Technische Universität
München

Fakultät für
Informatik

Master Thesis

Adding C++ Support to mbeddr

Language Engineering for C++ over the mbeddr Project C implementation

Presents: Zaur Molotnikov
Advisor: Dr. rer. nat. Daniel Ratiu
Supervisor: PD Dr. rer. nat. habil. Bernhard Schätz

Presentation Structure

- Introduction
- Projectional C++
- Evaluation

Presentation Structure

- Introduction
  - JetBrains MPS and mbeddr
  - Language modularity and extensibility
  - Introduction to Projectional C++
- Projectional C++
- Evaluation

JetBrains MPS

- Is a language engineering framework
- To describe a language:
  - Split a language on *concepts*
  - Define *concepts*
  - Provide additional automations
- After language definition:
  - MPS gives a *projectional editor*
  - A user can start using a language

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

mbeddr

- C implementation in MPS
  - Targets embedded domain
  - Contains extensions to C
  - Features analyses

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

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

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

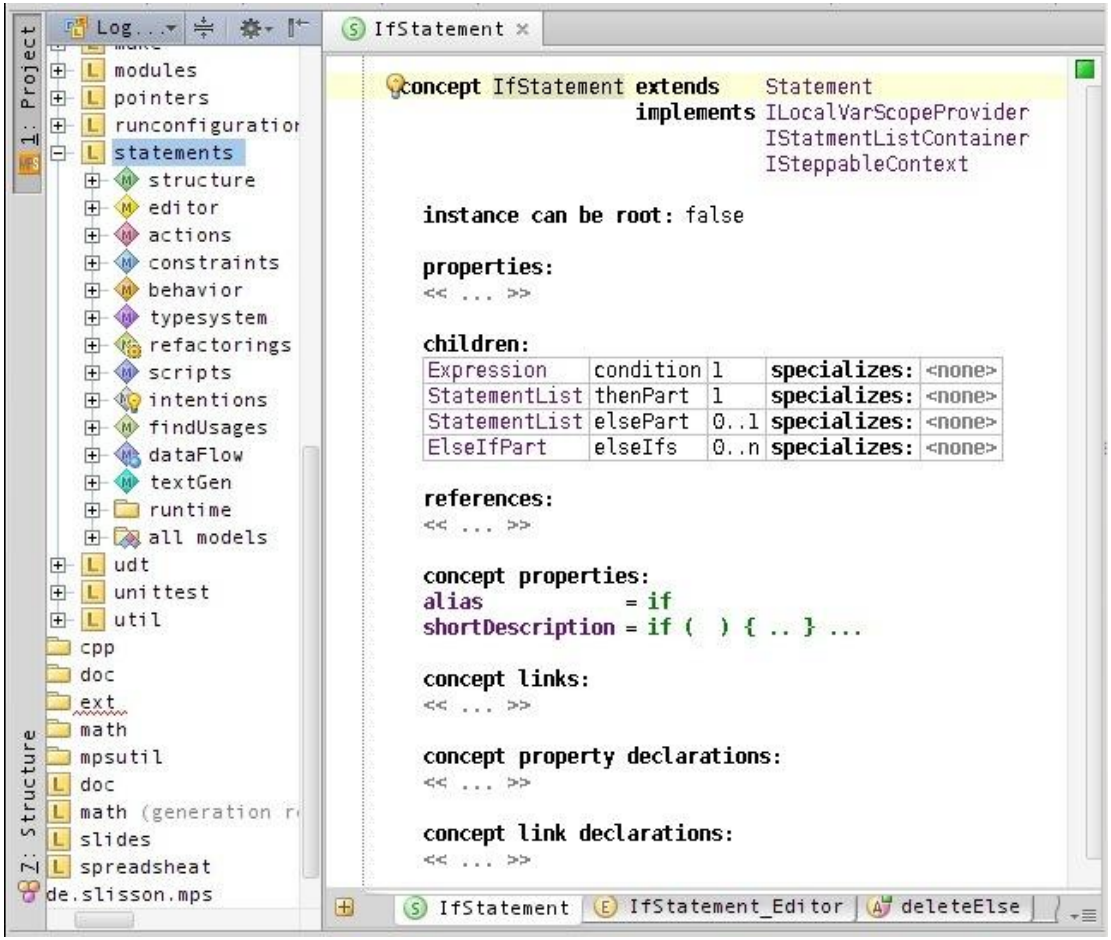
```
} nextMode (function)
```

Taken from [7], Ratiu 2012

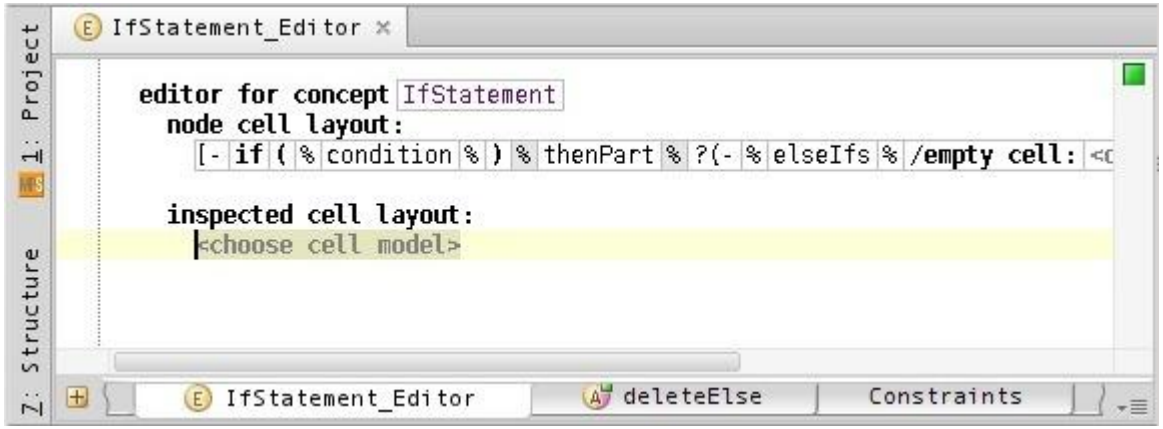
Defining a *Concept*

- A *concept* is defined in views on it
  - **Structure** view - properties and relationships
  - **Editor** view - the way to input and edit an instance
  - **Constraints** view - add context sensitive limitations
  - **Behavior** view - add methods like to a Java class
  - **Type system** view - for typed languages
  - **Non-type-system** checks - for warnings and errors
  - **TextGen** view - to generate to text
  - **Intentions** view - provide user-callable automations
  - Some other views - not related to this work

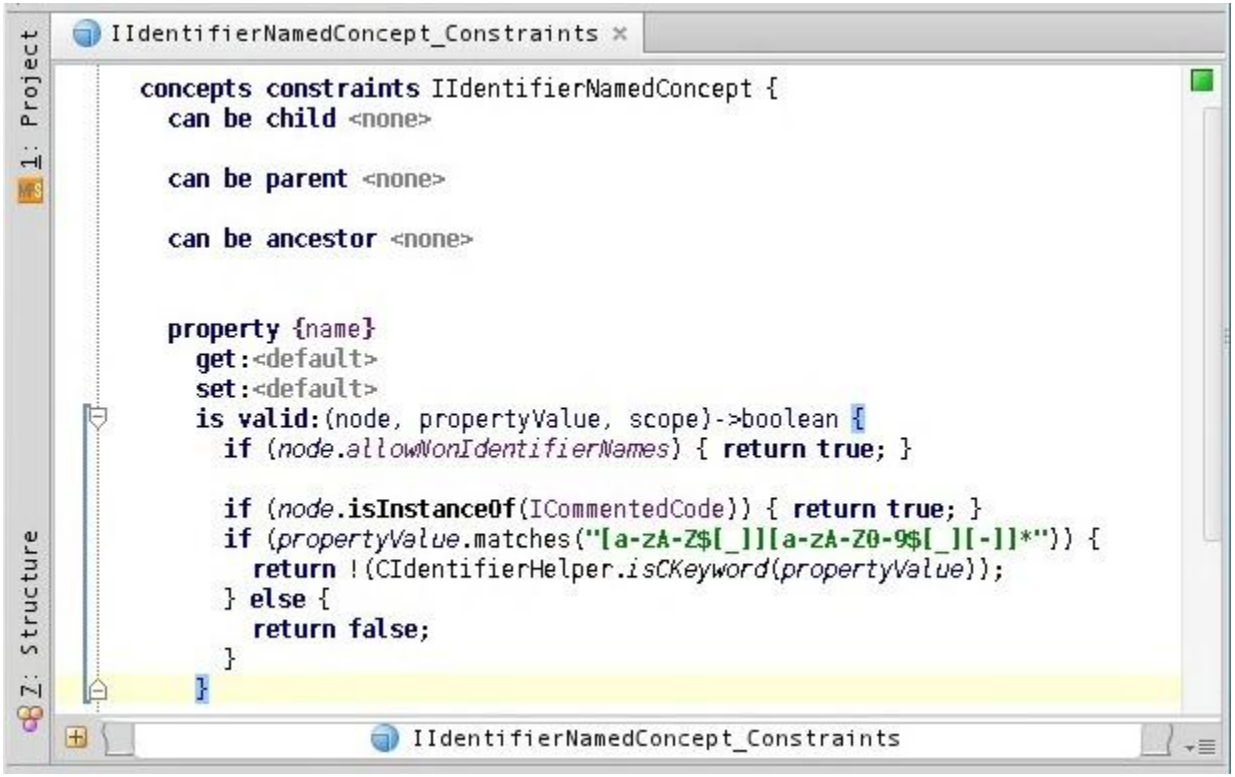
Structure View - MPS



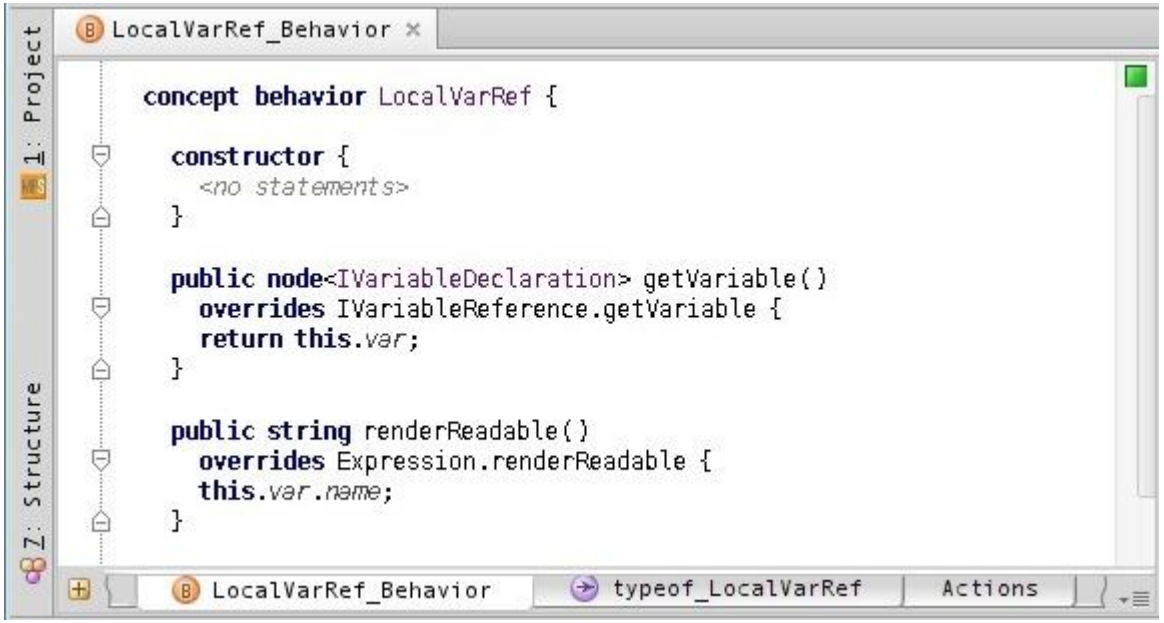
Editor View - MPS



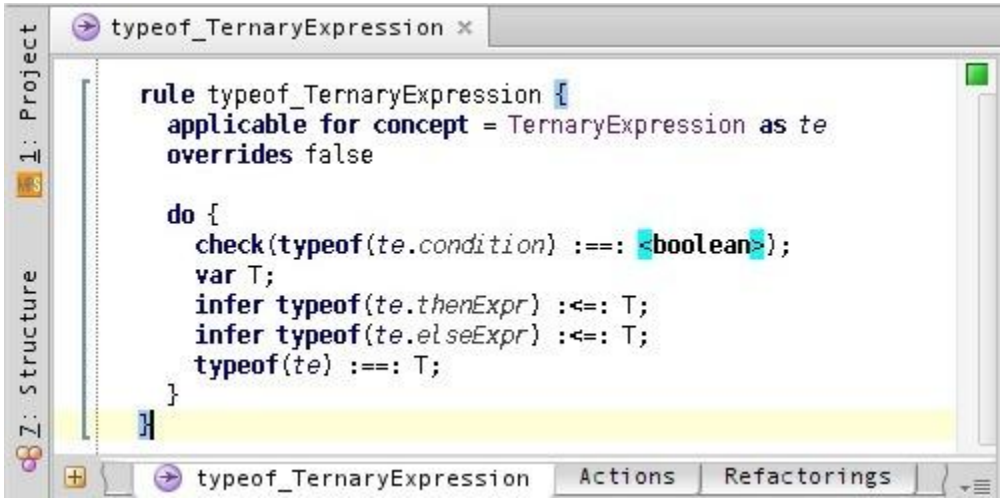
Constraints View - MPS



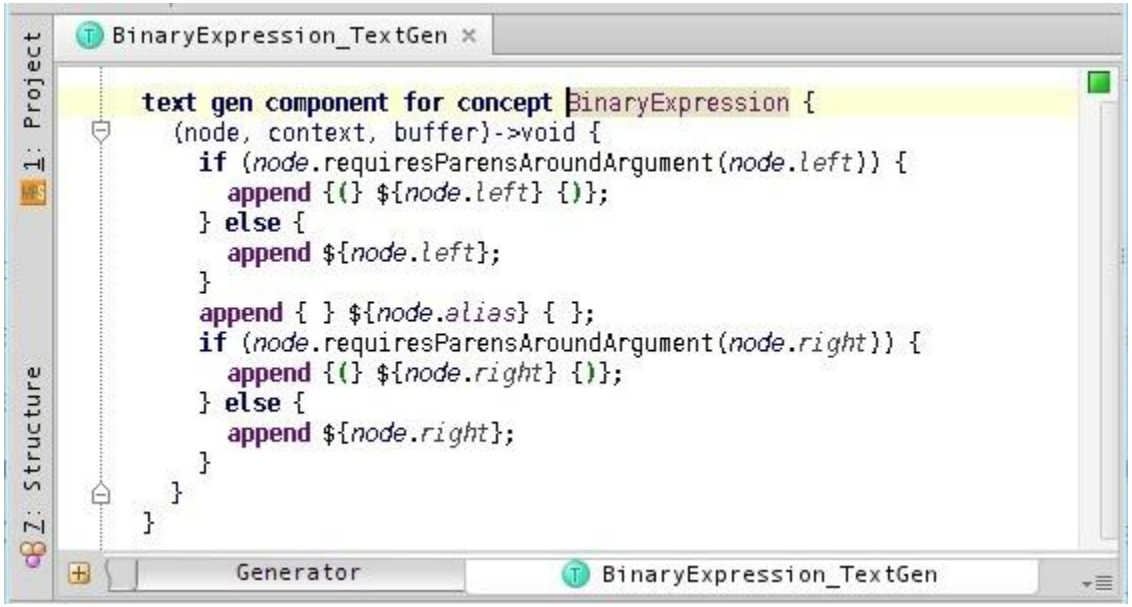
Behavior View - MPS



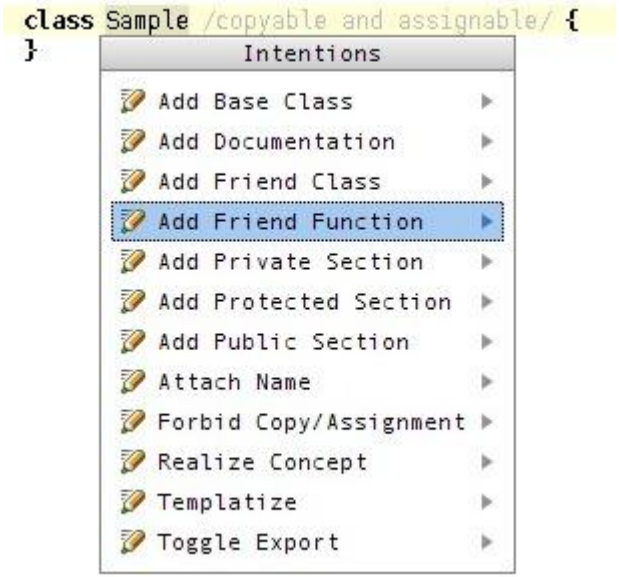
Type System View - MPS



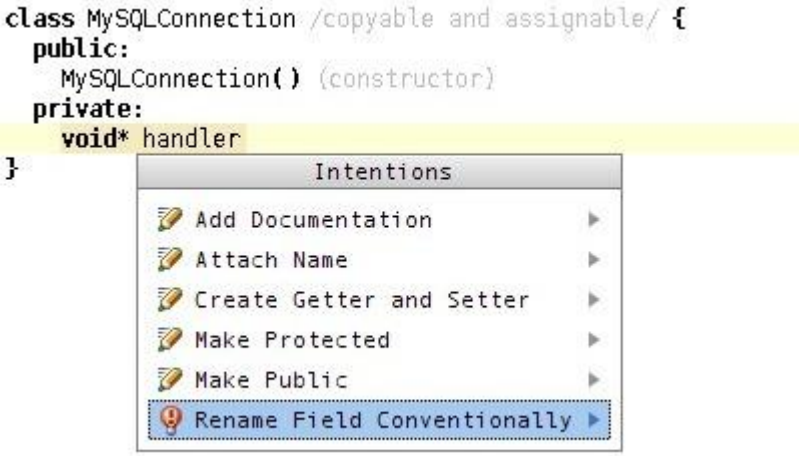
TextGen View - MPS



Intentions Example - MPS



Non-Type-System Checks Example - MPS



Language Modularity

- A language can use/include another one
  - Yellow zones - Expressions Language
  - Green zones - Statements Language
  - Uncolored zones - Modules Language

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

Language Extensibility

- A language can extend another one
  - State Machines language extends Expressions

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

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

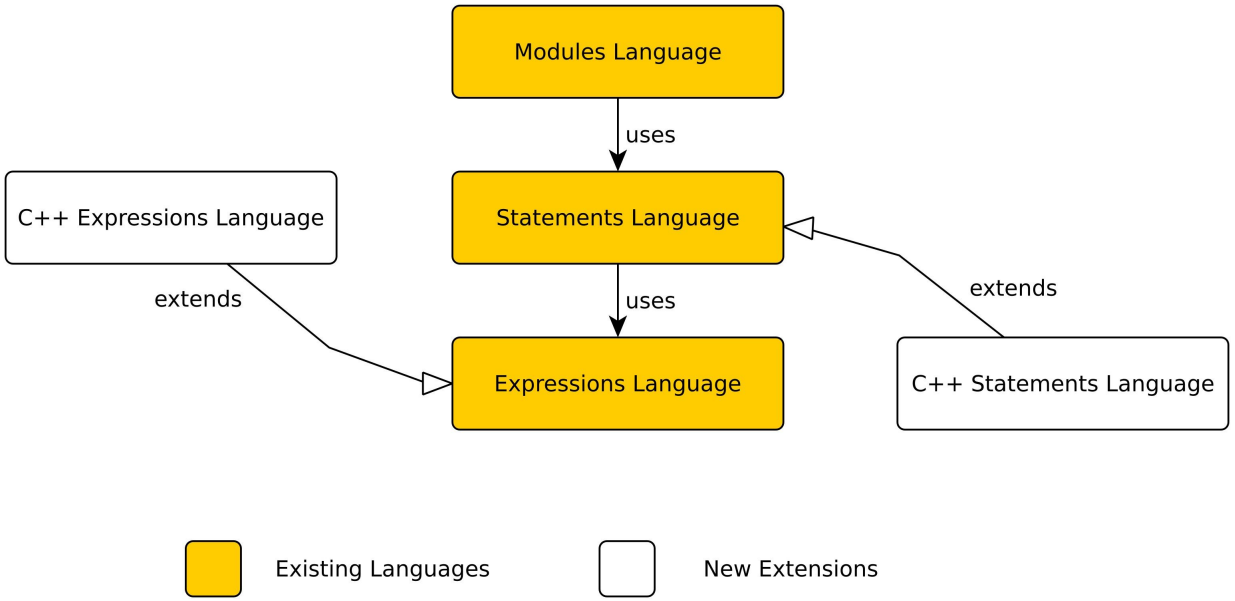
	mode == MANUAL	mode == AUTO
speed < 30	MANUAL	AUTO
speed > 30	MANUAL	MANUAL

```
;
} nextMode (function)
```

Taken from [7], Ratiu 2012

Projectional C++

- mbeddr supports only C
- C++ can be also desired by users
  - Extending mbeddr to support C++



Presentation Structure

- Introduction
  - JetBrains MPS and mbeddr
  - Language modularity and extensibility
  - Introduction to Projectional C++
- Projectional C++
- Evaluation

Presentation Structure

- Introduction
- Projectional C++
  - Points of Interest
  - One-Side-Awareness
  - C and C++
  - Object-Oriented Programming
  - Operator Overloading
  - Templates
  - Advanced Functionality
- Evaluation

Points of Interest

- **P1**: C++ is technically not based on C - does reuse make sense?
- **P2:** mbeddr restricts C to make it “better” - is something similar possible with C++?
- **P3:** How well MPS will support extending mbeddr languages with C++ - largest extension made in MPS?

One-Side-Awareness

- Extend mbeddr so, that only Projectional C++ is aware of mbeddr, and not vice versa
  - to not to make mbeddr code base bigger
  - to make support of mbeddr easier
  - to give an example of “pure” extension

- It is not as “simple” as in usual development in programming languages because
  - it is not just an interface and a usage of it since
  - all views on a language have to be extended and
  - all the new code has to reside in Projectional C++

C and C++

Differences:

- Reference type and boolean type in C++
  - result from extending expressions language
- Modules
  - no modules in C
  - modules in mbeddr
  - namespaces and classes in C++
  - Projectional C++ solution: modules and namespaces
- Memory allocation
  - operators **new** and **delete**
  - extensions for expressions and statement languages

C and C++

- Major differences between C and C++ were listed before.
- Otherwise similarities stay, C being a subset of C++:
  - Expressions
  - Types
  - Statements
  - Functions
  - ...
- **P1:** It makes sense to build C++ on top of C!

Object-Oriented Programming

- Object-Oriented Programming in C++ is supported through classes:
  - Classes declaration and copying
  - Encapsulation and access control
  - Polymorphism:
    - Polymorphic casting
    - Abstract classes and virtual functions

Classes Declaration and Copying

```
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
}
```

- **P2:** Declaring safe, analysing properties, providing intentions

Encapsulation and Access Control

```
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)
}
```

```
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)
```

- **P2:** New friend declaration, various access cases including inheritance, sections structure

Polymorphism

Here - polymorphism through virtual functions and inheritance.

In C++ polymorphism can be also achieved through template programming and operator overloading.

Polymorphic casting

```
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)
```

P2: Tracking typing and casts

Abstract Classes and Virtual Functions

- Have no syntax in C++

```
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()
}
```

Operator Overloading

- **P1:** One-side-aware reuse is achieved by extending mbeddr type system to be more general.

```
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
}
```

```
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;
```

Templates

- Implemented through C++ concepts
- Has a number of advantages and disadvantages

```
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)
}
```

Some Other Language Features

- Exceptions
- Standard output stub

- STL will require all features of the language

Advanced Functionality

Some additional features are present in Projectional C++ editor:

- Primitive renamings - due to projection
- Getter and setter generation
- Naming conventions
- Method implemented check
- Abstract class construction check
- Array deallocation check

More checks can be added (class virtuality, size, exceptions, data flow…)

Presentation Structure

- Introduction
- Projectional C++
  - Points of Interest
  - One-Side-Awareness
  - C and C++
  - Object-Oriented Programming
  - Operator Overloading
  - Templates
  - Advanced Functionality
- Evaluation

Presentation Structure

- Introduction
- Projectional C++
- Evaluation

Presentation Structure

- Introduction
- Projectional C++
- Evaluation

Presentation Structure

- Introduction
- Projectional C++
- Evaluation
  - Rebuilding a Language in Projection
  - MPS Extensibility
  - Analyses and Complexity
  - Future Work

Rebuilding a Language in Projection

Few principles discovered may apply to every language reconstructed:

- Target semantics - pure virtual functions, exts
- Store more information - overrides
- Configuration is a part of source - naming
- Hide redundant syntax - braces, etc.
- Make syntax human readable - pure virtuals
- Show core, hint on details - friend function
- Perform analyses - abstract classes

MPS Extensibility

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

P3: MPS can provide better support for extensibility

Analyses and Complexity

- Analyses were found to be useful, however
  - MPS does not support them explicitly
  - Computational complexity can be high enough
- Propositions for MPS evolution, APIs for analyses:
  - When analysis start?
  - Which scope do they have?
  - Result caching needed?
  - Prioritisation, concurrency limitations?
  - Informing user - can be improved and
  - Common solutions offered for reuse

Future Work

- Complete language support
- Investigating language use
- Importer, templates
- Debugger
- Extensions on top of Projectional C++
- JetBrains MPS Evolution

Templates and Importer

- Contradictory template nature - C++ concepts
- A text importer will have to provide them!
- Other approach can be considered.

```
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)
}
```

Potential Extensions

- Language constructions as emulated by preprocessor and templates (Alexandrescu)
- Signals and Slots (Qt, Objective-C)
- Object Oriented Design Patterns
- Higher level models with semantics, implemented by classes
- More? - Question by itself

MPS Evolution

- Support for extensibility
  - Analyze workarounds, introduce similar ways
  - Analyze poor extensibility, improve on it

- Support for analyses
  - Move common patterns inside MPS
  - Add running control
  - Improve on results indication

Thank you!

Thank you for your attention!

You are welcome to ask questions:

Zaur Molotnikov

zaur@zaurmolotnikov.com