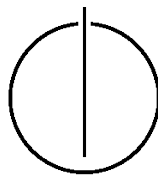# FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Adding C++ Support to MBEDDR

Zaur Molotnikov

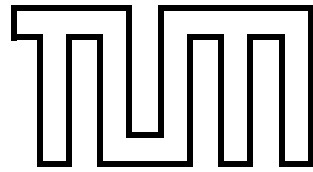# FAKULTÄT FÜR INFORMATIK
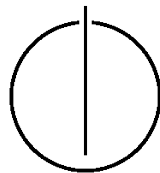
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's 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

I assure the single handed composition of this Master's Thesis only supported by declared resources.

Munich, September 16, 2013   Zaur Molotnikov

Acknowledgments

Abstract

In this work we describe the process of adding the C++ programming language support to the *mbeddr project*, an implementation of the C programming language with extensions in a projectional language engineering environment, *JetBrains MPS*.

While implementing the C++ programming language some well-known pitfalls of the language are taken into account in an attempt to build a better C++ language flavor. Several analyses are implemented to improve the programming experience of the end user. Lessons learned include generalized principles, which a language engineer could use to improve a language while recreating it in a projectional language engineering environment; an anlysis of support for language modularity and extensibility by *JetBrains MPS*; research on the typical problems occuring while implementing analyses in *JetBrains MPS*.

Contents

1 Introduction

1.1 Context

In embedded programming the C++ programming language is widely spread, [1]. Being a general purpose programming language, C++ does not provide, however, any special support for preogrammers of embedded systems.

By changing the language itself, together with a tool set for it, it is possible to get a better environment for a dedicated domain, for example, specifically for embedded programming. There are two known approaches to change the language itself.

Taking a Subset of a Language The first possible approach resembles dropping some language features, to get the language, which is simpler. As an example, a subset of C++, called *Embedded C++* can be brought, [2]. The approach taken in *Embedded C++* is omitting very many core features of C++: virtual base classes, exceptions, namespaces and templates. It allows for a higher degree of optimizations by compiler possible and makes the language much less sophisticated in general.

Embedded C++ was intended to ensure higher software quality through better understanding of the language by programmers; higher quality of compilers, through simplicity; better suitability for the embedded domain, through memory consumption considerations [3].

The C++ community has criticized the approach taken in *Embedded C++*, specifically for the inability of the limited language to take advantage of the C++ *stands for Standard Template Library, the standard library of the C++ language (STL)*, which requires the C++ language features, absent in *Embedded C++*, [4]. As a response to it IAR Systems have developed *Extended Embedded C++*, which includes many of the language features, omitted by *Embedded C++*, and a memory-aware version of *STL*, extending not only *Embedded C++*, but C++ in general [5].

Extending a Language The second approach to modify a language in order to get it more suitable for a specific domain[1] consists of extending the language with constructions, specific to the domain. The authors of the *mbeddr project* have taken such approach, to improve on the C programming language, [6].

Specific extensions may represent some often met idioms in the domain. For example, in embedded development a specification for a device might be given. The goal of the programmer could be to develop a software, which interacts with the

---

[1]Here we consider an improvement to a language to get a better suitability to any given domain, and not necessarily just embedded development. However, we have in mind the embedded domain as first to support, as *mbeddr* targets it directly.

device. The device specification may contain a state table or a state machine diagram, describing the device behavior or the way to control the device. A language developer could incorporate such originating from the domain of interest notions into the language.

For example, the Figure 1.1 demonstrates[2], how a decision table is built in into an *mbeddr* C code. The decision table provides a higher abstraction level, when compared to the existing initially C language constructs.

```
enum mode { MANUAL; AUTO; FAIL; }

mode nextMode(mode mode, int8_t speed) {
    return mode, FAIL
} nextMode (function)
```

	mode == MANUAL	mode == AUTO
speed < 30	MANUAL	AUTO
speed > 30	MANUAL	MANUAL

Taken from [7], Ratiu 2012

Figure 1.1: Example of a Decision Table, Added to C Language

Moreover, higher-level extensions could induce some higher-level semantics. An *Integrated Development Environment (IDE)* under construction could check the higher-level semantics for correctness on the programming stage. Such checks could improve quality of the software under development. For example, a *mbeddr* checks a given decision table for completeness of choices and their consistency, [7]. The Figure 1.1 demonstrates a decision table. The decision table takes `mode` and `speed` as input parameters and returns a new `mode` value as determined by the input parameters. A careful reader could see, that the exact value `30` of `speed` is not taken into account, and the default value `FAIL` is going to be returned. A programmer could invoke an analysis[3], to find out that the table is not covering the whole choice space.

1.1.1 *mbeddr*: a Language Engineering Project with JetBrains MPS

The *mbeddr* development team has used a special language engineering environment, *JetBrains MPS*, to support modular and incremental language development. A programmer using *JetBrains MPS* splits a language under development into special class-like items, called *concepts*. Concepts represent the *Abstract Syntax Tree (AST)* node types.

As an example of a *concept* an expression can be taken. It is possible to describe in *JetBrains MPS* different expression kinds, similar to object-oriented class hierarchy, allowing the objects to reference each other, and enabling polymorphism, in a way when any descendant can be used instead of its ancestor, e.g. binary minus expression can be used wherever an expression (any expression) is required. After various expression types were described to the language engineering environment

---

[2]The illustration is taken from [7]

[3]The analysis is described in detail in [7]

as *concepts*, the environment provides a chance to instantiate concrete expressions and edit them, acting as an editor for the language. The created editor serves as a key component in the *IDE* under construction. This editor contains the code for automations and analyses, if provided for the language. It works with text code generators, a compiler, a debugger and all other tools connected to the language. Thus we often use the term "editor" and the term "*IDE*" interchangeably.

Over the inheritance mechanisms, it is possible to extend languages, providing new *concepts* as descendants of the existing ones. For example, the expression *concept* can be extended to support a new sort of expressions, like decision tables. Thus, language modularity is achieved and incremental development is made to be possible.

Modularity is achieved as well, since one language is enabled to interact with another one. For example, expressions, described independently as a language, can be reused in any language, which has a need in expressions, like a language with statements of a programming language, because statements include expressions naturally.

Having in mind the opportunities, the language modularity in *JetBrains MPS* brings, it makes sense to recreate a general purpose programming language in *JetBrains MPS*. Building the general purpose programming language brings a basis to develop domain specific extensions to the well-known general purpose language. The editor for the general purpose language comes almost "for free", as a side product, after the language is split into *concepts*.

Later, from the code in the implemented general purpose language a text code can be generated for further processing, compiling, deployment. The language extensions, of-course, are not known to the existing tools which process the language. But they usually can be reduced to the base general purpose programming language statements, presenting the regular syntax of the taken general purpose language to the further tool chains, which expect only the basic language constructions, as an outcome. Thus the general purpose programming language is getting enhanced, remaining compatible with all the existing tools to process it. The developed editor forms a key component of a new *IDE*, as described above.

Additionally to the language modification itself, an *IDE* can be improved to support the domain specific development. Various analyses[4] can be built in into the code editor in order to detect inconsistencies, or, simply, "dangerous" constructs, and inform the programmer. Certain code formatting, or standard requirements could be enforced as well. The *IDE* could be enhanced with various automations, like support for code generation and refactorings.

As the new *IDE* works internally with an *AST*, described through the node types, or *concepts*, in order to perform a code analysis, a generation, or a transformation, there is no need to invoke parsers for the code, which is advantageous.

---

[4]analyses not only for extensions, but for the base language itself

1.2 Problem

Being a powerful extensible tool for an embedded systems developer, *mbeddr* does not support the C++ programming language. Supporting C++ would be an advantageous argument for *mbeddr*, as C++ empowers the developer with additional paradigms[5] of programming, mainly the object-oriented programming. Thus a problem appears, to support C++ within *mbeddr*. As *mbeddr* is built itself upon *JetBrains MPS* and language modularity principles, these principles have to be taken into account while extending *mbeddr*. This means, that the C++ programming language has to be developed as a modular extension for the C language provided by *mbeddr*.

1.3 Approach

Above we describe the two approaches used to make a language more suitable for a particular domain. In this work we use a mixture of the two approaches in an attempt to achieve a modular C++ language, which can later on serve as a basis to target some domain of interest. We try to modify the C++ language for it to be, on the one hand more, user-friendly, and, on the other hand, for it to be prepared to become a better embedded systems development tool, being safer, clearer and, in the future, including specific for the domain extensions. We built C++ itself as an extension to C and as a modular base for further extending. While limiting C++, we try to keep all the core features in it, to not to face the same problems as *Embedded C++* had.

We built C++ as a suitable base for the further language engineering, including the specialization of C++ for embedded development, and even more general, for any domain of choice. A special *IDE* is created together with a new C++ language flavor, which supports a C++ programmer.

During the creation of the C++ programming language in the way described, the language modularity in general is analyzed, and caveats of it are described together with the ways to avoid them. The newly created *IDE* features analyses. The question of their computational complexity is raised in general, together with the practical outcomes of it.

The approach taken in this work goes further into exploring the language modularity on the basis of *JetBrains MPS*. While building the C++ programming language itself with the goal of embedded domain specific extensions in mind, the C++ itself is being built as an extension to the C programming language, provided by *mbeddr*. The C++ flavor implemented in *JetBrains MPS* and discussed in this work we call *Projectional C++*. Although C++ is a separate from C language, the high degree of similarity allows to make use of the C programming language, implemented by the *mbeddr* as a foundation. Not only reuse of the basic C is achieved, but also the embedded extensions from the *mbeddr* are immediately supported by the newly built C++.

---

[5]C++ represents a multi-paradigm programming language, as functional programming or programming with templates can be seen as distinct paradigms. C, in comparison, is a single-paradigm language, with the only one procedure-oriented paradigm supported.

The ultimate goal during the reuse of *mbeddr* as a base for C++ is keeping *mbeddr* not modified towards the C++ programming language *only*, but instead, making, when needed, *mbeddr* more extensible *in general*, so that both resulting C++ and the base for it, *mbeddr*, can develop further being disjoint to a high degree. This independence of *mbeddr* from the C++ extension ensures, that the *mbeddr project* can develop further without looking back on C++, making the *Projectional C++* support an independent task.

1.4 Contribution

In this work we describe a C++ programming language implementation[6] on top of *mbeddr project*. Our contributions towards it are listed below.

Pure modular extension for *mbeddr*.   The task of a one-side-aware only extension is a challenge for the whole language modularity concept, provided by *JetBrains MPS*. This work explores further the support, provided by *JetBrains MPS* for the modular language construction, c.f. [7], and reviews it from the architectural point of view, summarizing finally the support for it, provided by *JetBrains MPS*.

Improved C++ programmer support.   This work contributes the C++ programming language with a number of automations. The automations include code generation and structuring. They are designed to compensate on some caveats of C++, or lack of support for several aspects in the language itself, like a support for a coding style.

Improving on C++ language itself.   C++ is known for a number of pitfalls, a programmer can be caught by. This work tries to improve on this situation by introducing analyses. The analyses are intended to increase understanding of a constructed code by a novice programmer, or to provide information to an experienced C++ professional. Analyses together with automations are provided to achieve an improvement in quality, security and understanding of the pure C++ code. The automations and analyses are mostly implemented as a programming on an *AST* in a Java-like programming language.

Analyses run-time and complexity problem researched.   As analyses and automations grow in complexity and quantity, the question of their computational complexity arises. In the *JetBrains MPS Application Programming Interface (API)* it is not explicitly defined, when analyses provided for a language take place, how much of a computational resource they can take advantage of, and how an end user should be informed on results and a progress. These aspects may affect the overall *IDE* behavior, including performance, as the analyses complexity may be high and the results of them could be of a high value. The question of analyses

---

[6]This implementation does not represent complete C++, and limitations are discussed along this work.

run-time, complexity and results presentation is raised and discussed in this work in general and in particular, suggesting improvements to *JetBrains MPS API*.

General language recreation principles.   Several principles, useful when recreating an existing language in a projectional environment, were formulated and supported by practical examples in this work.

Language extensibility in *JetBrains MPS* researched   Being one of the largest extensions to a language ever constructed in *JetBrains MPS*, *Projectional C++* represents a good basis to discover general language extensibility support limitations of *JetBrains MPS*. The support for extensibility in *JetBrains MPS* is reviewed considering each view on a language, workarounds for current *JetBrains MPS* version are offered, and improvements for future *JetBrains MPS* development are suggested.

Finally, it is fair to say, that the *mbeddr* team[7] have already grounded some practical foundations for the *Projectional C++*, before the start of this work. Some of them, were kept either as suitable (reference type) of for the further improvement (templates) and just described and analyzed here, some of them were considerably reworked (classes, inheritance, encapsulation and polymorphism).

Couching from the *mbeddr* team made us changing some of the implementation aspects to be different from what we planned originally (namespaces, operator overloading partially). And, of-course, we built some parts new, from scratch, without any influence (at least at the moment of being) *mbeddr* team at all (all analyses, construction and copying, some more).

The evaluation of the experience, gained by us during the practical implementation part, results into the extensibility analysis, research on complexity of checks and the run-time for them together with suggested *JetBrains MPS* improvements, and general guidelines for building projectional language implementation represent my own theoretical focus and commitment.

1.5 Structure of the Master's Thesis

In the Chapter 2 we describe in general two approaches, *IDE* developers can take when building a new *IDE* for a language, together with the language itself. We describe the traditional textual approach followed by the newer *projectional approach* used in this work. The language modularity and the way a language is described in a projectional environment are discussed. Finally, we shortly touch the problem of importing an existing code base into a projectional *IDE*.

The Chapter 3 is dedicated to the main technologies used in this Master's Thesis. At first an environment, providing all the facilities for building a projectional *IDE* is described, *JetBrains MPS*. Then, we describe the *mbeddr project* which serves as a modular basis for the present work practical achievements. Last, we give a number

---

[7]especially Markus Voelter

of references for the reader, who wants to get a better level of familiarity with the C++ programming language.

In the Chapter 4 we discuss the practical questions of the *Projectional C++* implementation. At first, we align, why the *mbeddr* C implementation can serve as a good basis for this work, and describe primitive extensions towards C++ for it. Next, we discuss all language features of C++ related to the object-oriented programming, and their implementation in the *Projectional C++*. After that, we describe the implementation of operator overloading and templates, as advanced C++ features. And finally advanced *IDE* functionality is described, including analyses and checks, supporting the C++ development.

In the Chapter 5 at first we compare the *projectional approach* and the textual approach. Then we list and describe the generalized principles, which could be used, when creating a projectional *IDE* for an existing language. Next, the *JetBrains MPS* support for language extensibility is reviewed. Later, we discuss analyses and their development problems, in a light of their computational complexity. Finally, briefly the outcomes of C++ concepts approach to C++ templates is evaluated.

The Chapter 6 concludes the Master's Thesis and suggests the potential future work.

2 Foundations

Before describing the technologies on which the current work is based, as well as the work itself, it makes sense to describe more general foundations and principles, around which the technology is built.

In the Section 2.1 we describe two approaches to create an *IDE* for a certain language, and mainly the projectional approach, which originates from the area of building new *Domain Specific Languages (DSLs)*.

In the Section 2.2 we describe a modular approach towards language engineering and extending, intensively used with the projectional approach to construct languages.

2.1 Building DSLs and IDEs

This section compares the traditional approach to build textual editors (or, broadly, *IDEs*) for a program code with the *projectional approach*, bringing up a motivation for the least.

2.1.1 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 a low-level code representation, especially when talking about syntax, which is only necessary for parsers to produce correct results, and not for the program intended semantics.

Parsers are used to construct so-called *AST*s from the textual program representation. *AST*s are structures in memory, usually graph-alike, reminding a control flow graph, where nodes are different statements, and edges are the ways, control passes from one statement to a next one.

For a 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 constructions 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 operations rely indeed on the higher-than-text level notions related to program such as a method, a variable, a statement, etc. A good editor has to be aware of these higher level program structures, to provide meaningful automations for the operations, which comprise the editor support to a user.

Nowadays, most of the editors work with a text, and, to provide assistance to a programmer, integrate with a parser/compiler front-end for the programming

language. Such way to extract a program structure during editing is not perfect for several reasons:

- The program edited as a text is not syntactically correct at every moment, having, for example, incomplete syntax constructions. Under such circumstances the parsing front-end can not be successfully invoked and returns error messages which are either not related to the program intended semantics, or false-positive warnings and errors.

- After a minor change of a 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 a large extent.

- 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 the refactoring correctly an editor must take into account various possible name collisions, as well as presume a compilable state of the program prior to the start of 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 preprocessor 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 2.1: Closing Several Blocks

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

- The textual representation of a program code, involves the need in formatting and preserving syntax. These both tasks, indeed, have nothing to do with the functionality of the program, and additionally load the developer, reducing the useful productivity. As an example, here we can mention the need to

close several blocks ending at the same point correctly, indenting the closing brace symmetrically to opening one. The Listing 2.1 demonstrates it in the last few lines.

2.1.2 Projectional Approach

Another approach an *IDE* creator can take when building an *IDE* is called *projectional approach.* Projectional editors do not work with a low-level textual representation of a program, but rather with a higher level structures, *AST*s. This approach is especially useful and is used when constructing new *DSL*s.

Working with *AST*s directly has several advantages over the conventional textual code editing:

- All syntax errors are no longer possible, as there is no syntax.

- There is no need to format the code on the level of indentation and look, since it is only needed for textual code.

- All features, which in textual approach require parsing, can be implemented without a parser involved, because *AST* is always known to the editor.

As most of the existing tools still expect a code in a textual form, code generation is used to convert an *AST* into a text code for the further use with these tools. The code generation step can be customized to provide support for a variety of compilers, when the compilers differ in parsing the code.

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

A model of a code is stored as an *AST* in a projectional editor. As in the Model-View-Controller pattern, the view for the model can be implemented separately, [8]. Thus the code may be presented in a number of different ways to the user. For example, an *AST* can be visualized as a graph, similar to the control flow graph. This visualization, however, is not always advantageous, being sometimes not compact and complicated to overview.

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

Figure 2.1: Example Projection of an *AST*, "Source Code" View

One of the well-spread way to visualize an *AST* is by visualizing its textual representation, as if it was written as a text code in the programming language, see the

Figure 2.1. There can be, in principle, many such textual visualizations, supporting different ways the code looks. Normally in the traditional approach this has to be achieved by reformatting, and thus changing, the source code. This is performed for the code to look similar across the developed software, and standards or coding guidelines are written to enforce the way to format the text code. This work to some extent is not needed when using a projectional editor, since the projection sets an fixes the way, code looks. Only higher-level than code formatting guidelines, have to be described an followed, unless captured as well by the projectional editor.

The textual projection of an *AST* looks similar to a text code. However the projectional nature of it has certain outcomes, which may be unusual for a programmer, who is used to editing the code as text.

The statements in a projectional editor are only selectable as whole. There is no way to just select the "while" word to cut or copy it, without selecting the condition and the block belonging to the statement. This behavior represents the position of the condition and `while`-body in an *AST* as children of the `while` statement. The statement can be selected all together only, including all of its children. Alternatively, one could select just the expression in the condition part, or just the `while` statement body.

Every block delimiters are just a part of the block visualization. They are organized in a proper way automatically, and there is no way to delete or confuse them, as well as there is no need to type them initially. Each closing brace can be marked with a parent statement name (through implementing such behavior in the *AST* visualization), enhancing navigation through the displayed code.

Among the benefits of the textual projection over text code are quicker code construction, after short learning, a more structured 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 listed above.

We discuss additionally the *projectional approach* and some of its basic principles, which we consider to be of practical value in the Section 5.2, when the approach is used to recreate an existing language in projection.

2.2 Modular Language Engineering

2.2.1 Describing a Language in Projection

When building a projectional editor for a language, the language must be given as a certain description of a possible *AST* in the language. As an *AST* represents a graph, the nodes and edges types, as well as their possible relationships must be described[1].

Nodes of an *AST* are described through defining their types. The node type in projectional editing is called a *concept*. *Concepts* are very similar to classes in object-

---

[1] Compare this with the textual approach, where a grammar for the language must be built, which is generally speaking complex, and some times even not a possible task, which leads to the increasing parser complexity, known problem in particular in the C++ area

oriented programming languages. They feature inheritance, they can implement interfaces, they can have internal data, similar to member fields, and they can feature behavior, similar to member functions. The difference with classes, however, is that the member fields are not usually encapsulated.

Edges of an *AST* are not described on their own, but instead as properties of nodes. A node can have children, or can reference other nodes. An example of a child relationship, can be a condition expression of the `if` statement. An example of a reference relationship can be a local variable usage, referencing the declaration of the local variable. The child and the reference relationships can have different cardinality, with minimal border from 0 or 1, to the maximal border of 1 or N, where N stands for just "many", or several.

The cardinality itself is not usually enough to restrict as desired node relationships. Special constraints can be added and checked for each relationship, which describe precisely, or provide a procedure to check, the validity of the relationship being established. A projectional editor must inform a user, every time, a constraint has not been satisfied, so that the user has a chance to correct the code, to match a description of a valid *AST*.

For a user to be able to manipulate an *AST*, for each *concept* an editor has to be created. The editor defines, how a node of a given *concept* should be represented to the user, which editing operations, and how, the user may perform on the node.

A minimal set of data was described above, which has to be defined for a language, to enable the projectional editing for it.

Additionally, constraints may be refined, involving some usual for typed languages type restrictions and checks. Generators can be added to transform *ASTs* given in a language. Text generators can be defined to generate a text code from an *AST*.

Behavior can be defined for a *concept*, to provide some method-like functionality to nodes of the *concept*. Additionally, some user-invokable functions can be described, to perform manipulations with an *AST*.

The process of defining a modular language in the *JetBrains MPS* environment is described additionally with practical details in the Section 3.1.

2.2.2 Language Modularity

As *concepts* feature inheritance, it is possible[2] to use a child *concept* node at a place where a parent *concept* node could be used. This creates a great opportunity for language extensibility. In order to extend a language at some point, just a passing base *concept* has to be determined and inherited from by a new *concept* which is meant to provide the language extension. It is possible to use the new *concept* immediately in a place of the base *concept*. As an example, one could think of extending statements of a language. The only a new statement is needed, a *concept* has to be created, which represents the new statement. Enabling the new statement consists of just inheriting it from the base statement *concept*, which exists in the extended statements language, Figure 2.2.

---

[2]to some extent, the extensibility is described separately in one of the following chapters
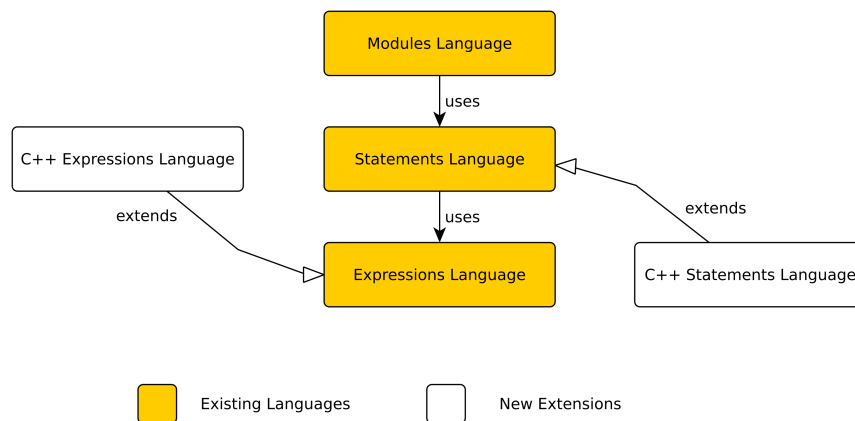
Figure 2.2: An Example of a Modular Language Reuse and Extension

Inheritance works over language borders, allowing to create a child *concept* in a language L2, separate from a language L1, where the original parent *concept* has been described. Thus the language L2 can be seen as a modular extension to the language L1.

Modular *DSL* creation is discussed in [9] and [10]. The language modularity, in a context of *JetBrains MPS* is described in [7] and [11].

The main focus of this work is the construction of the C++ programming language on top of the *mbeddr* C language implementation. Thus, modularity in language engineering plays a key role in the work.

While extending the C language of *mbeddr* with C++ specific *concepts*, all the aspects of the language description, as described in the Section 2.2.1, have to be extended. The newly introduced *concepts* for nodes of an *AST*, typical for C++, must inherit from some *concepts* of the original *mbeddr project* C language. Not only new node and edge types are introduced, but also constraints and other language description aspects have to be made incorporating them. The practical side of the language modularity and extensibility is discussed throughout this work.

2.2.3 Text Code Importers

When recreating an existing in a text form language in projection, it is natural to support some usual for language users code base. An example could be a standard library for the language. This code base is going to be present in a text form, as the language constructed has a textual nature. Thus, for a user of the language, to be able to use the code base, it has to be *imported* to the projectional editor created. The import process consists of parsing the text code, getting an *AST*, and converting the *AST* in an *AST*, as can be described by the language in projection.

The task of importing contains usually several principal challenges:

- The native text language can have more constructions, than the version, described in the projection. This happens, due to intentions to omit some of

them in projection[3], absence of them due to the simplicity of projectional implementation of the language, no need for them in projection, because the constructions are only specific to the textual nature of the code[4].

- The projectional language can contain more information about nodes, than is present in the textual language, thus the information has to be generated or manually added later.

- The technical work to create an importer can be considered significant, especially for complex languages, like the C++ programming language, as parsing front-ends may have complex *APIs*, or not be present at all.

---

[3]For example, dangerous constructions, like `reinterpret_cast` in C++.
[4]For example, preprocessor directives in C.

3 Technologies in Use

The C++ programming language, developed through out this Master's Thesis, is based on two technologies, which are introduced in this chapter. The first technology is the *JetBrains MPS* language engineering environment, which provides core foundations and means for incremental language construction. The second technology is the *mbeddr project*. These technologies are discussed in this chapter.

3.1 Jetbains MPS
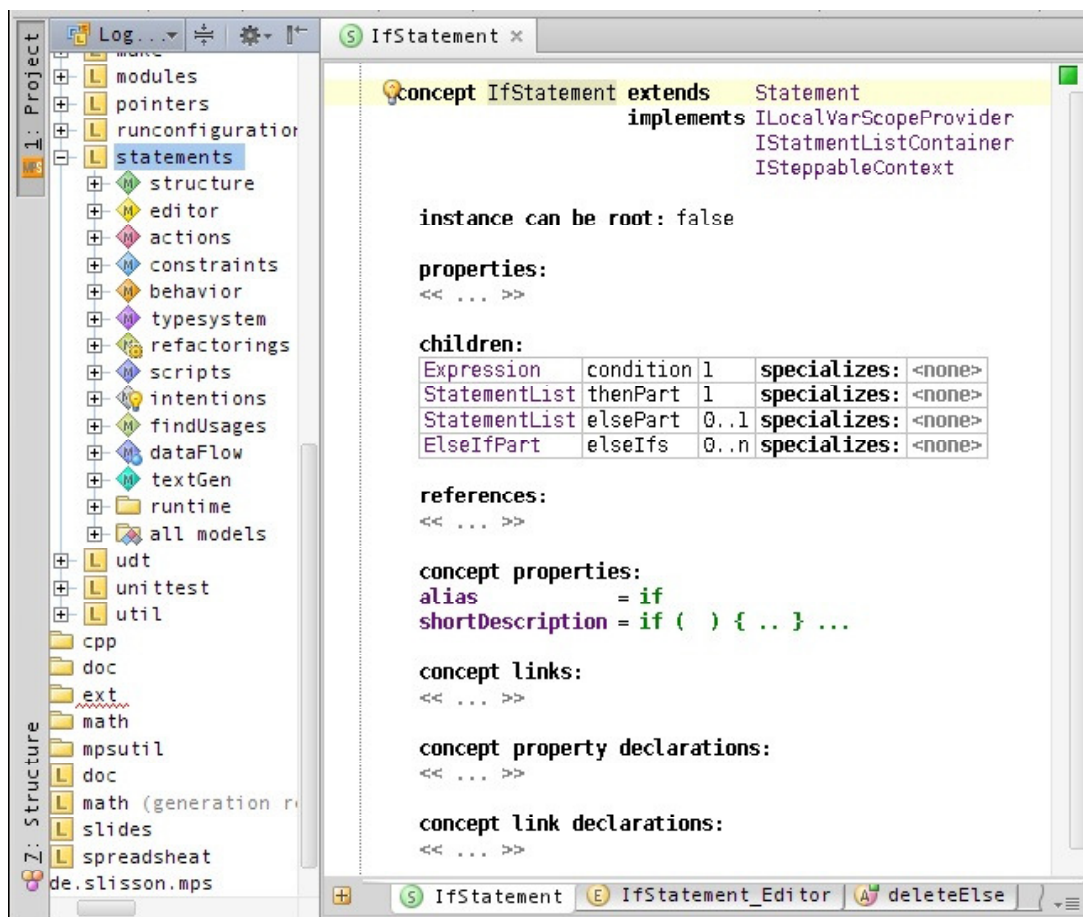
Figure 3.1: *JetBrains MPS* User Interface, **IfStatement** *Concept*

JetBrains MPS stands for JetBrains Meta Programming System. In this *IDE*-like software it is possible to develop *DSLs*. The approach taken in *JetBrains MPS* is

rather unique, and it is considered to be advantageous in many ways, [12].

In general, the way the language is described in *JetBrains MPS* corresponds to the way, the projectional editing is described in general in the Section 2.2. Here we will describe the process in practical details, as it is crucial for understanding the practical part of this Master's Thesis.

In this section we will go through a definition of one *concept*, describing the facilities, *JetBrains MPS* provides to support it. Each *concept* is described by several views on it.

Among such views there are a *concept* declaration (or language structure) view, an editor view, a behavior view, a constraints view, a type system view, a generator view and few more views. As a language consists mostly of the included in it *concepts*, the whole language is presented by the mentioned views as well, where each view on the language contains all views of the kind on each language *concept*. Left part of the Figure 3.1 demonstrates the views on the *statements* language.

The *mbeddr project* is a software, separate from *JetBrains MPS* but based on it, representing an extensible C language implementation with extensions. We will use *concepts* from the *mbeddr project* throughout this section as examples to demonstrate *JetBrains MPS*. Different C language parts are going to be decomposed into *concepts*, and these *concepts* are going to be defined using *JetBrains MPS*. The reader should not confuse though the *mbeddr project* and *JetBrains MPS* itself: the former is a software, developed in the latter. The *mbeddr project* and is used to demonstrate *JetBrains MPS* in this section.

Interestingly enough, *DSLs* are used in order to define new languages. Each view features a language used to describe a new *concept*. It is demonstrated in the following sections.

3.1.1 Concept Declaration

Concepts, as it is described in the Section 2.2, represent a class-like types for nodes of an *AST*. This terminology is kept in *JetBrains MPS* and MPS concept has the same meaning as *concept* term used in the Section 2.2. We use the term "concept" both in general, to describe an *AST* node type, and in particular referring to an MPS concept.

The Figure 3.1, on the right part, demonstrates a declaration of the **IfStatement** *concept* from the *mbeddr statements* language. It corresponds to the `if` statement of the C language.

At first, the *concept* is named, and a *base concept* is defined. For example, the **IfStatment** *concept* inherits from the **Statement** *concept*. This allows a node of the **IfStatement** *concept* to be used at any place at which a node of the parent **Statement** *concept* could be used, and inherits, like in object-oriented programming, all data and behavior of the **Statement**.

Next, it is defined, which interfaces the *concept* is going to implement. For example, by implementing **ISteppableContext**, the **IfStatement** supports the *mbeddr* debugger when stepping in the body of the **IfStatement**.

The "instance can be root" property defines, if it is meaningful to create a *concept* without a parent *concept* for it. In the case of **IfStatment** it does not make sense, as the statement should belong to some block. The `true` value can be used, e.g.

for modules, as they do not have any outer nodes, and can be seen as a document in *JetBrains MPS*.

The "properties" part defines if the described *concept* instances should have some primitive type data fields (string, boolean, integer). An example of a property could be a **name** property of a variable declaration. The **IfStatement** *concept* does not specify any properties, neither does it inherit any from the **Statement** concept.

The "children" section describe which nodes can be children of **IfStatement** on an *AST*. Each child is assigned with a role and cardinality. For example, a node of the **IfStatement** *concept* should have exactly one child of the **Expression** *concept*, it has a role "condition".

The "references" section describes in the similar way as in the "children" section, which nodes could be referenced by the node of a given *concept*. Referencing can be used, to bind a given node, to a node, located somewhere else on an *AST*. As an example, a variable usage in an expression shall reference the variable declaration, to express precisely, which variable is used.

Finally, some attributes of a *concept* follow, which do not have a primary importance for the discussion here. The "alias" is used to name a *concept* in a short way, to allow for quick instantiation in the editor. The "short description" is shown to hint a programmer on the alias meaning. A *concept* can be made abstract in the "concept properties" section. Abstract *concepts* are purely used in inheritance to create other non-abstract concepts with a common parent.

JetBrains MPS separately allows to define so-called *interface concepts*. *Interface concepts* are *concepts* which can not be instantiated, but which serve as a base for inheritance and implementing various declared by them behavior interfaces. A *concept* can have only one base *concept*, but can inherit from/implement many *interface concepts*. A reference or a child relationship can be typed with an *interface concept*.

3.1.2 Editor View
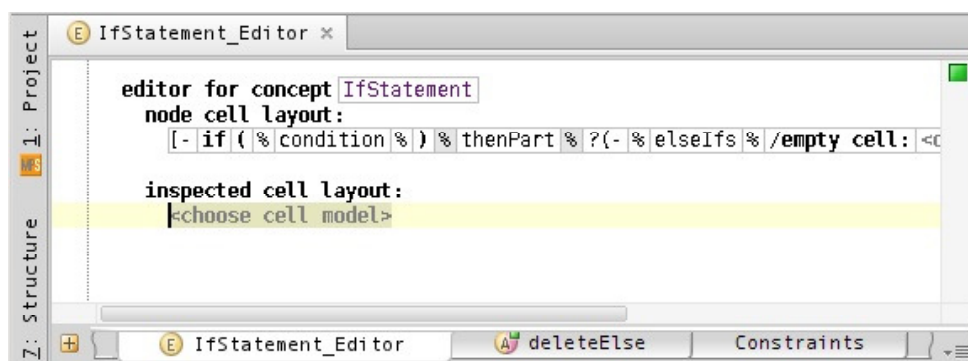
Figure 3.2: Editor View for the **IfStatement** *Concept*

The editor view, Figure 3.2, is designed to give a look for a node of 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 *JetBrains MPS* looks almost like a regular C++ code.

In the editor view one defines an editor for a *concept*. *JetBrains MPS* introduces here a bit confusing terminology, calling one editor for a given *concept* an "editor concept". Thus an error message "An editor concept not found for a concept X" would mean that no editor has been defined for the *concept* **X** in the editor view for it. In this work we call the content in the editor view for a *concept* **X** an "editor for the *concept* **X**". Similar terminology applies to all the following views related to a single *concept*.

The editor for a *concept* defines a visual representation for a node of the *concept*, using special syntax. For example, the Figure 3.2 defines an editor concept for **IfStatement**. At first the "constant" non-changeable by a programmer text is given, which is "if" and "(". Then the child **condition** is referenced, so that after the "if(" a user will be able to input an expression for the condition of the `if` statement. The editor can be configured to show or hide some parts of a node, depending on some condition, e.g. hiding `else` part of the `if` statement, if it is not defined. An editor concept is also responsible for all the interaction, a user experiences, when editing a node of the *concept*, to which this editor belongs.

3.1.3 Behavior View

The behavior view, can be used to define certain methods for a *concept*. A *concept* is represented there similar to a Java class, and it is possible to define methods in a Java-like language. *Concept* inheritance is taken into account like in Java.

A *concept* constructor can be defined there to initialize by default a newly created node of a *concept*.
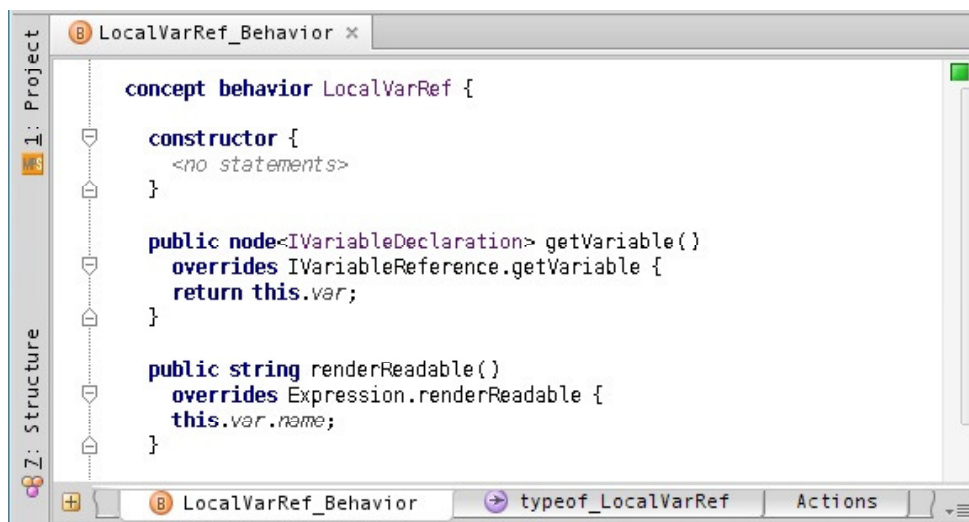
Figure 3.3: Behavior View for the **LocalVarRef** *Concept*

The Figure 3.3 shows the behavior of the **LocalVarRef** *concept*. This *concept* represents in the *mbeddr project* C language an expression, referencing a local variable. The local variable declaration is stored as a reference **var** in the *concept* nodes.

Two convenience methods are defined for the **LocalVarRef** *concept*, to get an easy access to local variable properties.

3.1.4 Constraints View


```
IIdentifierNamedConcept_Constraints ×

concepts constraints IIdentifierNamedConcept {
    can be child <none>

    can be parent <none>

    can be ancestor <none>


    property {name}
        get:<default>
        set:<default>
        is valid:(node, propertyValue, scope)->boolean {
            if (node.allowNonIdentifierNames) { return true; }

            if (node.isInstanceOf(ICommentedCode)) { return true; }
            if (propertyValue.matches("[a-zA-Z$[_]][a-zA-Z0-9$[_][-]]*")) {
                return !(CIdentifierHelper.isCKeyword(propertyValue));
            } else {
                return false;
            }
        }
    }

                    IIdentifierNamedConcept_Constraints
```

Figure 3.4: Editor View for the **IfStatement** *Concept*

The constraints view can be used, to limit in a desirable way *concept* property values and relationships, a node of the *concept* can have to other nodes. It is possible to take into account any sort of a context, and thus create a context-aware/sensitive *concept*.

The Figure 3.4 demonstrates constraints defined for a property **name**. This is a property of an *interface concept* called **IIdentifierNamedConcept**. All *concepts* which must have a name, confirming to the identifier naming restrictions, can implement the **IIdentifierNamedConcept**, to immediately get the desired characteristic.

The **name** property is programmatically restricted by the use of Java-like code snippet, Figure 3.4. In a similar way relationships to children and referents can be restricted.

It is possible to create a pure Java class within *JetBrains MPS* language, and use it in almost any *concept* view in *JetBrains MPS*. The Figure 3.4 demonstrates a use of the CIdentifierHelper class within the constraints for the *interface concept* **IIdentifierNamedConcept** to check a name property value on collisions with the C language keywords.

Constraints play an important role for the editor work. A programmer is presented with a list of choices, when inputting a new node on an *AST*. The choice of nodes is *defined* with the constraints. Such constrains are called "scoping con-

strains" in *JetBrains MPS*. A scope can be defined for a child or a reference of a *concept*. It is represented by a procedure, which takes a node in a context, and returns a list of possible children/referents after analyzing the given context. Exactly the resulting list is going to be presented to the programmer, as an options list for auto-completion, when building a child or a referent for the node.

3.1.5 Type System View

JetBrains MPS has a special support for creation of typed languages. Types are mainly used in expressions. An expressions may count on a certain sub-expression to have a given type, or a type, compatible with it. Whenever the expectation does not meet the reality, a warning or error can be displayed to a programmer.
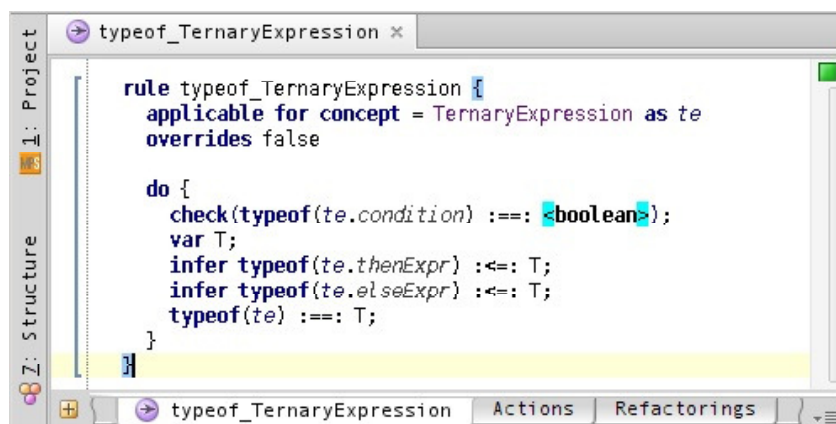
Figure 3.5: Type System View for the **TernaryExpression** *Concept*

A *concept*, representing nodes, which can have type in *mbeddr*, inherits from the **ITyped** *interface concept*. In the type system view a code in a special language has to be put, which will define the type, or type comparison rules, for the *concept*. Moreover, the type system language can be used to infer a type for a given node, see the Figure 3.5.

The Figure 3.5, demonstrates a use of the *JetBrains MPS* type system language to infer a type of the **TernaryExpression** *Concept* node. The syntax is on the one hand self-explanatory when reading, but, on the other hand, can be rather confusing when crafting.

When none of the existing type system rules can resolve a given typing problem, the so-called replacement rule can be invoked in *JetBrains MPS*. Replacement rules are defined for a given *concept*. A snippet of Java-like code must be given to explicitly analyze a node and assign a type to it.

3.1.6 Non-Type-System Checks

Additional to the type system and constraints limitations on an *AST* structure can be put via implementing non-type-system rules. In this work we often call them simply "checks".

For a given *concept* a code snippet can be given, to check all nodes of the *concept*. After the check is performed, there is a chance to mark a node as an error node, or create a warning on the node. It is possible to provide some textual hint for a language user, to identify a reason of the error or warning, see the Figure 4.19 for an example.

Non-type-system rules are used in this work to create *preventive analyses*, and are discussed additionally in the Section 5.4.3. Extensibility for checks is evaluated as extensibility for analyses, in the Section 5.3.9.

3.1.7 TextGen View

To make a use of a code in a projectional editor, further tools must be invoked on it, e.g. parser, compiler, etc. Normally they work with a textual representation of the code. In order to obtain the textual code from an *AST* in the projectional editor generators are invoked.

The generators can be of two kinds. The first kind of generators is dedicated to transform an *AST* in one *DSL* into another *AST* represented in a, usually, lower-level language. The second kind of generators is dedicated to transform an *AST* into text. Such generators are called "*TextGens*" in *JetBrains MPS*.
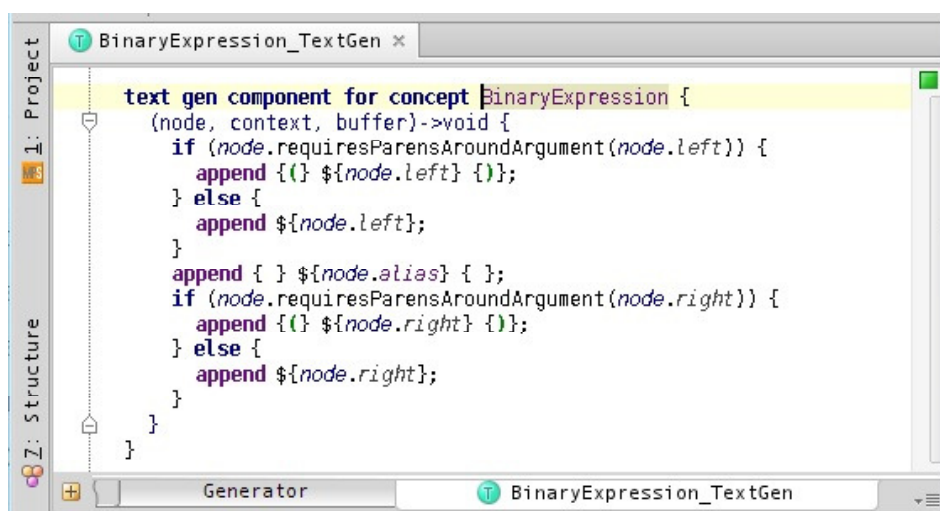
Figure 3.6: TextGen View for the **BinaryExpression** *Concept*

The Figure 3.6 demonstrates how a node of the **BinaryExpression** *concept* is converted to a text. It is noteworthy to say, that when rendering to a text the **left** and the **right** sub-expressions, the corresponding expression *TextGens* for them are invoked, making the text generation recursive.

3.1.8 Generator View

The generator view is one of the most complex *JetBrains MPS* views. A language engineer uses this view to define, how an *AST* composed in one language, has to be transformed into an *AST* in another language in *JetBrains MPS*. Alternatively, it

is used to define, how higher-level language extensions should be transformed to the basic constructions of a language, extended by them.

In this work we mostly use *TextGens* to produce a textual C++ code, when programming in the projectional editor has completed. Thus, the generator view does not have a strong connection to the present work. We do not describe it in details here.

3.1.9 Intentions

JetBrains MPS intentions are special procedures which can be used for automatic manipulations on an *AST* with a node of a given *concept*.


```
intention toggleConst for concept Type {
    error intention : false
    available in child nodes : true
    child filter : (editorContext, childNode, node)->boolean {
                    if (childNode.isInstanceOf(Type)) {
                        if (childNode.parent.isInstanceOf(Type)) {
                            boolean childCanBeConsted = childNode : Type.canBeConst();
                            boolean parentCanBeConsted = childNode.parent : Type.canBeConst();
                            return !(childCanBeConsted) && parentCanBeConsted;
                        }
                    }
                    false;
                }

    description(editorContext, node)->string {
        "Toggle Const for " + node.getPresentation();
    }

    isApplicable(editorContext, node)->boolean {
        return node.canBeConst();
    }

    execute(editorContext, node)->void {
        node.const = !(node.const);
    }
}
```

Figure 3.7: Toggle Const Property for a **Type** *Intention*

The Figure 3.7 demonstrates an intention, used to modify the const property of a node, belonging to the **Type** *concept*.

For an *intention* to be defined, one has to name it, specify a *concept*, to which the *intention* is applicable, provide a textual description for it, and finally specify a desired effect.

There is also a special kind of *intentions*, called "error intentions". They are not anyhow fundamentally different from the usual *intentions*, except their special purpose to fix an error, when one occurs. Error *intentions* are visualized using a red bulb icon in *JetBrains MPS*.

Intentions are accessible in the projectional editor from a context menu, when focused on a target node, see the Figure 3.8. They represent a useful mechanism to support a programmer with various automations, including automatic code generation.
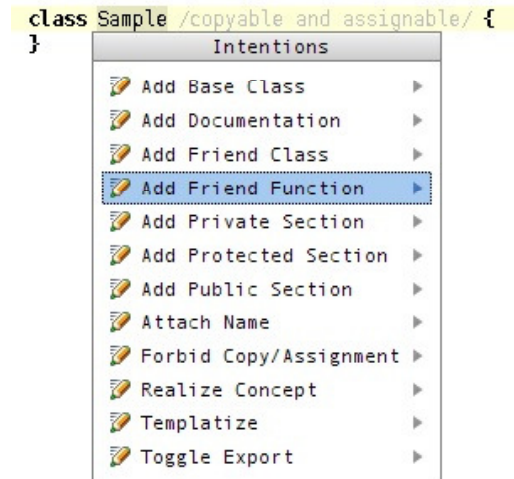
Figure 3.8: Intentions Available for the **Class** *Concept*

3.1.10 Other MPS Instruments

JetBrains MPS provides other instruments to enhance languages.

Actions are used to automate node deletions or editing. For example, deleting an array indexing expression, could be made to provide a substitution, an array expression itself, not indexed. Such behavior may seem more natural to a programmer, used to text editing.

JetBrains MPS provides a special support for refactorings. Special code snippets in a Java-like language can automate routine operations. As an example, a procedure of factoring out a local variable from an expression can be taken.

Additionally, it is possible to extend functionality of an editor under construction via creating special *JetBrains MPS* plug-ins. It can be in particular useful to implement some complex *analyses-on-demand*.

3.2 mbeddr Project

The *mbeddr project* is a software built with the use of *JetBrains MPS*. Mainly, it represents an implementation of the C programming language in the *JetBrains MPS* environment. Having embedded systems and software for them as the main focus, *mbeddr* provides certain language extensions to empower the programmer in embedded domain [13], [12].

Being a different from C language, the C++ programming language shares a lot of commonality with C. As *JetBrains MPS* allows, to some extent, see the Section

2.2.2, an incremental language construction, the *mbeddr project* represents a suitable basis for the C++ programming language implementation in *JetBrains MPS* .

We could not extend *mbeddr* with C++ pure incrementally. We had to perform some changes to *mbeddr* itself. The changes were introduced however, in a way to make *mbeddr* more extensible in general. Compare this to creating a separate branch of *mbeddr* designed only for C++. After such forking a separate team would have to support a newly created *mbeddr* version for C++. By making *mbeddr* more extensible, and by building C++ as a pure extension, we keep only one *mbeddr* version, and reduce the maintenance needed in the future, to keep C++ supported by *mbeddr*.

No matter the *mbeddr project* has (one) C language with extensions as an outcome, internally, as a *JetBrains MPS* software, it is represented as several *JetBrains MPS* languages.

All *mbeddr* languages are named starting from "*com.mbeddr.*" string. In this document we usually omit it, keeping the last word of the name only. For example, instead of *com.mbeddr.expressions* we call the language simply *expressions* here.

3.2.1 mbeddr Expressions Language

The *expressions* language contains definitions for all expressions, possible in the *mbeddr* C language. As in object-oriented programming languages, *concepts* of the `expressions` language form inheritance hierarchies. *JetBrains MPS* is capable of showing a given *concept* in a hierarchy.

Figure 3.9: *Concept* Hierarchy Example

The Figure 3.9 shows a hierarchy for the **MinusExpression** *concept*. In a similar way all expressions of the C programming language are implemented in the *expressions* language.

Whenever there is a need in the C++ programming language to extend the C programming language with a new expression kind, like object member reference, *new* expression and so on, a point of inheritance has to be found in the *mbeddr expressions* language to base a new *concept* on it.
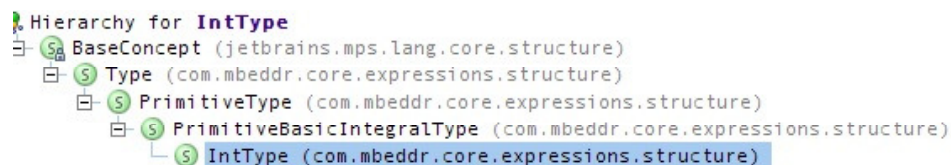
Figure 3.10: *mbeddr* Type Hierarchy Example

Additionally, the *expressions* language defines C language types. All *concepts* corresponding to C types are based (directly or indirectly) on the **Type** *concept*. For example, the hierarchy of the **IntType** *concept* is demonstrated in Figure 3.10.

In order to add a type to the *mbeddr* C language, one should inherit from the **Type** *concept* as well. Such inheritance automatically allows the new type to appear at all places, where a type in general can be found in the C language or its extensions.

3.2.2 mbeddr Statements Language

The *mbeddr statements* language contains definitions for the C language statements. The **Statement** *concept* serves as a base for inheritance, and represents by itself an empty line, or no-statement.

In order to create a new statement, like `delete` statement in C++, the inheritance should start from the **Statement** *concept*.

```
int16 abs(int16 x) {
  if (x >= 0) {
    return x;
  } else {
    return -x;
  } if
} abs (function)
```

Figure 3.11: Example of Multiple Languages Used Together

The *statements* language actively uses the *expressions* language, see the Figure 3.11. In an *mbeddr* code snippet the nodes, coming from *statements* language, are marked green, and the nodes, coming from *expressions* language are marked yellow[1]. As the example shows, **if** statement and **return** statement are coming from the *statements* language, but inside they contain, as children, expressions. This is an example of language modularity in *JetBrains MPS*, used by *mbeddr*.

3.2.3 Modules in mbeddr

In C (and in C++ as well) there is no clear notion of a module. The *mbeddr project* improves on it, defining modules, [12]. A C module is a *concept*, from which a header and a .c files are generated in *mbeddr*. Flagging an object (function, variable, structure, etc.) in a module as `exported` causes a declaration of the object to appear in the header file, and thus the object starts to be accessible by other modules or an external textual C code.

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

---

[1]Not marked with color is an instance of the *Function* concept, which comes from the *modules* language.

The implementation of modules in *mbeddr* can be found in the *modules* language. Functions are described there as well. For the **Function** concept example, see the Figure 3.11, the not marked with colors part.

Modules are further included into *JetBrains MPS models*. *Models* correspond to one single *JetBrains MPS* file unit. After a programmer finishes a code in a projectional editor, *JetBrains MPS* generates a textual C code, according to the rules, given by *mbeddr*. Each model is generated into a set of text C files. After the files have been generated, *mbeddr* launches a compiler to get an object code.

For the purpose of the code generation and compilation, every model should have a node of **TypeSizeConfiguration** and **BuildConfiguration** *concepts*. In the former, the programmer should specify all sizes for C language types, in the latter, the programmer should specify, how *mbeddr* has to invoke the compiler.

3.2.4 Pointers and Arrays in mbeddr

In the *pointers* language an array access and a pointer dereferencing expressions are defined as *concepts*. A similar syntax to them have overloaded operators with classes. Thus this language is also extended by *Projectional C++*, as some reuse is possible there as well. This is discussed in detail in the Section 4.3.

3.3 C++ Language

This work is basically an implementation of the C++ programming language in *JetBrains MPS*. No matter, the whole work is about the C++ implementation, it is infeasible to describe in detail the language here. Thus, we only give references to literature about the language in this chapter. Certain aspects of C++ are explained in depth during the description of the C++ implementation in *JetBrains MPS* itself in the Chapter 4.

Various literature is available, to get acquainted with the language closer. The most complete guide to the language, covering *STL* as well is [14]. A more easy-to-read and more suitable for beginners book on C++ is [15]. A newer book, oriented towards intermediate level C++ programmers and updated to the recent C++ 11 standard (referenced as [16]) and describing *STL* as well is [17].

A collection of techniques to get more effective C++ programs is gathered in [18]. Templates are explained in detail in [19]. A book dedicated to ad hoc polymorphism and advanced template meta programming can be also of interest as an approach to language engineering on top of C++ in a certain sense, [20].

The C++ programming language is a mature language, with long traditions, and high flexibility, [20] can serve as an example. It would not be possible to simplify the language, removing features from it, which would restrict the language use, most importantly the *STL* support. Thus in this work we try to research, how the editor can be more supportive for a user of the *existing* complex language, how to eliminate usual mistakes made while programming, as well as how to provide help in a code structuring.

4 Projectional C++ Implementation

In this chapter we talk about the implementation of the C++ language flavor, created by us. We call it, as said before, *Projectional C++*. At first we discuss the similarity and differences between C and C++ languages, making points, where extensions or changes were necessary to *mbeddr*. Next, we describe the support for object-oriented programming by *Projectional C++*. After, operator overloading implementation is discussed. Next, templates in C++ concepts apparoach to them are mentioned, followed by other C++ language features. Finally, we discuss the advanced *IDE* for *Projectional C++* features, including automations and analyses.

4.1 C and C++

Initially C++ appeared to be an extension to the C language, called "C with Classes" [21]. Till now a high degree of commonality can be found between the two languages. Mainly, the most of a C code is going to be a valid C++ code.

Some differences exist however, I introduce them here, together with the way they are adopted in *Projectional C++*.

4.1.1 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.

Reference Type

The reference type is basically a new construction in C++ which represent the old notion of pointer in C with a different syntax. In fact, behind the reference type stands a pointer to a base type, and it is dereferenced when needed.

The syntax for a variable of a type `T` can be used with a variable of a type reference to `T`, or `&T` as it is designated in C++. Thus the programmer benefits from a shorter syntax in comparison to a pointer dereferencing. It comes especially handy when passing arguments to methods by reference, which avoids creating a copy of an object to pass by value, and still preserves the short syntax without pointer dereferencing, when accessing the value.

Listing 4.1: Reference Type in a Copy Constructor

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

A 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 a constructor to give it a special meaning of a copy constructor for the class, as the Listing 4.1 demonstrates. This changes the whole semantics of the class, making it copyable with a given semantics. It is discussed in detail in the Section 4.2.1.

The reference type itself is implemented rather straightforward and similar to the pointer type of *mbeddr*. It inherits the **Type** *concept* of *mbeddr* as described in the Section 3.2.1, and has a base type as a child.

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 a 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 arguably 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 [16]. This can be suitable for the embedded domain better, conforming to the *mbeddr* spirit.

In the context of embedded systems, which *mbeddr* targets, it is often very important to know precisely, with which type the user is operating, as 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 a **TypeSizeConfiguration** node in each model, created with *mbeddr*, separately, see the Section 3.2.3.

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 the approach, when `bool` is translated to `int8_t`, one can consider `std::vector<bool>`. Since the word "bool" itself is never generated to a text, it is not possible to use the specialization of the template, which ensures storage optimization, through higher processing load when extracting a value from such a `vector` though.

The `bool` type from *mbeddr* is kept in *Projectional C++*. This decision is arguable, and can, of-course, be changed in the future.

4.1.2 Modules and C++

The C++ programming language is an improvement over the C language by itself. There is, however, no notion 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 clearly defined place for them. Potential name clashes are also disadvantageous. As an advantage, conditional compilation can be used in C++ to include different files with different implementations for one declaration, which is often used to

create a cross-platform C++ code. Additionally, classes can be used analogously to modules.

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

C++, however, does not elaborate on the disadvantage of having declarations and implementation not organized in one place, and being fully disjoint. *mbeddr* compensates on it, see the Section 3.2.3.

After comparing the two approaches of *mbeddr* and of the C++ programming language to improve on C, a hybrid approach has been taken. Firstly, C++ modules have been introduced. And secondly, namespaces can be defined and used in *Projectional C++*.

In *Projectional C++* modules are program modular elements, where a programmer declares and implements C++ classes, and .h 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 a header file, and introducing the changes in a declaration, and then rewriting a corresponding part of a .cpp file in strong correspondence with the header. The work decrease in order to change an interface can improve productivity of a programmer.

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

The need in the `using` directive does not emerge the same strong as in plain text editors, because the inputting of namespace names is made effective within the projectional editor. Long names of namespaces are appended to the end of method definitions for classes, to be exact about the namespace, but the declarations are kept looking short, with only the last namespace shown, Figure 4.1.

```
namespace MyNamespace {
  namespace MyInnerNamespace {
    class A /copyable and assignable/ {
      public:
        void DoSomething()
    }
    void A::DoSomething() from MyNamespace::MyInnerNamespace::A {

    }
  }
}
```

Figure 4.1: Example of Nested Namespaces

The TextGen is responsible for correct namespace resolution, and a *Projectional C++* programmer can benefit from short and clear namespace presentations, which are also quite handy to input.

4.1.3 Memory Allocation

In the C programming language memory (de-)allocation on the heap is happening via calls to dedicated library functions. C++ instead introduces separate language constructions for memory allocation and de-allocation. The `new` and `delete` keywords 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 a corresponding typing for the `new` expression, as described in the Section 3.2.1.

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 ([16]) 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 4.6.6.

4.2 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 support.

A class represents a new type in the C++ programming language. Each class may have data in a form of fields, and behavior in a 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 a creation of `public`, `protected` and `private` class sections.

Inheritance is implemented in C++ via allowing 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 a 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 sections.

4.2.1 Class Declaration and Copying

Visibility Sections

C++ Problematic   Instead of declaring visibility a modifier for individual class members (C#, Java), 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 a user to open and close the same section multiple times and declare sections without any particular order. This can be error-prone and confusing.

Various coding guidelines ([22], [23]) exist to enforce some restrictions on the visibility sections. In particular, the sections are allowed to be opened only once. This ensures, that a reader of the code will see an interface of a class (public section) in one place, "contents" of the class (private section) in one place, and opportunities to access members in an inheriting class (protected section) in one place, without a need to search through the whole class declaration.

Another typical requirement in coding standards, is the order of the sections. Usually the public section is required to be the 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 4.2: Sample Class Type Declaration

Implementation in *Projectional C++* The Figure 4.2, shows an example class declaration implemented in *Projectional C++*. The **Class** *concept* has the visibility sections as children. Each section is given a separate role (see the Section 3.1.1) and can appear 0 or 1 time. The editor for the **Class** *concept* orders the visibility sections so, that the public section always comes first, if present, followed by the private and finally the protected sections, when present.

The creation of a section is made with the a of *intentions*. The user uses *Alt+Enter* combination on a class declaration to create visibility sections. It should be more practical and fast for the user, compared to typing a keyword, a 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 the **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 concept. This limitation however is addressed in the newer *JetBrains MPS* 3.0 version, which was not tested completely with *mbeddr* by the time this work was made.

Constructors

C++ Problematic Constructors of a class 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 otherwise. Constructors gain an additional value, however, when participat-

ing 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 of the C++ language, adding conversion rules to it. Since this type extension can not be easily observed, as multiple consequent implicit conversions can take place, it is highly possible to get various *run-time* errors or unexpected behavior.

Listing 4.2: 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 printPerimeter(Circle c) {
  cout << "The perimeter is: " << c.getPerimeter();
}

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

int main(){
  // Potenitally unexpected behavior:
  printPerimeter(5);  //Prints: "The perimeter is: 31.4"
  return 0;
}
```

The Listing 4.2 demonstrates a simplified use case where the `printPerimeter()` function is invoked on `int` without any compiler error, and the resulting behavior is unexpected.

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

Implementation in *Projectional C++* The described problematic motivated the following decision. When a new constructor is created with one argument, it is by default declared to be explicit, a user must intentionally change it to get the type conversion behavior. Declaring a one argument constructor explicit is a safer *IDE* behavior by default. This is an example improvement to classical C++ which makes a code understood better by novices and safer for them.

Copying

C++ Problematic In the C++ programming language a programmer controls memory allocation fully on his/her own. This affects the way of copying for class instances. In C++ the programmer should define two methods for a class: a copy constructor and an assignment operator. These two methods work when an assignment like `a = b` happens, when instances of the 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. For example, if a pointer value is getting copied bitwise in a second instance and is deallocated twice in the following destructor calls.

Listing 4.3: Need in a Custom Copy Constructor

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

  ~Resource(){
    delete r;
  }
};

int main(){
  Resource a;

  // Pointers b.r and a.r are equal now.
  Resource b = a;
  return 0;
  // Here the two constructor calls happen.
  // The second call crashes the program.
}
```

The Listing 4.3 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, a programmer has to either define a proper copy-constructor or forbid copying of the objects for the class. The same applies to the assignment operator. Many standards require the two functions to be implemented in sync, i.e. implementing the same semantic behavior, [24]. This can be performed in an elegant way by implementing an assignment operator first and reusing it in a copy-constructor.

When not providing any copying behavior it comes logical to disable also the assignment behavior. This is done by a trick of declaring the corresponding functions in a `private` section of a class, without implementing them (as they never get called). To visually improve on such design, specialized macros exist in various libraries. For example the `DISALLOW_COPY_AND_ASSIGN` or the `Q_DISABLE_COPY` macros are described in [23] and [25]. An alternative to macros approach is a use of `boost::noncopyable` from the boost library, [26].

The use of macros in C++ appears often in similar cases, in order to perform some language-engineering tasks to add missing features to the language (copying or assignment deprecation in this case). Macros bear a pure textual nature, and are processed by the pre-processor. Some negative effects may come out:

- A need to preprocess 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 in general.

```
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 4.3: Hinting about Copyable and Assignable Class Properties

Implementation in *Projectional C++* The projectional editor allows for another solution, different from macros, or the use of a new library. In order to provide some support for a programmer regarding the copying issue, *Projectional C++* hints on a class declaration its assignable and copyable properties, see the Figure 4.3, and generates by default declarations of copy-constructor and assignment operator, when a class is created.

The copy constructor and the assignment operator are recognized by *Projectional C++*. Two *intentions* are provided on the **Class** *concept* to forbid or allow copying. The forbidding *intention* imitates the macros mentioned above, but displaying

and explaining the implementation to a user, see the Figure 4.4. The implementation of the intention consists of moving the declarations of two functions to the private section of the class. The allowing *intention* moves the declarations back to the public section, or creates them. The check for implementation, provided for all methods, flags the two declarations appropriately to finish the correct class declaration creation process, see the Section 4.6.4.

```
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 4.4: Class Made not Copyable by the *Forbid Copying* Intention

JetBrains MPS supports the *Projectional C++* implementation by providing read-only model accesses, special parts of the editor concept, see the Section 3.1.2, by which the hinting is implemented. The *intentions* enabled manipulations on the **Class** *concept*, 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 or libraries.

The whole work, a programmer needs to perform to forbid copying and assignment, consists 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, using them in a right way after (the corresponding macro requires exactly one parameter - the class name, and it has to go in the private class section). The boost library, [26], providing functionality to disable copying is often considered to be too heavy-weight to include, when it goes about compact development tasks, like the one described in this section.

4.2.2 Encapsulation and Inheritance

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

Various Cases of Access Control

C++ Problematic   In the C++ programming language a number of ways to govern the access control to class members exists. Before discussing the implementation of them in *Projectional C++*, we 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 a

code, which differ by the level of access they have to the class members. Among these locations are the class methods, friend functions of the class, and an external to the class code. Each member can be declared with a certain visibility/access type, and the inheritance of the base class can happen with one of the three inheritance modifiers.

```
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 4.5: Declaration of Two Classes with a Friend Function

Implementation in *Projectional C++* The Figure 4.5 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 of the class `A`.

This example demonstrates how, showing only the central information (the keyword `friend` and the friend function name) and just hinting on the details (the complete friend signature), can make the syntax more appealing. This is an example of the "Hide Redundant Syntax" principle, described in the Section 5.2.

The visibility plays no role for the friend function declarations themselves. That is why a decision was made to create a special section for friend declarations, called `friends`. This section name is not generated anyhow in the resulting C++ text. This allows for all the friend functions to appear in one place, and be easily observable.

The class `B` in the Figure 4.5 is inheriting publicly from the class `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 4.6.

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

Figure 4.6: 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 the second lines are making use of the 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 its 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. This means, no node in an *AST* is created and the code is in incomplete erroneous state.

The `compare()` function is declared in advance (Figure 4.5) 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 `printOut()` is not related anyhow to the classes declared. Thus, it represents "external" for the class `B` code. Only the public members of the class are accessible, but not the protected or the private members. The attempt to input them simply fails, they are highlighted red and are not bound to anything.

In this way *Projectional C++* gives for a programmer only a chance to input correct from the encapsulation point of view constructions. As members, accessible in each place of code, are provided by *Projectional C++*, instead of typing the member name, the programmer usually will have a choice from a short drop-down list of options.

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 represents nothing else, but an expression itself.

Thanks to *concept* inheritance in *JetBrains MPS* , see the Sections 2.2.2 and 3.1.1, creation of a new expression is reduced to inheriting from an existing one, and this is almost all what has to be done to implement object-oriented member access expressions. Inheriting the **OoDotOrArrowExpression** *concept* from the

Expression *concept*, we get the ability to use the expression, designed for member access, wherever an expression in general can be used.

The abstract *concept* **OoDotOrArrowExpression** serves as a parent for the two *concepts*: **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 class methods it is also possible in C++ to address class members as local variables. In the projectional implementation described, it is implemented as well. The member references are highlighted blue. This is made so do differentiate the members from any other local variables. This can be a subject for coding guidelines, similar to naming conventions as described in the Section 4.6.3.

4.2.3 Polymorphism

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

In this section we are writing only about the class-related virtual functions polymorphism, and the way it is implemented in *Projectional C++*.

Virtual Functions Polymorphism in C++

At first, we describe the polymorphism, as it is implemented in the C++ programming language, pointing out the places, where it could be improved.

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

Listing 4.4: Pure Virtual Function Syntax Example

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

The virtual functions in the base-interface class can be implemented as well, providing some "default", common enough behavior. 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[1] and a bit cumbersome requiring to type one reserved word, one punctuation sign and one digit to express a simple fact of pure virtuality or, simply, absence of implementation, see the Listing 4.4.

The approach of classes with virtual functions as interfaces is more flexible compared to languages with the notion of an interface directly introduced. In C++ it is possible to create partially implemented base classes, what can not be done when implementing an interface in Java or C#, where the approach is "all or none" regarding the implementation of interface functions. The presence of interfaces in the language can be though considered a more clean way to program.

In order to implement a declared in a base class function, a descendant must declare and implement a virtual function, matching the full (including a return type) signature of the declared in the base class function, see the Listing 4.6. The connection to the "interface" function declaration stays subtle however. It is not immediately clear, whether the declared new function in the descendant is an override of an existing in the base class function or an independent declaration of an entirely new function in the descendant class. This knowledge affects the changing process greatly, as the override should change from the interface, together with all the implementations. The absence of a clear, explicit override syntax we call here an "*override syntax absence*".

Whenever a class and all of its ancestors do not provide an implementation of a certain virtual function, created as a *pure virtual* in the declaring class, the class is called an *abstract* class. It is not possible to construct instances of an *abstract* class. C++ however does not have any special syntax to explicitly declare a class *abstract*, see the Listing 4.4. The programmer usually has to be aware (from documentation, implementation, or, the worst case, compilation errors) whether a given class is *abstract*. We will call this phenomenon an "*abstract class syntax absence*".

Overriding a function is an active action of the programmer, and it is initiated by the programmer. I.e. the programmer wants to state, that a new function is designed to override an existing one, and which is the overridden function. The abstract property of a class, oppositely, is not a quality a programmer directly gives to a class in C++. It can be rather deduced from the analysis on the base classes automatically, by editor. So in this case no actions are needed from the programmer side. Because of this, the two similar, at the first glance, absence of syntax phenomena are resolved or differently from in the *Projectional C++* implementation.

In order to use an interface, declared in some class, the using code has to get a pointer to an instance of any inheriting the interface descendant class instance. Thus, the type system has to allow a pointer to the descendant to be treated in the way as a pointer to the base class would be. The same should hold, normally, for the reference types, but it is not used very often in practice, and is omitted in the implementation.

As this typing rule represents the core of polymorphic behavior, we will start from it below, describing further the polymorphism in the *Projectional C++* implementation.

---

[1]Especially when not knowing about the zero pointer value semantics.

Special Typing for a Pointer to a Class

The problem solved here is enabling the usage of pointers to descendants instead of pointers to ancestors, see the Listing 4.5.

Listing 4.5: Example Usage of a Pointer to Descendant Class

```
class Shape {};

class Circle: public Shape {};

void draw(Shape* shape){
  // Some drawing implementation,
  // delegating, perhaps, to the object.
}

int main(){

  Circle* c = new Circle();

  // A call bellow happens
  // with a Circle* type instead of
  // the declared Shape* type.
  draw(c);

  return 0;
}
```

The **PointerType** *concept* is implemented in the *pointers* language in *mbeddr*. Following the goal of non-invasive changes to *mbeddr* the *Projectional C++* implementation needs to add the typing rules for pointers *to classes* without changing the *pointers mbeddr* language, where the typing system for the 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, like in the Listing 4.5. 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 *Projectional C++* extensions[2] by design.

In a case like this a replacement rule can be used, see the Section 3.1.5. *Projectional C++* provides such a replacement rule for the pointer type, which checks, whether a class pointed to, is a passing descendant of the class required by an expression, where the pointer was used, and performs the necessary typing.

In the Section 5.3 we discuss more on the approach taken here, and its limitations.

---

[2]The **ClassType** *concept* in this case.

4.2.4 Safer Casting for a Pointer to Class

C++ Problematic   The type casting for class types in C++ has a number of disadvantages.

- The `const_cast` is usually a signal of a design error, since the declared `const` property of a value is taken away.

- The `static_cast` allows for wrong conversions from a base class to a derived class, which could cause a failure in the run-time, as no checks are performed.

- The `reinterpret_cast` allows to ignore the possibility of a completely meaningless conversion, which can lead to run-time crashes, neither run-time nor compile-time checks are performed.

- The `dynamic_cast` has usually low compile-time support, and relies on the run-time type information to perform checks at the run-time. This information may not be available, being, for example omitted for the sake of performance gains.

Implementation in *Projectional C++*   A new construction for type conversions is introduced, which can be considered a slight improvement over the existing in C++ casts. The **as** construction translates to the C++ `dynamic_cast`. Additionally, checks are added to improve the security of the construction.

```
class NonPoly /copyable and assignable/ {
  public:
    void hello()
    int32 getFive()
    NonPoly() (constructor)
}

class NPChild : public NonPoly /copyable and assignable/ {
  public:
    NPChild() (constructor)
}

testcase NonPolymorphicCasting {

  NonPoly* parent = new NonPoly();

  ( parent as NPChild* )->hello();

  assert(0) ( parent as NonPoly* )->getFive() == 5;

} NonPolymorphicCasting(test case)
```

Figure 4.7: Example of the **as** Construction

The Figure 4.7 demonstrates some of the introduced in *Projectional C++* features related to the class type conversions.

The new **as** construction is being checked in the coding-time. For example, the cast from `parent` to `NPChild` is signaled as an error, since these class hierarchy is not polymorphic, and the `dynamic_cast` in the basis of **as** would not work on it.

The next cast from `parent` to `NonPoly` is signaled with warning, because the `parent` pointer does *already* have the requested type. Such casting may signal a lack of understanding by the programmer while coding. The **as** construction returns `0` at the run-time when conversion fails. Thus a good programming style should include a check for it at the run-time.

When **as** is used to cast a polymorphic class, it checks, whether the cast makes sense. For this the related classes must be in the same class hierarchy. When a zero result is possible, the node is highlighted with warning, for the programmer to take an action of checking. Casting between unrelated classes is presented as an error.

Some implicit type conversions are always meaningful, thus a conversion from a pointer to a child class to a pointer to a parent class is recognized by *Projectional C++* and is allowed, as described in the Section 4.2.3.

The casting can be further improved through introducing new analyses. At first, a fast type of cast, like `static_cast` or even `reinterpret_cast` can be used in translation, when a data-flow analysis can prove, that the conversion is always meaningful. Secondly, unnecessary casts can be eliminated in this way.

Overriding a Virtual Function

The Listing 4.6 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 4.6 demonstrates as well the *override syntax absence* in C++.

Listing 4.6: 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 4.8 the same example is demonstrated, but in *Projectional C++*. In fact, the code shown in the Listing 4.6 can be generated from the demonstrated projectional sample in the Figure 4.8, by invoking all *TextGens* in *JetBrains MPS*.

When a method declaration is created, it can be set to be an override of a method in a base class. Right after 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. This is yet another work saver for a programmer.

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. Thus if the latter changes, it is going to be indicated in the former. The overridden method is shown next to the override declaration, see the Figure 4.8, which compensates on the *override syntax absence*.

```cpp
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 4.8: Example of an Overridden Function - *Projectional C++*

The additional, not belonging to C++, syntax in *Projectional C++*, should not confuse the reader. It is only present in the projectional editor, and, when an *AST* is generated into a text code, a regular C++ syntax is achieved. However in this case an *AST* stores *more* than needed to generate and compile a C++ code.

This subsection is one of the examples, that storing more information can be useful, it is generalized in the Section 5.2.

Pure Virtual Functions

In the example code in the Listing 4.6 one improvement to could be made. As the `getArea()` for a random shape can not be determined, it makes sense to make the `getArea()` function *pure virtual*. As said before, a *pure virtual* function has no implementation in the declaring class, and serves only the overriding purpose. Semantically in C++ it sets the pointer in `vptr` table to 0 and thus has reflecting the zeroing syntax, see the similar example in the Listing 4.4. This syntax can be seen as not obvious, as it reflects more the under-the-hood implementation of the mechanism, rather than the original programmer intention to built a basis for an override chain, while taking the advantage of polymorphism. Additionally, as discussed above, declaring a *pure virtual* function requires a significant amount of the syntactical overhead. These were the disadvantages of the C++ programming language, *Projectional C++* tries to improve on.

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;`" a programmer has to input only the name `getArea`, pick the type `double`, and toggle the "Make pure virtual" intention for the declaration in the `Shape` class. The result of the intention work can be seen in the Figure 4.9.

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

Figure 4.9: Example of a Pure Virtual Method - *Projectional C++*

The word `pure` is added by the projectional editor. This makes the reading of the code easy and natural. The "` = 0`" part is preserved for the C++ programmer with habits, used to the original C++ 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 at all.

This example demostrates, how the lower-level syntax can be made more readable. The general principle on it is formulated in the Section 5.2.

Abstract Classes

If any of *pure virtual* functions, in an inheritance chain, leading to a class, is not implemented by this chain, or inside the class itself, the class is called an *abstract* class. It is not possible to construct an instance of an *abstract* class, as such classes have not implemented methods.

A programmer has to know which classes are *abstract*, and are intended for inheritance and further implementation only. One of the reasons for this, is to not to try instantiation of an *abstract* class. Another reason is to exactly identify *abstract* classes, designed to serve as extension points. Above we discuss the *abstract class syntax absence* in C++. It leads to the need for the programmer to determine somehow him/herself if a given class is *abstract*, the source code representation of a class does not give any information on it, unless some naming conventions require *abstract* classes to be named specially. An improvement to this situation would be a behavior, when an editor can perform an analysis and determine if a class is *abstract*, hinting on it to the user. In the case of the projectional editor, this analysis is especially computationally efficient[3], as an *AST* is readily available, and a quick analysis can be performed by a simple recursive algorithm on inheritance chains.

After an *abstract* class has been determined, it is possible to modify the editor representation for it, and show that it is *abstract*, see the Figure 4.10. 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 implementations, e.g. `PushButton` in the example, or check boxes, radio buttons and similar, which could be alternatively created.

---

[3]In comparison to textual editors, with the need to parse, see the Section Section 5.1 on comparison to the textual approach, and Section 5.4 on complexity of analyses.

```
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 4.10: 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 a code.

This example demonstrates a general principle, see the Section 5.2, of an advised practice to perform quick analyses and inform the user.

Additionally, the creation and usage of *abstract* class instances is checked, and forbidden by type analysis. This is described separately in the Section 4.6.

4.3 Operator Overloading

Listing 4.7: String Concatenation with Overloaded Operator +

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

int main( )
{
    string s1 ( "Blue" );
    string s2 ( "berry" );
    cout << "The string concatenation is: " << s1 + s2 << endl;
}
```

C++ Problematic The operator overloading feature in C++ allows to redefine semantics for a given operator when used with certain types. Indeed, this represents a language engineering methods, when some language constructions (operators) are extended to support new features (work with new types). The projectional way to support such needs would be extending the language with new constructions.

However, moving towards the *STL* support, it is good to provide the operator over-loading usual for C++.

Most often, the operator overloading feature is used together with class types. A very common examples are iterators with their increment, equality and derefer-encing; standard input and output streams with stream operators << and >>; and `std::string` concatenation with operator +.

In the Listing 4.7 one can find 4 overloaded operators:

- for `std::string` output,

- for `const char*` output,

- for `std::endl` output, and

- for `std::string` concatenation.

Implementation in *Projectional C++*   We will discuss the implementation of the operator overloading in *Projectional C++* based on the example. The Figure 4.11 shows (partially) a definition of the `Coords` class.

```
class Coords /copyable and assignable/ {
  public:
    Coords() (constructor)
    Coords(int32 xx, int32 yy) (constructor)
    Coords operator + (Coords arg )
    Coords operator - (Coords arg )
    int32 operator [] (int32 index )
    int32 getX()
    int32 getY()
  private:
    int32 mX
    int32 mY
}

int32 Coords::operator[](int32 index) from Coords {
  return (index == 0) ? this->getX() : this->getY();
}

Coords Coords::operator+(Coords arg) from Coords {
  return Coords(this->mX + arg.mX, this->mY + arg.mY);
}

Coords Coords::operator-(Coords arg) from Coords {
  return Coords(this->mX - arg.mX, this->mY - arg.mY);
}
```

Figure 4.11: Operator Overloading: Definition

The class represents a vector in a two-dimensional space. Three operators are overloaded for it:

- `operator +` performs the addition of two vectors,

- `operator -` subtracts one vector from another one,

- `operator []` allows to reference the x and the y coordinates of a vector in the array fashion.

Other members of the `Coords` class should be self-describing, given the vector semantics.

Inside the class the overloaded operators get declared. A reader can notice a high degree of similarity between an operator overload declaration and a method declaration, as both have a return type and arguments. Thus a common base *concept* was created, **AbstractImplementableAsMethod**. It allows to link an implementation to it, or a method definition, in C++ terminology.

The difference between a method declaration and an overloaded operator declaration, is that the overloaded operator declaration has an operator designator instead of a name. An abstract *concept* **OperatorDesignator** has been created to serve as a base for all possible operator designators for overloading, like +, −, [] in this example and more.

Creating a concrete operator designator, and inheriting it from the abstract *concept* **OperatorDesignator**, makes it possible to define a new operator for overloading. Thus declaration and definition of an operator overload task is resolved in a modular way.

Left stays the way, overloaded operators can be incorporated into the existing hierarchy of expressions.

```
Coords v1 = Coords(1, 2);
Coords v2 = Coords(2, 3);
Coords v3 = v1 + v2;

assert(0) v3.getX() == 3;
assert(1) v3.getY() == 5;

Coords v4 = v2 - v1;

assert(2) v4.getX() == 1;
assert(3) v4.getY() == 1;
assert(4) v4[1] == 1;
```

Figure 4.12: Operator Overloading: Usage

The Figure 4.12 demonstrates, how the defined operator overloads could be used. The `assert` statements are taken from the special *mbeddr* language designed to create test cases. They test the condition and if it is `false` the execution stops.

At first, in the example two `Coords` objects are created. Next, the third object is created as their sum. The + operator used is nothing else but an instance of the **PlusBinaryExpression** from *mbeddr*. Next, the newly created `v3` object gets tested. After that, the `v4` object is created as a difference between `v2` and `v1`. Again, the operator −, used to calculate the difference, is an instance of **MinusBinaryExpression** in *mbeddr*. Then, `v4` values get tested. And finally, the [] operator is used with `v4` to get its second coordinate. The operator used is an instance of the reused from *mbeddr* **ArrayAccessExpression** *concept*.

The reuse of existing expressions from *mbeddr* is highly beneficial. For example, the **ArrayAccessExpression** features a complex behavior. At first the indexed

object is typed as an expression. After `[` gets typed by the programmer, the indexed object is transformed into an instance of the **ArrayAccessExpression** *concept*. The **ArrayAccessExpression** has two children. First is the expression which corresponds to the array. The indexed object is moved to be a child of the new **ArrayAccessExpression**. The second child is an expression, which represents the index. It gets the input focus, after the replacement happens. Next, if the indexed expression is about to be deleted, a special action[4] takes place, which replaces the **ArrayAccessExpression** with its child with the role[5] `array`. Thus the deleting of index resembles the usual behavior as in text editors, where just the index disappears, not the whole expression together with the indexed object, as it would happen in projection without the action provided.

The decision to not to create special expressions, which would represent the usage of the overloaded operators, helps to avoid code duplication, reprogramming the behavior of the existing expressions in *mbeddr*.

Alternatively, the expressions in *mbeddr* could be inherited and changed. But the degree of changes, possible through inheritance is limited, as described in the Section 5.3, and creation of the second set of the same-looking expressions could potentially confuse a programmer, when he/she is going to instantiate one of the many + operators, for example.

The existing expressions could not be reused as they were, however. Their type system, specified in *mbeddr*, is not aware of the **ClassType**, introduced by *Projectional C++*. Thus the class objects would get rejected by the type system, as the type for the resulting expression would not be figured out by *mbeddr*.

Changing the *mbeddr* type system, hard coding the necessary behavior for classes, would introduce a dependency of *mbeddr* code on the *Projectional C++* code. The goal of this work, however, is to avoid such situations, as described in the Section 1.3. The solution for this problem was found to be as follows. For each of the expression kinds in *mbeddr* an *interface concept* could be introduced, which would participate in the type system for the expression, and, if found to be one of the expression parts, could be delegated with the task of the type calculation. Then, the **ClassType** is changed to implement the newly created *interface concepts*, and the code is added to it, to compute the typing. The **ClassType** looks up the corresponding **Class** for the presence of overloads, and if an overload for the operation in question is found, the overloaded operator return type is taken as the result, otherwise an error is reported.

For example, the **ClassType** implements a newly created in *mbeddr* concept, *interface concept* **ISelfTypingInBinaryExpression**. Thus, when a type of a binary expression is calculated, the **ClassType** is delegated with the task to determine the possibility to be involved in the binary operation under question, and to calculate the resulting type. In the Figure 4.12 when calculating `v3`, the **PlusBinaryExpression** delegates to the **ClassType** corresponding to `Coords` class, to figure out, if the operation is possible. According to the declarations in the Figure 4.11 the `Coords` class has an overload for the + operator, the argument type is `Coords` class, thus the operation is determined to be possible, and the re-

---

[4]See the Section 3.1.10
[5]See the Section 3.1.1

turn type of the operator, namely `Coords` class, is taken as the result type for the **PlusBinaryExpression**.

As the **ISelfTypingInBinaryExpression** *interface concept* is declared in the *mbeddr project* code, the only dependency created by using it, is the existing dependency of *Projectional C++* on *mbeddr*, keeping *mbeddr* development separate from *Projectional C++*.

The current *Projectional C++* implementation has some limitations in the operator overloading:

- Not every operator is ready for overloading, and *mbeddr* has to be further extended to support it, this is, however, not an entirely new task, but rather repetitive application of the approach described above.

- C++ features not only member operators, but also static operator overloads, this is not supported by *Projectional C++* in the current version, a very similar to member operators approach can be taken to implement it.

4.4 Templates

C++ Problematic   Templates represent a powerful tool in the C++ programming language. In fact, templates, especially used together with a preprocessor, represent a language engineering tool, which allows for extending C++ with additional constructions, not originally present in the language, [20]. In this regard an approach taken in the projectional editing is an alternative, as the projectional language modularity is considered to be a basis for the language extending, see the Section 2.2.2.

However, templates have to be supported to some extent in *Projectional C++*, as *STL* is based on templates entirely. Without supporting templates, it would not be possible to provide a usual for a C++ programmer standard library. Mostly templates are used to abstract over some type in *STL*.

Several Disadvantages of Templates in C++   Templates bear a pure textual structure in C++. A template code is not even syntactically checked before instantiation, i.e. template code is not even parsed before it gets explicitly used, or instantiated, in a compiled, working, code.

When instantiating a template, assumptions, taken in the template code on a template parameter, are checked against a concrete template parameter. These assumptions are implicit in C++. The template parameter is assumed to be capable of everything, which is possible in C++. Participation in all types of expressions and statements is assumed, in each role.

The described features of a template code can present a source of various kinds of errors, for instance:

- A syntactic error found on the time, the template code is used for an instantiation first, and not before, when the code is created

- A template code receives precise semantic meaning, only during instantiation. For a user to understand, what the template code really does, the user

has to know the implementation of the template, namely, how exactly the real parameters are going to be used.

- Assumptions, put on a template parameter, may conflict, when one template parameter is to be used with several template code fragments, putting different assumptions on the parameter.

- The assumptions are hidden from a programmer being implicit requirements in the template code. Staying implicit, they can be not fully observed, or understood wrongly.

The Conflict brought by the Textual Nature of Templates In the projectional editing, constraints are required, as they define, what can be constructed in principle, see the Section 3.1.4. In other words, before using a certain object to construct with it nodes on an *AST*, in the projectional editor it has to be precisely known, what this object is capable of doing, how it can be used. This is defined through the special "scoping constraints", see the Section 3.1.4. For example, when an instance of a class is used, it has to be known *in advance*, which methods the class has, in order to call them on the instance. When using, for instance, a template parameter type, it is not known in C++, what are its capabilities. They are being determined later, as the mentioned above implicit assumptions, put on to the type, when it was used by template code. This example represents the "unconstrained" nature of template parameters in C++. It is a contradiction in nature of templates in C++ and the projectional editing with constraints in general. A special modification to C++ language is presented below as a way to resolve the formulated problem.

Implementation in *Projectional C++* As a way to support templates in *Projectional C++* we introduce a term a "C++ concept". This term is not to be confused with *concepts* in projectional editing.

A C++ concept explicitly describes assumptions put on a template parameter by a template code. A template code is required to declare, which C++ concept a template parameter belongs to, before the parameter can be used. The C++ concept defines for the parameter the way the parameter could be used in the template code through a mechanism, similar to giving a type to the parameter. This generalized type is described by the C++ concept itself. Thus the C++ concept describes, how the parameter can be used. In *Projectional C++* the scoping constraints code is analyzing the C++ concept of the parameter, to determine, how it could be used.

The Figure 4.13 demonstrates how a template code can be composed with the use of C++ concepts.

At first, the `Comparable` C++ concept is declared. It puts a requirement on a type to have a function `compare()` in its public section. Next, a class called `NumberWrapper` is declared to `realize` the `Comparable` concept. The editor will check the class upon the requirement.

```
concept Comparable {
  public:
    int8 compare(Comparable c1)
}

realizes Comparable
class NumberWrapper /copyable and assignable/ {
  public:
    int8 compare(NumberWrapper other)
    NumberWrapper(int8 v) (constructor)
  private:
    int8 mValue
}

template <class T: Comparable>
class OrderedList /copyable and assignable/ {
  public:
    OrderedList() (constructor)
    int8 compare(T first, T other)
}

int8 OrderedList::compare(T element, T other) from OrderedList {
  return element.compare(other);
}
```

Figure 4.13: Template Code Sample in *Projectional C++*

A template class `OrderedList` declares a template parameter `T` and specifies that it satisfies the `Comparable` concept. This makes an object of type `T` usable as if it was satisfying the requirements as declared by `Comparable`. This is demonstrated in the `OrderedList::compare()` function.

When instantiating a template, it is checked if either the parameter given realizes the needed C++ concept, or, more flexibly, if it satisfies the C++ concept requirements, without declaring the realization of the C++ concept explicitly.

The support for templates by *Projectional C++* is, of-course, limited, when compared to traditional C++ facilities. It is evaluated in the Section 5.5.

4.5 Other C++ Language Features

There are some C++ languages features, which are to be worked on with *Projectional C++*. In this section we briefly mention them.

4.5.1 Exceptions

Exceptions are not yet supported by *Projectional C++*. The primitive support could be easily achieved by implementing `try{}catch()` and `throw` statements.

However, it represents a special task for the future, to make a special, improved over original C++, exceptions support. This may include:

- An analysis for exceptions to be caught always, including a check for correct exception typing.

- An *informative analysis* to tell a user, which functions could generate exceptions, and the types of the exceptions should be expected.

4.5.2 Standard Output Stub

Since *STL* is not yet supported by *Projectional C++*. There is no way to use the C++ standard output, which is very common for various C++ programs.

A special statement has been developed to provide a primitive support for the standard output stream `std::cout`. The **cout << x** statement is capable of having every *JetBrains MPS concept* instead of **x**. This reflects the high flexibility of the `std::cout` shift operator overloads.

This facility is designed to be replaced later by a proper calls to *STL* version for *Projectional C++*.

4.6 Advanced IDE Functionality

As a projectional editor works directly with an *AST*, it is possible to provide some programming on an *AST* to improve the user experience. We call this additional programming as an advanced editor functionality, and discuss it in this section.

4.6.1 Primitive Renaming Refactoring

Directly from the nature of the projectional editor, without any additional effort, comes a feature to perform primitive renaming refactorings[6].

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

The renaming refactoring comes out of the box, without any additional efforts, thanks to the nature of the projectional editing.

As an example - renaming a class or a method would mean just changing its name where it is declared first. No search for usages and multiple replacements are going to be involved.

4.6.2 Getter and Setter Generation

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

---

[6]More complex renaming, like renaming a function *together with* all its overrides, are not covered by the default behavior of a projectional editor

The job of declaring and prototyping the two functions can be automated with a *JetBrains MPS intention*, see the Figure 4.14.
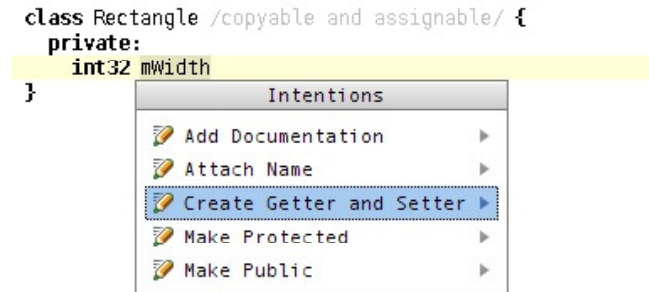
Figure 4.14: Calling the Generation of Getter and Setter

The result of the *intention* work is, as expected, two methods declared and prototyped, see the Figure 4.15.

Figure 4.15: Getter and Setter Generated

The way the editor names the getter and the setter can be controlled through the **NamingConventions** *concept*, which is discussed in the Section 4.6.3.

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 is designed to return a `bool` value by default. This is made to remind a programmer, to include a check of the value and return `false`, if the check locates a wrong value passed. Secondly, the parameter of the setter is typed appropriately. As C++ by itself imply a 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, see the Figure 4.16.

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

Figure 4.16: Setter Works with a Constant Reference for Classes

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

4.6.3 Naming Conventions

In C++ development projects some code writing conventions are usually agreed upon. For example, in Google coding style guide for C++, [23], it is stated that all member fields must end up with "_" sign. For the consistency and uniformity purposes each project has to have some agreements on a way a code is composed. They can include code formatting rules and naming rules.

To some extent the formatting conventions are fixed by the way the projectional editor shows an *AST*, see, for example, the Section 4.2.1. The naming rules, however, should be additionally controlled. The naming rules, accepted in a project, we call naming conventions here.

In order to perform the naming conventions control a new *concept* has been introduced, called **CppNamingConventions**, see the Figure 4.17. It is instantiated once per *JetBrains MPS model*.

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

Figure 4.17: **CppNamingConventions** *Concept* Instance

In a **CppNamingConventions** instance one can give a standard prefixes for getters and setters, which are used during the code generation, see the Section 4.6.2. The argument prefix is used in the setter generation, to name the setter argument.

The member prefix is used on 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 *intention* is available, see the Section 3.1.9, which automatically renames the field, see the Figure 4.18.

```
class MySQLConnection /copyable and assignable/ {
  public:
    MySQLConnection() (constructor)
  private:
    void* handler
}
```

Intentions

- 📝 Add Documentation ▶
- 📝 Attach Name ▶
- 📝 Create Getter and Setter ▶
- 📝 Make Protected ▶
- 📝 Make Public ▶
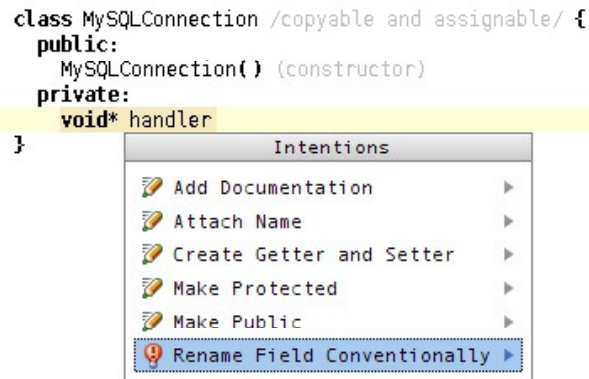- 🔴 Rename Field Conventionally ▶

Figure 4.18: Intention to Rename a Field - Naming Conventions

The **CppNamingConventions** *concept* was created to demonstrate some new possibilities, which can be added to *Projectional C++*. In this case naming conventions can be seen as a part of project configuration, in a very common sense. This is generalized in the Section 5.2.

Some remarks on the **CppNamingConventions** *concept* are given additionally in the Section 5.1.1.

4.6.4 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 a module in the language itself (more on this in the Section 4.1.2) usually different coding standards have to improve on it, requiring, for example in the Qt Project [27], one class has to be defined in two files, one header file with declaration, and one implementation file. The both files have to be named with the class name.

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

After introducing the new module *concept* as in the Section 4.1.2, it is possible to check each method and find out if they all are implemented on the very early coding stage.

A check is introduced in *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.

4.6.5 Abstract Class Construction Check

In this work we have already defined the notion of *abstract* classes in C++ and some related to them questions were discussed in the Sections 4.2.4 and in 4.2.3. Mainly the *abstract class syntax absence* and the problem of determining *abstract* classes were discussed.

```
abstract class MyObject /copyable and assignable/ {
  public:
    pure virtual string getName()  = 0
}
                         Error: Can not be an abstract class
void greet(MyObject objec) {

} greet (function)
```
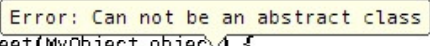
Figure 4.19: Abstract Class Construction Check Signaling an Error

An opened question which stayed, however, is providing some help to a programmer using such classes. Namely, any instantiation of an *abstract* class has to be forbidden. The *abstract* class 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, marking the corresponding nodes as error nodes. *JetBrains 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, see the Section 3.1.6.

Among the places, where a class can be instantiated are variable declarations field declarations, method and function arguments. The return type of the function can imply an *abstract* class instantiation too. 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, see the Figure 4.19.

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 Section 5.4.

4.6.6 Array Deallocation Check

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

the second deallocates the 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 it as undefined, [16]. A simple check is performed, and array deallocation is marked, when it is performed wrongly.

Of-course, a complete check for this can only be performed by a complex analysis of the data-flow. This is left for the future work.

4.6.7 Analyses to Implement as a Next Step

The advanced *IDE* functionality can be grown incrementally. Among the necessary checks to increase *Projectional C++* quality one can find:

- Detection of virtual classes.

- Calculation of class-sizes.

- A check for virtual destructor in a virtual class.

- A check for a default constructor when an array of objects is created.

- A check that an exception thrown gets caught eventually.

And indeed many more. Such checks are left for the future work. When growing the number of checks and/or computational complexity of individual checks, one should mind the remarks on analyses and complexity, made in the Section 5.4.

5 Lessons Learned

In this chapter we present the lessons, learned while building *Projectional C++*.

5.1 Comparison with Textual Approach

It is important to compare the *projectional approach* to build editors for languages to the well-adopted textual approach. To the opinion of author the *projectional approach* has a number of advantages, which could make it more popular in the future, as well as, naturally, some disadvantages.

Below we present the advantageous aspects of the projectional editing. We group them by the party, which benefits from the advantage.

Advantages for an end-user, or a programmer, who uses a projectional *IDE*:

- In a projectional editor the programmer can potentially type less, as the editor is aware *completely* about all the possible constructions, dislike various code-completing helpers in the textual editors, which can refuse to work at all in some cases[1]. Thus the programmer may rely on the auto-completion much more, which can make the typing itself faster.

- As an *AST* is readily available in a projectional editor, there is no need to pre-parse the code in order to perform an analysis on it to detect mistakes or for another purpose. In a textual editor before any analysis the program has to be parsed first. Parsing is a computationally expensive operation. Repeating it after each time a piece of code is modified puts a high load on the developer machine and can slow it down significantly.

- Different naming conventions and coding standards can be replaced by decorations and the way a projectional editor projects an *AST*. This can increase the code readability, and make some of coding guidelines not necessary anymore, reducing the amount of work, a programmer has to accomplish. For example, a member variable, can be shown differently from a local variable in an method.

- Nowadays, in *JetBrains MPS* the approach of generating a text code is taken, before a projected *AST* could be further processed. For example, *mbeddr* is designed to at first generate a textual C code, before it can be compiled into an object code. However, taking into account the fact, that an *AST* in a projectional editor represents the same, or even more rich, information as an *AST* resulting after parsing a text code, in the future it is possible, to process

---

[1] when a program is in unparsable state, or when there is no source code for a library, sometimes, when a code base is too big to be indexed, the second problem is well known to C++ programmers

the projected *AST* itself, without generating a text code at all. For example, produce an object code from it directly. Thus, in the future, the compilation/processing of a code from a projectional editor can be made much faster and effective when compared to the textual approach, where parsing takes place.

Advantages for an *IDE* developer:

- In order to support refactorings an ultimate "understanding" of a code is required by the environment. It is not always possible, however. Especially this problem is well-known with complicated languages like the C++ programming language. The parsing problem, and various checks related to it distract the *IDE* creator, from focusing on the refactoring implementation itself. As an *AST* is available in a projectional *IDE*, the implementation of refactorings represents a more straightforward task, disjoint from parsing.

- Projectional editors have a flexibility to adopt new language constructions. As a projectional editor allows for language modularity (see the Section 2.2.2), it is possible to extend a language, without modifying an editor significantly, installing extensions for it or similar.

As problematic or disadvantageous the following aspects of projectional editors can be concerned, they usually affect both *IDE* user and developer:

- A special problem with the projectional approach is moving a segment of code. As all nodes of an *AST* get referenced by the use of their internal identifiers, and their names do not participate in it, after a piece of code has been moved to another location on the *AST*, a special processes of binding the nodes has to be executed. For example, if there was a variable x in the code defined before the moved snippet, and in the new location for the snippet another variable with the same name x is defined, the new x will not get referenced by the moved code snippet, as it references the old one (not available anymore) by its internal identifier. In *JetBrains MPS* the manual process of rebinding is always required, i.e. the programmer will have to go over all nodes, which reference the old context, and input them again, to connect to the new context[2]. In textual editors, there is no such problem, but another one exists: the moved code snippet can change its semantics in a wrong way after being associated with a new context. Fitting, however, immediately, it may cause a false confidence in correctness of the operation performed. For example, a reference to a variable can start to reference another, locally defined with a narrower scope.

- A very specific issue relevant to projectional editors is the format to store the projectional code and a version control for it. In *JetBrains MPS* an *AST* gets serialized as XML, and is stored in a file. The XML resulted, is, generally speaking, not a human-readable code, despite the XML nature. The line-by

---

[2]Sometimes a special function to re-analyze and rebind nodes can be used in *JetBrains MPS*, however it does not work flawlessly and represents in any case an additional concern and work

line merging as employed in regular text-oriented version control systems (CVS, SVN, Git, Mercurial) does not apply, breaking the XML, or asking the user to merge, presenting with unreadable XML code. The approach taken in *JetBrains MPS* is providing a special merge driver, which handles merging for the projectional code in a proper way. The driver is not perfect however, still requiring the user to finish the job manually sometimes, especially when it comes to non-trivial conflict resolutions.

- New languages are developed themselves using a special defining projectional languages in *JetBrains MPS*. So the evolution of a language under development, and its version control is also an issue. Each new incremental iteration of a language in *JetBrains MPS* gets internally an increment in the language version number. When a second language is referencing the modified one, the version number is taken into account. The update for the first language then requires an update to the second, using the first, language, as well as to all other languages, which use the updated one. If two updates happened at the same time for a language, then two *different* versions with the same version number appear, which presents a problem for the referencing the changed language, as two different language versions get the same version signature. All these version control issues are still up to be thoroughly though of, and are not yet implemented well enough in *JetBrains MPS*.

- The question of language evolution and code in it is to be researched on. When a language defined in a projectional editor is modified, the old code may turn to be not matching the language description anymore. For example, nodes may stop satisfying constraints, or have a child/reference with a role, deleted in the new language version, or, vice versa, have a nothing in a place where the new version language requires some child, reference or property. This renders the code in a former language version incorrect in a newer one. The question of the code update to a new language version appears. For example, a script can be provided with each new language version, which updates the code to the version in question. This is not automated however in *JetBrains MPS*, so that the update process is seamless for the language user. This is another problem to work on in the future. The problem described, of-course, is not relevant to the text code editors, as it is, since the language evolution there is rather untypical.

5.1.1 Some Additional Remarks on Naming Conventions

Few remarks concerning the **CppNamingConventions** *concept* are to be made here:

- The marking of special variables or object, keywords or type names, can be performed by 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.

- 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 an increase of a computational load in the projectional editor, this is discussed also in the Section 5.4.

- The **CppNamingConventions** in *Projectional C++* is just a *concept* in a *Jet-Brains MPS model*. It is similar to the **CppImplementationModule** or the **BuildConfiguration** *concepts*. Dislike many conventional editors, which store such settings as a workspace, a project or an *IDE* configuration, *Projectional C++* makes it a part of a program code itself. It is advantageous, since all programmers in a team have to follow the same naming conventions set up *once* in a project on the start phase, and no environment (re-)configuration is needed to work on the project by any of joining developers. The principle to store a configuration data together with a code is generalized in the Section 5.2. The projectional approach is discussed and compared to the conventional one in the Section 5.1.

- The **CppNamingConventions** *concept* can, of-course, be extended to incorporate class names, function names, additional naming for methods, like "is"- prefix for the boolean return type.

5.2 Rebuilding a Language in Projection

In this chapter we formulate some of the general principles, which can be taken into account when designing new languages in projectional editors, which are meant to represent, especially, the existing already text languages.

5.2.1 Target Semantics

When implementing a language for a projectional editor, one should target semantics of it, rather than an existing syntax. For example, extensions can be provided, which raise the abstraction level to be closer to the application of the language. The code in the target text language is generated then from the higher level constructions.

Another place to think of when targeting semantics is, where the target language constructions are low-level and full of compiler-helping syntax. These constructions can be cleaned out, helping the programmer to focus on their semantics, instead of typing and syntax. C++ brings various examples of it: *pure virtual* method declaration, pointer-to-method type usage, and similar.

5.2.2 Store More Information

A language in a projectional editor can, and often should, contain *more* information as it is needed to just generate a text in a target text language. This information may be used to improve the generation results, or analysis. Example is the overridden method link in the override in *Projectional C++*, see the Section 4.2.4.

A disadvantage of this way can be a problem to extend the information, taken from an importer of the native text language, see the Section 2.2.3.

5.2.3 Configuration as a Part of Source Code

Usually a code project consists not only of a source code, but of some configuration for it, like naming conventions, generator configuration, build configuration, another specific to the project information.

It presents advantageous to store this configuration together with code as a part of it. This eliminates the need to separately configure an environment of each developer before the development process may start.

Usually editor preferences are not shared among users, like in Eclipse for example, stored in a individual for each machine workspace part. This brings a need to configure each development environment separately, and maintain the similarity of configuration. Individually, separately from code, only the pure developer or machine specific configuration parts should be stored.

5.2.4 Hide Redundant Syntax

Usually, text languages contain a lot of syntax, which helps parsers to process a code. In C-like languages these are semicolons, curly braces, braces and triangular brackets.

This syntax has nothing to do usually with the code semantics, so it could be not projected at all, without changing the meaning of the code. Decreasing the amount of projected symbols, the code can be made more readable.

Formatting[3] can also be considered the redundant syntax, and addressed by projection.

5.2.5 Make Old Syntax Readable

Whenever a syntax of a target language happened to be not well readable by itself, a projectional editor can change it. The amount of punctuation can be lowered and some syntactical constructions named in a human language.

As an example we are bringing here a *pure virtual* method declaration, see the Section 4.2.4.

5.2.6 Show the Core, Hint on Details

Not necessarily all information represents a core meaning of a language construction. The most important information can be shown first, and the rest can be shown as a hint, especially when it can be figured out by the projectional editor automatically.

As an example we can consider a friend function declaration in *Projectional C++*, see the Section 4.2.2, where the function full signature is hidden from the main

---

[3]Formatting is important for some languages, like Python, and less relevant to C, being just a not necessary syntax complication

construction, and is just hinted upon, for simplicity, since usually it is clear, which function is declared to be a friend function.

5.2.7 Perform Analyses and Inform a User

Performing various analyses on an *AST* in a projectional editor, it is possible to improve an overall editing experience.

We suggest a logical division of analyses on two types: *informative analysis* and *preventive analysis*.

An *informative analysis* can be performed to find out more about the edited code. Various newly found properties can be shown as hints for a user. *Abstract* classes determination is an example of an *informative analysis*, see the Section 4.2.4. Another example is a class copying and assignment analysis, see the Section 4.2.1.

A *preventive analysis* can be performed to prevent programming mistakes. In general, this topic can go deep, and it is considered separately in, e.g. [7]. In this work as an example of such an analysis for C++ the detection of abstract class instantiation attempts can serve, see the Section 4.6.5.

When implementing an analysis, one should mind the complexity of it, and take a decision, whether to implement it as a *self-running analysis*, or on as an *analysis-on-demand*, see the Section 5.4.

5.3 Projectional Language Extensibility

In general extensibility and modularity are very important concepts for projectional editing, see the Section 2.2.2.

In *JetBrains MPS* extensibility consists of creating a new language and there some new *concepts*, which represent an extension. When creating *concepts* in *JetBrains MPS* one deals with several views on them, see the Section 3.1. In this section we consider extensibility, and the most important, how well it is supported by *JetBrains MPS*, per view on a language.

5.3.1 Structure Extensibility

This view is the same as the view on *concept* declarations. Extensibility is implemented here by inheriting from a base *concept*. It is rather straightforward, since it is very similar to the object-oriented programming languages concept of inheritance, with one difference, actually originating from the Java notion of an interface, as there is one base *concept*, but several *interface concepts* which could be implemented.

5.3.2 Editor Extensibility

It is hard to extend a language in terms of changing an editor concept (see the Section 3.1.2) for an existing *concept* in *JetBrains MPS*. Namely, there are no direct ways to perform inheritance there. So, if the way a *concept* is edited or displayed has to be changed, one should take work around ways to achieve the needed behavior.

One way is inheritance from the *concept*, and changing the editor concept for the descendant. Another way, is mixing some interface calls into the editor, and overriding the methods in the inheriting concept. This way limits the editor to the way it was originally programmed. These both workarounds, however, do not change the way instances of the old *concept* are presented.

In the new *JetBrains MPS* 3.0 an ability to have several editors for one *concept* is present. It is up to the future research to learn, how and if it will change the editor extensibility.

5.3.3 Constraints Extensibility

Constraints, similar to the editor concepts, do not extend well in *JetBrains MPS*. Namely, there is no way to newly define constraints for an existing *concept*. When creating a new *concept*, there is no way to reuse constraints from an existing one, as they are not accessible programmatically.

As an example, when extending the *mbeddr* C language to *Projectional C++*, the naming of identifiers had to be changed, so that C++ keywords are accounted too. The naming rules for identifiers are defined in **IIdentifierNamedConcept** in *mbeddr*. So in its constraints concept a limiting behavior is programmed. There is, however, no way to reuse it explicitly. Another example would be the reference constraints concept, where a variable declaration would be forbidden to use an *abstract* class as a type. It is not possible to redefine those constraints, just adding a language, and not editing the existing one directly.

As a workaround two methods are proposed. First, the constraining behavior can be taken out into a separate code fragment in a Java-like class, the abstract factory pattern, [8], could be implemented and a polymorphic behavior achieved in this way. Second, it is possible to create a check, additionally constraining and marking as errors the places, where the new constraints have been violated.

The first workaround is work intensive to implement originally in the base language. The second workaround does not allow for extending the constraints, making them weaker, and lets the not allowed nodes stay in the code completion lists. Additionally, it is noteworthy to mention, that it is possible to redefine a constraint for a certain role, when inheriting from a *concept*. In the future, a research could be made, on how *JetBrains MPS* could be improved to allow for easier constraints extensibility.

5.3.4 Behavior Extensibility

Behavior concepts are very similar to Java classes. It is easy to extend behavior of a *concept* in *JetBrains MPS* as all the object-oriented methods for polymorphism can be involved in the very same way as for Java classes.

5.3.5 TextGens Extensibility

TextGens are very well extensible. When generating a text for a *concept*, the children or the references are output with the textgen **append** command, without specifying, how exactly it should be done for each particular child or referenced *concept*.

Polymorphically the matching *TextGens* are invoked, and this extends the *TextGen* of the previously defined *concept*.

For example, when generating a text for a function, each statement of the body is generated to a text. For each statement the *TextGen* of the statement itself is executed. Thus, if we add a new kind of statements, all we need is to define a *TextGen* for it, and it will be polymorphically invoked from the *TextGen* for the function. The text for the function will be generated correctly, without the need to change the function *TextGen* implementation anyhow.

5.3.6 Generators Extensibility

Generators perform language transformations in *JetBrains MPS*, see the Section 3.1.8. As this work does not directly implement language transformations, it is left for the future research, to investigate, how extensible the generators are.

5.3.7 Intentions Extensibility

There may be a need to modify an *intention* when extending a language. There is no direct way to do it in *JetBrains MPS*. For example, there is no explicit way to forbid an *intention* to work on a newly defined *concept*, no matter it is based on some *concept* for which the *intention* is defined.

As a workaround an injection of an external polymorphic code from a Java-like class can be taken, similarly to the way it is described above for constraints, with the same practical outcome of a high programming intensity when creating the base language to be extended.

Alternatively, an *intention* can delegate its function to the *concept* under question, so that the whole manipulation is made by the target *concept* itself. This will lead, however in a necessity to define a general behavior in the base concept, which can be behaving wrong in one of the particular cases, or to the need to implement the polymorphic methods called in the *intention* in *each* of the descendant *concepts*. This problem is solved in general object-oriented programming, being well-known.

5.3.8 Type System Extensibility

Type System in *JetBrains MPS* is, generally speaking, another way of constraining. And exactly as in the situation with constraints there is no direct way to change the type system behavior, when extending an existing language with a new one. For example, when a new kind of expressions is added, which has to be typed differently, when used together with some existing expression type, and the existing expression type system is determining the type, it is not possible to redefine this behavior or extend it naturally with some provided by *JetBrains MPS* tools.

Workarounds consist of injecting a polymorphic behavior in the type system calls, or using *JetBrains MPS* own workaround, like the replacement rule, as for example in the Section 4.2.3. Using the replacement rule is acceptable, when it is used once. Later, if used multiple times, it can make the typing system behavior unobservable, as many fragments of code, spread in numerous languages, will decide in undefined order a type for a some expressions.

Hopefully, in future versions of *JetBrains MPS* this is going to be taken care of, and some ways to extend naturally, by the means of the typing system itself will be present.

5.3.9 Analyses Extensibility

Analyses[4] are also subjects to challenge the language extensibility. If an analysis is relying on some concrete *concepts* to work, and is not assuming that a new language could extend those, a problem happens. For example, *mbeddr* data flow analysis, can analyze a C++ code snippet. Being unaware of C++ statements and expressions, it fails, or presents to a user false-positive warnings and errors.

Either an analysis should delegate some features to *concept*, or employ some external polymorphic calls as possible workarounds. In the first case the analysis stops to be a separate programming from the language itself, as delegated analysis methods are found now in the *concepts* themselves. In the second case the development can get more complex initially, as the code should be written in the very beginning polymorphic and abstract enough to allow a non-invasive modification in the future.

Again, some specific support for analyses from *JetBrains MPS* itself could improve the situation, by providing some special means to develop extensible analyses.

5.3.10 Extensibility Overview

Here we summarize in a table the degree of extensibility support in *JetBrains MPS*, provided for languages, per view on a language, and a quality of workarounds, which are known till now.

View	Extensibility Support	Workarounds Quality
Structure	High	-
Editor	No	Poor
Constraints	Low	Good
Behavior	High	-
TextGen	High	-
Generators	-	-
Intentions	No	Medium
Type System	Low	Medium
Analyses	No	Medium

As an outcome it appears, that *JetBrains MPS* does not provide a high degree of extensibility, which would allow a pure modular language engineering. The views, which are not extensibility aware could be improved, and the task of improvement can be not trivial[5].

---

[4]Including but not limiting to non-type-system rules. For example, a complex *analysis-on-demand* can be implemented in a *JetBrains MPS* plug-in.

[5]The authors imply this statement from the absence of good workarounds currently.

5.4 Analyses and Complexity

As it is stated before (see the Section 5.2.7) when constructing a language with an editor for it, it makes sense to provide the language with some analyses. Two types of analyses by application are considered: an *informative analysis* and a *preventive analysis*. Two types of analyses by the way they are initiated are defined: a *self-running analysis* and an *analysis-on-demand*.

A work flow of an analysis can be split into three major steps: initiation, running and reporting the result to a user. On each of the steps the environment (*JetBrains MPS* in this case) can provide certain support for the analysis, making the development of it easier. Below we discuss it step-by-step.

5.4.1 Initiating an Analysis

It can be beneficial for the overall *IDE* performance to implement all computationally heavy analyses as *analyses-on-demand*. Thus a user invokes them when needed, and the question of initiation is solved.

For the *self-running analyses* the question of automatic initiation is to be decided upon. In *JetBrains MPS* there is no way to define, when a certain check is to be run. Thus two problems occur: some checks run too often, and decrease unnecessarily the system performance, or other checks do not happen often enough[6], and the user is not informed on time on some important changes in analyses results.

As a solution to this an *API* extension for *JetBrains MPS* can be proposed: for each analysis it should be possible to define an event, on which the analysis has to repeat, as well as which nodes of an *AST* have to be covered.

For example, for the *informative analysis* identifying *abstract* classes (see the Section 4.2.4) the running event can be defined through the declaration of a new method or a new inheritance relationship. The affected nodes of an *AST* can be correspondingly described as the classes, changed by the modification.

Thus the analysis will never run when changes made do not require it, and will always run when it has to, and only on the relevant nodes.

5.4.2 Running an Analysis

After an analysis has been initiated, the running phase comes. Some analyses can be computationally complex, as they can require external tools running, like SMT solvers, for example, [7], or involve themselves complex algorithms.

As an example of a computationally intensive analysis, the *preventive analysis* for *abstract* classes instantiation can be taken (see the Section 4.6.5). At first, the amount of nodes to analyze is high, as they include local variable declarations, function parameters, function return types, class member variables and more. Each of the nodes is checked like this: at first the type is extracted, then the type is checked to be a **ClassType**. When an instance of the **ClassType** is found, the **Class** is checked on being abstract. For this all of its method declarations are checked on being *pure virtual*, and then all the base classes are checked on being abstract. If

---

[6]There is a way to re-run *self-running analyses* in *JetBrains MPS* manually, by pressing F5, but this is not an automatic solution, and thus is error-prone.

we take a reasonably big code base, like the Qt library, which contains about 1000 classes[7], the amount of nodes to check and the associated computations can be immense[8].

Another example of a computationally intensive analysis could be brought, if **CppNamingConventions** *concept* would be implemented for all named objects, and in a complex way, including regular expressions, as described in the Section 5.1.1.

As analyses can be started automatically, the computational load on an *IDE* can be increased as some analyses may start in parallel.

This all together may present a performance problem, and a certain support from *JetBrains MPS* is needed to handle the task of efficient analysis development:

- As discussed in the previous subsection, it should be possible to control, when *self-running analyses* start, and the scope of their work.

- A value caching is needed, to store the information figured out by the analyses. For example, for each class, it could be saved, if it was found to be an *abstract* class, and the cached value should be valid, until the event to re-run the analysis is detected. It may be necessary, to retain the values after the *IDE* stops.

- As an analysis may take some time to execute, there has to be a way, to inform the user about a background process running, so that a user interface is more clear about analyses and their run time. Currently, *JetBrains MPS* does not inform the user about checks, executed in the background.

- There has to be a way to limit and prioritize analyses running at the same time. For example, *preventive analyses* could be preferred over *informative analyses*, and the total amount of analyses running in parallel could be limited to some number.

The task of providing a better support for analyses in *JetBrains MPS* represents a challenge which lies out of scope for this work. However, this work demonstrates by example the need in such specific support in *JetBrains MPS*.

5.4.3 Reporting Results of an Analysis

Reporting a result after an analysis has completed is very important, since it has to convey the information to the user in a right way, so that the user can take all corresponding actions when needed.

For the time being three forms of reporting have been used in *JetBrains MPS*:

- An error or a warning markup by non-type-system checks, see the Section 3.1.6.

---

[7]There are 989 classes in Qt 5.1

[8]If we assume, that each class out of 1000 classes in a library has 1 parent, 3 member fields, 3 methods, each with 1 parameter, 1 return type, and 1 local variable, we get up to 100 000 nodes to be addressed by a single check execution.

- An *informative analysis* hinting in the editor, as for *abstract* classes for example, see the Section 4.2.4.

- A plug-in created with a custom user interface for an *analysis-on-demand*, as described in [7].

A problem with non-type-system checks is that an error could appear in a root node A after a modification of a node B. For example, if B is a class, changed to be *abstract*, and A is an implementation module, where the class B was instantiated. A must turn to be invalid, but the error is not going to be seen to the programmer, as he/she is working in another root node, in one, where the B class is defined.

JetBrains MPS needs to provide some support which will allow to modify the way, root nodes look in the project explorer[9], if an error is found in a node. For example, if a modification of one module, causes an error in another one, this capability is explicitly useful and should immediately come into action, to inform the programmer about a problem at the moment, when the problem appears.

Alternatively, some user interface element may be provided to output analysis messages in some form to the user, like a log window with hyper references to error nodes.

An *informative analysis* analysis, as for example copyability and assignability analysis (see the Section 4.2.1), is integrated with the editor for the **Class**[10] *concept*. This makes the editor and the analysis being built in dependence on each other. This, in turn, makes the code more complex. Instead, some methods to decorate nodes could be provided, separately from an editor implementation. This would allow to control the way, how *informative analysis* results are displayed to the user separately.

Finally, when creating plug-in analyses, the user interface for them has to be created in standard Java libraries for a user interface creation. Some patterns could be figured out, and offered as a *JetBrains MPS API* for a language developer, which would allow, for example, a parallel execution of the analysis code or an external tool, presenting the results in some commonly met form (as a table, a diagram or decorations for code), and invalidating the results of the analysis, when needed.

5.5 Templates

The support for templates, as described in the Section 4.4, presents some pro- and contra- arguments to a potential *Projectional C++* user.

Among the disadvantages are:

- Incompleteness of the support for templates, thus it will not allow every construction, possible in textual C++.

- The need to prepare C++ concepts before writing the template code, there questions of code duplication can appear, when declaring two C++ concepts with the same content.

---

[9]See the left window panel on the Figure 3.1.
[10]See the Section 3.1.2 for editor concept or editor view

- Importing of native C++ code is made more complex, as C++ concepts have to be generated, this involves parsing the template code and extracting requirements for the template parameters, and establishing existing classes to be compatible with the extracted C++ concepts.

Advantageous are the following points:

- The requirements on a template parameter are made explicit for a programmer through the use of C++ concepts.

- Checks for C++ concept compliance make the template code programming and usage more predictable.

The challenges present above as disadvantages of the C++ concepts approach could be resolved in the future by improving on the C++ concepts through:

- Making the template support with C++ concepts more complete.

- Developing techniques to auto-generate C++ concepts from a given code, preparing it to be generalized.

- Developing a special importer, which will figure the concept definitions out of the existing textual C++ code.

Additionally, it could be of a practical interest to impose not only syntactical, but semantic assumptions on a template parameter with the use of the C++ concept. The way to describe the semantic constraints, and to check them later on, can represent a challenging task.

6 Conclusion

Here we briefly review the accomplished contributions and a future work possible.

6.1 Overview of the Work Performed

In this work we were extending the *mbeddr* C language with programming constructs from the C++ programming language, which resulted in a software, we call here *Projectional C++*.

A new *IDE* for C++ has been created, which supports projectional editing for a subset of the C++ programming language. The new *IDE* has been designed with several goals in mind.

First, *Projectional C++* should serve as a modular basis for the future extensions to C++. The potentially possible extensions are figured out and proposed as a future work.

Second, the experience of the *Projectional C++* programming has to be more safe and informative, when compared to the regular C++ programming. Various pitfalls, usual for C++ programming, have been explained in this work, and tools have been introduced within the new *IDE*, designed to compensate on the pitfalls.

While improving on the C++ programming experience, an application of analyses has been found to be useful. Various analyses have been applied in *Projectional C++*. The analyses have been classified orthogonally by the purpose to *informative analyses* and *preventive analyses*, and by the running type to *analyses-on-demand* and *self-running analyses*.

A method to support, to some extent, a code project guidelines, precisely, naming conventions, has been proposed in this work. The idea to store the project related information together with the source code has been formulated, its advantages have been listed.

The projectional approach to create an *IDE* has been compared to a traditional textual approach. The advantages of the projectional approach have been listed. The potential problems while developing, using and evolving a projectional *IDE* and a code produced by it, have been identified.

Projectional C++ as a language has been designed, to represent, in the end, a complete enough subset of the C++ programming language for *STL* to be recreated in it in the future. The creation of a projectional *STL* copy was out of scope for this work, however. The completeness of the *Projectional C++* language was described in this work, and the future work was proposed, which is going to be needed in order to support *STL*.

Generalized principles of language re-engineering in a projectional *IDE* have been discovered and formulated. The principles can be reused when creating an

IDE for a language, to make the language more readable, more expressive, higher cross-platform, and safer and more convenient to use for a programmer.

Various language modularity and extensibility problems have been considered on practice while developing the *mbeddr* C++ extension. *Projectional C++* has been developed in such a way, that the *mbeddr project* languages do not have a dependency on *Projectional C++*, which allows for, to a certain degree, separate development of the projects.

In parallel, while extending *mbeddr* to build a new C++ *IDE*, the facilities for extensibility, provided by *JetBrains MPS* were researched. Extensibility has been analyzed separately for each view on a language. Whenever the degree of extensibility was not considered to be high, workarounds and future *JetBrains MPS* improvements were proposed.

The question of building analyses in *JetBrains MPS* for a new language has been discussed. Three phases of analysis running were identified, and for each phase potential problems were highlighted. *JetBrains MPS* improvements have been proposed to make the analysis creation more productive.

Finally the potential future work has been described, including the ways to get the full C++ language support, the need to test the new *IDE* on practice, the problem to create an importer for the textual C++ code and its potential complications have been described.

6.2 Future Work

Here we briefly describe the way, *Projectional C++* could be developed in the future.

6.2.1 Full Language Support

One of the main target for *Projectional C++* in the future has to be developing all of the original C++ features. This is needed both for the convenience of a *Projectional C++* user, and for the ability to import an existing code base, which can use all the constructions, possible in C++.

One of the big challenges for the language completeness is a development of full templates support. A very important part of the C++ programming language, is undoubtedly *STL*. In order to acquire all language capabilities, as it is usual for a C++ programmer, after supporting all features of the language itself, *STL* has to be implemented in *Projectional C++*. Alternatively , if a powerful importer is developed first, the *STL* could be potentially imported.

6.2.2 Investigating the Language Use

In order to continue the *Projectional C++* development a user has to be found for the *Projectional C++*, who will use the language on practice. This will allow to figure out in the fastest way, which language features left are the most desirable on practice, which new features, similar to those described in the Section 4.6, could first be developed.

No matter the C++ programming language is not completely supported by *Projectional C++*, the ability to use the object-oriented programming paradigm while using *mbeddr* represents a qualitative improvement over the existing C language in *mbeddr*. This fact represents an advantage, which can attract any *mbeddr* user towards *Projectional C++* even before the C++ programming language is entirely supported.

6.2.3 Extending Projectional C++

Following one of the goals to create *Projectional C++*, extensions for it could be created in a modular fashion.

As examples of such extensions could be:

- Language constructions emulated by preprocessor and templates as in [20].

- Classes extensions to support messaging as in Objective-C, or signals and slots as in the Qt project.

- Object-oriented design patterns could be researched for the ability to be supported on the language level.

- Higher-level models, which would generate to classes together with semantic analyses for them.

Other *Projectional C++* extensions can be invented, including specific extensions for various domains.

6.2.4 Projectional C++ Importer and Other Tools

An importer for *Projectional C++* could be developed, which would allow to reuse in *Projectional C++* the existing textual code base. One of the special challenges to develop such an importer would be a conversion from regular C++ templates into the *Projectional C++* templates with C++ concepts. A debugger of *mbeddr* could be extended in order to support the *Projectional C++*.

6.2.5 JetBrains MPS Evolution

This work suggests some improvements for *JetBrains MPS* itself, see the Section 5.3 and the Section 5.4 for example.

If *JetBrains MPS* gets updated, taking some of the mentioned potential improvements into consideration, *Projectional C++* could be improved benefiting from the new *JetBrains MPS* features.

Appendix

Glossary

abstract is a class in C++, having at least one method declared pure virtual in ancestors, but never implemented in the ancestors or the class itself, instances of such class can not be created. 32, 41, 46, 47, 58, 66, 67, 70–72

abstract class syntax absence a phenomenon in C++, representing the absence of any syntax to explicitly declare a class abstract. 41, 46, 58

action is a JetBrains MPS term, which corresponds to automations, specific to actions, occurring while editing. 25

analysis-on-demand is an analysis, which a user must invoke explicitly in the *IDE*, it is usually computationally expensive, c.f. *self-running analysis* . 25, 66, 69, 70, 72, 75

API Application Programming Interface. 5, 6, 15, 46, 70, 72

AST Abstract Syntax Tree. 2, 3, 5, 9, 11–14, 18, 19, 21–24, 39, 45, 46, 52, 54, 56, 58, 61, 62, 66, 70, 81, 82

base concept is a JetBrains MPS concept, which serves as an inheritance base or parent concept for a given concept, e.g. Statement concept for IfStatement concept. 18

concept is a class-like type, describing a node type in an *AST* when talking about projectional editing. 2, 3, 12–14, 17–28, 30, 32, 33, 36, 37, 39, 40, 42, 49, 50, 52, 54–57, 63, 64, 66–69, 71, 72, 82

DSL Domain Specific Language. 9, 11, 14, 17, 18, 23

Embedded C++ is a language subset of the C++ programming language, intended to support embedded software development. 1, 4

Extended Embedded C++ is a improvement on Embedded C++, bringing back the omitted language features, and a memory-aware *STL* version. 1

IDE Integrated Development Environment. 2–7, 9, 11, 17, 29, 35, 57, 59, 61, 62, 64, 70, 71, 75, 76, 81, 82

informative analysis is performed to inform a user of an *IDE* about some code properties, to enhance understanding of a code. 54, 66, 70–72, 75

intention is a special procedure in *JetBrains MPS* which can be used for automatic manipulations on an *AST* with a node of a given *concept*. 24, 33, 36, 37, 46, 55, 56, 68

interface concept is a JetBrains MPS concept, which serves for inheriting a concept behavior interface, can not serve as a base concept. 19, 21, 22, 50, 51, 66

JetBrains MPS is a language engineering environment allowing to construct incrementally defined domain specific languages. ix, 2–7, 13, 14, 17–28, 33, 37, 39, 44, 54–56, 58, 61–64, 66–72, 76, 77, 82

mbeddr same as the mbeddr project. xi, 1, 2, 4–7, 14, 18, 22, 25–31, 33, 42, 49–51, 61, 67, 69, 75–77

mbeddr project is a JetBrains MPS based language workbench, representing C language and domain specific extensions for the embedded software development. ix, 1, 5, 6, 14, 17, 18, 20, 25–27, 51, 76

model corresponds in JetBrains MPS to one single JetBrains MPS file unit, in the mbeddr context it corresponds to C or C++ project, like one library or one executable.. 28, 56, 64

override syntax absence a phenomenon in C++, consisting in absence of any syntax to explicitly designate a method, to be an override of another method. 41, 44, 45

preventive analysis is performed to inform a user about potential mistakes in advance, in order to prevent them. 23, 66, 70, 71, 75

projectional approach is an approach to create an editor for a language, or, speaking broadly, an *IDE*, when the editor is aware of an *AST* for a code, and shows the code to a user, projecting an *AST* itself, and allowing to edit the *AST* directly. 6, 7, 9, 11, 12, 61

Projectional C++ is a C++ flavor introduced in this work, together with an *IDE* for it, based on the mbeddr C implementation in the JetBrains MPS environment. 4–7, 28–31, 33, 35–46, 48, 50–54, 57, 59, 61, 63–65, 67, 72, 75–77

pure virtual are virtual methods in a C++ class, for which intentionally no implementation is provided, they serve purely for overloading purposes, describing an interface. 40, 41, 45, 46, 57, 64, 65, 70

self-running analysis is an analysis, which determines itself the point of time, when it is performed, or this moment is determined automatically by an *IDE*, there is no need to explicitly run such analysis. 66, 70, 71, 75, 81

STL stands for Standard Template Library, the standard library of the C++ language. 1, 28, 48, 51, 54, 75, 76, 81

TextGen is an special kind of generator in JetBrains MPS, dedicated to produce a textual representation of a node of a given concept. 23, 24, 44, 67, 68

Bibliography

[1] VDC Research. Survey on embedded programming languages, http://blog.vdcresearch.com/embedded_sw/2010/09/what-languages-do-you-use-to-develop-software.html.

[2] Embedded C++. Official website, http://www.caravan.net/ec2plus/.

[3] Embedded C++. Objectives behind limitting C++, http://www.caravan.net/ec2plus/objectives/ppt/ec2ppt03.html.

[4] Bjarne Stroustrup. Quote on Embedded C++, http://www.stroustrup.com/bs_faq.html#ec++.

[5] IAR Systems. Extended Embedded C++, http://www.testech-elect.com/iar/extended_embedded_c++.htm.

[6] Markus Voelter, Daniel Ratiu, Bernhard Schätz, and Bernd Kolb. mbeddr: an extensible c-based programming language and ide for embedded systems. In *SPLASH*, pages 121–140, 2012.

[7] D. Ratiu, M. Voelter, Z. Molotnikov, and B. Schätz. Implementing modular domain specific languages and analyses. In *Proceedings of the 9th Workshop on Model-Driven Engineering, Verification and Validation*, 2012.

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

[9] Paul Hudak. Modular Domain Specific Languages and Tools. In *ICSR '98*, jun 1998.

[10] Eric Van Wyk. Modular Domain-Specific Language Extensions. In *1st ECOOP Workshop on Domain-Specific Program Development (DSPD)*, 2006.

[11] Markus Voelter. Language and IDE Development, Modularization and Composition with MPS. In *GTTSE 2011*, LNCS. Springer, 2011.

[12] 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.

[13] 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.

[14] Bjarne Stroustrup. *The C++ Programming Language, Special Edition*. Addison-Wesley, 2000.

[15] Herbert Schildt. *C++: A Beginner's Guide, Second Edition*. McGraw-Hill Osborne Media, 2003.

[16] Standard for Programming Language C++ (C++ 11), ISO/IEC.

[17] Stephen Prata. *C++ Primer Plus, Sixth Edition*. Addison-Wesley Professional, 2011.

[18] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005.

[19] David Vandevoorde Nicolai Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 2002.

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

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

[22] Class layout recomendations, Possibility.com. http://www.possibility.com/cpp/cppcodingstandard.html.

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

[24] Open Office C++ Coding Standard, http://wiki.openoffice.org/wiki/cpp_coding_standards.

[25] Qt Project Documentation Page, contains Q_DISABLE_COPY macro. http://qt-project.org/doc/qt-5.0/qtcore/qobject.html.

[26] Boost C++ Libraries. Official Website, http://www.boost.org/.

[27] Qt Project. The Qt Project Coding Guidelines, http://qt-project.org/wiki/category:developing_qt.