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 16, 2013

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

München, den 16. 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

Part I.

Introduction and Theory

1. Introduction

Here I introduce the main concepts and existing work, relevant to this Master Thesis, as well as present the motivation for the topic.

1.1. Projectional Editing

This section compares the traditional approach to build textual editors for the program code with the projectional approach, bringing up motivation for the least.

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, indenting 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.
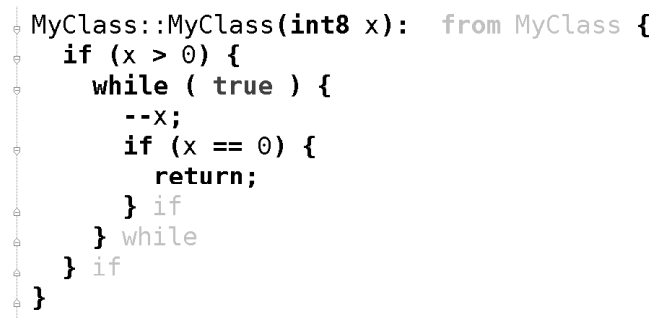
1.1.2. Projectional Approach

Another approach which can be taken in the editor for a language is called projectional approach. Projectional editors do not work with a low-level textual representation of a program, but rather with a higher level concept, ASTs.

Working with ASTs directly has several advantages over the conventional text editing. Firstly, all syntax errors are no longer possible, as there is no syntax. Secondly, there is no need to format the code anyhow, since it is only typical for textual code. Thirdly, all features, which in textual approach require parsing, can be implemented without parser, because AST is always known to the editor.

Projectional editors have to display somehow the AST to the developer, in order for him/her to work with it. Such visualization of an AST is called "projection", giving a name to the editor class.

The model of code is stored as AST in the projectional editor. As in the Model-View-Controller pattern ([7]) the view for the model can be implemented separately. Thus the

```
MyClass::MyClass(int8 x):   from MyClass {
  if (x > 0) {
    while ( true ) {
      --x;
      if (x == 0) {
        return;
      } if
    } while
  } if
}
```

Figure 1.1.: Example projection of an AST, "source code" view

code may look in a many different ways to the user. For example, the AST can be visual-
ized as a graph, similar to control flow graph. This visualization, however, is not always
advantageous being sometimes not compact and complicated to overview.

One of the well-accepted way to visualize AST is by visualizing its textual representation, as if it would was written as a code in the programming language, see Figure 1.1. The statements in the projectional are only selected as whole. There is now way to just select the "while" word for cut or copy, without selecting the condition and the block belonging to the statement. This behavior represents the position of the condition and while-body in the AST as children of the while statement. The statement can be selected all, only with children.

Additionally, each block delimiters are just a part of block visualization. They are organized in a proper way automatically, and there is no way to delete or confuse them, as well as type them initially. Each closing brace is marked with the parent statement name (not a problem to implement such behavior in viewing AST), enhancing navigation through the AST's projection.

As one can see the projection to text in a projectional editor looks almost the same, as the conventional textual editor. This can cause some confusion for the developer at first, as attempts to edit this textual visualization as a real text will fail. Eventually, however, advantages of such visualization overwhelm the disadvantages. Among the benefits of the textual projection over text are quicker code construction after short learning, better way to select code fragments, since not individual characters or lines, but rather AST nodes or groups of nodes are selected, plus all the advantages, the projectional editing brings by itself, as discussed above.

I discuss additionally the projectional approach and some of its basic principles in the Part V.

1.2. C++ in Projection

The goal of this Master Thesis is to research different aspects connected with implementation of the C++ programming language in a projectional editor. As a tool base for this Jetbrains MPS has been selected together with the based on it mbeddr project. This section overviews the technologies.

1.2.1. Jetbrains MPS

Jetbrains MPS stands for Jetbrains Meta Programming System. In this IDE-like software it is possible to develop domain specific languages. The approach taken in the Jetbrains MPS is rather unique, and it is considered to be advantageous in many ways, [11]. One defines a language in it not through the canonical grammar approach, but instead through defining so called concepts, and relationships between them similar to those in ER diagrams. A logical part of a language can be made a concept. Related to C++, class, method declaration, field declaration can be represented as a concept.

Each concept can have children, which should be assigned with a role and cardinality. For example, the class concept can have children of method concept, with a role called "methods", and cardinality 0..n. m Concepts can form hierarchies as classes normally do in object oriented programming languages. For this each concept should have a base concept (inheritance), and can implement several interface concepts (interface implementation).

Concepts can be abstract, for the use purely only in inheritance to create other non-abstract concepts with a common parent.

Each concept is described by several views on it. Among such views there are editor, behavior, constraints, type system, generator and few more views.

The editor view is designed to give a look for a concept, and a way to input it. This is where the projection of an AST node is defined. As editors are mostly defined to look like text, a program in the C++ programming languageimplemented in MPS looks almost like a regular C++ code.

The behavior view, can be used to define certain behavioral methods for a concept. A concept is represented there similar to a java class, and it is possible to define the methods in a Java-like language.

The constraints view can be used, to limit in a desirable way relationships the concept can have to other concepts.

The type system view is used mainly to define a type for each instance of a concept. The type is used later (for example, in the constraints), to determine suitability of the concept instance for a place, where it is used.

The generator view, as well as the textgen view, is used to define the way, the concept instance is going to be transformed when generation of the AST is invoked by the MPS user. In general, the generation is needed in the end of the programming, to obtain the source code from the AST in a textual representation. This textual representation is needed because all existing tools (like compilers) expect text code nowadays. It is worth mentioning that, in theory, one could implement a compiler, which works directly from the AST in the editor, eliminating thus the need in generators to text.

The representation of a new language created in concepts and views to them, present a seamless approach to creating a new language with a projectional editor. Thus Jetbrains MPS is a good suitable technology for the practical implementation of the Master Thesis goal.

1.2.2. mbeddr Project

The mbeddr project represent mainly an implementation of the C programming language in the Jetbrains MPS environment. Having embedded systems and software for them as a main focus, mbeddr provides certain language extensions to empower the programmer in the mentioned domain [10], [11].

Being a different language the C++ programming language shares a lot of commonality with C. As Jetbrains MPS allows to some extent incremental language construction, as the basis for the C++ programming language implementation in Jetbrains MPS the mbeddr project was considered to be suitable.

The use of mbeddr however, could not be purely incremental, and required some changes to the mbeddr itself. The changes were introduced however, in a way to make mbeddr simply more extensible, without creating a dedicated branch working as a basis for the C++ programming language only.

1.2.3. Motivation

The selection of the topic for the Master Thesiscan be motivated in several ways.

At first, the author is not aware of any implementations of C++ editing in a projectional editor. The goal to research such opportunities for one of the most complicated, traditionally highly text-oriented, object oriented languages is new.

Secondly, extending the mbeddr project with the C++ programming language has a practical interest for the mbeddr project users. Being comparatively lightweight, the C++ programming language supports object-oriented programming paradigms and more. It is thus of a high interest for embedded domain developers, using mbeddr, to be able to have the C++ programming language in the arsenal.

1.2.4. Special Goals

This work strives for some special goals, which are described below. These goals are targeted to either enhance the user experience, or to make the implementation more technically elegant. All the goals have some influence on the implementation, which is discussed in the following chapters, mentioning the goals.

Special Goal 1: Embedded-Programming Domain

Being an extension for the mbeddr project the discussed in this Master Thesis C++ implementation is following the general mbeddr goals. Among these goals are targeting the embedded-programming domain.

Special Goal 2: Enhancing the C++ Experience

It is hard to improve on one of the most beautiful programming languages. However, the usage of it can be improved with special features of the editor, making the language, when possible, more supportive to the user. This can focus towards an experience user-in-a-hurry, or to a novice, who is learning C++.

Special Goal 3: Non-invasive Language Extending

The implementation of the C++ programming language discussed in this work is an extension of the mbeddr project. Developing "on top of" the mbeddr project was not always possible through just the us of the mbeddr project as a layer of abstraction, one down to the layer, representing C++. However such a layered architecture was desirable. Due to this sometimes modifications to the mbeddr project were required.

The current goal consists of implementing all modifications to the mbeddr project in such a way, that only the C++ implementation has to be aware of the mbeddr project implementation details, but not the opposite! This is required to allow a separate development of the mbeddr project without a look back to the C++ extension and still remaining the C++ extension working with a newer mbeddr versions.

Such architecture is similar to layered architecture, but does not exactly represents one. The difference is that the usage of the underlying mbeddr project can not be described only as the use of the interfaces, provided by it. Sometimes there is simply no interfaces as such, and other ways have to be applied to make use of the mbeddr project in a needed way. All the cases where the pure use of the mbeddr layer is not possible are discussed

separately in the work below. Thus, this work researches also the practical modularity and extensibility aspects of the language development and its support in Jetbrains MPS.

Part II.

C++ and Projectional Editing

2. C and C++

Initially C++ appeared to be an extesnion to the C language, called "C with Classes" [9]. Till now the high degree of commonality can be found between the two languages. Mainly, the most of the C code is going to be valid C++ code.

2.1. Reference Type and Boolean Type

Among primitive types and operations there are two major differences relevant to practice. They are discussed in the following subsections.

2.1.1. Reference Type

Reference type is basically a new construct in C++ which represent the old notion of pointer in C with a different syntax. The syntax for a variable of type `T` can be used with the variable of type reference to `T`, or `&T` as it is designated in C++. It comes especially handy when passing arguments to methods by reference, which avoids creating a copy of an object to pass by value.

Listing 2.1: Reference Type in a Copy Constructor

```
class MyClass {
    MyClass( const MyClass& original );
};
```

The value of the reference type in C++ is bigger however, than just a new syntax for pointers. A constant reference to a class object has to be used in constructor to give it a special meaning of a copy constructor, as the Listing 2.1 demonstrates.

2.1.2. Boolean Type

The C++ programming language has a special `bool` type to represent the two logical values. There is no such type in C.

No matter, the `bool` type is not present in C, for the convenience of the user the mbeddr C implementation introduced a type `bool`, which is translated to `int8_t`, together with `true` and `false` values, converting to `1` and `0` respectively. This implementation can be considered better than the original C++ `bool` type present in the generated code, because the C++ programming language standard does not define explicitly the size of the `bool` type [5].

In the context of embedded systems, which we target, see 1.2.4, it is often very important to know precisely, with which type the user is operating, as the limited resources

are important to consider. Substituting the `bool` type with the `int8_t` type ensures that the size of the `bool` variables is known. Also, it can be changed as needed in the `TypeSizeConfiguration` in each model created with mbeddr separately.

The C++ standard explicitly allows the `bool` type to participate in integer promotions. This ensures further the compatibility of the custom-written text code, which may use actual `bool` type, with the code generated with substitution of `bool` to `int8_t`, as the user `bool` will be promoted.

Among limitations of this approach one can take as an example `std::vectoor<bool>`. Since the word "bool" itself is never generated, it is not possible to use the specialization of template, which ensures storage optimization, through, though, higher processing load when extracting a value from such vector.

2.2. Modules

In C++ as well as in C there is no clear concept of a module. The mbeddr project improves on it, defining modules, [11]. A C module is a concept, from which the header and the .c files are generated in mbeddr. Flagging an object (function, variable, structure, etc.) in a module as `exported` causes the declaration of the object to appear in the header, and thus the object starts to be usable by other modules.

Another issue with the C language is the one and only global namespace. The mbeddr project improves on it by introducing so-called name mangling. All names of the module contents are prefixed with the module name, when generated to the C text code. Thus the object with the same name but from different modules do not cause a name clash.

The C++ programming language is an improvement over the C language by itself. There is, however, no concept of a module, which can bring certain advantages and disadvantages. As a disadvantage can be named a potential disorder in placing implementations for classes and functions, as the language itself does not enforce a clear one place for them. As an advantage, conditional compilation can be used to include different files with different implementations, which is often used to create cross-platform code. Additionally, classes can be used analogously to modules.

The name clash problem is solved in C++ by introducing the namespaces. The programmer defines a namespace and then places there all the exported code. The nodes with equal names from different namespaces do not clash.

When comparing the two approaches of mbeddr and of the C++ programming language to improve on C, the hybrid approach has been picked. Firstly, the C++ implementation modules has been introduced. And secondly, the namespaces can be defined and used.

Modules are convenient program modular elements, where a programmer declares and implements C++ classes, and the header and .cpp files are automatically generated from them. This eliminates the need to create and control two files for each logical program element, like a class, and keep them synchronized with each other. For example, changing an interface of a class would require at first opening the header file, and introducing the changes in the declaration, and then rewriting the corresponding part of the .cpp file in strong connection with the header.

Namespaces are introduced in the projectional C++ to avoid inter-modular name clashes, they work similar to the namespaces in C++.

As auto-completion works very effective in the projectional editor, the need in `using` directive does not emerge the same as in plain text editors. The long names of namespaces are appended to the end of function declarations, to be exact about the namespace, but the declarations are kept looking short, with only the last namespace shown. In general, the generator is responsible in generating correct namespace resolution, and the projectional C++ programmer can benefit from short and clear namespace presentations, which are also quite handy to input.

2.3. Memory Allocation

In the C programming language memory (de-)allocation on the heap is happening via calls to the dedicated library functions. C++ instead introduces separate language construction for memory allocation and de-allocation.

The `new` and `delete` together with their array versions are simply an expression and a statement correspondingly, which perform the required operations.

Creating them is relatively easily possible by inheriting from the Expression and Statement concepts, and introducing the corresponding typing.

When a segment of memory is allocated as an array, it has to be de-allocated as an array either. As it is stated in the C++ standard ([5]) the behavior of simple de-allocation of an memory segment, allocated as an array, results into undefined behavior. A simple check is introduced in order to control this, see the Section 6.6.

3. C++ Object-Oriented Programming

The C++ programming language is a multi-paradigm programming language. The ability to support the object-oriented programming, is incorporated via classes.

A class represents a new type in the C++ programming language. Each class may have data in the form of fields, and behavior in the form of methods. Two types of methods are special for C++ - constructors and destructors, they have special meaning and syntax.

Encapsulation is enabled via governing access permissions to fields and methods of a class. The access control is governed with the creation of `public`, `protected` and `private` class sections.

Inheritance is implemented in C++ via allowing for each class to have one, or even many base classes. Inheritance from a base class is performed under a certain access control modifier. There is no pure notion of an interface, but rather abstract classes are introduced.

Polymorphism is implemented via pointer-to-class type compatibility over inheritance-connected classes.

The implementation of these C++ features in a projectional editor environment is discussed in the following chapters.

3.1. Class declaration

The C++ programming language is a mature language, with long traditions, and high flexibility, [6] can serve as an example. Although the language is also famous for being complex. It will not be possible to simplify language, removing features from it, which will restrict the language use. In this work I try to research, how the editor can be more supportive for the user, to eliminate usual mistakes made while programming, as well as provide help in structuring the code.

3.1.1. Visibility Sections

Instead of declaring visibility type for individual class members, visibility sections are created in C++.

The sections can be opened with a string `private:`, `protected:` or `public:` within a class declaration, and closed when another section is opened or when the class declaration ends. This allows the user to open and close the same section multiple times and declare sections without any particular order.

Various coding guidelines ([1], [2]) exist to enforce some restrictions on the visibility sections.

In particular, the sections are allowed to be opened only once. This ensures, the reader of the code will see interface of the class (public section) in one place, "contents" of the class (private section) in one place, and opportunities to access members in the inheriting

class (protected section) in one place, without the need to scroll through through the whole class declaration.

Another typical requirement in coding standards, is the order of the section. Usually the public section is required to be first, for the class users to see immediately the public interface, the class provides.

```
class WebPage {
  public:
    explicit WebPage() (constructor)
    WebPage(const WebPage& original) (copy constructor)
    boolean loadFromFile(string path)
    string getHtml()
  private:
    int32 visits
  protected:
    string html
}
```

Figure 3.1.: Sample class type declaration

The Figure 3.1, shows an example class declaration implemented in the projectional editor. The class concept has the visibility sections as children. Each section is given a separate role, and can appear 0 or 1 times. The editor for the class concept orders the visibility sections so, that the public section always comes first, followed by the private and protected sections.

The creation of a section is made with the use of so called *intention* in MPS. The user uses *Alt+Enter* combination on the class declaration to create visibility sections. It should be more practical and fast for the user, compared to typing the keyword, colon and indenting the result.

A question arises on how to support another way to represent a class, so that it will reflect requirements from a different coding standard. And as a way to resolve it a definition of another editor for a class concept can be offered, Unfortunately, the current version of Jetbrains MPS 2.5 does not support a definition of multiple editors for the same concepts. This limitation however is addressed in the newer 3.0 version.

3.1.2. Constructors

Constructors are special methods of a class, used to construct the class instances.

Constructors have special syntax and no return type, being similar to class methods. Additional value, however, the constructors gain, when participating in type transformations. Namely, when a constructor of a class B exist from a type T, instances of the type T can be used whenever the B class instance is required. The constructor will be *implicitly* called and a temporary object of class B is going to be created as a mediator.

Thus constructors extend the type system. Since this type extension can not be easily observed, it is highly possible to get various *run-time* errors or unexpected behavior.

Listing 3.1: Example of an Implicit Constructor Error

```cpp
#include<iostream>
using namespace std;

/*
 * API definition
 */

class Circle{
private:
  int r;
public:
  Circle(int radius){
    r = radius;
  }

  float getPerimeter(){
    return 2*3.14*r;
  }
};

void print(Circle c) {
  cout << "The perimeter is: " << c.getPerimeter();
}

/*
 * Use case by user of the API
 */

int main(){
  print(5);   //Prints "The perimeter is: 31.4"
  return 0;
}
```

The Listing 3.1 demonstrates a simplified use case where the function `print()` is invoked on `int` without any compiler error, and the resulting behavior is unexpected.

To avoid similar, it is possible to deprecate participation of a constructor in type conversions, adding a word `explicit` to the constructor declaration.

The described problematic motivated the following decision. When a new constructor is created, it is by default declared to be explicit, the user must intentionally change it to get the type conversion behavior. Such behavior is safer by default.

3.1.3. Copying

In the C++ programming language the programmer controls memory allocation fully on his/her own. This affect the way of copying the object. In C++ a programmer should

define two methods for a class: a copy constructor and an assignment operator. These two methods work when assignment like a = b happens, when instances of a class are passed by value to a function, when one object is initialized with a value of another object and so on.

C++ serves here sometimes dangerously generating default copy constructor and assignment operator, which by default represent a bitwise cloning of an object. This can lead to problems.

Listing 3.2: Need in a Custom Copy constructor

```cpp
class Resource{
  int* r;
public:
  Resource(){
    r = new int(0);
  }
  ~Resource(){
    delete r;
  }
};

int main(){
  Resource a;
  Resource b = a; // b.r is the same address as a.r
  return 0;
}
```

The Listing 3.2 demonstrates a program which crashes upon execution as destructors of a and b are deallocating memory with the same address, after default copy-constructor copies the address from a into b.

To avoid the described problem, the programmer has to either define proper copy-constructor or forbid copying of the objects of the class. The same applies to the assignment operator. Many standards require the two functions to be implemented in sync, [3]. This can be performed be implementing the assignment operator first and reusing it in the copy-constructor. To simplify the first, specialized macros exist, for example DISALLOW_COPY_AND_ASSIGN or Q_DISABLE_COPY, [2], [4].

The use of macros in C++ appears often in similar cases, in order to perform some language-engineering tasks to add the missing features to the language. Macros bear pure textual nature, and are processed by the pre-processor. Some negative effects may come out: need to pre-process reduces the speed of compilation and hides the resulting code from a programmer, macros lead to error prone programming, as no type checks are possible, macros make code less analyzable by automatic analyzers.

```
class A /copyable and assignable/ {
  public:
    explicit A() (constructor)
    A& operator = (const A& originial ) (makes class assignable)
    A(const A& originial) (copy constructor)
    int16 getX()
  private:
    int16 x
}
```

Figure 3.2.: Hinting about Copyable and Assignable Class Properties

The projectional editor allows for another solution, different from macros. In order to provide some support for the programmer regarding the copying issue, the projectional C++ implementation hints on the class declaration its assignable and copyable properties, Figure 3.2, and generates by default the declarations of copy-constructor and assignment operator.

The copy constructor and the assignment operator are detected automatically. Two intentions are provided on the Class concept to forbid or allow copying. The forbidding intention imitate the macros mentioned above, but displaying and explaining the implementation to the userFigure 3.3. The implementation consists of moving the declarations of two functions to the private section of the class. Implementation is not required for such functions.

```
class A /neither copyable nor assignable/ {
  public:
    explicit A() (constructor)
    int16 getX()
  private:
    int16 x
    A& operator = (const A& o ) (makes class not assignable)
    A(const A& o) (makes class not copyable)
}
```

Figure 3.3.: Class Made not Copyable by the *Forbid Copying* Intention

The Jetbrains MPS supports the implementation by providing read-only model accesses, special parts of editor concept, by which the hinting is implemented. The intentions allowed manipulations on a class, which made it possible to automate the allowing or forbidding of copying/assignment, making the implementation clear to the user, without the use of macros. The whole work to forbid copying and assignment contains of a call to an intention, one key-stroke. There is no need to include a header file with macros, and look up the documentation for them (symmetric macro requires exactly one parameter - the class name, and it has to go in the private class section).

3.2. Encapsulation and Inheritance

Encapsulation and inheritance are considered here together, because from the language-engineering point of view, they just decide the access for class members. In other words, the projectional C++ implementation has to track encapsulation and inheritance related definitions and provide access to the class members accordingly.

3.2.1. Various Cases of Access Control

In the C++ programming language exists a number of ways to govern access control to class members. Before discussing the implementation of them in a projectional C++, I briefly review them with an example.

All members, a class has, are either declared in the class, or inherited from its base classes.

The members can be accessed in a number of different locations in the code, which differ by the level of access they have to the class members. Among these locations are the class methods, friend functions, belonging to the class and external to the class code.

Each member can be declared with a certain visibility/access type, and the inheritance of the member can happen with one of the three inheritance modifiers.

The access depends on the object, on which the member is requested, and on its type as well.

```
class A /copyable and assignable/ {
  public:
    int8 valAPublic
  private:
    int8 valAPrivate
  protected:
    int8 valAProtected
  friends:
    friend compare (boolean compare(const A& a1, const A& a2))
}

class B : public A /copyable and assignable/ {
  public:
    B(const B& originial) (copy constructor)
}
```

Figure 3.4.: Declaration of two classes with a friend function

The Figure 3.4 shows a declaration of two classes. The class A has all three public, protected and private fields. A function `compare()` is declared to be a friend function. The visibility plays no role for the friend function declarations, that is why a decision was made to create a special section for friends, it is not generated anyhow in the resulting C++ textual code.

The class B in the Figure 3.4 is inheriting publicly from A, which means, that public members of A remain being public in B. The class B declares a copy constructor.

Such declaration can be utilized as shown in the figure Figure 3.5.

```
B::B(const B& originial)   from B {
  this->valAPublic = originial.valAPublic;
  this->valAProtected = originial.valAProtected;
  this->valAPrivate;
}

boolean compare(const A& a1, const A& a2) {
  return a1.valAPrivate >= a2.valAPrivate;
} compare (function)

void external() {
  B b;
  cout << b.valAPublic;
  cout << b.valAProtected;
  cout << b.valAPrivate;
} external (function)
```

Figure 3.5.: Visibility resolution

In the copy-constructor the visibility resolution happens after `this` pointer and after the `original` object. Arrow expression and dot expression are used for this. The first and second lines are making use of public and protected fields of the base class `A`. The use of the private field is however not possible, since private fields of a class are only accessible to methods of the same class and friend functions.

It is even not possible to input the not-allowed member, as the projectional editor does not bind the text to anything, and it remains red, invalid, not usable further.

The `compare()` function is declared in advance (Figure 3.4) to be a friend function of the class `A`. Thus, it is not a problem for this function to access even private fields of `A`, for comparison purposes in this example case.

The function `external()` is not related anyhow to classes declared. Thus, it represent "external" from the class point of view code. Only public members are accessible, but not protected or private. The attempt to input them, simply fails, they are highlighted red, and not bound to anything.

3.2.2. Expressions to Address Class Members

Members are usually accessed relatively to some object. The object can be designated as an expression of type class or a pointer to class, in particular, `this` expression. The resulting access represent nothing else, but expression itself. One of the greatest ideas in Jetbrains MPS to allow extensibility is concept inheritance. Once a need to create a new concept arises, serving as a concept known before, the new concept has to inherit from the existing one, and this is almost all what has to be done. Thus inheriting the OoDotOrArrowExpression concept from the Expression concept, we get the ability to use the expression, designed for member access, wherever an expression can be used.

The abstract concept OoDotOrArrowExpression serves as a parent for OoDotExpression and OoArrowExpression. The commonality between the two, is that an object is accessed in the left part, and a member is selected in the right part, as well as the way to decide the

access to members. The access is defined then, which left part is going to be possible in such expressions.

Within the class methods it is also possible in C++ to address class members as local variables. In the projectional implementation described, it is not possible. Instead, `this` expression has to be used. It makes typing a little bit slower, but allows to easily distinguish between members of the class and other variables or functions.

3.3. Polymorphism

There are several ways to achieve polymorphic behavior in C++. The purists of the language differentiate between the polymorphism based on virtual functions or on templates. More general opinion can include in the notion of the polymorphism also functions overloading and operator overloading, also called *ad hoc* polymorphism. Occasionally various operations with `void *` are also classified as the polymorphic programming.

In this section I am writing only about the class-related virtual functions polymorphism.

3.3.1. Virtual Functions Polymorphism in C++

Starting the description of the projectional implementation, I find it good to spend some time describing the way, the polymorphism is implemented in the C++ programming language itself.

Dislike many other popular object-oriented programming languages in the C++ there is no pure notion of an interface. Instead, a base class is used as an interface declaration for its descendants. Functions designated to be a part of the interface must be declared `virtual` and they can be overloaded in the subclasses.

The virtual functions in the base-interface class can be implemented as well, providing some "default" implementation. Otherwise they are left *pure virtual*, meaning that no implementation is provided and the pointer to the function in the table of pointers to virtual functions is zeroed. The syntax for pure virtual functions is rather not obvious, when not knowing about the zero pointer semantics, and a bit cumbersome requiring to type one reserved word, one punctuation sign and one number to express simple fact of pure virtuality or, simply, absence of implementation, Listing 3.3.

Listing 3.3: Pure Virtual Function Syntax Example

```cpp
class Animal
{
public:
    virtual void voice() = 0;
};
```

Such approach is more flexible compared to languages with the notion of interface directly introduced. In C++ it is possible to create partially implemented base classes, what can not be done when implementing and interface in Java or C#, where the approach is "all or none" regarding the implementation of interface functions.

In order to implement a declared in a base class function the descendant must declare and implement a virtual function, matching the full (with return type) signature of the declared function, Listing 3.5. The connection to the "interface" function declaration stays subtle, and it is not immediately clear, whether the declared implementation is an override or a new independent declaration. This knowledge affects the changing process greatly, as the override should change from the interface, together with all the implementation. The absence of a clear override syntax I call here "override syntax absence".

Whenever a class and all of its ancestors does not provide an implementation of a certain virtual function, created as a pure virtual in the declaring class, the class is called *abstract*. It is not possible to construct instances of abstract classes. C++ however does not have any special syntax for abstract classes, Listing 3.3. The programmer usually has to be aware (from documentation, implementation, or, the worst case, compilation errors) whether a given class is abstract. I will call this phenomenon as "abstract class syntax absence".

When the overriding a function interface is a active action of the programmer, and is initiated by the programmer, the abstract property of the class, oppositely, can be deduced from the analysis on the base classes automatically, by editor.

In order to use an interface declared in some class, the using code has to get a pointer to instance of any implementing the interface descendant class instance. Thus typing system has to allow a pointer to the descendant to be treated in the way as a pointer to the base would be. The same should hold normally for the reference types, but it is not used very often on practice, and omitted in the implementation. As this typing rule represents the core of polymorphic behavior, I will start from it, describing further the projectional C++ implementation.

3.3.2. Pointer to Class Special Typing

The problem solved here is enabling the usage of pointers to descendants instead of pointers to ancestors, Listing 3.4.

Listing 3.4: Example Usage of a Pointer to Descendant Class

```
class Shape {};

class Circle: public Shape {};

void draw(Shape * shape){
  // ...
}

int main(){

  Circle* c = new Circle();

  // Call with a Circle* parameter instead of the declared Shape*
  draw(c);
```

```
    return 0;
}
```

The pointer type is implemented in a separate language in mbeddr. Following the Special Goal 3 (see 1.2.4) the C++ implementation needs to add the typing rules for pointer *to classes* without changing the pointer mbeddr language, where the typing system for pointer type is defined.

In the case when a type of a pointer to a base class is expected, a pointer to a subclass should also be accepted. The type system of the pointer language will try to check the compatibility of the two pointer types. It will fail to do so, as the mbeddr languages are not aware of the C++ extensions by design.

When none of the existing type system rules can resolve the given situation, the so called replacement rule can be invoked in MPS. The C++ extension provides a replacement rule, which checks, whether a class pointed to, is a passing descendant of the class required by an expression, where the pointer was used.

In the Part III I discuss more on the approach taken here, its limitations and the potential ways to overcome them.

3.3.3. Overriding a Virtual Function

The Listing 3.5 demonstrates, how overriding of a virtual function happens in C++. The function `getArea()` is initially defined in the class `Shape` and is then overridden in the class `Circle`. The Listing 3.5 demonstrates as well the "override syntax absence" in C++.

Listing 3.5: Example of an Overridden Function - Text Code

```cpp
class Shape {
  public:
    Shape();
    virtual double getArea();
};

class Circle : public Shape {
  public:
    Circle(double r);
    virtual double getArea();
  private:
    double mRadius;
};
```

In the Figure 3.6 the same example is demonstrated, but in the projectional C++ implementation. Indeed, the Listing 3.5 is the generation result part from the demonstrated projectional sample.

When a method declaration is created, it can be set to be an override of a method in a base class. Only when an empty method declaration is created, it can be set to be an

override, so that the only thing which stays, is to pick a method from a base class to be overridden. The override is automatically named, the parameters and the return type are set accordingly, the `virtual` property is immediately set.

After the override has been linked to the overridden method, the projectional editor checks, if the override full signature stays precisely the same as the one of the overridden method. The overridden method is shown next to the override declaration Figure 3.6, which compensates on the override syntax absence.

```
class Shape /copyable and assignable/ {
  public:
    Shape() (constructor)
    virtual double getArea()
}

class Circle : public Shape /copyable and assignable/ {
  public:
    Circle(double r) (constructor)
    virtual double getArea() overrides Shape::getArea()
  private:
    double mRadius
}
```

Figure 3.6.: Example of an Overridden Function - Projection

The additional syntax on the projectional view, should not confuse the reader. It is only present in the projectional editor, and, when the AST is generated into code, the regular C++ syntax is achieved. However in this case the AST stores *more* than needed to generate and compile the C++ code. This subsection demonstrates, why it can be useful.

3.3.4. Pure Virtual Functions

In the example above one improvement to the code can be made. As the `getArea()` for a random shape can not be determined, it makes sense to make the `getArea()` function pure virtual. A pure virtual function has no implementation in the declaring class, and serves only overriding purpose. Semantically in C++ it sets the pointer in `vptr` to 0 and thus has reflecting this syntax, see the similar example in Listing 3.3.

This syntax can be seen as not obvious, as it reflect more the under-the-hood implementation of the mechanism, rather then the original programmer intention to built a basis for an overrides chain.

Additionally, as discussed above, declaring a pure virtual function requires a significant amount of the syntactical overhead.

In the projectional C++ implementation, one intention is reserved to make a function pure virtual. The intention automatically sets the needed pure virtual and virtual flags of the function declaration, and the projection changes. That is why out of the statement `virtual double getArea() = 0;` the programmer has to input only the name `getArea`, pick the type `double`, and toggle the pure virtual intention for the declaration.

The result of the intention work in projection can be seen in the Figure 3.7.

```
pure virtual double getArea() = 0
```

Figure 3.7.: Example of a Pure Virtual Method - Projection

The word `pure` is added by the projection editor. This makes the reading of the code easy and natural. The = 0 part is preserved for the C++ programmer, used to this syntax. And, as in many other cases, the semicolon is omitted as it is nothing but syntactic help to the compiler, which does not have to appear in the projection.

3.3.5. Abstract Classes

If any of the pure virtual functions in the inheritance chain leading to a class is not implemented by this chain, or inside the class itself, the class is called abstract. It is not possible to construct an instance of an abstract class, as such classes are partially implemented. The programmer has to know which classes are abstract, and are intended for inheritance and further implementation only.

Above I discuss the absence of the abstract class syntax in C++. It leads to the need for the programmer to determine somehow him/herself if a given class is abstract.

And editor can however perform an analysis and determine if the class is abstract. In the case of the projectional editor, this analysis is especially computationally efficient, as the AST is readily available, and the quick analysis can be performed by a simple recursive algorithm on the inheritance chains.

After an abstract class is determined, it is possible to modify the editor representation for it, and show that it is abstract, Figure 3.8.

In this example a typical class hierarchy is created to support user interface programming. A user of this API, when searching for a button, could try using the `Button` class, which is designed to be abstract, and serve as a base for the further implementation, e.g. `PushButton` in the example, or check boxes, radio buttons and similar, later.

```
abstract class Widget /copyable and assignable/ {
  public:
    explicit Widget(Widget* parent) (constructor)
    pure virtual Size getDimensions()  = 0
}

abstract class Button : public Widget /copyable and assignable/ {
  public:
    Button() (constructor)
    pure virtual boolean isPressed()  = 0
}

class PushButton : public Button /copyable and assignable/ {
  public:
    PushButton() (constructor)
    virtual Size getDimensions() overrides Widget::getDimensions()
    virtual boolean isPressed() overrides Button::isPressed()
}
```

Figure 3.8.: Determining Abstract Classes

The projectional editor, however, checks on the fly, if a certain class is abstract, adding a special `abstract` word in front of its declaration. This makes the reading easier, and allows for quicker understanding of the code.

Additionally, the creation and usage of abstract class instances is checked, and forbidden by type analysis. This is described separately in the Chapter 6.

4. Operator Overloading

5. Templates

6. Advanced Editor Functionality

As the projectional editor work directly with an AST, it is possible to provide some programming on the AST to improve the user experience. I call this additional programming as advanced editor functionality, and discuss it in this chapter.

6.1. Renaming Refactoring

Directly from the nature of the projectional editor, without any additional effort, comes a feature to perform the renaming refactorings.

If a node of an AST gets to be referenced somewhere else, it is referenced by the use of its unique identifier. The name of the node is a property of the node, which is not playing any role in the referencing the node from somewhere else. Thus renaming, dislike the way it is performed in text editors, does not involve replacing of the name all around the code. Instead, just the name property of a node is changed.

As an example - renaming a class or a method would mean just changing its name where it is declared first.

As a disadvantage here one can think of moving a code from one place to another. For example, moving a method from one class to another. The member fields of the source class, even if present in the destination class, will not immediately fit into the copied new code, as it is still referencing not available anymore identifiers of the source class member fields. This effect is also mentioned in the Chapter 7.

6.2. Getter and Setter Generation

In order to provide access to encapsulated class properties, usually expressed as member fields in C++, two access functions are defined, known as a getter and a setter. The getter is used to read the property, and the setter is used to set the property to a new value, after checking the validity of the new value.

The job of declaring and prototyping the two functions can be automated with an MPS intention, Figure 6.1.

```
class Rectangle /copyable and assignable/ {
  private:
    int32 mWidth
}
                        Intentions
            Add Documentation          ►
            Attach Name                ►
            Create Getter and Setter   ►
            Make Protected             ►
            Make Public                ►
```
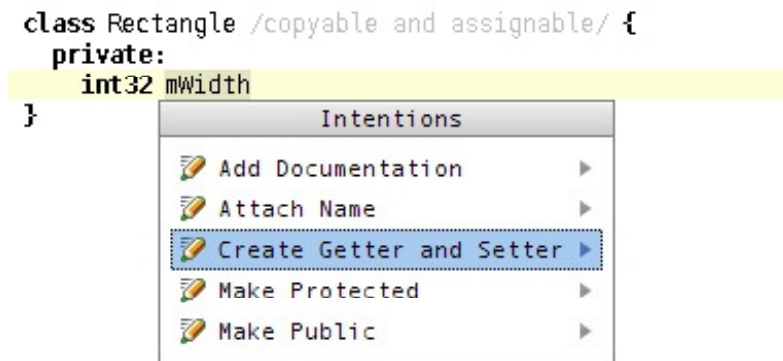
Figure 6.1.: Calling the Generation of Getter and Setter

The result of the intention work is, as expected, two methods declared and prototyped, Figure 6.2.

```
class Rectangle /copyable and assignable/ {
  public:
    int32 getWidth()
    boolean setWidth(const int32 theWidth)
  private:
    int32 mWidth
}

boolean Rectangle::setWidth(const int32 theWidth) from Rectangle {
  this->mWidth = theWidth;
  return true;
}

int32 Rectangle::getWidth() from Rectangle {
  return this->mWidth;
}
```

Figure 6.2.: Getter and Setter Generated

The way the editor names the getter and setter can be controlled through Naming Conventions concept, which is discussed in the Section 6.3.

The getter and setter are declared in the public section of the class, which is automatically created, if not found there already.

The getter is rather simple, it just returns the value of the member field.

The setter is somewhat more complicated. Firstly, it designed to return `boolean` by default. This is made to remind the programmer, to include checking of the value and return `false`. Secondly, the parameter of the getter is typed appropriately. As C++ by itself and the Special Goal 1 (see 1.2.4) imply the performance maximization, the type of the parameter for the setter depends on the member field type. And when the type

represents a composite structure, like a class, it is passed by a constant reference, instead of value, Figure 6.3.

```
boolean Widget::setSurface(const Rectangle& theSurface)
  this->mSurface = theSurface;
  return true;
}
```

Figure 6.3.: Setter Works with a Constant Reference for Classes

Passing a composite parameter as a value involves an overhead of allocating the necessary memory and then copying the contents in the newly allocated memory. To access the parameter in both cases pointer arithmetic is going to follow.

6.3. Naming Conventions

In the C++ development projects some code writing conventions are usually agreed upon. For example, in Google coding style guide for C++ [2] the member fields must end up with "_" sign. For the consistency and uniformity purposes each project has to have some agreements on code composing. They can include code formatting rules and naming rules.

To some extent the formatting conventions are fixed by the way the projectional editor shows the AST, see, for example, the Section 3.1.1.

The naming rules, however, should be additionally controlled. The naming rules accepted in a project I call naming conventions here.

In order to perform this a new concept has been introduced, called C++ Naming Conventions, Figure 6.4.

```
C++ Naming Conventions
  Member prefix:   m
  Getter prefix:   get
  Setter prefix:   set
  Setter argument prefix: | the
```

Figure 6.4.: C++ Naming Conventions Concept

In the naming conventions one can give a standard prefixes for getters and setters, which are used during the code generation for them, see the Section 6.2. The argument prefix is also used in the generation, to name the setter argument.

The member prefix is used on the member fields to check their naming. It possible to change the prefix and the editor will control each of the member field names to comply.

Whenever a field is not named in a proper way, an error extension is available, which automatically renames the field, Figure 6.5.
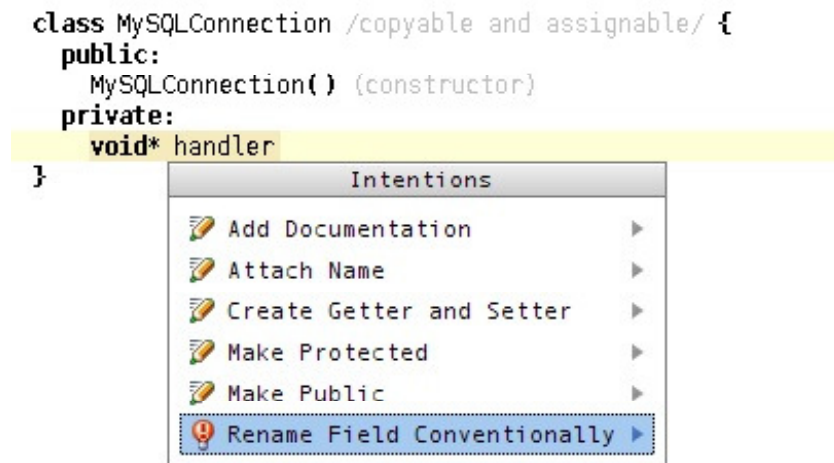
Figure 6.5.: Intension to Rename a Field - Naming Conventions

The naming conventions concept was created to demonstrate some new possibilities, which can be added to the projectional C++. Few remarks are needed to explain potential changes to the concept and its difference from the similar functionality taken in the conventional text editors.

First, the marking of special variables or object, keywords or type names, can be performed be changing editors for them. The special naming will not be needed then. Such way would mean however no opportunity to introduce a project-dependent marking, as it starts to be common for all projects, being a part of the projectional C++ implementation.

Second, the naming checks can be generalized to use compliance to regular expressions, instead of just prefixes. As an advantage comes the ability to adopt a much broader set of different naming conventions. Disadvantageous is the increase of the computational load in the projectional editor, this is discussed also in the Chapter 9.

Third, the naming conventions in projectional C++ is just a concept in an MPS model. It is no different from a C++ Implementation Module or Build Configuration. Dislike many conventional editors, which store such settings as a workspace, project or IDE configuration, the projectional C++ makes it a part of the program code itself. It is advantageous, since all the programmers in a team have to follow the same naming conventions set up once in a project, and no environment (re-)configuration is needed to work on the project. The projectional approach is also discussed and compared to the conventional one in the Chapter 7.

And last, the naming conventions can of-course be extended to incorporate class names, function names, additional naming for methods, like "is"- prefix for the boolean return type.

6.4. Method Implemented Check

When using the C++ programming language it is possible to declare a class in one file, and then implement its methods in other, potentially several, files. As there is no clear concept of module in the language itself (more on this in the Section 2.2) usually different coding standards improve on it, requiring, for example in the Qt Project [8], one class to be declared in a the-same-named header file, and the implementation for it to be in one .cpp file.

As the language itself does not argue about the place for the method, IDEs usually do not check if the method is implemented. Neither do compilers. And only at the linking stage it can appear, that the linker is not able to locate the method. The error message from the linker can be rather obscure, as the linker operates on a lover abstraction level, rather than the programmer using the C++ language.

After introducing the new module concept as in the Section 2.2, it is possible to check each method and find out if they all are implemented.

A check is introduced in the projectional C++, which respects inheritance chains, with overloaded functions and pure virtual functions, and creates a warning for the programmer hinting that a certain method has not been implemented. Together with the class declaration, generating the assignment operator declaration and a copy constructor, the method implemented check makes the programmer to fully define the class together with the most important methods, affecting the memory operations behind the class instances life-cycle.

6.5. Abstract Class Construction Check

In this work I have already defined the notion of abstract classes in C++ and some related to them questions in 3.3.5 and in 3.3.1. Mainly the absence of syntax for the abstract classes and the problem of determination those was discussed.


```
abstract class MyObject /copyable and assignable/ {
  public:
    pure virtual string getName()  = 0
}

                        ┌─────────────────────────────────────┐
                        │ Error: Can not be an abstract class  │
                        └─────────────────────────────────────┘
void greet(MyObject object) {

} greet (function)
```
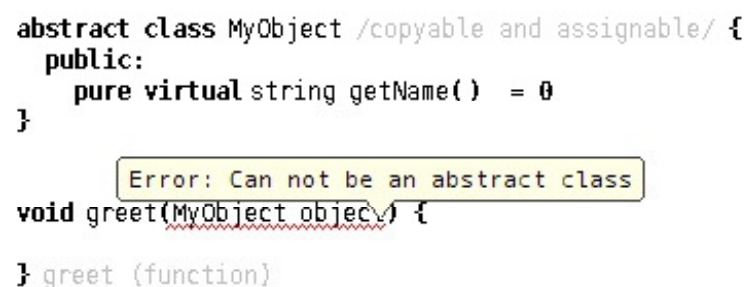
Figure 6.6.: The Abstract Class ConstructREal Zaj Zion Check Signaling an Error

An opened question which stayed, however, is providing some help to the programmer in the usage of such classes. Namely, any instantiation of an abstract class has to be forbidden. The abstract cREal Zaj Zlass as a type, however, should stay, and the pointer type to it should not be anyhow affected by the fact that the class is abstract.

The decision was taken to check the places, where an instance of an abstract class can appear, and deprecate such situations. MPS supports so-called non-type-system checks, which come useful in this situation. Any node on the AST can be analyzed, and if the error happens to be detected, there is a chance to associate it with the node and a textual error message, which will clarify the programmer, what exactly went wrong.

Among the places, where a class can be instantiated are variable declarations field declarations, method and function arguments, return types of those. For each of such places a check is performed, and if the class type is detected, it is checked against being abstract. Error message appears to indicate the problem if found, Figure 6.6.

The procedure of determining if a class is abstract, can happen every time, when the check is performed. These are all variable declarations, function and method arguments and return types, and memory allocations. The amount of computational load can be significant. The question of this complexity is discussed in the Chapter 9.

6.6. Array Deallocation Check

Whenever some allocated on the heap data is about to be deallocated, in C++ two operators can be involved, `delete` or `delete[]`. The first one is dedicated to work with memory, allocated for a single object be he operator `new`, the second deallocates memory, allocated for an array of object by the operator `new[]`.

The C++ standard does no specify any concrete behavior for wrong deallocation of an array with an operator `delete` (without square brackets), describing

Part III.

MPS Language Extensibility

Part IV.

Comparison of Projectional and Traditional Approaches

7. Comparison with Textual Approach

Part V.

More on the Projectional Approach

8. Generalized Principles of the Projectional Approach

In this chapter I formulate some of the general principles, which can be taken into account when designing new languages in the projectional editors.

8.1. Targeting Semantics

8.2. Store More Information

8.3. Configuration as a Part of Source Code

8.4. Hide Redundant Syntax

9. Analysis and Complexity

Part VI.

Conclusion

10. Conclusion

Appendix

A. Detailed Descriptions

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

Bibliography

[1] Class layout recomendations, possibility.com. http://www.possibility.com/cpp/cppcodingstandard

[2] Google style recomendations for c++, http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml.

[3] Open office c++ coding standard, http://wiki.openoffice.org/wiki/cpp_coding_standards.

[4] Qt project documentation page, contains q_disable_copy macro. http://qt-project.org/doc/qt-5.0/qtcore/qobject.html.

[5] Standard for programming language c++ (c++11).

[6] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, 2001.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[8] Qt Project. The qt project coding guidelines, http://qt-project.org/wiki/category:developing_qt.

[9] Bjarne Stroustrup. Classes: An abstract data type facility for the c language. *Sigplan Notices*, 17:41–52, 1982.

[10] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: an Extensible C-based Programming Language and IDE for Embedded Systems. In *Proceedings of SPLASH Wavefront 2012*, 2012.

[11] Markus Voelter. Embedded Software Development with Projectional Language Workbenches. In Dorina Petriu, Nicolas Rouquette, and Oystein Haugen, editors, *Model Driven Engineering Languages and Systems, 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010. Proceedings*, Lecture Notes in Computer Science. Springer, 2010.