick Fix is the name presented at the user interface.

Such a resolution can be called by hitting `Ctrl-1` if the cursor is at the right place in the source code.

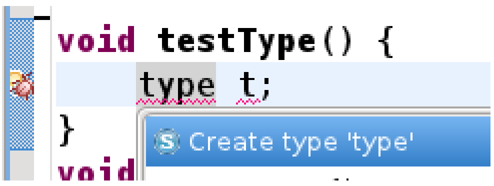


```
void testPerimeter() {
    geometry::square s;
    ASSERT
}

void runSu
```

Figure 3.4.: Multiple resolution to a missing name

3.2.4. Code Analysis and Change

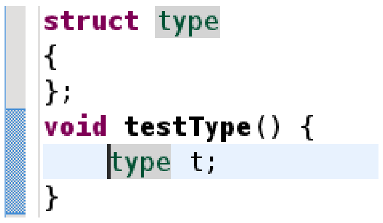
If the programmer decides to apply such a Quick Fix on a certain source selection, a mechanism is needed for analyzing what code has to be generated or changed and at which position it should be inserted. Such an analysis is done using visitors on the AST to collect information about a situation and in which context the Quick Fix is called and trying to generate the expected result.



```
void testType() {
    type t;
}
void
```

Figure 3.5.: Resolution to create a type is displayed

Changes to the code are done carefully. Impacts of a Quick Fix are kept as local as possible. This has several reasons. First, mistakes are kept local and second, the programmer does not need to switch the source code file to control or correct the newly generated code. The programmer can stay in the test file until he integrates his changes permanently. Until then there is no need to change the source file, minimizing context switch and maximizing focus on the current work.



```
struct type
{
};
void testType() {
    type t;
}
```

Figure 3.6.: A new type was generated above the test

3.2.5. User Guidance

After code has been generated, some situations require the programmer to review the changes or to take a decision, like choosing a meaningful name for a newly created variable. To avoid user dialog, linked mode of Eclipse jFace[Ecl11d] may be used that allows to toggle between specific text selections. linked mode is probably best known from the *Rename Refactoring* both in CDT and JDT.

Linked mode is explained in more detail in section 3.8.5.

3.3. Used Extension Points

Extension points [Ecl11f] provide a way to extend Eclipse without changing its source code. Instead of hard-coding changes, new features can be added by adding an entry inside the `plugin.xml` file. Using a certain interface allows communication between the extension point and the new feature.

Extension Points were used to add Quick Fixes, checkers and the key binding for the *Extract Type / Function* features. They are explained in the following sections.

3.3.1. Checkers

To add a new checker to the Codan infrastructure, first the specific checker name must be registered at the Extension Point: `org.eclipse.cdt.codan.ui.codanMarkerResolution`. This is done via the `plugin.xml` in the plug-in project. In addition it must be specified there what kind of problem the checker reports, his name and the severity. (error or warning).

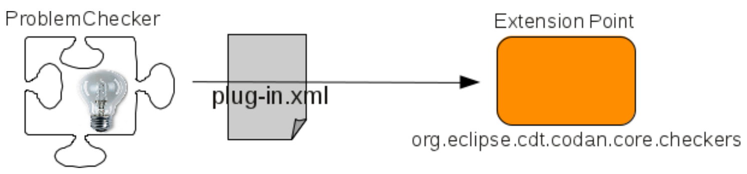


Figure 3.7.: Registering a checker at the extension point

```
<extension point="org.eclipse.cdt.codan.core.
   checkers">
  <checker
      class="ch.hsr.eclipse.cdt.codan.checkers.
         MissingOperatorChecker"
      id="ch.hsr.eclipse.cdt.codan.checkers.
         missingoperatorchecker"
      name="Missing Operator Checker">
    <problem
      defaultSeverity="Error"
      id="ch.hsr.eclipse.cdt.codan.checkers.
         MissingOperatorResolutionProblem_HSR"
      name="Operator could not be resolved">
    </problem>
  </checker>
</extension>
```

Listing 3.1: Registering a checker with Codan extension point

For the logic of the checkers, the class `AbstractIndexAstChecker` was extended. The checker processes the AST with a visitor and collects information about context and tries to find out problems by either resolving names or more complex methods.

3.3.2. Codan Marker Resolution

New Quick Fixes are added by implementing the `ICodanMarker-Resolution` interface. Like new checkers, the implementing class must be registered at the `plugin.xml` configuration file in the plugin project.

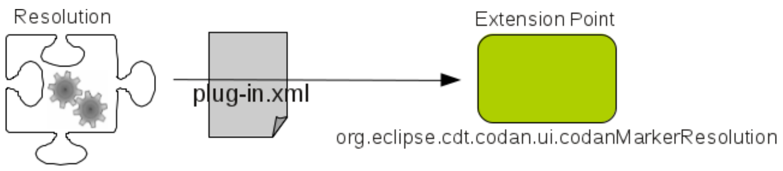


Figure 3.8.: Registering a marker resolution at the extension point

Each Quick Fix is registered to a certain problem id which a checker reports. The `plugin.xml` file acts as the adapter between checkers and Quick Fixes.

```
<extension point="org.eclipse.cdt.codan.ui.
   codanMarkerResolution">
  <resolution
    class="ch.hsr.eclipse.cdt.createtype.
       CreateTypeQuickFix"
    problemId="ch.hsr.eclipse.cdt.codan.checkers.
       TypeResolutionProblem_HSR">
  </resolution>
</extension>
```

Many problem ids that are interesting for offering Quick Fixes are listed and reported in the class `ProblemBindingChecker`.
Because the checker class cannot not be altered directly, all Codan bindings are disabled inside the Activator class and replaced by problem ids ending in "_HSR". All problem ids are referenced in the `ProblemIdCollection` class located in the plug-in package for convenience.

3.3.3. Marker Resolution Generator

The resolution generator of Codan has a problem. It does not allow a dynamic number of the same Quick Fix to be displayed for a single reported problem. Why would anyone need this?

Having overloaded functions with different argument and a wrong function call with no arguments.

```
void func(int) {}

void func(int, int) {}

void call() {
  func();
}
```

Listing 3.2: Overloaded functions and a wrong function call

Here two Quick Fixes need to be displayed, first for the function with one and second the function with two arguments. This is not possible with the standard Codan way of registering a single instance of a Quick Fix to a number of same problem ids.

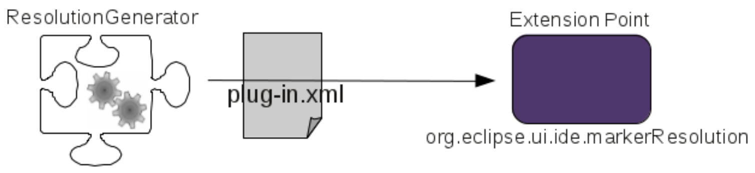


Figure 3.9.: Registering a marker resolution at the extension point

`org.eclipse.ui.ide.markerResolution` is used to implement a separate resolution generator for *Appending Argument*. This is needed to display multiple times the same Quick Fix .

```
<extension point="org.eclipse.ui.ide.
    markerResolution">
  <markerResolutionGenerator
      class="ch.hsr.eclipse.cdt.addArgument.
          AddArgumentQFGenerator"
      markerType="org.eclipse.cdt.codan.core.
          codanProblem">
  </markerResolutionGenerator>
</extension>
```

Listing 3.3: Registering a resolution generator

3.3.4. Commands, Bindings, ActionSets

These are the default extension points for adding commands to menus and key bindings in Eclipse.

3.3.5. Quick-Fixes for a Specific `messagePattern`

Optionally, the xml attribute `messagePattern` can be used to offer more specific Quick Fixes for the same problem id depending on the marker message.

```
  <resolution ...
    messagePattern="Function '*' is undefined">
  </resolution>
</extension>
```

This feature was not used but could be considered if the plug-in is integrated into CDT. Be aware that the messagePattern attribute of a problem id has another meaning.

3.4. Marker Resolution Refactorings

Behind every Quick Fix, a class derived from refactoring classes does change the source code. However *Marker Resolution Refactorings* are not really refactorings but are the same infratstructure to rewrite the source code in the editor. These refactoring classes contain the logic of the Quick Fixes and are the heart of the developed plug-in. This section introduces the implemented marker resolution refactorings.

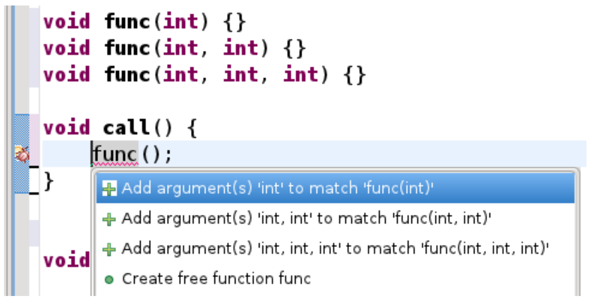
3.4.1. Create Type or Namespace

Weather a type or a namespace should be created is pre-determined in some situations. Hence, it is possible that both Quick Fixes are offered to the user.

This is also discussed in section 2.2.3

3.4.2. Append / Remove Argument

Append Argument is special because it is the only Quick Fix that is offered multiple times for a single problem marker. Hence, the refactoring has to know which of these Quick Fixes was applied. To find out which overload should be used, each Quick Fix is aware of the candidate number it is linked to. The refactoring retrieves the sorted list of candidates itself independently and uses the passed in number to find the same candidate again.



```
void func(int) {}
void func(int, int) {}
void func(int, int, int) {}

void call() {
    func();
}

void
```

Add argument(s) 'int' to match 'func(int)'
Add argument(s) 'int, int' to match 'func(int, int)'
Add argument(s) 'int, int, int' to match 'func(int, int, int)'
Create free function func

Figure 3.10.: Offering three times the same Quick Fix type

3.4.3. Create Function

This is the most complex set of Quick Fixes. Different refactorings are needed for free, member, operator, constructor and static functions. The latter function types mostly differ in their way of creating the newly inserted function definition. This logic has been wrapped in a strategy pattern and classes have been grouped into three packages: common, strategy and Quick Fix classes. Figure 3.11 describes the package structure with the different function creation strategies.

The different Quick Fix classes are responsible for showing an appropriate image and label and each declare which strategy should be used. Figure 3.12 visualizes the role of the strategy during the calculation of changes inside the refactoring object.

The `FunctionCreationStrategy` is the most common of the strategies and is used for creating member, non-member and indirectly also for static functions. For creating operator and constructor function definitions, different functionality was required each.

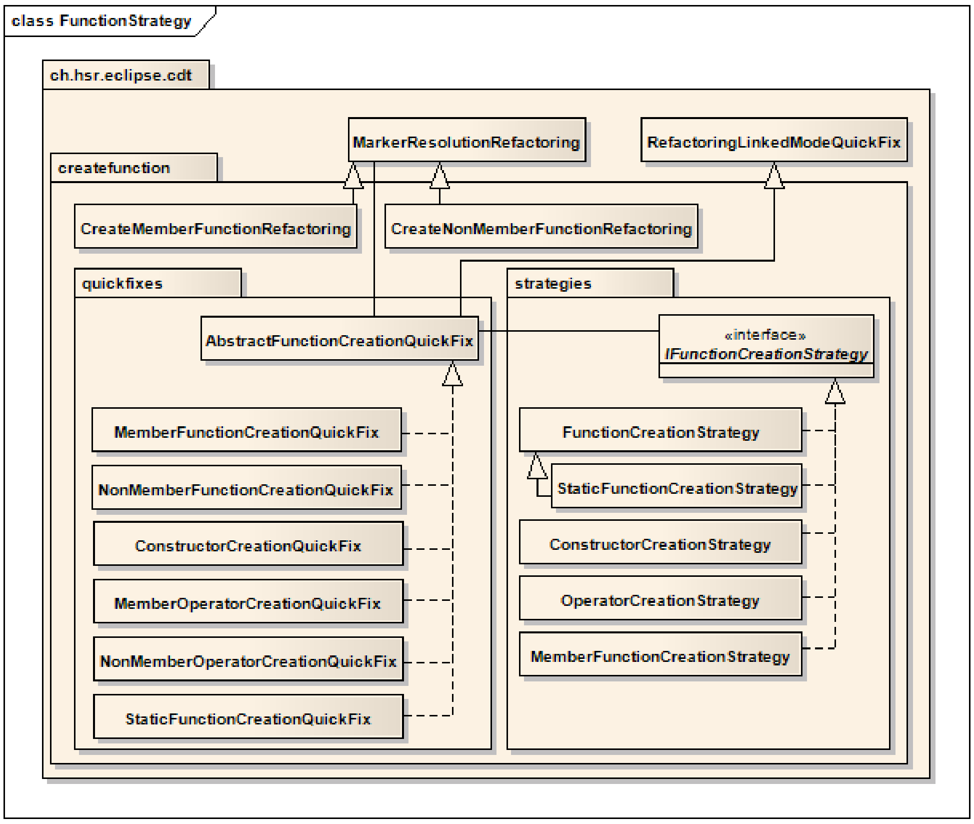

class FunctionStrategy
ch.hsr.eclipse.cdt
MarkerResolutionRefactoring
RefactoringLinkedModeQuickFix
createfunction
CreateMemberFunctionRefactoring
CreateNonMemberFunctionRefactoring
quickfixes
strategies
AbstractFunctionCreationQuickFix
«interface» IFunctionCreationStrategy
MemberFunctionCreationQuickFix
NonMemberFunctionCreationQuickFix
ConstructorCreationQuickFix
MemberOperatorCreationQuickFix
NonMemberOperatorCreationQuickFix
StaticFunctionCreationQuickFix
FunctionCreationStrategy
StaticFunctionCreationStrategy
ConstructorCreationStrategy
OperatorCreationStrategy
MemberFunctionCreationStrategy

Figure 3.11.: Package overview for the create function features

3.4.4. Create Variable

Create Local Variable was introduced by the CDT community at the time of the start of this thesis. Nevertheless, the Quick Fix was continued since it has better return type recognition for basic types (boolean, double, etc.) and linked mode was added for convenient change of type. However a linked mode position to change the name was not implemented since the name was already chose by writing it before `Creating Variable`.

For creating member variables another refactoring was developed but placed into the same package since it also creates variables.

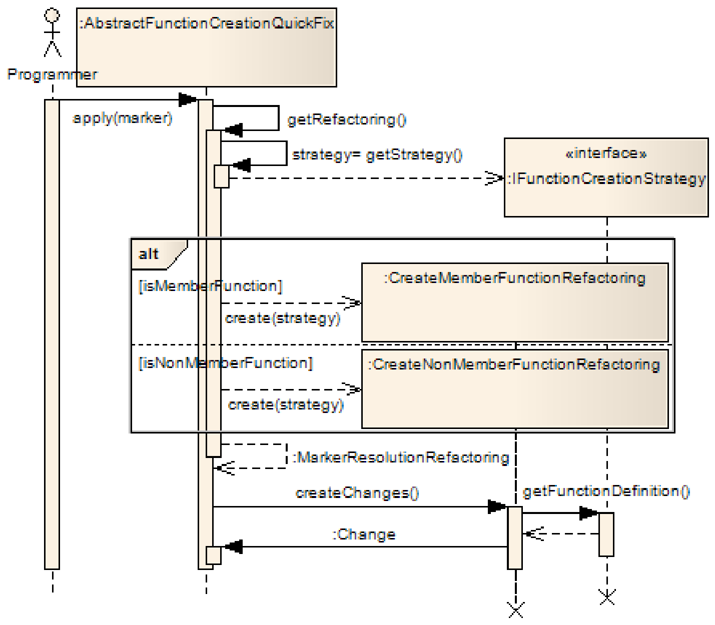


Figure 3.12.: Sequence diagram of the function creation process

3.4.5. Change Visibility

This refactoring is not directly useful for TDD but the visibility logic was needed for other refactorings.

3.4.6. Extract Type and Function

This refactoring is the only one which is not invoked through a Quick Fix but via a key combination (Alt-Shift-P was free, stands for extract type). The extract refactoring does both, extract types and functions, since behavior is almost the same.

The extract refactoring supports moving functions and types into a separate header file. However, it is only a small step towards a good *Move Refactoring* which is required for convenient TDD.

3.5. Architecture

A lot of architecture is already predetermined by Eclipse, CDT and Codan. This section discusses the overall architecture of the developed plug-in.

3.5.1. Package Structure

The classes are structured into several packages of separated interests. Classes used by more than one parts were moved to the top level package.

All checkers were put into a single package as detecting errors and correcting them are two different parts.

The Quick Fixes which represent the user interface part of the plug-in, the logic that decides whether to provide a Quick Fix and the actual logic for changing the source code are located together in a single package for each provided Quick Fix .

3.5.2. Class Diagram

Each Quick Fix consists of at least two classes, extending `MarkerResolutionRefactoring` or `RefactoringLinkedModeQuickFix` respectively. Latter uses the `ChangeRecorder` to find out the positions to be shown in linked mode based on the changes generated by the refactoring class.

Separate of the Quick Fixes is the extract functionality which is registered to a key binding instead of a problem.

Checkers all define a visitor that handles problems on the AST. Each visitor extends class `AbstractResolutionProblemVisitor` which helps filering problem nodes.

Figure 3.13 shows all important classes of the plug-in without helper classes.

3.5.3. Basic Call Flow

Whenever the user makes changes inside the C++ source code editor, a Codan reconciling job is scheduled. After 50ms of pause without typing, this job starts. After the Abstract Syntax Tree has been updated, the Codan reconciler job runs all registered checkers, reporting problems to the central CodanMarkerResolutionRegistry. See figure 3.14. A list of all markers and resolutions is requested by the editor and displayed appropriately. This process is visualized in figure 3.15.

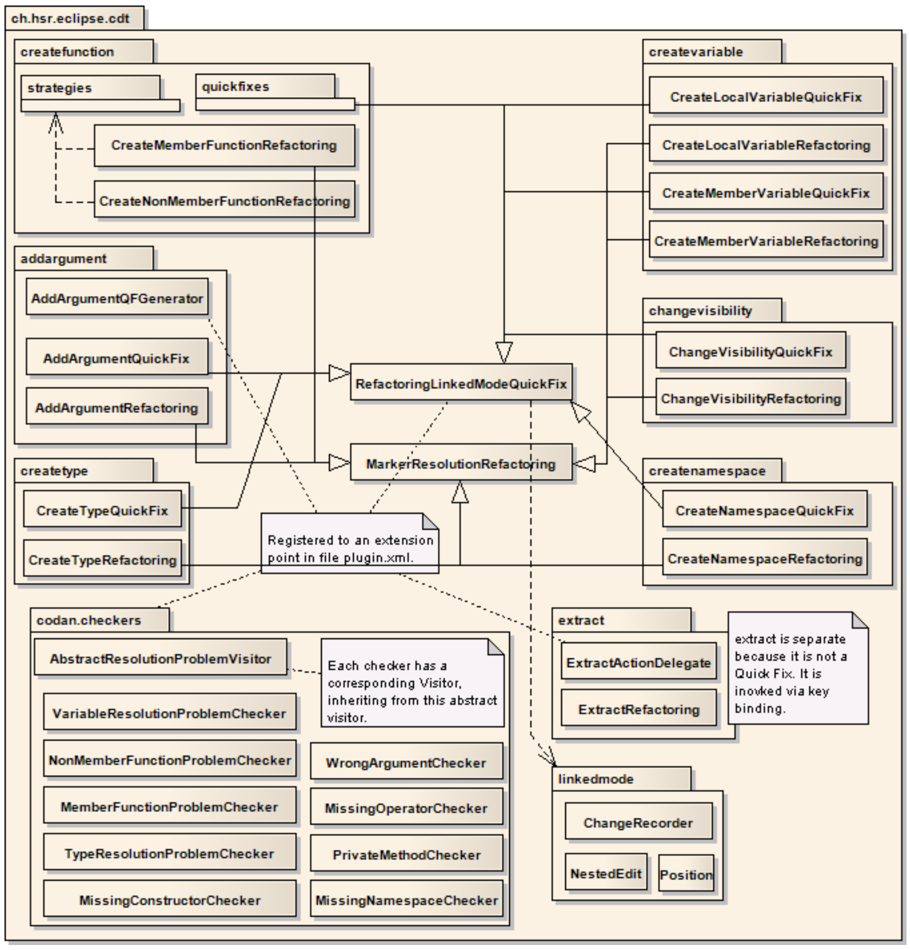

ch.hsr.eclipse.cdt
createfunction
strategies
quickfixes
CreateMemberFunctionRefactoring
CreateNonMemberFunctionRefactoring
addargument
AddArgumentQFGenerator
AddArgumentQuickFix
AddArgumentRefactoring
createtype
CreateTypeQuickFix
CreateTypeRefactoring
createvariable
CreateLocalVariableQuickFix
CreateLocalVariableRefactoring
CreateMemberVariableQuickFix
CreateMemberVariableRefactoring
changevisibility
ChangeVisibilityQuickFix
ChangeVisibilityRefactoring
RefactoringLinkedModeQuickFix
MarkerResolutionRefactoring
createnamespace
CreateNamespaceQuickFix
CreateNamespaceRefactoring
Registered to an extension point in file plugin.xml.
codan.checkers
AbstractResolutionProblemVisitor
Each checker has a corresponding Visitor, inheriting from this abstract visitor.
VariableResolutionProblemChecker
NonMemberFunctionProblemChecker
WrongArgumentChecker
MemberFunctionProblemChecker
MissingOperatorChecker
TypeResolutionProblemChecker
PrivateMethodChecker
MissingConstructorChecker
MissingNamespaceChecker
extract
ExtractActionDelegate
ExtractRefactoring
extract is separate because it is not a Quick Fix. It is invoked via key binding.
linkedmode
ChangeRecorder
NestedEdit
Position

Figure 3.13.: Reduced class diagram of the entire plug-in

Checkers produce a problem marker which can be resolved by applying a Quick Fix . The Quick Fix in turn performs a certain refactoring, depending on the problem. Inside the refactoring the affected problem node is analyzed, a new node is created and a suitable insertion point is determined.

The refactoring returns a set of changes which is not yet performed. Changes are actually performed by the Quick Fix class inside a `ChangeRecorder`. The information gained by the `ChangeRecorder` is used to calculate offsets of newly inserted code for use with linked mode editing. See figure 3.16.

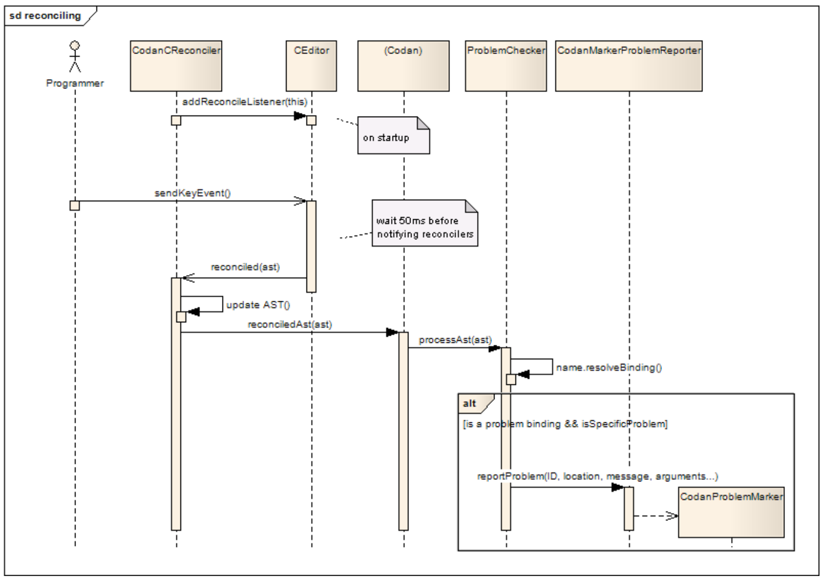


Figure 3.14.: The AST is updated after code changes, then checkers are invoked

3.5.4. Design Decisions

The choice of architecture is critical to performance and influences the flexibility of the Quick Fixes. The following sections discuss implications of the existing and chosen design.

More Logic for Codan Checkers

The need for more logic inside checkers actually rises from the nature of Quick Fixes. The method `isApplicable()` of the Quick Fix is called by the user interface for every displayed marker and therefore needs to be very responsive. Otherwise the GUI is frozen until it can display all the labels. This also means that access to the AST is not realistic because it is too slow to retrieve information about the situation. Therefore all information needed to display in the label of the Quick Fix is passed in inside the marker.

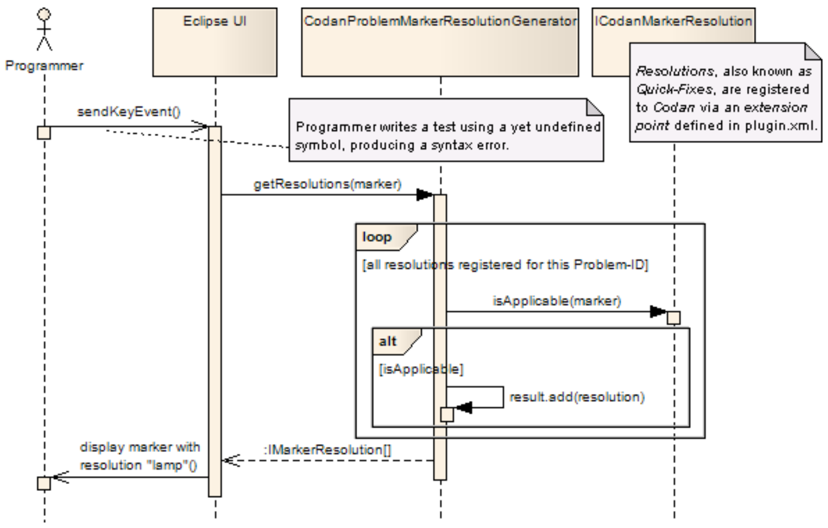

Programmer
Eclipse UI
CodanProblemMarkerResolutionGenerator
ICodanMarkerResolution
Resolutions, also known as Quick-Fixes, are registered to Codan via an extension point defined in plugin.xml.
sendKeyEvent()
Programmer writes a test using a yet undefined symbol, producing a syntax error.
getResolutions(marker)
loop
[all resolutions registered for this Problem-ID]
isApplicable(marker)
alt
[isApplicable]
result.add(resolution)
display marker with resolution "lamp"()
:IMarkerResolution[]

Figure 3.15.: Registered Quick Fixes are searched for each problem marker

Passing Information to the Quick-Fix

A Quick Fix needs to know at least two details: First, if it is applicable and second, what its label is for a specific marker. This information has to be passed to the Quick Fix as it should not access the AST itself.

A checker has three ways for passing information to a Quick Fix. Either the checker reports more granular problems, the `messagePattern` attribute is used or more information is passed via the problem markers argument list. Because the marker message is displayed to the user, it should not be used to transport internal details. Multiple problem ids for the same problem are confusing so it was decided to pass internal information via the open marker parameter list. Listing 3.4 shows how problems are reported inside a checker.

```
AbstractIndexAstChecker.reportProblem(
   <Error ID>,
   <IASTNode to be marked with red underline>,
   <hover message to be displayed>,
   [user defined arguments, ...]
);
```

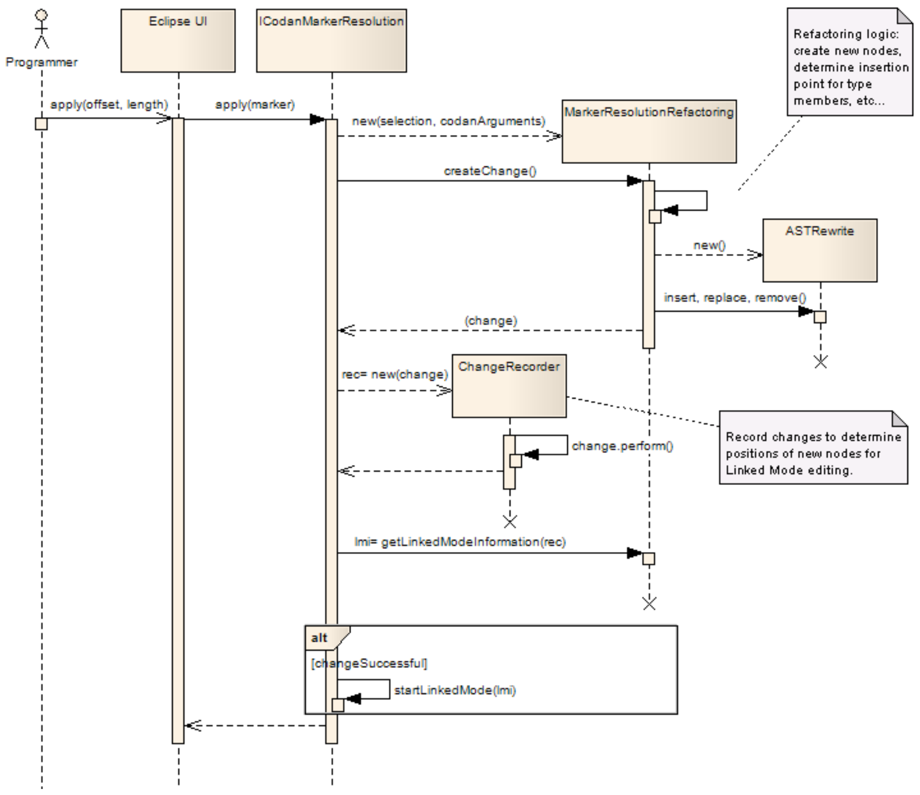

Programmer
Eclipse UI
ICodanMarkerResolution
MarkerResolutionRefactoring
ASTRewrite
ChangeRecorder
apply(offset, length)
apply(marker)
new(selection, codanArguments)
createChange()
new()
insert, replace, remove()
(change)
rec= new(change)
change.perform()
lmi= getLinkedModeInformation(rec)
alt
[changeSuccessful]
startLinkedMode(lmi)
Refactoring logic: create new nodes, determine insertion point for type members, etc...
Record changes to determine positions of new nodes for Linked Mode editing.

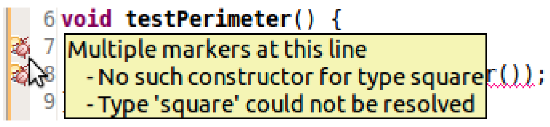
Figure 3.16.: User applies a Quick Fix

Listing 3.4: Reporting a problem inside a checker

Other Benefits of Extended Checkers

Checkers with more logic have other benefits too. As the first argument of a reported problem is used as the hover message of a marker, the checker is able to chose a more specific message. In addition, because the checker has access to the AST, it can already decide about the needed resolution strategy which allows to produce a more user friendly Quick Fix text.

Checkers cannot pass whole `ASTNodes` to the Quick Fix because if the AST is destroyed after a marker has been created, the marker still holds a reference to an `ASTNode` which does not exist anymore. This means that even if the AST is not valid anymore,



```
void testPerimeter() {
Multiple markers at this line
  - No such constructor for type square
  - Type 'square' could not be resolved
r());
```

Figure 3.17.: Hover messages are defined by the problem markers checker

there exists a marker with indirect references to the AST which cannot be freed by the garbage collector.

To prevent such a memory leak and access to an invalid AST, all arguments appended to a reported problem are serialized and passed as a string to the marker.

This has some consequences to the whole plug-in. Every problematic node found in a checker needs the be searched again in the same AST. Since most nodes have a selection, a position in the text of the source file, they can be retrieved again relatively easy. However passing additional information like other nodes in the context of the problematic node is not possible and should be avoided, because all non-serializable Information is lost between detecting the problem and reacting to it.

Keeping this in mind, gathering too much information in the checker is useless as long as it cannot be passed to the Quick Fix.

Also it makes it hard for sophisticated resolution generators to generate appropriate solutions to a problem. A solution maybe depends on the situation where the problem occurred. This leaves no other option than passing at least some of the context information as a string to the marker to search and generate a resolution.

Single QuickFix for Multiple Problems

The design of the Codan resolution generator implies that only one Quick Fix exists for a whole class of problems with the same problem id. Since the apply() method of a Quick Fix takes a marker as argument, one Quick Fix is basically enough.

However, the `getLabel()` method, as defined in the Eclipse framework (not CDT or Codan), does not take a marker as an argument. This makes it uncomfortable to display the name of a missing function or variable in the label. This is another problem from the fact that there is just a single instance of a Quick

Fix created by Codan. If every resolution would be newly created from a resolution generator, a label could be passed in the constructor of that

It is possible to back up the marker in the last `isApplicable()` call. This way, specific labels may be displayed even with only one Quick Fix per problem type. This is definitly a hack and fails if `isApplicable()` is not called directly before `getLabel()`.

MarkerResolutionGenerator for Append Argument

Add Argument needed its own marker resolution generator. Why? The problem with the resolution generator of Codan is, it can only produce one Quick Fix per problem. In case of multiple overloads, different Add Argument Quick Fixes have to be displayed. This could be solved by reporting the problem several times for each overload.

Disabled Codan Problems

Since a separate plug-in was developed and Codan checkers were duplicated to change their behavior, the old checkers are disabled. This prevents from the issue that some problems got reported twice. It is done inside the `Activator` class of the plug-in. There a listener waits for Codan to be loaded and then disables all Codan problems since checkers can only be disabled if all there to be reported problems were disabled. From this time on only the self written checkers are active and report problems.

3.5.5. Important Class Overview

Here is a short description about some classes used in the implementation and their duties.

Checkers

To write a checker, an AST-visitor[Gam95] is defined that resolves the binding for every `IASTName` inside the AST. If the binding cannot be resolved properly, the returned binding is of the type IProblemBinding, containing additional information about the problem. Depending on the type of the problem binding, problem markers with different problem ids are reported. A more detailed description is found in section 3.3.1 how checkers work.

QuickFixes

Each resolution action for a problem is implemented as a separate Quick Fix. This is the simple most way to decide which problem is solved by which Quick Fix, statically defined in the `plugin.xml`. The Quick Fix class itself is the glue between the reported problems and the logic to correct the problem. As a side effect it displays information about the correction to be done. Due to the fact that it is not straight forward to dynamically display information about the Quick Fix in the label various tricks were applied to this class. See section 3.5.4 how this is done.

CodanArgument

To achieve a proper refactoring some information is needed from the marker. Markers do support open argument lists to pass additional information. This information, no matter what kind of object, will be serialized. See section 3.3.1 for this marker problem.

A lot of problems came up with retrieving information with the wrong position from the argument map of the marker. To avoid this the `CodanArgument` class was created. It serves as an adapter before and after serialization. Passing information to `CodanArgument` will put the information in the correct order.To retrieve information from a marker, the marker is passed to the constructor of `CodanArguments` through which the information of the marker can be fetched.

Refactorings

The logic to correct the problem is done in a refactoring. Early versions of some refactorings used a text replacement method but were removed from our package since this is a quick and dirty way of correcting a problem.

The proper way to do this and how it is done now is trying to collect all needed information from the context of the AST, creating new nodes, insert them into the AST and let the AST infrastructure rewrite the code.

Quick Fix Generator

The static assignment between problems and Quick Fixes may be more memory efficient but raises a big problem. See section 3.5.4 With the `AddArgumentQFGenerator` it is shown how dynamically a Quick Fix is put in context with a problem.

Strategies

Creating the different kinds of functions, like operators or static functions would produce a lot of duplicated code as already mentioned in 3.4.3. For Example operators can be implemented as a member or as non-member function. The usual implementation approach to have such shared logik did not produce the expected result of a good readable code. It was decided to use a strategy based implementation which results in returning the various kinds of functions: Operators, static functions or normal functions. The actual refactoring then retrieves the created function an places it in the correct member or non-member context.

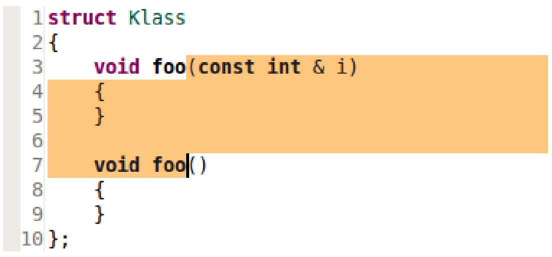
ChangeRecorder

To enable linked mode, exact positions of the newly inserted code elements have to be known. `ChangeRecorder` is responsible for determining these positions. Before this class was created, several means to find the correct locations were analyzed:

1. Search the document for the inserted name
2. Guess positions with the knowledge that code is inserted at a certain offset
3. Save a reference of the newly inserted element and read its location after insertion
4. Wait for the AST to be reconciled and search for the affected node
5. Analyze generated changes

The first is the fastest way to get the locations, however it is obvious that this solution will break as soon as another element has the same name. Item two was the first implemented approach which worked well in first place. However this solution was too unstable because the insertion point was too variable. The third approach failed due to the fact that newly inserted nodes are disposed after insertion and new nodes (with new references) are created by the parser when the AST is reconciled. Searching for the new node inside the reconciled AST was also tried. However, this solution introduced a delay of aound half a second before linked mode could be started which is not acceptable.

In the end, analyzing the generated changes was the last alternative. It introduces an indirect coupling to the `ChangeGenerator`. However, generated code must stick to syntax rules, so only whitespaces may change. Only one feature complicates extracting the changes: To avoid that `ChangeGeneratorVisitor` reformats the whole file, generated changes are reduced to the smallest necessary text edit. Most complexity of `ChangeRecorder` origins from the structure of these text edits. For example, figure 3.18 highlights the contents of a change during an overload of a member function.



```
struct Klass
{
    void foo(const int & i)
    {
    }

    void foo()
    {
    }
};
```

Figure 3.18.: Reduced text changes provided by `ChangeGenerator`

3.6. Testing

This project could not have been created without the use of tests. About 230 tests ensure correct functionality of the implemented features. This way, even at the end of the project, refactorings could be safely executed. In this section, the different kinds of tests run throughout the project are discussed.

3.6.1. Unit Tests

Unit tests profit from the advantage of being able to run without an Eclipse instance. They run through instantly, however not all functionality could have been tested this way.

Unit tests were used to ensure correct positions for linked mode editing. The changes created by the refactorings are subject to change and the mechanism that analyzes changes needs to be flexible especially in connection with whitespaces.

3.6.2. Refactoring Tests

Refactoring tests are ensuring correctness of the Quick Fix logic. Some of the tests are explicitly written to test compatibility with CUTE macros. To avoid importing all CUTE headers, a simplified CUTE structure was used together with the possibility to put them as external files and include them in the various tests if they need it. Earlier without the possibility of the external file test infrastructure, every definition of the specific cute macro was included in every test which needed the macro. Using the external file infrastructure reduced the test code for about 400 lines of code.

Later in the project it was noticed that these refactoring tests were very slow and slowed down the development a little bit. Waiting 120 second for tests to complete is no option since it leads to the behavior of postponing the tests.

However the project was analyzed with the *Testing and Performing Tools Platform* [Ecl11e]. It was discovered that opening a document in the IDE to retrieve the marker and the selection is very inefficient and slow. The code was refactored so that the document was created with the content of the tests and the selection was extracted directly from the Marker, rather than using the document.

With this refactoring, tests were sped up by about 30 percent.

3.6.3. Integration Tests

For each Quick Fix, an integration test checks for a specific code example whether

- markers of the appropriate type are generated
- the positions of the markers match the actual position of the problem
- the message of the marker is correct
- the Quick Fix message is correct
- the correct code is generated
- there is a displayed image in the Quick Fix label

Also later in the project the integration tests ran too slow. They were examined and it was detected that there is no need to run the indexer[Ecl11c] before every check on the marker position or a correct marker message. Running the indexer once before all tests is enough.

3.6.4. Tests on Large Projects - Performance

To test how the implemented checkers and Quick Fixes behave in a bigger project, Webkit[web10] was analyzed.

Webkit

Despite some rumors from mail on the CDT-mailing list[Esk11], Codan behaved quite performant in this test.

Various problems and bugs were found and fixed. Webkit makes intensive use of `friends`. To prevent a lot of false-positive messages a feature to not complain about accessing private members from outside if it is a friend class was implemented.

False-positive messages all about constructors were detected and could be fixed after the test. The problem resides in the fact that a declaration of a variable in a function can be constructor call weather a declaration inside type definition is not a constructor call.

However some limitation were discovered in context with templates. With multiple nested instances of `vectors` or other data structures using iterators and operators. What CDT does is trying to instantiate the templated classes during "coding time" and tries to resolve functions to see if they match. This functionality is needed for the auto completion but does only work to a certain level of template nesting. The Operator checker were not able to detect the defined operators on such a nested templated data structure like `vector<vector<pair`.

3.7. Known Issues

A lot of time was invested to avoid false positive on checkers and the generation of wrong code. But since no software is perfect and time is limited some issues remain unresolved.

3.7.1. Checker Performance

Static analysis is most helpful if it is done while typing. However, for big projects, checking the code for errors consumes too much time. Checkers may be turned off at the code analysis preferences page.

3.7.2. False Positives

It is possible that checkers are generating false positives - markers for problems that do not exist. Even if the compiler is able to build the code without errors, false positives will place red markers on affected source lines. If a specific checker is producing false positives, it may be turned off using the code analysis preferences page and a bug should be reported in Bugzilla[Ecl11b].

3.7.3. Chaos with Basic Types

During analysing code of types it was detected that in CDT there seems to be a chaos how types like `int, float` or `double` are defined. Especially with arguments it is important to compare two basic types.

One way of the definition is in the class `IASTSimpleDeclSpecifier`. There, basic types are defined as constant integers. Then there is the interface `IBasicType` which defines a new enum called `Kind` in which all the basic types are defined again. `BasicType` defines the kind of basic types for example for parameters

(ICPPParameter). BasicType was once used to return the same integer value as IASTSimpleDeclSpecifier but this getType() method is marked as deprecated.

So if one wants to compare a ICPPParameter with a IASTSimpleDeclSpecifier there is no chance not to run into a *marked as deprecated* warning.

3.7.4. Stopping at Typedef

CDT has good support for detecting and handling typedef. For example overwriting the output operator operator<<() as a free ouput operator, the stream to show the information usually is std::cout. Now the type of std::cout is defined as a std::ostream. However, std::ostream is a typedef which looks like this.

```
typedef basic_ostream<char> ostream;
```

Listing 3.5: typedef of std::ostream

This means when detecting the return type in a statement that looks like the following listing 3.6 it would insert a basic_ostream<char> for the operator<<() while digging out the required return type.

```
void test() {
  A a;
  std::cout << a;
}
```

Listing 3.6: Instance of type a should be send to the output

For sure, nobody does that and it looks ugly.

So, the easiest solution is to stop at the typedef and not dig deeper to the actual type.

This however brings up another problem. Consider the following code:

```
struct A {
};

typedef A X;

void test() {
  X a;
  Y a = a.giveMeA();
}
```

Listing 3.7: Typedef'ed class

Now when creating the `giveMeA()` function it expects a return type of `A`. But considering the typedef between the definition and the actual call, the result looks much more like the listing 3.8.

```
struct A {
  X giveMeA() {
      return *this;
  }
};

typedef A X;

void test() {
  X a;
  Y a = a.giveMeA();
}
```

Listing 3.8: Typedef'ed class returning itself

This dilemma was detected quite late and remained unresolved at the time of documenting.

3.7.5. AST Parser and Syntax Errors

When creating a whole templated class by writing the class name and the actual parameter type as template argument there is no problem in detecting the class name and the template parameter. See section 2.2.6

```
void test() {
  A<int> a;
}
```

Listing 3.9: Instantiating a templated class which does not exist

This however can be detected and a templated class can be created without a problem. See section **??**

If on the other side an instantiation of the following code in 3.10 is written, the Parser is unable to detect that this is a class instantiation with an non-type parameter. In this case the AST is build as: Problematic AST name 'A' operator<5 operator>.

```
void test() {
  A<5> a;
}
```

Listing 3.10: Instantiating a templated class with non-type parameter

Even stranger, if the class is defined as in listing 3.11, the parser has no problem detecting that it is a non-type parameter template instantiation. This problem remained unresolved.

```
template<int i>
class A { ... };
```

Listing 3.11: Class defined with a non-typed template argument

3.8. Solved Issues

This section explains some problem encountered during the project which could be solved.

3.8.1. Detecting Constructors

Detecting constructors and its callers is a little different than detecting unresolved normal functions or variables. Since there is no AST element identifying a constructor call, the AST must be carefully analyzed. To find a matching constructor which is already defined, there can be used a function called `findImplitlyCalledConstructor()` in the class `CPPSemantics`. This function takes a `Declarator` as an argument. However, having no constructor found does not mean that this is no constructor defined. `Declarators` do appear a lot in the AST. Therefore, the AST is checked on a specific combination of some AST elements which do only appear in constructor call. Then this combination is reported as call to a undefined constructor.

3.8.2. Detecting Operators

Like detecting constructors, detecting operators is a non-trivial business but a little easier.

Generally all expressions in an AST are analyzed. If such an expressions is ether of type `IASTUnaryExpression` of `IASTBinaryExpression`, then the implicit names of the expression is retrieved. The following code in listing 3.8.2 and 3.8.2 shows the relation between expressions and implicit names.

```
void test() {
  A a;
  a++;
}
```

Such a expression is actually nothing else than:

```

void test() {
  A a;
  a.operator++(int);
}
```

Since the definition of a operator looks like:

```
struct A {
  A operator++(int) {
  }
};
```

Such a *implicit name* is only found if an operator already exists. If no operator is found the name of the operator is copied with the help from the class `OverloadableOperator` and reported.

3.8.3. Deleting Function Creates no Marker

Detecting problems with missing names and other issues depends on resolving the corresponding binding for a name. Resolving names uses the index to check the bindings. If the indexer settings are not set correctly problems like missing functions were not detected. The indexer needs to be automatically updated.

3.8.4. Codan Marker Patch

While writing the Quick Fix for creating new functions, odd behavior was detected when macros were used. The reported problem marker position included the whole macro instead of the pure function name. This way it was difficult to determine the name of the function to be created. To find the correct name, the problem binding had to be searched again inside the Quick Fix . To solve this, it had to be found out why codan was reporting the wrong positions.

The `ProblemBindingChecker` of Codan relied on the `getRawSignature()` method of `IASTNode`, to determine the name of an affected node. However, if a node is contained inside a macro, `getRawSignature()` returns the text of the macro node instead of the actual name of the node. To solve the problem, `getSimpleID()` method of `IName` was used to get the name of the node. Another solution would have been to change all `getRawSignature()` methods which could have been very laborious.

The Problem has been reported in Bug 341089[Kal11] and a patch has been filed.

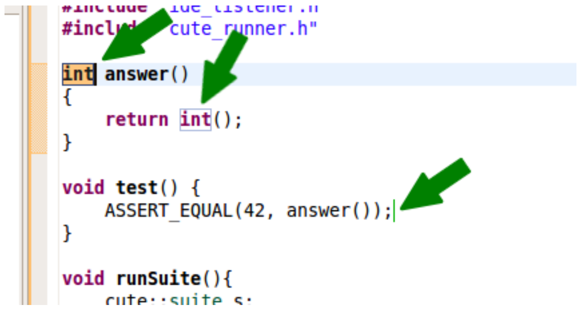
3.8.5. Create Linked Mode Positions

Whenever code is generated, the programmer may adapt some parts of the new code: Function parameters may be renamed, the return type may be changed or a `class` may be chosen in place of a `struct`. To enable this functionality a mechanism called linked mode is provided by Eclipse JFace.

Linked mode has three powerful features: First, it allows to edit identical text parts simultaneously. Second, it allows to jump between important edit positions. Last, an optional exit, position may be defined to determine the place where the editors cursor is place after the return key has been pressed.

Gathering the Text Positions

It is difficult to find the right positions of the newly inserted nodes in a timely manner. A straightforward solution would be to use the AST. However, the AST first has to recover from the perfromed changes and the linked mode has to be available to



```
#include "ide_listener.h"
#include "cute_runner.h"

int answer()
{
    return int();
}

void test() {
    ASSERT_EQUAL(42, answer());
}

void runSuite(){
    cute::suite s;
```

Figure 3.19.: Two linked mode positions and an exit position

the user immediately. Guessing the number of whitespaces would not provide a stable solution. To have immediate access to the positions, the text changes provided by the rewriter[Gra06] are analyzed to determine the linked mode positions.

The problem has been solved with a class `ChangeRecorder` that wraps the original changes provided by the rewriter. Using the size of the inserted text and the name of the inserted node, the `ChangeRecorder` is able to calculate the code positions for the linked mode.

4. Conclusions, Interpretation and Future Work

A lot of features have been implemented, problems have been encountered and experiences have been collected throughout this thesis. This chapter summarizes what has been reached and where future potential is possible.

4.1. Conclusions

It was encouraging that even during implementation, Quick Fixes were already helpful for writing new test cases. However, what has been done cannot be regarded as "complete" TDD support. There is still a lot of work to do and functionality which is not obviously supportive for TDD is also needed to provide a full set of Quick Fixes for writing software test-first. For some features, it is difficult to decide whether they support a test-first or test-last approach, as the discussion around *Change Argument* versus *Change Parameter* showed. In the end, every programmer will use the provided Quick Fixes in its own way.

As `JDT` demonstrates, there is far more left to do than the few Quick Fixes implemented during this bachelor thesis. As the choice of implemented features has impact on code quality and on checker execution time, future projects should think about what should be implemented and what not. At this point, we would like to refer to the "assignment-to-itself-checker".

Execution speed during manual and automated testing was satisfying, even for large projects like Webkit. However complexity grows linearly as more checkers will be added and at some point, one will have to decide which checkers are really needed.

The TDD for CDT plug-in supplements the *Toggle Function Definition* [KS10, Som11a] which has been developed in 2010. Both features in combination are a powerful tool for fast development.

4.2. Interpretation

This section reflects the opinion of the authors about the developed project.

4.2.1. Limitations

There was a big potential for supporting special cases that will never be used. Therefore, focus has been set on implementing basic functionality for a broad set of Quick Fixes . Special cases may be added later if they are needed. Hence, it is possible that checkers will produce false positives or false negatives for some code.

Add Argument has been artificially limited by inserting "_" instead of a default constructor even if it was possible to determine such a default value. We think, this simplification is needed because the programmer should program deliberately. In Java, the Quick Fixes that insert `null` references is a dangerous feature and we are convinced that the selection and design of Quick Fixes has influence on clean code.

At last, the fact that own checkers have been implemented that in some cases duplicate functionality from Codan will make integration into CDT harder. CDT community has to be convinced that these checkers are needed to provide meaningful labels for both markers and Quick Fixes .

4.2.2. Personal Review

We remember that this section was the first one we have read when we were browsing references. Following a short and personal summary of the authors for future CDT plug-in thesis students.

Martin Schwab

Working with Eclipse is challenging because it is such a large project. A lot of functionality already exists at some place and may be reused. In some cases, e.g. the definition lookup mechanism, code is available but bound together with other functionality so it cannot be reused. This inspired a lot of discussions about code smells, refactoring and architecture which was very interesting.

Although after a while, adding new Quick Fixes was pure work, new challenges had to be solved throughout the project, especially with *Add Argument*.

Personally, I enjoyed working with legacy code over designing a new system from scratch. Eclipse is decorated with concrete implementations of patterns and a lot can be learnt by browsing its code. Although there is still is a long way to go to have functionality like in JDT, I would be happy if with this project a first stone had been laid down for future Quick Fixes in CDT.

Thomas Kallenberg

As in the semester thesis half a year ago, the productive environment with my team member and the good relationship with the supervisor and the IFS team made fun. Having a deeper look inside the CDT team through the mailing list [?] gave an extra motivated feeling to be part of the development cycle of the Eclipse CDT. As we heard that our semester thesis was merged in CDT 8 we were much more motivated to bring good results in this thesis too.

What came out was mostly hard work. The most difficult part was to connect the multiple parts which were already available in Eclipse to a unit we needed. But unlike the semester thesis we could compare to the JDT infrastructure providing code assistance for Java and learn from their mistakes. Very funny to see the selection algorithm behind JDT proposals and how primitive it is working in some cases. But much more shocking if you think about how much more functionality and logic is needed to provide better results.

4.2.3. Insights

For future projects, in this section we collected positive and negative experiences throughout this bachelor thesis.

Co-location was encouraged by HSR by providing fast development machines and a big desk. This was very productive and written project management could be reduced to a minimum.

Redmine was used to track issues and as information radiator for project supervisors. We have spent as few time as possible with this tool, and the mechanism to refer to a certain issue inside the commit message (e.g. `refs #123`) was a big help to reach this goal of avoiding double work. What we would definitely not do is writing documentation inside the wiki because all has to be moved to the actual documentation at a later time. We tried to begin the documentation earlier this time, but we had to rewrite a lot of it, so writing the documentation in the beginning provides no bigger benefit.

Git was used for revision control and this time, only a master and one development branch were used constantly. By using rebase, a lot of merge commits could be avoided, resulting in a cleaner repository history.

At last, for any project that we will work with in future, we will follow the guideline *Cheap desk, nice chair* from [Bec03].

4.3. Future Work

For TDD support, a subset of possible Quick Fixes was selected and implemented. However, a lot more Quick Fixes could be used for daily programming and also for TDD. For interested students, this is a field with a lot of potential and it is a thankful work because it is really useful.

Concerning this plug-in, some work will be needed to bundle it with CUTE and more work to integrate it into CDT. An interesting statistical work that was not done during this project due to will be to measure the time saved by the Quick Fixes . A striking difference in speed was noticed during writing tests, however, no numbers have been collected to be able to promote a speed-up of twenty percent. Even if no time is saved, the plug-in was worth its effort because of less typing. It will be interesting to follow up future improvements for TDD and Quick Fixes.

A. User Manual

This chapter explains the installation and usage of the plug-in for Eclipse.

A.1. Installation of the Plug-in

This plug-in was developed for Eclipse CDT v8.0.0 using Eclipse Indigo M5. Follow these instructions to use the plug-in:

- Get a recent Eclipse Indigo (3.7) at
  `http://download.eclipse.org/eclipse/downloads/`
- Install a recent CDT v8.0.0 build, available at
  `http://download.eclipse.org/tools/cdt/builds/8.0.0/`
- Install the Tdd4Cdt plug-in using the update site at
  `http://sinv-56042.edu.hsr.ch/updatesite`
- Optionally, the CUTE unit testing framework may be installed using the update site at
  `http://sinv-56042.edu.hsr.ch/downloads/cute/`
  Please notice that this is a pre-release of CUTE. The publicly available version did not yet run on Indigo at the time of this writing.

Be aware that the update site is hosted on a virtual server that will be shut down at the end of the semester. A copy of the update site is available on the attached CD. Eclipse update site supports local paths: use the path
`file:/mnt/cdrom/tdd/ch.hsr.eclipse.update/target/site`

A.2. Building The Source Code

To build the source code, Maven 3 is needed, available at `http://maven.apache.org/download.html`.
Copy the git repository stored on the attached CD and enter the directory `ch.hsr.eclipse.parent`. To build the project, run

```
mvn clean install
```

After a successful build, the plug-in may be installed using the update site from the `ch.hsr.eclipse.update/target/site` directory.

Documentation is available in directory `ch.hsr.eclipse.doc`. To build the documentation, run

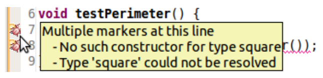
```
mvn clean latex:latex
```

A.3. Using the Quick Fix

Quick Fixes are available whenever a bug (A.1a) is displayed together with a Quick Fix lamp (A.1b). If the mouse pointer is held above the bug symbol for a moment, a hover message (A.1c)lists all problems at the current line.

(a) Bug

(b) Bug with Quick Fix



```
void testPerimeter() {
Multiple markers at this line
  - No such constructor for type square
  - Type 'square' could not be resolved
```

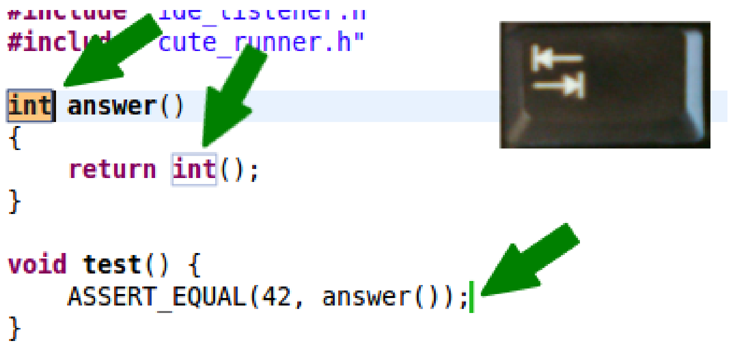
(c) Hover message

Figure A.1.: Display of markers and Quick Fixes

Quick Fixes are context sensitive: Before calling the Quick Fix menu, place the cursor on the element to be fixed. The cursor automatically jumps to the nearest problem on the current line. To open the menu with a list of Quick Fixes , press the `Ctrl-1` key combination or select Quick Fix from the *"Edit"* menu in Eclipse. The Quick Fix is applied by pressing the `return` key or with a double-click on the appropriate Quick Fix.

A.3.1. Editing Generated Code

Some Quick Fixes offer the option to choose another symbol name or a more specific return type. For this purpose, linked mode editing is started after the code has been generated. See figure D.4 for an example of linked mode editing. The tabulator key is



```
#include "cute_runner.h"

int answer()
{
    return int();
}

void test() {
    ASSERT_EQUAL(42, answer());
}
```

Figure A.2.: Usage of linked mode after code generation.

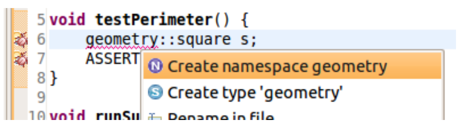
used to switch to the next marked position whereas the escape key exits linked mode at the current selection. Pressing the return key at any time jumps back to the initial position which is indicated by a green bar.

Whenever focus is lost or a region outside of a linked text is edited, linked Mode exits automatically. Further Quick Fixes may be applied without having to exit linked mode.

A.3.2. Available Quick Fixes

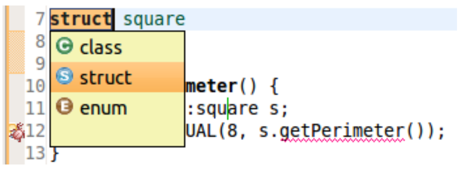
The following Quick Fixes are available. Each Quick Fix is shown together with a code example.

- Create new types and namespaces



```
void testPerimeter() {
    geometry::square s;
    ASSERT
}

void runSu
```

Figure A.3.: Choice between a new class and a new namespace



```
struct square

meter() {
:square s;
UAL(8, s.getPerimeter());
}
```

Figure A.4.: Dropdown list on a newly created type definition

- Add member functions, -variables, constructors and operators

```cpp
struct square {};
void test() {
  square s1(2), s2(3);
  s1.size = s2.size;
  ASSERT_EQUAL(16, s1 + s2);
}
```

- Create Local Variable

```cpp
void test() {
  ASSERT_EQUAL(16, square1);
}
```

- Add arguments to match a certain candidate function

```cpp
double add(int first, square second);
double add(double first, double second);
void test() {
  ASSERT_EQUAL(16, add(8));
}
```

- Change visibility of type members

```cpp
class square {
  bool privateMember();
};
void test() {
  square s;
  ASSERT(s.privateMember());
}
```

A.4. Extracting Types and Functions

The *Extract Type or Function* feature may be executed with the key `Alt-Shift-P`. To be applicable, the current selection has to be inside a type or function definition.

```cpp
struct square {
  square(double s): side(s) {}
  int getPerimeter() { // Current cursor position
    return side*side;
  }
private:
  int side;
};

void test() {
  square s;
  ASSERT(8, s.getPerimeter());
}
```

Listing A.1: Code before type extraction

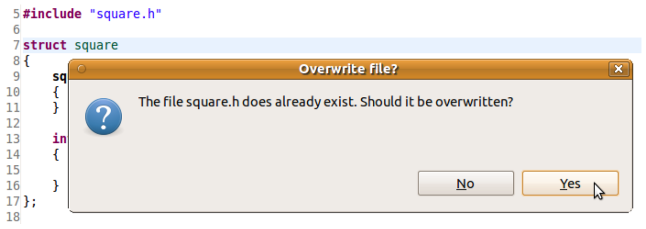
```cpp
#include "square.h"

void test() {
  square s;
  ASSERT(8, s.getPerimeter());
}
```

Listing A.2: Code after class `square` has been extracted to a file

Member functions cannot be extracted separately. Their corresponding outmost type definition is extracted instead. Namespaces are not extracted. Listings A.1 and A.2 show a before and after example of type extraction.

A.4.1. Target-Destination Conflicts

Types and functions are always extracted to a header file with the same name as the affected declaration. If a header file with the same name already exists, the user is asked whether she wants



```
#include "square.h"

struct square
{
    sq
    {
    }

    in
    {

    }
};

```

Overwrite file?

The file square.h does already exist. Should it be overwritten?

Figure A.5.: Overwrite dialog if destination file already exists

to overwrite the old file. Moving a declaration into a header file with another name is not implemented.

By accepting this message, the whole header file is overwritten.

B. Extension Manual

This chapter describes how to write new Quick Fixes for instant source code correction and how to test them.

B.1. Checkers

Using the normal way of Codan checkers should be fine. This means extending the `AbstractIndexAstChecker` class and process the AST with a visitor. If one should react on a problem with a unresolved name it is probably best to modify the `ProblemBindingChecker` and integrate them into the official CDT.

The glue between a reported problem and the appropriated Quick Fix is the problem id. In the `plugin.xml` the binding of a Quick Fix and the problem id is made.

For more information on how to configure Checkers and register them refer to section 3.3.2 and 3.3.1.

B.2. Quick Fix

Writing a Quick Fix has been made easy by extending the `RefactoringLinkedModeQuickFix` and implementing the appropriate methods. Most important is the `getRefactoring()` method where the actual code refactoring is created.

Second there is the `configureLinkedMode()` method were groups and positions of the linked mode can be set to edit the code later. To add a group add the position and the length of the element. The delivered `ChangeRecorder` helps to detect changed the code-parts after editing.

If a list of proposals should be presented to the programmer, there is the `addProposal()` method in the passed in `LinkedModeInformation`.

B.3. Refactoring

The returned refactoring in the Quick Fix is of the type `MarkerResolutionRefactoring` which is a `CRefactoring2` with some small changes.

B.4. Tests

Tests are important in this area where small changes can produce wrong code. So there are two kinds of tests.

B.4.1. Refactoring Tests

First there are Refactoring tests. Extending `MarkerResolutionRefactoringTest` provides an infrastructure to test code which produces markers and applies the appropriate Quick Fix. The code and the expected outcome is written in a `.rts` file.

B.4.2. Quick Fix Tests

In contrast to the refactoring tests, the Quick Fix tests do a more detailed level of testing. Here various other properties can be tested like the marker length or the marker text.

C. Project Setup

C.1. Configuration Management

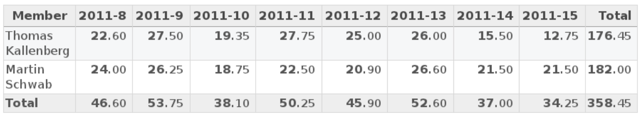
The configuration management was exactly the same as in the semester thesis [KS10] with minor differences between some versions of the used software.

D. Time Management

Focus in this chapter is the time management and presents some charts explaining where the time was spent and by whom.

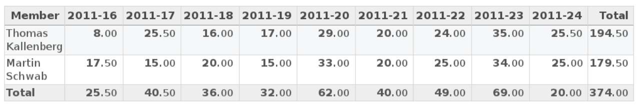
D.1. Time Budget

Compared to the previous semester thesis [KS10] one change in the organization was done to prevent the issue thesis where time was not logged for three or four weeks. Since the Redmine project management tool [red10] is able to analyze a commit message, commit messages can be linked to an issue. Putting the spent time on a issue in that commit message together with the issue number it was possible to massively reduce logging effort. Hence time logging could be done while working on issues directly without the risk of postponing it indefinitely. In the last week some time was taken on credit since this chapter has to be written and the time chart was freezed.



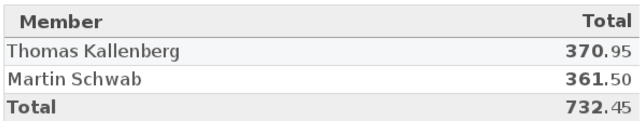
Member	2011-8	2011-9	2011-10	2011-11	2011-12	2011-13	2011-14	2011-15	Total
Thomas Kallenberg	22.60	27.50	19.35	27.75	25.00	26.00	15.50	12.75	176.45
Martin Schwab	24.00	26.25	18.75	22.50	20.90	26.60	21.50	21.50	182.00
Total	46.60	53.75	38.10	50.25	45.90	52.60	37.00	34.25	358.45

Figure D.1.: Used time for the first part



Member	2011-16	2011-17	2011-18	2011-19	2011-20	2011-21	2011-22	2011-23	2011-24	Total
Thomas Kallenberg	8.00	25.50	16.00	17.00	29.00	20.00	24.00	35.00	25.50	194.50
Martin Schwab	17.50	15.00	20.00	15.00	33.00	20.00	25.00	34.00	25.00	179.50
Total	25.50	40.50	36.00	32.00	62.00	40.00	49.00	69.00	20.00	374.00

Figure D.2.: Used time for second part

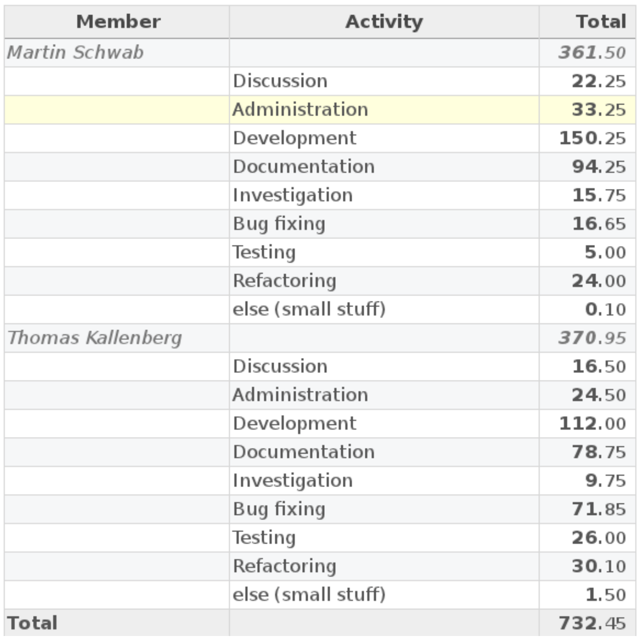


Member	Total
Thomas Kallenberg	370.95
Martin Schwab	361.50
Total	732.45

Figure D.3.: Used time by team members

D.1.1. Activities

Most time was spent on implementation and documentation. Bug-fixing took some time too. Administration took quite some time too since the process of creating an issue in the wiki, estimating the required time to complete, chooseing a category and logging time afterwards was quite time consuming.



Member	Activity	Total
Martin Schwab		*361.50*
	Discussion	22.25
	Administration	33.25
	Development	150.25
	Documentation	94.25
	Investigation	15.75
	Bug fixing	16.65
	Testing	5.00
	Refactoring	24.00
	else (small stuff)	0.10
Thomas Kallenberg		*370.95*
	Discussion	16.50
	Administration	24.50
	Development	112.00
	Documentation	78.75
	Investigation	9.75
	Bug fixing	71.85
	Testing	26.00
	Refactoring	30.10
	else (small stuff)	1.50
Total		732.45

Figure D.4.: Used time by categorie

After all the team thinks that time management with Redmine is quite time consuming. Planing over more than one week and even visualize it is nearly impossible. Here a spreadsheet or something that is faster to administrate would be a better choice because switching between the two web pages, the visualization and the issue manipulation page of Redmine is tedious.Many bugs in the visualization graph prevent it from using it properly as a planing and visualization tool resulting in loosing time without any benefit.

D.2. Declaration of Originality

Ich erkläre hiermit,

- dass ich die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt habe, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass ich sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben habe.[HSR11]

Ort, Datum:

________________________, ________________________

Thomas Kallenberg

____

Martin Schwab

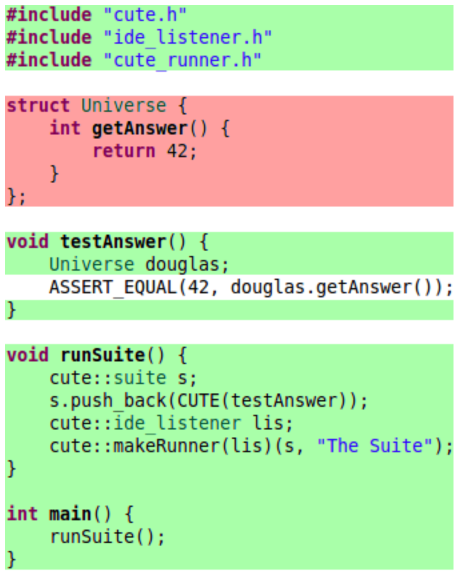
____

Task Agreement for Bachelor Thesis: TDD for CDT

During the bachelor thesis in spring semester 2011, a plug-in for Eclipse CDT is developed. It has to provide several features that facilitate the process of Test-Driven Development (TDD), especially writing CUTE tests. This is achieved by offering functions to generate appropriate code where code is missing. In the TDD life cycle of *Red – Green – Refactor – Integrate*, this project is dedicated to speed up the process of satisfying the compiler with syntactically correct code.

Current Situation

Test-Driven Development has become a very useful and important software development process in modern software development. Fast and safe Test-Driven Development needs support by the development environment. Eclipse JDT, a development environment for Java does offer efficient and intuitive support for TDD. The horizon of this bachelor thesis is to develop TDD functionality for Eclipse CDT. Test automation in Eclipse CDT is already possible using a separate plug-in like CUTE. This plug-in supports generating new test code. Adapting or creating code starting from parse errors as is useful for TDD is not yet supported. The figure on the right visualizes which code may already be generated by CDT.



```
#include "cute.h"
#include "ide_listener.h"
#include "cute_runner.h"

struct Universe {
    int getAnswer() {
        return 42;
    }
};

void testAnswer() {
    Universe douglas;
    ASSERT_EQUAL(42, douglas.getAnswer());
}

void runSuite() {
    cute::suite s;
    s.push_back(CUTE(testAnswer));
    cute::ide_listener lis;
    cute::makeRunner(lis)(s, "The Suite");
}

int main() {
    runSuite();
}
```

Green: Code may be generated.
Red: No automation available.

Basic Objectives

Whenever a user writes a test for a program element that does not exist yet, a parse error is produced and the code is underlined by a red waved line by the IDE. Starting from this problem marker which is already part of the functionality in CDT 8, it shall be possible to generate the missing code using a simple interface, ideally a keystroke. The task may be split up into the following subtasks:

- Automation of (member) function creation
- Automation of class creation (in current file)
- Offer above as quick-fixes for corresponding error markers
- Provide technical documentation with a user guide

Also included in the basic objectives is support for changing the signature so the following items should work.

Glossary

activator a class inside Eclipse plug-ins that controls the lifecycle of plug-ins, see OSGi[Ecla].

autocomplete context sensitive text assistance feature to complete previously defined words.

candidate in case of an abmiguous function call, a proposal of a function with the same name but other arguments.

commit incremental change to a project under version control.

compiler program that transforms source code into machine executable representation.

constructor as defined in the C++ specification [ISO11] *12.1*.

CUTE C++ unit test framework developed by Prof. Peter Sommerlad (supervisor of this thesis).

Eclipse CDT C/C++ Development Tooling, an Integrated Development Environment for C++.

friend a function or class that is given permission to use the private and protected meber names from the class.[ISO11] *11.4*.

garbage collector automatic memory management, tool for freeing unused memory.

header file a file with extension .h or .hpp that contains forward declarations that may be embedded in source files.

implicit name is used to resolve uses of implicit bindings, such as overloaded operators.[K$^+$11].

initializer list a way to initalize variable in classes as defined in the C++ specification [ISO11] *18.9*.

Integrated Development Environment smart text editor with tools optimized for software development.

keywords reserved character sequence in programming language, for C++ defined in [ISO11], appendix A.1.

linked mode editing a jFace feature which simplifies editing groups of text.

macro is a preprocessor directive as defined in the C++ specification [ISO11] *16.3*.

member function a member function, as defined in the C++ specification [ISO11] *17.6.4.5*.

namespace container for declarations in C++, as defined in the C++ specification [ISO11] *7.3*.

operator similar to functions, may be overloaded as described in the C++ specification [ISO11] *13.5*.

overloading as defined in the C++ specification [ISO11] *13*.

parameter value that is accepted by a function declaration in case of a call.

parse error even with correct syntax, an error can occur because of wrong semantic.

parser tranforms source code into other representation formats.

plug-in used to extend an existing system with new functionality, Eclipse CDT in this case.

pointer as defined in the C++ specification [ISO11] *8.3.1*.

problem marker are displayed inside an Eclipse editor, usually with a red bullet. Markers are registered to a certain location in code and may be underlined with color, by default yellow for warnings and red for errors.

proposal menu editable dropdown list, displayed during linked mode editing, for quick access of predefined options.

quick fix context-sensitive menu which may be called by pressing `Ctrl-1` or via *Edit → Quick Fix*.

refactoring functionality for modifcation of source code. In this context, a refactoring is a subclass of CDT `CRefactoring2` class but not a refactoring in the sense of [Fow99].

rewriter is used to make modify the source code using the AST.

scope as defined in the C++ specification [ISO11] *3.3*.

serialize process of expressing a data structure as a stream of data.

shadowing occurs when a variable declared within a certain scope (decision block, method, or inner class) has the same name as a variable declared in an outer scope. This outer variable is said to be shadowed [Wik10b].

static code checker tool that scans an Abstract Syntax Tree for potential problems. It is called static because it does not rely on compiled source.

symbol an abstraction, tokens of which may be marks or a configuration of marks which form a particular pattern [Wik10a].

template arguments as defined in the C++ specification [ISO11] *14.3*.

template parameter as defined in the C++ specification [ISO11] *14.1*.

templated class a family of classes or functions or an alias for a family of types. [ISO11] *14*.

Test-Driven Development a development guideline which promotes designing software test-first.

unit testing defines fine granular constraints for source code to ensure expected behavior.

wizard modal dialog collecting user decisions for complex tasks.

Bibliography

[Bau10] Sebastian Bauer. *Eclipse für C/C++ Programmierer*. dpunkt, 11 2010.

[Bec03] Beck, Kent. *Test-Driven Development by Example*. Pearson Education, Inc., 2003.

[Ecla] Eclipse. Equinox – OSGi. http://www.eclipse.org/equinox/.

[Eclb] Eclipse SDK Documentation. Platform Plug-in Developer Guide – Contributing Marker Resolution. `http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/guide/wrkAdv_markerresolution.htm`. [Online; accessed 01-March-2011].

[Ecl10a] Eclipse Bugs. Bug 309760: Provide checker for name resolution problems found by indexer. `https://bugs.eclipse.org/bugs/show_bug.cgi?id=309760`, 2010. [Online; accessed 08-March-2011].

[Ecl10b] Eclipse Bugs. Bug 319196: Quick fixes for creating a variable. `https://bugs.eclipse.org/bugs/show_bug.cgi?id=319196`, 2010. [Online; accessed 08-March-2011].

[Ecl11a] Eclipse. Eclipse. `http://www.eclipse.org/`, 2011. [Online; accessed 15-June-2011].

[Ecl11b] Eclipse. Eclipse Bugzilla. `http://bugs.eclipse.org`, 2011. [Online; accessed 15-June-2011].

[Ecl11c] Eclipse. Eclipse CDT Indexer. `http://help.eclipse.org/indigo/topic/org.eclipse.cdt.doc.isv/reference/extension-points/`

org_eclipse_cdt_core_CIndexer.html, 2011. [Online; accessed 15-June-2011].

[Ecl11d] Eclipse. Eclipse JFace. http://wiki.eclipse.org/index.php/JFace, 2011. [Online; accessed 15-June-2011].

[Ecl11e] Eclipse. Eclipse Test and Performance Tools Platform (TPTP). www.eclipse.org/tptp, 2011. [Online; accessed 14-June-2011].

[Ecl11f] Eclipse. Extension Point. http://www.eclipse.org/resources/?category=Extension%20points, 2011. [Online; accessed 15-June-2011].

[Ecl11g] Eclipse CDT Wiki. CDT/designs/StaticAnalysis. http://wiki.eclipse.org/CDT/designs/StaticAnalysis, 2011. [Online; accessed 08-March-2011].

[Esk11] Jesper Eskilson. [cdt-dev] Codan performance. http://dev.eclipse.org/mhonarc/lists/cdt-dev/msg21516.html, 2011. [Online; accessed 15-June-2011].

[Fow99] Martin Fowler. *Refactoring, Improving the Design of Existing Code.* Addison-Wesley, 1999.

[Gam95] Gamma, Erich; Helm, Richard; Johnson, Ralph E. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Prentice Hall, 12 1995.

[Gra06] Leo Graf, Emnuel; Buettiker. C++ Refactoring Support for Eclipse CDT. http://leo.freeflux.net/blog/archiv/c-refactoring-support-fuer-eclipse-cdt.html, 2006. [Online; accessed 15-June-2011].

[HSR11] Hochschule für Technik Rapperswil HSR. Muster einer Erklaerung ist integrierender Bestandteil der SA BA DA. HSR internal document: Muster_einer_Erklaerung__ist_integrierender_Bestandteil_der_SA_BA_DA.doc, 2011. [Online; accessed 15-June-2011].

[ISO11] WG21 ISO/IEC. *IS 14882: Programming Languages – C++*. International Organization for Standardization, Geneva, Switzerland, March 2011.

[JUN11] JUnit.org – Resources for Test Driven Development. `http://www.junit.org/`, 2011. [Online; accessed 08-March-2011].

[K$^+$11] Mike Kucera et al. CDT source code, class `org.eclipse.cdt.core.dom.ast.IASTImplicitName` in project org.eclipse.cdt.core, 2011. SVN version v201105161139.

[Kal11] Thomas Kallenberg. Bug 341089 – Codan does not handle missing functions in macros correctly. `https://bugs.eclipse.org/bugs/show_bug.cgi?id=341089`, 2011. [Online; accessed 15-June-2011].

[KS10] Thomas Kallenberg and Martin Schwab. One touch C++ code automation for Eclipse CDT. `http://eprints3.hsr.ch/158/1/OneTouchToggleRefactoring.pdf`, December 2010. [Online; accessed 06-June-2011].

[Mic05] Sun Microsystems. Top Level Type Declarations. `http://java.sun.com/docs/books/jls/third_edition/html/packages.html#26783`, 2005. [Online; accessed 15-June-2011].

[Mic11] Microsoft Developer Network. MSDN Library - C++ Language Reference - Unions. `http://msdn.microsoft.com/en-us/library/5dxy4b7b%28v=VS.100%29.aspx`, 2011. [Online; accessed 21-Mai-2011].

[red10] Redmine – Project Management Web Application (Version 0.9.3-1). `http://www.redmine.org/`, 2010. [Online; accessed 22-December-2010].

[Som08] Prof. Peter Sommerlad. C++ Refactoring and TDD with Eclipse CDT. `http://accu.org/index.php/conferences/accu_conference_2008/`

accu2008_sessions#C++%20Refactoring%20and%20TDD%20with%20Eclipse%20CDT, 2008. [Online; accessed 15-June-2011].

[Som11a] Prof. Peter Sommerlad. Agile C++ Through Advanced IDE Features. http://accu.org/index.php/conferences/accu_conference_2011/accu2011_sessions#Agile%20C++%20through%20Advanced%20IDE%20Features, April 2011. [Online; accessed 15-June-2011].

[Som11b] Sommerlad, Prof. Peter. CUTE – C++ Unit Testing Easier. http://www.cute-test.com/, 2011. [Online; accessed 01-March-2011].

[Sut09] Herb Sutter. Const-Correctness. http://gotw.ca/gotw/006.htm, 2009. [Online; accessed 15-June-2011].

[Tho05] Andrew Thomas, David; Hunt. *The Pragmatic Programmer – From Journeyman to Master*. Addison-Wesley Longman, Amsterdam, 01 2005.

[web10] The WebKit Open Source Project (Version SVN head: r71799). http://webkit.org/, 2010. [Online; accessed 22-December-2010].

[Wes11] Tomasz Wesolowski. Bug 319196 - Quick fixes for creating a variable. https://bugs.eclipse.org/bugs/show_bug.cgi?id=319196, 2011. [Online; accessed 10-June-2011].

[Wik10a] Wikipedia. Symbol (formal). http://en.wikipedia.org/wiki/Symbol_%28formal%29, 2010. [Online; accessed 15-June-2011].

[Wik10b] Wikipedia. Variable Shadowing. http://en.wikipedia.org/wiki/Variable_shadowing, 2010. [Online; accessed 15-June-2011].