JetBRAINS november 2004

Language Oriented Programming: The Next Programming Paradigm

Sergey Dmitriev, JetBrains

It is time to begin the next technology revolution in software development, and the shape of this revolution is becoming more and more clear. The next programming paradigm is nearly upon us. It is not yet fully formed – different parts have different names: Intentional programming, MDA, generative programming, etc. I suggest uniting all of these new approaches under one name, 'language-oriented programming', and this article explains the main principles of this new programming paradigm.

Today's mainstream approach to programming has some crucial built-in assumptions which hold us back like chains around our necks, though most programmers don't realize this. With all the progress made so far in programming, we are still in the Stone Age. We've got our trusty stone axe (object-oriented programming), which serves us well, but tends to chip and crack when used against the hardest problems. To advance beyond stone, we must tame fire. Only then can we forge new tools and spark a new age of invention and an explosion of new technologies.

I'm talking about the limitations of programming which force the programmer to think like the computer rather than having the computer think more like the programmer. These are serious, deeply-ingrained limitations which will take a lot of effort to overcome. I'm not being pretentious when I say that this will be the next big paradigm shift in programming. We will need to completely redefine the way we write programs.

In this article, I present my view and my current work toward Language Oriented Programming (LOP). First I will show what is wrong with mainstream programming today, then I'll explain the concept of LOP by using the example of my existing implementation, the Meta Programming System (MPS). This article is intended to give you a bird's-eye-view of LOP, to spark interest in the idea, and hopefully to generate feedback and discussion.

Part I. LANGUAGE ORIENTED PROGRAMMING OVERVIEW

Language Oriented Programming and the Meta Programming System

Ideally, being a programmer means I can do anything on a computer. I have complete freedom, complete control. But in reality, programmers today have very restricted freedom. Sure, I can do anything on a computer, but some things take me years of effort when it should take much less time. Something is wrong here.

Programmers are restricted because they are heavily dependent on programming infrastructure which they cannot easily change, namely the languages and environments that they use. If I want some extension to a language, I must wait for the language designer to update it. If I want some extra power from my IDE, I must wait for the IDE vendor to add the new features. It is this dependence which limits my complete freedom. Sure, I can write my own compiler or IDE. In fact, this is why I started work on IntelliJ IDEA, because I was tired of being dependent on the existing weak Java IDEs. But this takes a lot of time and effort and is simply not practical for most programmers. There is a big difference between theoretical freedom and practical freedom. When I talk about freedom here, I mean practical freedom.

The way to gain freedom is to reduce our level of dependency. For example, one of the main goals of Java is to reduce dependency on the operating system, giving developers the freedom to deploy on different operating systems. So, to gain freedom over languages and environments, we should reduce our dependency on them.

Why is this a problem? Any general-purpose language, like Java or C++, gives us the ability to do anything we want with a computer. This is true, at least in theory anyway, but general-purpose languages tend to be unproductive as I will explore later. Alternatively, we could use domain-specific languages (DSLs, aka 'little languages'), which are tailored to be highly productive in a specific problem domain, such as SQL for writing database queries. The strength of DSLs, domain specificity, is also their weakness, since any real-world program will involve many different domains.

It's not a question of general-purpose versus domain-specific. I want all freedoms. I want to be able to do anything, and also be highly productive at the same time. There aren't any good ways to do this yet. Ideally, I would be able to use different languages for each specialized part of the program, all working together coherently. And the environment would fully support these languages with refactoring, code completion, navigation, and all the other productivity tools that are available for mainstream languages.

To achieve this independence, I need the freedom to create, reuse, and modify languages and environments. For this freedom to be practical, it needs to be easy to achieve. If we solve this problem of easily developing languages and environments, it will be a giant leap forward for programmers. This is where Language Oriented Programming comes in.

To understand what Language Oriented Programming is, let's first take a look at today's mainstream programming. It goes something like this:

Think:

You have a task to program, so you form a conceptual model in your head about how to solve the problem.

Choose:

You choose some general-purpose language (such as Java or C++) for writing the solution.

Program:

You write the solution by performing a difficult mapping of your conceptual model into the programming language.

The **Program** step is the bottleneck because the mapping is not easy or natural in most cases (see Figure 1). This method has proved ineffective for programmers to express complex programs. In contrast, here is how LOP would work:

Think:

You have a task to program, so you form a conceptual model in your head about how to solve the problem.

Choose:

You choose some specialized DSLs for writing the solution.

Create:

If there are no appropriate DSLs for your problem, then you create ones that fit your problem.

Program:

You write the solution by performing a relatively straightforward mapping of your conceptual model into the DSLs. Now, the Program step is much less of a bottleneck because the DSLs make it much easier to translate the problem into something the computer can understand (See Figure 2). It may appear that the difficulty has simply shifted to the Create step. However, a combination of tool support and applying LOP to itself will make this step much easier.

The motivation behind LOP goes something like this: I want to be able to work in terms of the concepts and notions of the problem I am trying to solve, instead of being forced to translate my ideas into the notions that a generalpurpose language is able to understand (e.g. classes, methods, loops, conditionals, etc.). To achieve this, I need to use domain-specific languages. How do I get them? I create them.

I have begun development of a universal platform (the Meta Programming System) for designing domainspecific languages along with their supporting tools and environments. It will allow programmers to define languages as easily as they can write programs today. The platform will fully support LOP, giving programmers the freedom to use the most suitable language for each part of their programs, rather than tying them down to one fixed general-purpose programming language.

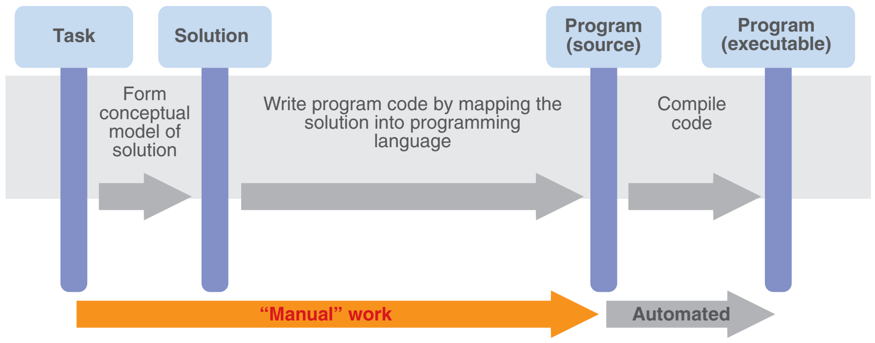

Task
Solution
Program (source)
Program (executable)
Form conceptual model of solution
Write program code by mapping the solution into programming language
Compile code
"Manual" work
Automated

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

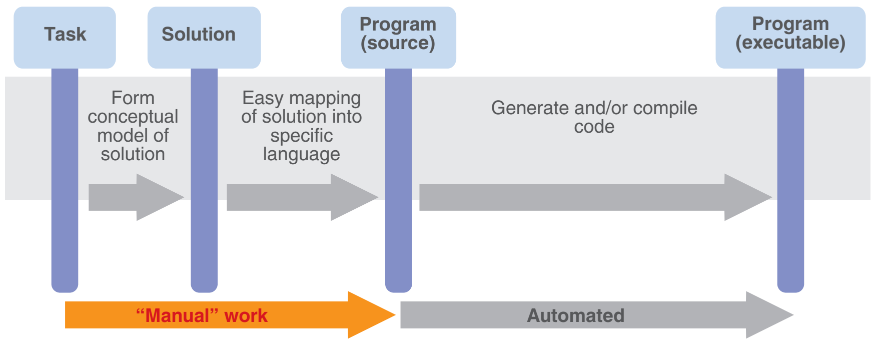

Task
Solution
Program (source)
Program (executable)
Form conceptual model of solution
Easy mapping of solution into specific language
Generate and/or compile code
"Manual" work
Automated

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

MPS is just one example of Language Oriented Programming. Although I will use MPS as an example, LOP could be achieved in many different ways. You might know of some alternatives yourself. The concept of LOP is not the same as its implementation, just as the concept of OOP is not the same as Java or C++ or Smalltalk.

What Is Wrong with Mainstream Programming

You know the old saying, "If it ain't broke, don't fix it". Mainstream programming is definitely broken. I see many problems with it, and most of them stem from the fact that there is no way for a general-purpose language to fully support arbitrary domains, and likewise there can be no universal domain-specific language. Here are the three worst problems with mainstream programming that will be solved by LOP:

Time Delay to Implement Ideas

For me, the most serious problem is that there is a very long gap between when I know exactly how to solve a problem and when I have successfully communicated this solution to the computer as a program. I can explain the problem and solution to another programmer in a matter of hours, but encoding this solution into the computer takes much longer. This is because with a programmer I can use natural language which is very rich, but for the computer, I must use a general-purpose programming language which is much less expressive. Programming languages today have only tens of notions that can be expressed. A natural language has tens of thousands of notions which can be expressed succinctly. So, to explain a program to another programmer, I can just express very high-level ideas, but for the computer, I must express every single step and every detail.

In mainstream programming, most of the time spent 'programming' is really just finding ways to express natural language concepts in terms of programming level abstractions, which is difficult, not very creative, and more or less a waste of time.

For example, today a good deal of development time is spent on object-oriented design (OOD). This is actually a fairly creative process where the programmer expresses classes, hierarchies, relationships, and such. The purpose of this exercise is to express the program in objectoriented terms such as classes and methods.

The process of OOD is necessary because these classes and methods are the only abstractions that objectoriented languages understand. It seems like it is necessary and creative, but with Language Oriented Programming, OOD is not needed at all.

Understanding and Maintaining Existing Code

The next problem I have is in understanding and maintaining existing code. Whether it is written by another programmer or by me, the problem is the same. Because general-purpose languages require me to translate highlevel domain concepts into low-level programming features, most of the big picture is lost in the resulting program. When I come back to the program later, I have to reverse engineer the program to understand what I originally intended, and what the model in my head was. Basically, I must mentally reconstruct the information that was lost in the original translation to the generalpurpose programming language.

The traditional way to address this problem is to write comments or other forms of documentation to capture the design and model information. This has proven to be quite a weak solution for a number of reasons, not the least of which is the cost of writing such auxiliary documentation, and the tendency of documentation to grow out-of-synch with code. Additionally, and not as frequently recognized, is the fact that documentation cannot be directly connected to the concept it is documenting. Comments are tied to the source code in a single location, but the concept may be represented in the code in many places. Other types of documentation are entirely separated from the code and can only indirectly reference the code. Ideally, the code should be self-documenting. I should read the code itself to understand the code, not some comments or external documentation.

Domain Learning Curve

The third major problem is with domain-specific extensions to the language. For example, in OOP the primary method of extending the language is with class libraries. The problem is that libraries are not expressed in terms of domain concepts, but in lower-level general-purpose abstractions such as classes and methods. So, the libraries rarely represent the domain directly. They must introduce extra complications (such as the runtime behavior of a class) to complete the mapping. Two good and common examples are graphical user interface libraries and database libraries.

Learning such libraries is not a simple task, even if you are an expert in the domain. Since there is no direct mapping from domain to language, you must learn this mapping. This presents a steep learning curve. Usually we attempt to solve this problem with extensive tutorials and documentation, but learning this takes a lot of time. As a library becomes more complex, it becomes much more difficult to learn, and programmers lose motivation to learn it.

Even after learning such a complicated mapping, it remains very easy to misuse the library because the environment (such as compiler and editor) isn’t able to help you use the library correctly. To these tools, a call to a method on a GUI object is the same as a call to a method on a DB object—they are both just method calls on objects, nothing more. It is up to the user to remember which classes and methods need to be invoked, and in what order, and so on.

And even if you are an expert in the domain and alsoan expert user of the library, there is still the problem of the verbosity of programs written using the library. Relatively simple domain concepts require complicated gestures to invoke correctly. Anyone who has used Swing, for example, is aware of this. It just takes too long to write simple things, and complex things are even worse.

Details of LOP

What Is a Program in LOP?

Today, ninety-nine percent of programmers think programming means writing out a set of instructions for the computer to follow. We were taught that computers are modeled after the Turing machine, and so they ‘think’ in terms of sets of instructions. But this view of programming is flawed. It confuses the means of programming with the goal. I want to show you how LOP is better than traditional programming, but first I must make something clear: A program in LOP is not a set of instructions. So what is a program then?

When I have a problem to solve, I think of the solution in my head. This solution is represented in words, notions, concepts, thoughts, or whatever you want to call them. It is a model in my head of how to solve the problem. I almost never think of it as a set of instructions, but instead as a set of inter-related concepts that are specific to the domain I'm working in. For example, if I'm thinking in the GUI domain, I think *'I want this button to go here, this field to go here, and this combobox should have a list of some data in it.'* I might even picture it in my head, without any words at all.

I say that this mental model is a solution because I can explain this model to another programmer in enough detail that the programmer could sit down and write a program (e.g. in Java) which will solve the problem. I don't need to explain the solution in terms of a programming languages – it could be in almost any form. To explain how to lay out a GUI form, I could just draw the form, for example. If this drawing has enough detail, then the drawing itself represents the solution. Such domain-specific representations should be the program. In other words, there should be a method that allows me to use this representation as an actual program, not just as a way of communicating with other programmers. So this leads to my informal definition of a program: A program is any unambiguous solution to a problem. Or, more exactly: A program is any precisely defined model of a solution to some problem in some domain, expressed using domain concepts.

This is the main reason I think programmers should have the freedom to create their own languages – so they can express solutions in more natural forms. General-purpose languages are unambiguous, but too verbose and errorprone. Natural language (e.g. English) is very rich, but currently it is too difficult because it is very informal and ambiguous. We need to be able to easily create formal, precisely defined, domain-specific languages. So Language Oriented Programming will not just be writing programs, but also creating the languages in which to write our programs. Our programs will be written closer to the problem domain instead of in the computer's set-ofinstructions domain, and so they will be much easier to write.

Programs and Text

Everyone is used to the idea that a program is stored as text, i.e. a stream of characters. And why shouldn't it be? After all, there are countless tools for editing, displaying, and manipulating text. Central parts of programming languages today are their grammars, parsers, compilers, and line-oriented debuggers. But a program's text is just one representation of the program. Programs are not text. Forcing programs into text form causes lots of problems that you might not even be aware of. We need a different way to store and work with our programs.

When a compiler compiles source code, it parses the text into a tree-like graph structure called an abstract syntax tree. Programmers do essentially the same operation mentally when they read source code. We still have to think about the tree-like structure of the program. That's why we have brackets and braces and parentheses. It's also why we need to format and indent code and follow coding conventions, so that it is easier to read the source.

Why do we resort to text storage? Because currently, the most convenient and universal way to read and edit programs is with a text editor. But we pay a price because text representations of programs have big drawbacks, the most important of which is that text-based programming languages are very difficult to extend. If programs are stored as text, you need an unambiguous grammar to parse the program. As features are added to the language, it becomes increasingly difficult to add new extensions without making the language ambiguous. We would need to invent more types of brackets, operators, keywords, rules of ordering, nesting, etc. Language designers spend enormous amounts of time thinking about text syntax and trying to find new ways to extend it.

If we are going to make creating languages easy, we need to separate the representation and storage of the program from the program itself. We should store programs directly as a structured graph, since this allows us to make any extensions we like to the language. Sometimes, we wouldn't even need to consider text storage at all. A good example of this today is an Excel spreadsheet. Ninety-nine percent of people don't need to deal with the stored format at all, and there are always import and export features when the issue comes up. The only real reason we use text today is because we don't have any better editors than text editors. But we can change this.

The problem is that text editors are stupid and don't know how to work with the underlying graph structure of programs. But with the right tools, the editor could work directly with the graph structure, and give us freedom to use any visual representation we like in the editor. We could render the program as text, tables, diagrams, trees, or anything else. We could even use different representations for different purposes, e.g. a graphical representation for viewing, and a textual representation for editing. We could use domain specific representations for different parts of the code, e.g. graphical math symbols for math formulas, graphic charts for charts, rows and columns for spreadsheets, etc. We could use the most appropriate representation for the problem domain, which might be text, but is not limited to text. The best representation depends on how we think about the problem domain. This flexibility of representation would also enable us to make our editors more powerful than ever, since different representations could have specialized ways to edit them.

What Is a Language in LOP?

Lastly, I should clarify what I mean by 'language'. In LOP, a language is defined by three main things: Structure, editor, and semantics. Its structure defines its abstract syntax, what concepts are supported and how they can be arranged. Its editor defines its concrete syntax, how it should be rendered and edited. Its semantics define its behavior, how it should be interpreted and/or how it should be transformed into executable code. Of course, languages can also have other aspects,such as constraints and type systems.

Part II. INTRODUCTION TO META PROGRAMMING SYSTEM

Creating Languages in MPS

I have explained *why* we need to easily create new languages. But *how* can we make it easy? If you turn the question around and apply Language Oriented Programming to itself, you will soon see the answer. This calls for a little self-referential bootstrapping, which can seem tricky, but be patient. Once you understand this, you will 'get' the real power of LOP.

Recall that the idea of LOP is to make it easy to create special domain-specific languages, and those DSLs will make writing our programs easier. But as I've shown, 'programs' in LOP are not restricted to mean the typical 'set-of-instructions' programs you are used to. Any unambiguous solution to some problem in some domain is a 'program'. So if you imagine the domain of 'creating new languages', then a 'program' in that domain would actually be a definition of a new language itself, which can be thought of as a solution just like any other solution.

So, applying the idea of LOP, the way to make 'creating new languages' easy is to create special DSLs dedicated to the domain of 'creating new languages'. By using these language-building DSLs, we can make it easy to make new languages. Let's look at some of these language-building languages to give you a better idea of how this works. This is an overview; future articles will describe these languages in more detail.

Structure Language

At the bare minimum, we need to define the 'structure' of a new language. This is how we will be able to write 'precisely defined' programs. The structure of a language doesn't mean its textual grammar—as I mentioned, there may not even be a textual representation of the program, but only a graph representation.

In most cases, while practicing LOP, you work with two 'levels' of programming, the meta level and the program level. You define the language in the meta level, and write the program in the program level. When defining the structure of a new language, you would use a language-structure DSL to define your new language, which would reside in the program level.

In MPS, each node in the program level has a 'type' which is just a link to another node in the meta level. The node in the program level is said to be an 'instance' of the type. The meta level 'type' node defines what relationships its instances can have and also what properties they will have. The language for describing this meta level language structure is called simply the Structure Language.

To define a language's abstract syntax with the Structure Language, you should just enumerate all the types in the language. The types simply represent the features, or concepts, that the language supports. Each concept should be defined by its name, the internal properties of its instances, and the relationships (basically links) its instances can have with other nodes (see Figure 3).

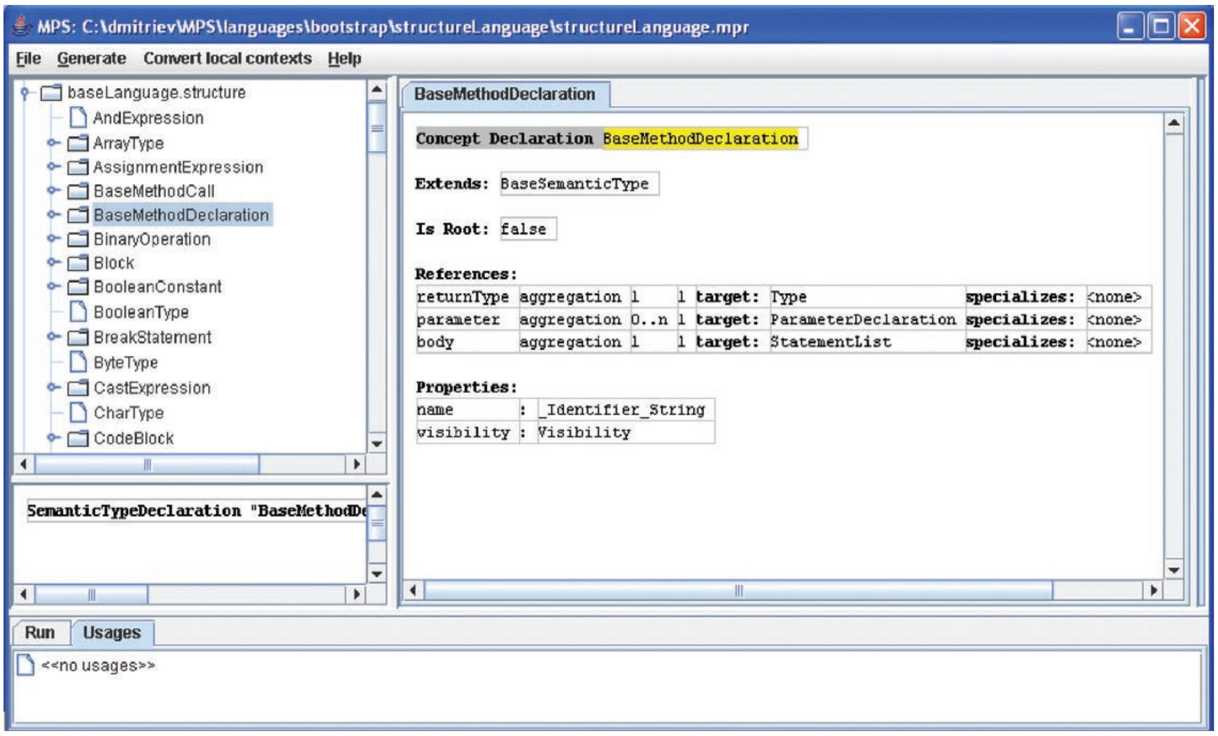
There are two kinds of relationships possible. The first kind is an aggregation-like relationship which forms the parent-child tree structure of concept models. The second kind is a non-aggregating, freeform relationship which can link any node to any other node in the system. Relationships have two ends, the source end and the target end. Relationships have roles, and for every role you define the name of the role, the cardinalities of each end, and the type of the target nodes. Cardinalities can be 1, 0..1, 0..n, or 1..n, which lets you restrict how many links can be created for this relationship. The relationship target type can be used to restrict what types of nodes can be linked together.

So, using the new language to write a program would involve creating instances of the concepts in the language, assigning values to the properties of the instances, and linking the nodes in the program together according to the relationships defined by the language concepts. All of this will be supported by powerful editors which you can define for your language.

Editor Language

So, what is the interface for writing and manipulating concept models? We need some sort of editor for our languages. But we don't want a generic editor; experience has shown that generic editors aren't as usable as we want them to be. We want writing models to be fast, so we want specialized editors tailored to our language concepts. In a way, the editor is part of the language, and our goal is to create new languages easily, so creating new editors should also be easy. Essentially, we need a language for creating editors. In MPS, it is called the Editor Language.

When people hear me say that our programs will be stored as graphs and we will have special editors, I'm sure many will think that I'm talking about diagram editors. This is not the case. Even though the programs are graphs, the editors don't have to render as diagrams. In fact, diagram editing is usable in only a small percentage of cases (i.e. when it is appropriate, such as with database tables). In contrast, there is a



```
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
```

Figure 3: Definition of the "Method" concept in the Structure Language

much better source of inspiration for our Editor Language, and that ironically happens to be text editors.

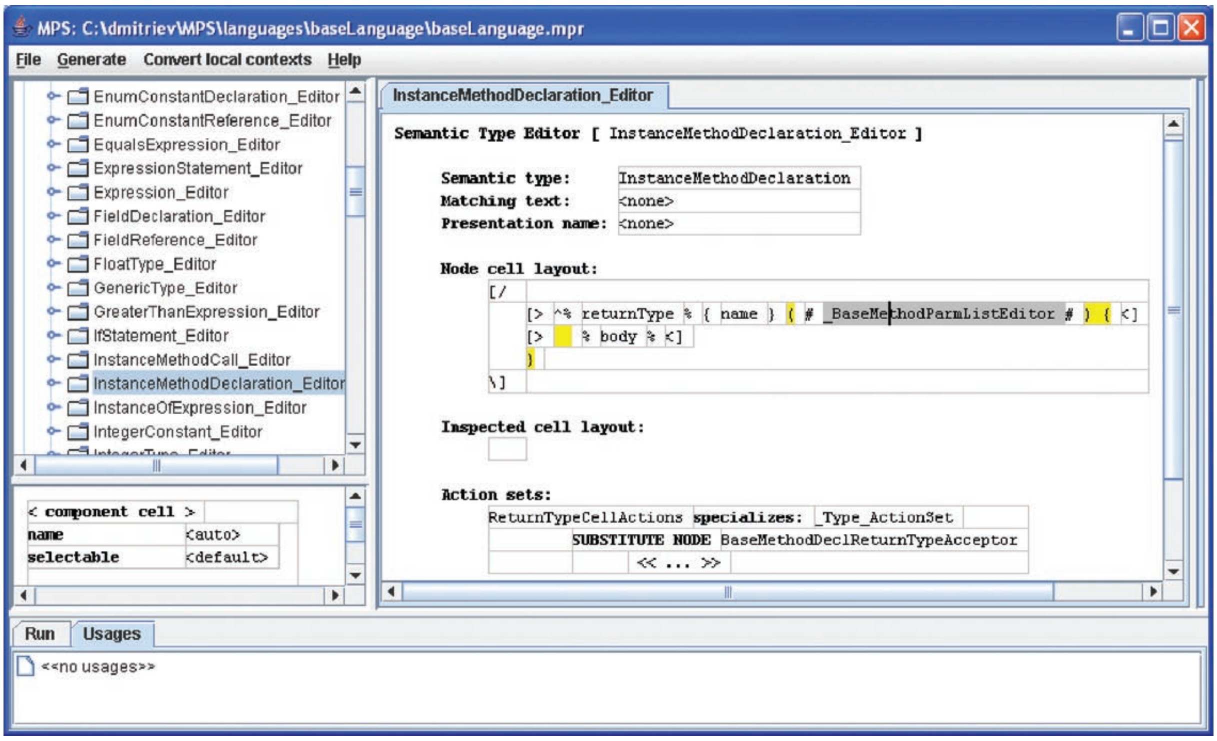
If you look at a typical program in a text editor, you can imagine that the editor is divided into rectangular cells. Some cells would contain required symbols like keywords, braces, and parentheses, and other cells would contain user-defined symbols like class and method names. Larger cells would be composed of smaller cells, like a method block containing statements, which might also have their own nested blocks. In fact, any well-formed program in any mainstream programming language could be composed into a set of rectangular cells. Well, in the Editor Language, you don't have to imagine these cells, because the editors simply are composed of rectangular cells (see Figure 4).

The usage of cells has some interesting advantages. First, the cells can perfectly mimic, and even outdo, standard text editors while working directly on the program graph instead of text. Second, cells are not limited to text; you could have anything like color choosers, math symbols, charts, vector graphics, or anything else in a cell. In the end, even this cell layout is optional and the programmer could provide a different mechanism. The cell layout is just a useful default.

So the Editor Language helps you define the layout of cells for each concept in the language. You can define which parts are constant, like braces or other decorations, and which parts are variable and need the user to define them. The Editor Language also helps you add powerful features to your own editors, like auto-complete, refactoring, browsing, syntax highlighting, error highlighting, and anything else you can think of. So you can add the power of today's editors, like IntelliJ IDEA, to your own custom languages. This is possible because programs and languages are structured as graphs, and because we have a specialized Editor Language that helps us create powerful editors.

Transformation Language

The Structure Language and Editor Language together already provide some power. You could use them to communicate ideas to other people, for example to draw UML diagrams or to write other types of static documents. However, most of the time we want our code to do something, so we have to find a way to make it executable. There are two main ways to do this: Interpretation and compilation.



MPS: C:\dmitriev\MPS\languages\baseLanguage\baseLanguage.mpr
File
Generate
Convert local contexts
Help
InstanceMethodDeclaration_Editor
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>
```

```
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>>
```

Figure 4: Definition of an editor for the "Method" concept

Interpretation is supported by DSLs to help define how the computer should interpret the program. Compilation is supported by DSLs to help define how to generate executable code from our program. I will discuss support for interpretation in future articles. Right now I want to show how MPS supports compilation.

Compilation means to take source code and generate some form of executable code from it. There are many possibilities for the format of the resulting code. To generate executable code, you could generate natively executable machine code or bytecode that runs in a virtual machine. Alternatively, you could generate source code in a different language (e.g. Java or C++), and later use an existing compiler to turn that into executable code. Along the same lines, you could even generate source code in some interpreted language, and use the existing interpreter to execute the code.

To avoid dealing with such a wide variety of target formats, our approach is to do everything in MPS. First, you define a target language in MPS using the Structure Language. This target language should have a direct, one-to-one mapping to the target format. For example, if your target format were machine code, you would define a Java-like target language. The target language doesn't have to support all the features of the target format, just as long as there is a simple, one-to-one mapping for all of the language features that you need.

So now there are two phases to compilation, a simple translation from the target language to the final result, and a more complex transformation from the initial source language to the intermediate target language. The translation phase is trivial, so we can focus on the more interesting transformation phase. Essentially, the problem is now simplified into how to transform models from one language to another. But the source language and target language could be radically different, making transformations very complex, for example by mapping one source node to many target nodes scattered throughout the target model. We want to make it as easy as possible to define transformations, so we need a model-transformation DSL to help us. In MPS, this DSL is called the Transformation Language.

There are three main approaches to code generation, which we would like to use together to define model transformations. The first is an iterative approach, where you enumerate all the nodes in the source model, inspect each one, and based on that information generate some resulting target nodes in the target model. The second approach is to use templates and macros to define how to generate code in the target language. The third approach is to use search patterns to find where in the source model to apply transformations.

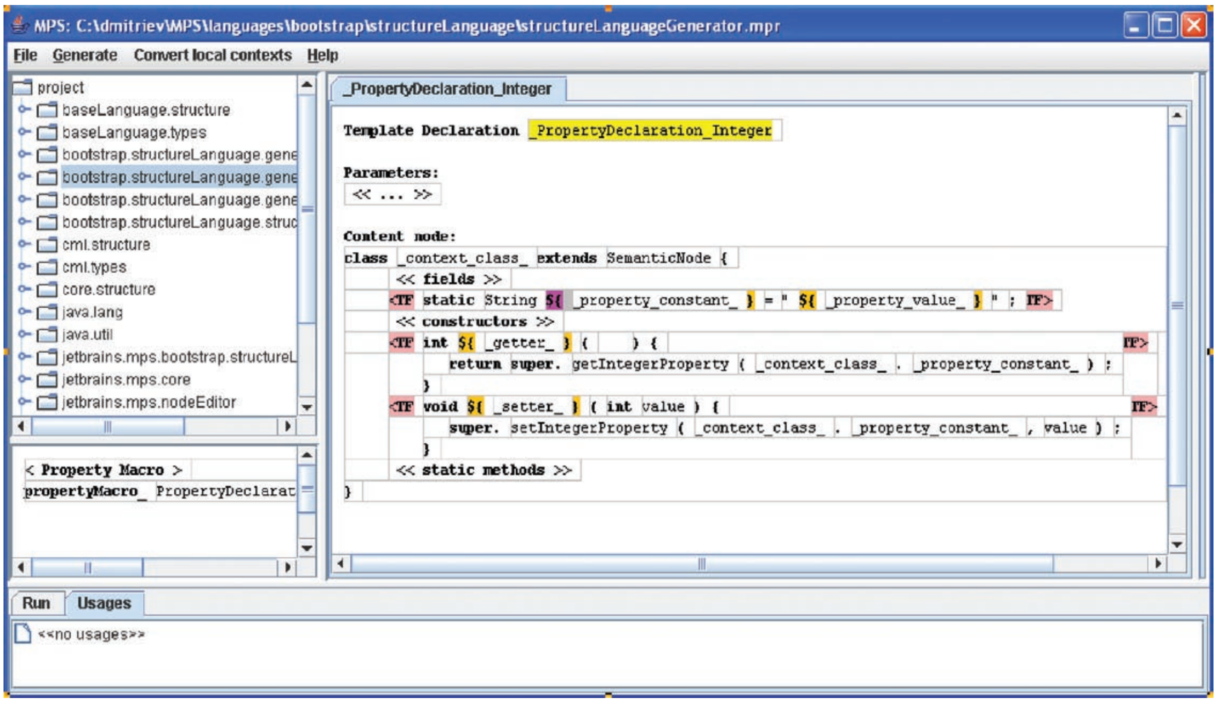
We combine these approaches by defining DSLs to support each approach. The DSLs will all work together to help you define transformations from one language to another. For example, the iterative approach inspired the Model Query Language, which makes it easy to enumerate nodes and gather information from a concept model. You can imagine this as something like SQL for concept models. As a bonus, having a powerful query language is useful for more than just code generation (e.g. making editors smarter).

Templates

The template approach works something like Velocity or XSLT. Templates look like the target language, but allow you to add macros in any part of the template. Macros are essentially bits of code that are executed when you run the transformation. The macros allow you to inspect the source model (using the Model Query Language), and use that information to 'fill in the blanks' in the template to generate the final target code.

In Figure 5, you can see the definition of a template for generating Java code for a "Property" concept. The template adds field declarations, getters, and setters for the property. This template is part of the generator that translates code from the Structure Language into Java.

Since the templates look like the target language, you can imagine that templates are written in a special language that is based on the target language. This is in fact how it works. Instead of manually creating a new template language for each possible target language, we actually have a generator which generates the template language for you. It basically copies the target language and adds in all the special template features like macros and such. Even the template


MPS: C:\dmitriev\MPS\languages\bootstrap\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
<<no usages>>

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

```
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 >>
}
```

Figure 5: Template for generating Java code for the “Property” concept

editors are generated from the target language’s editors, so you don’t have to hand code them either.

When you use a template language, you can think of it as writing code in the target language where some parts of the code are ‘parameterized’ or ‘calculated’ with macros. This technique helps simplify code generation enormously. Templates can also be used for other tasks like refactoring, code optimizers, and more.

Patterns

The model pattern-matching approach gives us a powerful way to search models, as an alternative to the Model Query Language. You can imagine patterns as regular expressions for concept models. Similar to the template approach, we will generate a pattern language based on the source language. The pattern language looks like the source language, but adds features which help you to define flexible criteria for performing complex matching on the source model. You can imagine this approach as a powerful search-and-replace technique. Again, the pattern languages are useful for more than just code generation. For example, they would be very useful for writing automatic code inspections for the source language’s editors.

Remember that the Model Query Language, template languages, and pattern languages are all supported by powerful editors with auto-complete, refactoring, reference checking, error checking, and so on. Even complex queries, macros, and patterns will be easy to write. Code generation has never seen this level of power.

Using Languages Together

The previous section on code generation raises some interesting issues about how languages can work together. There are in fact several ways to achieve it. In MPS, all the concept models know about each other. Since languages are concept models too, this means that all the languages know about each other, and can potentially be interlinked.

Languages can have different relationships to each other. You could create a new language by extending an existing one, inheriting all of its concepts, modifying some of them, and adding your own. One language could reference concepts from another language. You could even ‘plug’ one language into another. I will discuss this in more detail in future articles.

Platforms, Frameworks, Libraries, and Languages

Our system for supporting Language Oriented Programming needs more than just meta-programming capabilities to make it useful. It should also support all the things that programmers have come to rely upon from today's programming languages: Collections, userinterface, networking, database connectivity, etc. Programmers don't choose languages solely based on the language itself. For instance, much of the power of Java comes not only from the language, but from the hundreds and hundreds of frameworks and APIs available for Java programmers to choose from. It's not the Java language they are buying into, but the entire Java platform. MPS will also have a supporting platform of its own.

Before I get into the specifics, let's talk briefly about frameworks. What is a framework? In mainstream programming, it usually means a set of classes and methods packaged up into a class library. Let's look a little closer at this and see what we can see through the lens of LOP.

Why do we want to package up classes and methods into libraries? Most programmers would recite what their professors once told them and say, "Reuse." But that just leaves another question in its place. Why do we want to reuse some set of classes? The answer is because the set of classes is useful for solving certain types of problems, like making GUIs, or accessing databases, or whatever. You might say that a class library corresponds to some domain. Lo and behold, we see the connection. Class libraries are wannabe DSLs! This sad fact really frustrates me.

Domain-specific languages exist today in the form of class libraries, except they aren't languages, have none of the advantages of languages, and have all the limitations of classes and methods. Specifically, classes and methods are immediately tied to a specific runtime behavior which can't be modified or extended, because that behavior is defined by the concepts of 'class' and 'method'. Because they are not languages, class libraries are rarely supported intelligently by the environment (compiler and editor, for example).

Should we be stuck with wannabe DSLs, or should we have the freedom to use a real DSL when a DSL is called for? Freedom, of course. Any class library is a good candidate for creating a full-fledged DSL for our platform. For example, all the libraries in the JDK should be DSLs for the MPS platform. Some of these DSLs are not so critical at the outset, but others will have a big impact on the power and reusability of the platform right from the beginning. I want to talk briefly about the three most important platform languages that will be provided with MPS: The Base Language, the Collection Language, and the User Interface Language.

Base Language

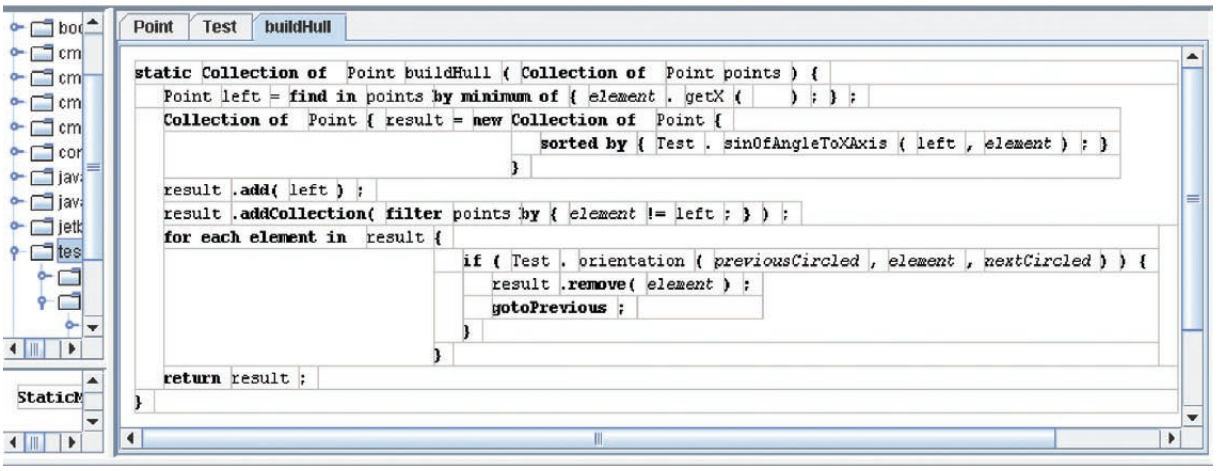
The first thing we need is a language for the simplest programming domain, which is general-purpose imperative programming. This simple language would support such nearly-universal language features as arithmetic, conditionals, loops, functions, variables, and so on. In MPS we have such a language, which is called the Base Language.

The need for such a language should be clear. For example, if we want to add two numbers together, we should be able to say 'a + b' as simple as that. We won't need to use it everywhere, but it will be needed in some part of nearly all programs, wherever it is the most appropriate tool for the job.

The Base Language is so named because it is a good foundation for many languages that need basic programming support like variables, statements, loops, etc. It can be used in three ways. You can extend it to create your own language based on it, you can reference its concepts in your programs, and you can generate your code to the Base Language. There will be various generators available to transform the Base Language into other languages like Java, C++, etc. Not every language needs to use the Base Language, of course, but it's a good starting point in many cases.

Collection Language

The next most important language we need is a language for working with collections. The need for collection support is ubiquitous. Every major mainstream language has some sort of support for collections. For example, in Java you have java.util, in C++ you have STL. Everybody needs collections. If different DSLs had their own collection support, there would be a



```
static Collection of  Point buildHull ( Collection of  Point points ) {
  Point left = find in points by minimum of { element . getX (     ) ; } ;
  Collection of  Point { result = new Collection of  Point {
                                  sorted by { Test . sinOfAngleToXAxis ( left , element ) ; }
                                }
  result .add( left ) ;
  result .addCollection( filter points by { element != left ; } ) ;
  for each element in  result {
                            if ( Test . orientation ( previousCircled , element , nextCircled ) ) {
                              result .remove( element ) ;
                              gotoPrevious ;
                            }
                          }
  return result ;
}
```

Figure 6: Convex hull algorithm using the Collection Language

Babylon of different collection languages, each incompatible with each other. This is why MPS must provide a single Collection Language which everyone uses.

In many mainstream languages, collections are not language features but class libraries. A good example is Java's java.util package. The support is technically there, but it is so inconvenient, messy, and error prone.

Yuck! Most Java code today is littered with lines and lines of redundant, repetitive code for handling collections. Figure 6 shows an example of how a Collection Language beats the tar out of a class library. The example is an algorithm for finding the convex hull of a given set of points. More details about the Collection Language will be forthcoming in future articles.

User Interface Language

The User Interface Language is the next most important DSL for our platform. Interestingly, the Editor Language I mentioned previously could conceivably be used for providing user interfaces, but a full-fledged language for graphical user interfaces would be more flexible. The benefits of such a language would be enormous. Java Swing code is a perfect example of a class library wanting to be a DSL. The features are there, but are easy to misuse [3], and Swing code is a complete mess. Many environments today include GUI builders to simplify user-interface creation. The User Interface Language will take that mission to a higher plane. I will discuss this language in more detail in future articles.

Getting Started with MPS

I can already hear some of the skeptical responses to LOP: "Sounds great, but our project is already underway and switching to LOP at this point isn't practical," or "Sounds great, but it's too risky to start a real-life project with an untested method like LOP," or "Sounds great, but when will it be ready for prime time? OOP took 20 years to become mainstream."

The good news is that you won't have to dive head-first into the unknown; you can dip your toe in and see if the water is nice first. You can try just a little bit of LOP on your project to see if it provides a practical advantage, and then try a bit more if you like it. Here are two possible applications of LOP which you will be able to try in the near future with MPS.

Using MPS on Java Applications

There is already a prototype plugin for IntelliJ IDEA which will allow you to include MPS concept models in your project. The models will automatically be translated into Java source code in the background as you edit them. So, you will be able to write part of your Java applications using MPS, as much or as little as you want. This means that you get all the power of MPS, such as the ability to create and use specialized DSLs, to make whatever language extensions you want, as well as to use customizable editors with code completion, error highlighting, refactoring, etc. The plugin will be tightly integrated with IDEA, allowing you to embed Java code in your MPS models, navigate to embedded or generated Java code, and even perform concept-level debugging similar to the JSP debugging support already available in IDEA. More integration features are planned, as well. This will be an important new tool available to Java developers using IDEA.

Configuring and Scripting Your Applications

There's a pattern I've seen many times. An application starts off needing some form of configuration, be it a simple options file, or a more complete deployment descriptor file. Eventually, configurations become more complex, and the application ends up needing a scripting language. For simple configuration files, XML is popular. For scripting languages, you can create your own, or borrow a general-purpose one like VBScript, Python/Jython, Tcl, Javascript, or maybe Lisp. Each of these solutions has at least some of the standard drawbacks of mainstream programming: Long time to implement, steep learning curve, hard to extend, poor environment support, etc.

Alternatively, you could create your own configuration/ scripting language with MPS. Users of your application would then have an easy-to-use, intelligent editor for writing their scripts, including syntax highlighting, error highlighting, code completion, navigation, etc. It will take a fraction of the time to create and integrate the language into your application. You will be able to redistribute the MPS runtime for use with this kind of application.

Conclusion

The ideas underlying LOP and MPS are not new, and have actually been around for more than 20 years [1]. The term Language Oriented Programming itself has been around for at least 10 years [2]. What is new is that these ideas have silently saturated the software development community, and their time has finally come. With this article, I hope to provide a seed around which these ideas can crystallize into new discussions, opinions, critiques, experiments, research, and real-life projects.

And so, I invite you to take part in this new paradigm in whatever way you can. Add a comment below, or send me an email at mps_article@jetbrains.com. Find out more about MPS at http://www.jetbrains.com/mps and watch for updates. Take a fresh look at websites, magazines, blogs, and books from the perspective of LOP, and think about how much easier things could be. Think about your own projects and how often what you're doing is actually designing and using little specialized languages cobbled together with classes and methods. What do you think about that? I want to know.

I have already seen first-hand how Language Oriented Programming can drastically improve software development, as I have been using the concept of LOP to develop MPS itself. MPS is currently not ready for the real world, but it is getting there. There is also no documentation yet, except for this article. I will publish more articles soon, which will explore MPS in more depth. Also, I plan to make MPS available to download and try out in the coming months, so keep your ears open.

There are other projects out there that follow similar approaches, notably from Intentional Software [4], and Xactium [5].

So have fun exploring, and let me know what you find.

Acknowledgements

I would like to thank Rob Harwood for his help in editing this article. I would also like to thank the following people for their reviews, comments, and suggestions: Igor Alshannikov, Florian Hehlen, Jack Herrington, Guillaume Laforge, Vaclav Pech, Thomas Singer, Dmitry Skavish, David Stennett, and Timur Zambalayev.

About the Author

Sergey Dmitriev (http://www.sergeydmitriev.com) is the co-founder and CEO of JetBrains Inc. (http://www.jetbrains.com), makers of the IntelliJ IDEA Java IDE.

References

Articles:

[1] Donald E. Knuth. Literate programming. The Computer Journal, 27, 97-111, May 1984.

[2] M. Ward. Language Oriented Programming. Software - Concepts and Tools, 15, 147-161 1994, http://www.dur.ac.uk/martin.ward/martin/papers/middle-out-t.pdf

Intentional Programming articles:

Charles Simonyi. *The Death of Computer Languages, The Birth of Intentional Programming*. 1995. ftp://ftp.research.microsoft.com/pub/tr/tr-95-52.doc also ftp://ftp.research.microsoft.com/pub/tr/tr-95-52.ps

John Brockman. Intentional Programming: *A Talk With Charles Simonyi*. Edge. 2000. http://www.edge.org/digerati/simonyi/simonyi_p1.html

Microsoft Research. *Intentional Programming.* http://www.cse.unsw.edu.au/~cs3141/ip.asf (video)

Charles Simonyi. *Intentional Programming: Asymptotic Fun?* http://www.hpcc.gov/iwg/sdp/vanderbilt/position_papers/simonyi.pdf

Books:

Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools and Applications.* Addison-Wesley, 2000. ISBN: 0201309777.

Jack Herrington. *Code Generation in Action*. Manning, 2003. ISBN: 1930110979. http://www.codegeneration.net/cgia/

Xactium. Applied Metamodelling: *A Foundation for Language Driven Development*. 2004. http://albini.xactium.com/content/index.php?option=com_remository&Itemid=28

Other Resources on the Web:

[3] Matt Quail. Totally Gridbag. http://madbean.com/blog/2004/17/

Jack Herrington. Code Generation Network. http://www.codegeneration.net/

[4] Intentional Software http://www.intentsoft.com

[5] Xactium http://www.xactium.com

Intentional Programming interviews

Sergey Dmitriev. http://codegeneration.net/tiki-read_article.php?articleId=60

Charles Symonyi. http://codegeneration.net/tiki-read_article.php?articleId=61

Krzystof Czarnecki. http://codegeneration.net/tiki-read_article.php?articleId=64

Andy Evans. http://codegeneration.net/tiki-read_article.php?articleId=68