Language Oriented Programming: The Next Programming Paradigm

Paper Presentation by Tommy Carpenter

Original Author: Sergey Dmitriev

February 4, 2012

Outline

What is this paper about?

The Problem

Generalized programming languages (Java/C++) force us to translate our ideas on how to solve problems into restrictive PL constructs and environments. Ideas are lost in this downconversion and this way of programming is not natural or efficient.

What is this paper about?

The Problem

Generalized programming languages (Java/C++) force us to translate our ideas on how to solve problems into restrictive PL constructs and environments. Ideas are lost in this downconversion and this way of programming is not natural or efficient.

The Solution

Language Oriented Programming (LOP) & Meta Programming System (MPS). We should be able to work in terms of the concepts and notions of problems we are trying to solve using DSLs (LOP). If we don't have a DSL that fits our needs, we should be able to **easily** modify one or create a new one (MPS).

What’s wrong with our current programming methodology?

Imagine if...

- programming was closer to explaining the solution to a problem in NL
- domain experts could program directly or better express their ideas to programmers
- it was easy to view a program years later and quickly extend it
- we could eliminate time and Tom ↓

Completion time is too long

- Long time between mentally solving a problem and a working implementation
- We can explain the solution to a complex problem to others in minutes, but it takes d/m/y to write it in a way computers understand. Why?
  - PLs are much less expressive than NL
  - Can't give a computer high level ideas: must implement every detail
  - We waste time converting NL ideas into PL constructs, e.g., OOD

Maintenance is hard

- Looking back at code is difficult!
- Info about “what is going on” is lost in the downconversion from NL to PL
- Looking back, its hard to build the mental model of the solution again
- Extensive code documentation is time consuming and quickly becomes out of sync
- Wouldn’t it be awesome if code was *self documenting*?

DS extensions are weak

- Forex, specific class libraries in Java
- Steep learning curve for domain experts to learn how to use these libraries as opposed to DSLs
- Can lose motivation when drudging through code documentation

Proposed Solution: Language Oriented Programming (LOP) and the Meta Programming System (MPS), a framework for LOP

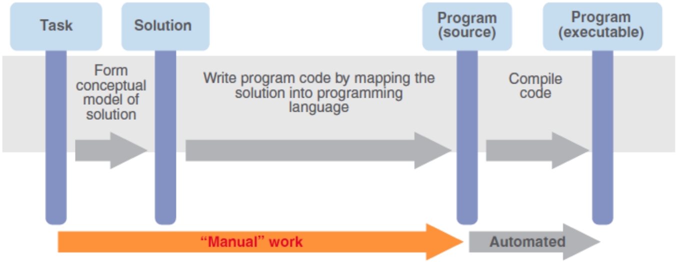

Task
Solution
Program (source)
Program (executable)
Form conceptual model of solution
Write program code by mapping the solution into programming language
Compile code
"Manual" work
Automated

Figure 1: Mainstream programming with a general-purpose language.

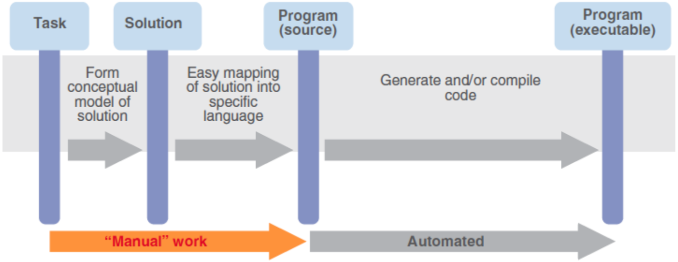

Task
Solution
Program (source)
Program (executable)
Form conceptual model of solution
Easy mapping of solution into specific language
Generate and/or compile code
"Manual" work
Automated

Figure 2: Language-oriented programming with domain-specific languages.

LOP Uses

With LOP, we can:

- think of programs as solutions to problems in their domains
- “lay out” solutions like they are in our heads
- represent different parts of code with their domain specific symbols, e.g., math equations, pictures...
- edit a program’s “graphical structure” directly, instead of just editing text
- easily create or extend a DSL if one isn’t available or sufficient

LOP

An L in LOP has *a structure, an editor, and semantics*:

- structure: “abstract syntax”. Defines what concepts are supported and how they interact
- editor: “concrete syntax”. Defines how we edit the structure (read: program) and how it is displayed
- semantics: defines how the structure is interpreted or compiled

MPS

So how do we create these languages?

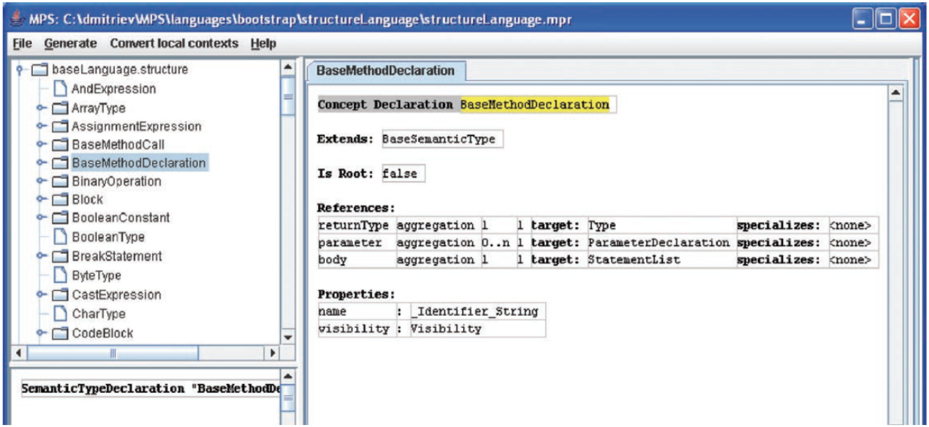
- Need a “bootstrapping” DSL whose domain is the creation of languages for other domains. “Language building languages”
- A program in this bootstrapping domain is a DSL
- MPS has a *structure language*, an *editor language*, and a *transformation language* used to define other DSLs
- Defining these new DSLs is “metaprogramming”: we are programming the DSLs we will program in

MPS Structure Language

We use the MPS Structure Language to:

- define the *types*, like objects or “concepts” we want to represent
- define the properties of the types, and define how *instances* of various types interact with each other
  - child/parent (like inheritance) vs. freeform relationships (define arbitrary relationships between types)

To use the newly created language: define instances of types, populate property values, link instances together



```
SemanticTypeDeclaration "BaseMethodDe
```

```
Concept Declaration BaseMethodDeclaration

Extends: BaseSemanticType

Is Root: false

References:
returnType aggregation 1    1 target: Type                 specializes: <none>
parameter  aggregation 0..n 1 target: ParameterDeclaration specializes: <none>
body       aggregation 1    1 target: StatementList        specializes: <none>

Properties:
name       : _Identifier_String
visibility : Visibility
```

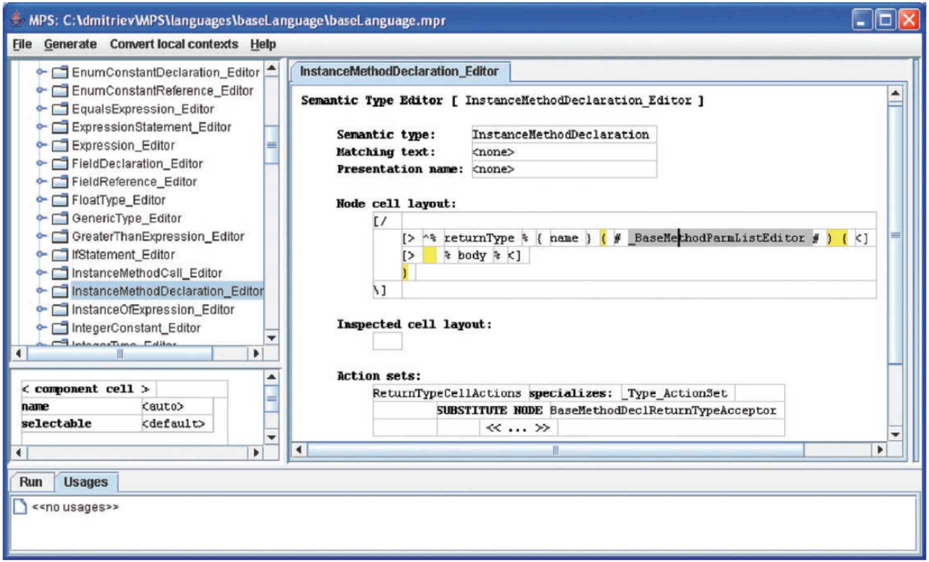
MPS Editor

confusion warning - vague

The MPS Editor is *not* a diagram editor even though programs are internally represented as graphs

We use the MPS Editor to:

- create a template of *cells*
  - cells can hold anything
- add features like autocomplete, refactoring, model checking...


MPS: C:\dmitriev\MPS\languages\baseLanguage\baseLanguage.mpr
File
Generate
Convert local contexts
Help
Run
Usages

EnumConstantDeclaration_Editor
EnumConstantReference_Editor
EqualsExpression_Editor
ExpressionStatement_Editor
Expression_Editor
FieldDeclaration_Editor
FieldReference_Editor
FloatType_Editor
GenericType_Editor
GreaterThanExpression_Editor
IfStatement_Editor
InstanceMethodCall_Editor
InstanceMethodDeclaration_Editor
InstanceOfExpression_Editor
IntegerConstant_Editor

```
< component cell >
name         <auto>
selectable   <default>
```

InstanceMethodDeclaration_Editor

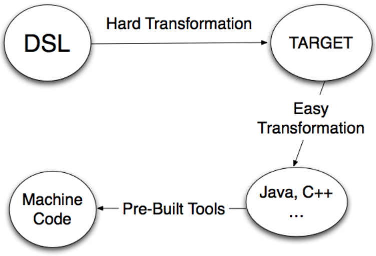
```
Semantic Type Editor [ InstanceMethodDeclaration_Editor ]

    Semantic type:      InstanceMethodDeclaration
    Matching text:      <none>
    Presentation name:  <none>

    Node cell layout:
        [/
            [> ^% returnType % { name } ( # _BaseMethodParmListEditor # ) { <]
            [>   % body % <]
            }
        /]

    Inspected cell layout:

    Action sets:
        ReturnTypeCellActions specializes: _Type_ActionSet
                 SUBSTITUTE NODE BaseMethodDeclReturnTypeAcceptor
                           << ... >>
```

<<no usages>>

MPS Transformation Language

- We want our code to *execute* or be *interpreted*!
- Could write a direct transformation from the DSL to machine code (essentially a compiler), but thats hard.

MPS compilation approach:



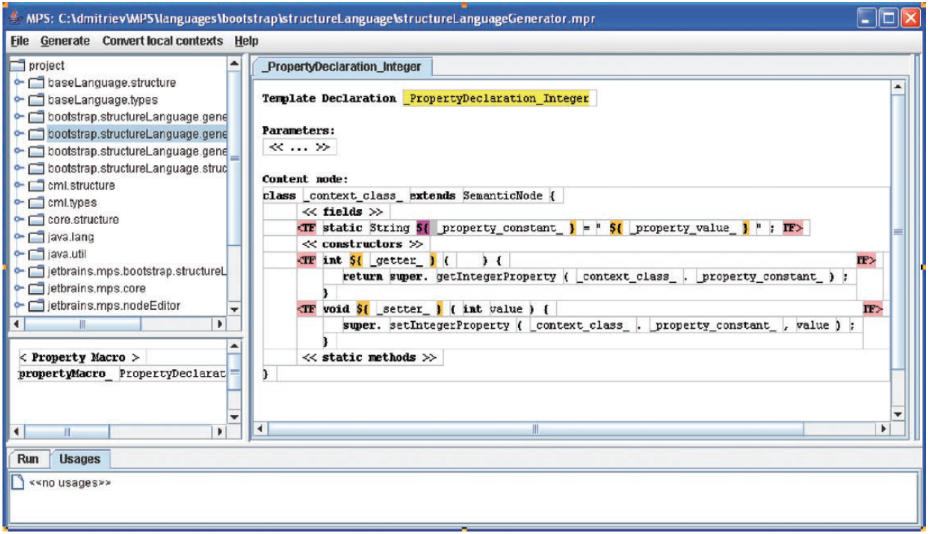
Use the Transformation Language for the hard part. Employ three separate DSLs that use: *an iterative approach*, *templates/macros*, and *search patterns*

Iterative Approach

- Enumerate code in DSL and produce target language directly
- Forex, IBM's Model Query Language. MQL essentially allows you to search for DSL code and upon matches, generate target source code
- Think of this as "find DSL code and replace with target code"

Templates and Macros

- Build a template in the target language, and write macros that are executed when you run the transformation
- Uses the iterative approach to inspect, then executes the macros that fill in the templates
- Forex, Velocity, XSLT


MPS: C:\dmitriev\MPS\languages\bootstrap\structureLanguage\structureLanguage\structureLanguageGenerator.mpr
File
Generate
Convert local contexts
Help
project
baseLanguage.structure
baseLanguage.types
bootstrap.structureLanguage.gene
bootstrap.structureLanguage.gene
bootstrap.structureLanguage.gene
bootstrap.structureLanguage.struc
cml.structure
cml.types
core.structure
java.lang
java.util
jetbrains.mps.bootstrap.structureL
jetbrains.mps.core
jetbrains.mps.nodeEditor
_PropertyDeclaration_Integer
Run
Usages

```
Template Declaration _PropertyDeclaration_Integer

Parameters:
<< ... >>

Content node:
class _context_class_ extends SemanticNode {
      << fields >>
      <TF static String ${ _property_constant_ } = " ${ _property_value_ } " ; TF>
      << constructors >>
      <TF int ${ _getter_ } (     ) {                                              TF>
          return super. getIntegerProperty ( _context_class_ . _property_constant_ ) ;
          }
      <TF void ${ _setter_ } ( int value ) {                                       TF>
          super. setIntegerProperty ( _context_class_ . _property_constant_ , value ) ;
          }
      << static methods >>
}
```

```
< Property Macro >
propertyMacro_ PropertyDeclarat
```

<<no usages>>

Patterns

- Not described in detail
- Can build a regex type of MQL

Platforms, libraries, extensions...

- Need some common functionality we never have to reimplement when making new DSLs
- In MPS libraries are languages you can use with your DSL
- The *base language* provides arithmetic, variables, loops, conditionals etc
- The *collections language* provides several container types, like those in Java.util or C++.STD
- The *UI language* provides the ability to design UIs, like java.swing
- Others: networking, database connectivity...

Conclusions

Conclusions

- Mainstream programming has disadvantages we can all agree with
- DSLs seem powerful, and people are starting to work on systems like MPS
- For DSLs to really take off, we need a good environment for building and transforming them
- “99% Java reimplemented with MPS. (wiki)” Professional GUI released in Apr. 2010. Project seems alive, you can download V2. Documentation and future articles *were* released. Q2 2012 MPS 3.0 release (Jetbrains)

Discussion Points

- Its not clear what the difference between the three types of transformation DSLs are. They are fundamentally the same (search, find, generate), just different ways of doing so.
- Paper is motivated well, but not clear if this particular solution is the answer. Using MPS seems like a lot of work; domains are small. Perhaps awesome for a company with limited product scope
- Some cool ideas. Perhaps a Maple like interface that compiles to the language of your choice, useful!