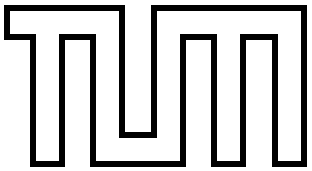


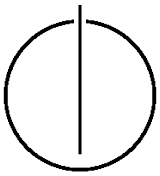
FAKULTÄT FÜR INFORMATIK

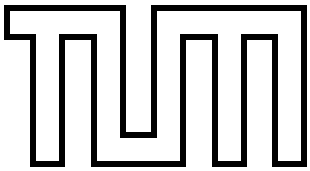
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master Thesis in Informatics

Adding C++ Support to MBEDDR

Zaur Molotnikov





FAKULTÄT FÜR INFORMATIK

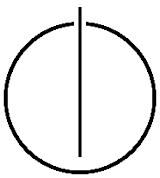
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master Thesis in Informatics

Adding C++ Support to MBEDDR

C++ Unterstützung für MBEDDR

Author:	Zaur Molotnikov
Supervisor:	Dr. Bernhard Schätz
Advisor:	Dr. Daniel Ratiu
Date:	September 15, 2013



Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. September 2013 Zaur Molotnikov

Acknowledgments

If someone contributed to the thesis... might be good to thank them here.

Abstract

An abstracts abstracts the thesis!

Contents

Acknowledgements vii

Abstract ix

Outline of the Thesis xiii

I. Introduction and Theory 1

1. Introduction 3
1.1. Projectional Editing 3
1.1.1. Traditional Approach 3
1.1.2. Projectional Approach 4

Appendix 7

A. Detailed Descriptions 7

Bibliography 9

Outline of the Thesis

Part I: Introduction and Theory

CHAPTER 1: INTRODUCTION

This chapter presents an overview of the thesis and it purpose. Furthermore, it will discuss the sense of life in a very general approach.

CHAPTER 2: THEORY

No thesis without theory.

Part II: The Real Work

CHAPTER 3: OVERVIEW

This chapter presents the requirements for the process.

Part I.

Introduction and Theory

1. Introduction

Here I introduce the main concepts and existing work, relevant to this Master Thesis.

1.1. Projectional Editing

1.1.1. Traditional Approach

Traditionally programming languages are used in a textual form in text files, forming programs. However the textual nature is not typical for the structure of programs themselves, being rather low-level representation of code. Parsers are used to construct so called abstract syntax trees (ASTs) from the textual program representation. ASTs are structures in memory, usually graph-alike, reminding sometimes control flow graph, where nodes are different statements and edges are the ways control passes from one statement to the next one.

For the developer, using an editor, the degree to which the editor can support the development process is important. For this, the editor has to recognize the programming language construction and provide possible assistance. Among such assistance can be code formatting, syntax validation, source code transformations (including refactoring support), code analyses and verification, source code generation and others. Many of these operation rely indeed on the higher than text level notions related to program such as method, variable, statement. A good editor has to be aware of these higher level program structure.

Nowadays most of the editors work with text, and to provide assistance to programmer integrate with parser/compiler front-end for the programming language. This way to extract the program structure during editing is not perfect for several reasons. First of all, the program being edited as text is not syntactically correct at every moment, being incomplete, for example. Under this circumstances the parsing front-end can not be successfully invoked and returns error messages which are usually false-positive. Secondly, after minor editing of the code, usually the whole text file has to be processed again. Such compiler calls are usually computationally expensive, they slow down, sometimes significantly, the performance of the developer machine. Various techniques exist to speed it up, including partial and pre- compilation, but the problem is still relevant to large extent.

Moreover, the textual nature of the code complicates certain operations additionally. As an example, we can take a refactoring to rename a method. Every usage of the method, being renamed, has to be found and changed. To implement it correctly an editor must take into account various possible name collisions, as well as presume a compilable state of the program to even start the refactoring.

Not to mention the parsing problem itself. Parsing a program in a complex language like C++ is a difficult problem, it involves the need to resolve correctly scoping and typing, templates and related issues, work with pre-processor directives incorporated in the code.

In this regard different compilers treat C++ in a different way, creating dialects, which may represent obstacles for the code to be purely cross-platform.

Listing 1.1: Closing several blocks

```
class MyClass {
  void doSomething() {
    while(true) {
      try {
        // ...
        }
      catch(Exception e) {
      }
    }
  }
};
```

The textual representation of program code, involves the need in formatting and preserving syntax. These both tasks, indeed, have nothing to do with program functionality, and addtionally load the developer, reducing productivity. As an example, here I can mention the need to close several blocks ending at the same point correctly, identing the closing brace symmetrically to opening one, and putting in the exact amount of braces. The Listing 1.1 demonstrates it in the last few lines.

1.1.2. Projectional Approach

SomethingSomethingSomethingSomethingSomethingSomethingSomethingSomethingSomethingSomethingSomet
SomethingSomethingSomethingSomethingSomethingSomethingSomethingSomethingSomethingSomethingSomet
SomethingSomethingSomethingSomethingSomethingSomethingSomethingSomethingSomethingSomethingSomet

Appendix

A. Detailed Descriptions

Here come the details that are not supposed to be in the regular text.

Bibliography