

# Language Oriented Programming: The Next Programming Paradigm

Nidhi Saluja, Pooja Dhiman

Lecturer - MCA Deptt.  
SDDIET, BARWALA

*Abstract - Today's mainstream approach to programming has some crucial built-in assumptions, though most programmers don't realize this. The limitations of programming force the programmer to think like the computer rather than having the computer think more like the programmer. These are serious limitations which will take a lot of effort to overcome. Though object-oriented programming serves us well, but it tends to chip and crack when used against the hardest problems. To advance beyond these limitations, we must forge new tools and spark a new age of invention and an explosion of new technologies. If the problem of easily developing languages and environments is solved, it will be a giant leap forward for programmers. For the next big paradigm shift in programming, we will need to completely redefine the way we write programs. Different parts of next technologies have different names like Intentional programming, MDA, generative programming, etc. All of these new approaches can be united under one name, 'Language-Oriented Programming Paradigm'. This paper, focusing on Language Oriented Programming, surveys and analyses the problems in mainstream programming, the current work being done towards LOP, and explains the concept of LOP by using its existing implementation in Meta Programming System(MPS). This article is intended to give a bird's-eye-view of LOP and to spark interest in the idea.*

Keywords – Programming Paradigm, Meta Programming System(MPS), Language Oriented Programming(LOP), Domain Specific Language(DSL)

## 1. Introduction

The word 'programming' means the process of writing computer programs and the word 'paradigm' is used to denote a typical pattern of

programming. Hence, programming paradigm is a fundamental style of programming regarding how solutions to problems are to be formulated in a programming language.

Programming languages advocate different programming paradigms. Some languages are designed to support one particular paradigm, while other programming languages support multiple paradigms.

## Relationship between programming paradigms and programming languages:

The relationship between programming paradigms and programming languages can be complex since a programming language can support multiple paradigms. For example, C++ is designed to support elements of procedural programming, object-oriented programming and generic programming. However, designers and programmers decide how to build a program using those paradigm elements. One can write a purely procedural program in C++, or a purely object-oriented program, or one can write a program that contains elements of both paradigms.

There are various programming paradigms. The four main paradigms are explained here:

1. **Imperative paradigm:** The word 'imperative' means a command or an order. The 'first do this, next do that' is a short phrase which really in a nutshell describes the spirit of the imperative paradigm. The phrase also reflects that the order to the commands is important.
2. **Functional paradigm:** Functional programming originates from a purely mathematical discipline: the theory of functions. So, it is a simpler and cleaner programming paradigm. In a nutshell, functional paradigm can be expressed by a phrase: 'Evaluate an expression and use the resulting value for something'.

3. **Logic paradigm:** The logic paradigm fits extremely well when applied in problem domains that deal with the extraction of knowledge from basic facts and relations. The logical paradigm seems less natural in the more general areas of computation. In a nutshell, logic paradigm can be expressed by a phrase: ‘Answer a question via search for a solution’.
4. **Object-oriented paradigm:** The object-oriented paradigm has gained great popularity in the recent decade. The primary and most direct reason is the strong support of encapsulation and the logical grouping of program aspects. These properties are very important when programs become larger and larger. An object-oriented program is constructed with the outset in concepts, which are important in the problem domain of interest. In a nutshell, object-oriented paradigm can be expressed by a phrase: ‘Send messages between objects to simulate the temporal evolution of a set of real world phenomena’.

The other paradigms are also possible like Language-oriented programming, ARS based programming, and Grammar-oriented programming

### Language-Oriented Programming:

Language oriented programming is a fundamental style of computer programming, via metaprogramming in which, rather than solving problems in general-purpose programming languages, the programmer creates one or more domain-specific programming languages for the problem first, and solves the problem in those languages. The idea of LOP is to make it easy to

create special domain-specific languages, and those DSLs will make writing the programs easier.

Programmers using the mainstream programming are restricted because they are heavily dependent on programming infrastructure which they cannot easily change.

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 but general-purpose languages tend to be unproductive. Alternatively, we could use domain-specific languages (DSLs), which are tailored to be highly productive in a specific problem domain, such as SQL for writing database queries.

Extension to a language or some extra power from IDE can be achieved by waiting for the language designer to update it, or for the IDE vendor to add the new features. The programmer can write his own compiler or IDE also. But this takes a lot of time and effort and is simply not practical for most programmers. LOP gives the way to create, reuse, and modify languages and environments, in an easy way.

Today’s mainstream programming goes something like this:

**Think:** There is a task to program, so one forms a conceptual model in one’s head about how to solve the problem.

**Choose:** Some general-purpose language (such as Java or C++) is chosen for writing the solution.

**Program:** Writing the solution by performing a difficult mapping of the conceptual model into the programming language.

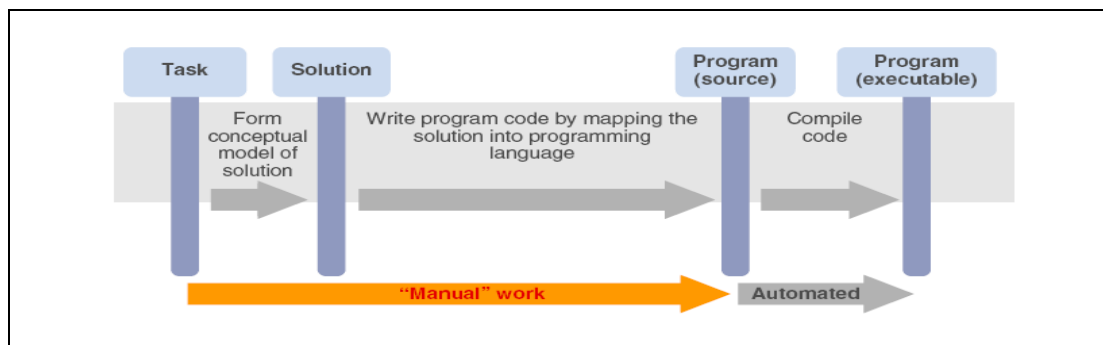


Figure 1: Mainstream programming with a general-purpose language.

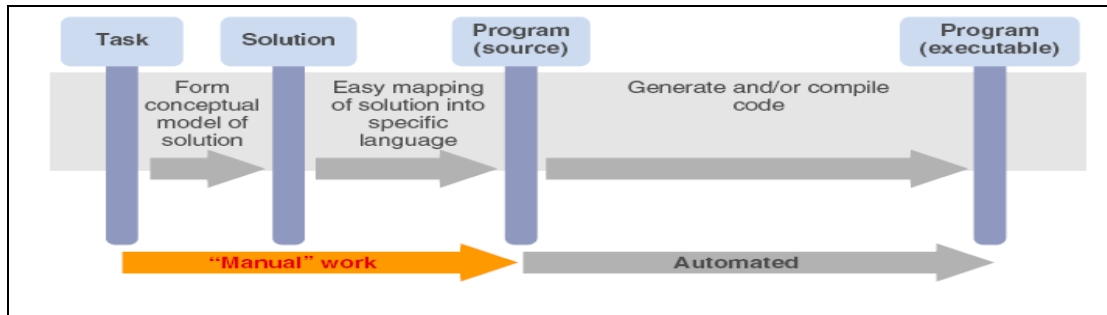


Figure 2: Language-oriented programming with domain-specific languages.

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:** There is a task to program, so one forms a conceptual model in one's head about how to solve the problem.

**Choose:** Some specialized DSLs are chosen for writing the solution.

**Create:** If there are no appropriate DSLs for the problem, then create the appropriate DSL that fits the problem.

**Program:** Writing the solution by performing a relatively straightforward mapping of the 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.

### Problems in Mainstream Programming

There are many problems with Mainstream Programming, and most of them stem from the fact that there is no way for a general-purpose language to fully support arbitrary domains. Arbitrary Domains help in easily creating new languages. Following are the three major problems with mainstream programming that will be solved by LOP:

#### 1. Time Delay to Implement Ideas

There is a very long gap between when formulating the solution of a problem mentally and when this solution is successfully communicated to the computer as a program. The reason is that, for the computer, one must use a general-purpose programming language which is much less expressive. We must express every single step and every detail. In mainstream programming, most of

the time spent on 'programming' is really just finding ways to express natural language concepts in terms of programming level abstractions. 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.

#### 2. Understanding and Maintaining Existing Code

Whether the existing code is written by another programmer or by the same programmer, programmer has to reverse engineer the program to understand what he originally intended. Because general-purpose languages require the translation of high level domain concepts into low-level programming features. Basically, programmer must mentally reconstruct the information that was lost in the original translation to the general purpose 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. But this has proven to be quite a weak solution for a number of reasons, including the cost of writing such auxiliary documentation, and the tendency of documentation to grow out-of-synch with code. Ideally, the code should be self-documenting. One should read the code itself to understand the code, not some comments or external documentation.

#### 3. 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 introduce extra complications (such as the runtime behavior of a class) to complete the mapping. Common examples are graphical user interface libraries and database libraries.

Learning such libraries is a complicated task, even for an expert in the domain. Since there is no direct

mapping from domain to language, we must learn this mapping. This presents a steep learning curve, but learning this takes a lot of time. 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 in the use the library correctly.

## Details of LOP

### Program in LOP

Programs in LOP are not restricted to mean the typical 'set-of-instructions'. A program in LOP, 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 means that programmer need not explain the solution in terms of a programming language—it could be in almost any form. For example, the layout of a GUI form can be explained by just drawing the form. If this drawing has enough detail, then the drawing itself represents the solution.

So Language Oriented Programming will not just be writing programs, but also creating the languages in which to write our programs. In LOP, programs are written closer to the problem domain instead of in the computer's set-of instructions domain, and so they will be much easier to write.

### Programs and Text

A program is stored as text, i.e. a stream of characters. And, there are countless tools for editing, displaying, and manipulating text. But a program's text is just one representation of the program. Forcing programs into text form have big drawbacks, the most important of which is that text-based programming languages are very difficult to extend. As features are added to the language, it becomes increasingly difficult to add new extensions without making the language ambiguous.

In order 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, because when a compiler compiles source code, it parses the text into a tree-like graph structure called an abstract syntax tree, consisting of brackets, braces and parentheses. This allows to make any extensions we like to the language. Text editors 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 can render the program as text, tables, diagrams, trees, etc.

Sometimes, we wouldn't even need to consider text storage at all. A good example of this today is an Excel spreadsheet. Most of people don't need to deal with the stored format, and there are import and export features.

### Language in LOP

In LOP, a language is defined by three main things: Structure, Editor, and Semantics.

Its structure defines its abstract syntax, the supported concepts and their possible arrangement. Its editor defines its concrete syntax, how it should be rendered and edited.

Its semantics define its behavior, the way to interpret it and/or the way to transform it into executable code.

Of course, languages can also have other aspects, such as constraints and type systems.

## Introduction to Meta Programming System

### Creating Languages in MPS

*“To create new languages easily: apply Language Oriented Programming to itself, i.e. perform a little self-referential bootstrapping.”*

The idea of LOP is to make it easy to create special domain-specific languages, and DSLs will make writing our programs easier. This is the real power of LOP.

In LOP, any unambiguous solution to some problem in some domain is a 'program'. So in the domain of 'creating new languages', 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. Some the languages-building languages are explained below:

#### (i) 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 means its textual grammar or its graph representation.

In most cases, while practicing LOP, we work with two 'levels' of programming:

1. **Meta level** – In this, language is defined
2. **Program level** – In this, program is written

When defining the structure of a new language, we use a language-structure DSL which resides 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 the relationships its instances can have and also the properties they will have. The language for describing this meta level language structure is called the 'Structure Language'.

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 we can define for the language.

#### (ii) Editor Language

Editor for the language acts as the interface for writing and manipulating concept models. Experience has shown that generic editors aren't as usable as we want them to be. In order to write models fast, we want specialized editors tailored to our language concepts. We can consider the editor as a part of the language, and the 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'.

Storing the programs as graphs and having special editors in LOP, doesn't imply editors to be diagram editors. Actually, 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 much better source of inspiration for Editor Language – the text editor only.

A well-formed program in any mainstream programming language could be composed into a set of 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. So the Editor Language helps to define the layout of cells for each concept in the language. We can define which parts are constant (like braces), and

which parts are variable. The Editor Language also helps to add powerful features to the new editors, like auto-complete, browsing, syntax highlighting, error highlighting, etc.

#### (iii) Transformation Language

The Structure Language and Editor Language together already provide some power. We can use them to write static documents. However, to make code executable, there are two main ways:

1. **Interpretation** – Interpretation is supported by DSLs to help define the way computer should interpret the program.
2. **Compilation** – Compilation is supported by DSLs to help define the way to generate executable code from our program.

#### MPS support for 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, the target format may be:

- (a) natively executable machine code or bytecode that runs in a virtual machine, or,
- (b) source code in a different language (e.g. Java or C++), and later use an existing compiler to turn that into executable code. We can even generate source code in some interpreted language, and use the existing interpreter to execute the code.

To avoid dealing with a wide variety of target formats, the approach is to do everything in MPS. So first, we 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 the target format is machine code, we define a target language in MPS that represent machine code, and if the target format is Java source code, we define a Java-like target language. The target language doesn't have to support all the features of the target format, as long as there is a simple, one-to-one mapping for all of the language features that are needed.

There are two phases to compilation:

- (a) a simple translation from the target language to the final result, and
- (b) a more complex transformation from the initial source language to the intermediate target language.

But if the source language and target language are different, this transformation becomes very complex. To define transformations easily, we need a model-transformation DSL. In MPS, this DSL is called the 'Transformation Language'.

To define model transformations, there are three main approaches to code generation which can be used together:

1. The first is an **iterative approach**, where we 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. For example, the iterative approach inspired the Model Query Language, which makes it easy to enumerate nodes and gather information from a concept model.
2. The second approach is to use **templates and macros** to define how to generate code in the target language.
3. 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 define transformations from one language to another.

#### (iv) **Templates**

Templates look like the target language, but allow to add macros in any part of the template. Macros are essentially bits of code that are executed when transformation are executed. The macros allow 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.

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

Template language can be thought of 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 code optimizers, etc.

### **Using Languages Together**

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. A new language can be created 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.

### **Platforms, Frameworks, Libraries, and Languages**

The system for supporting Language Oriented Programming needs more than just meta-programming capabilities to make it useful. LOP should also support all the reliable concepts of today’s programming languages such as Collections, user-interface, networking, database connectivity, etc. For example, programmer buys the entire Java platform, and not just the Java language. This means that programmers don’t choose languages solely based on the language itself, as much of the power of Java comes not only from the language, but from the hundreds of frameworks and APIs available for Java programmers to choose from. MPS also have a supporting ‘platform’ of its own.

‘Framework’ means a set of classes and methods packaged up into a class library. We need to reuse set of classes because some set of classes are useful for solving certain types of problems, like making GUIs, or accessing databases, etc.

A ‘class library’ corresponds to some domain. Any class library is a good candidate for creating a full-fledged DSL for the platform of the ‘language’. Three most important platform languages that will be provided with MPS are:

- (a) Base Language,
- (b) Collection Language, and
- (c) User Interface Language.

#### **(a) Base Language**

Firstly, a language for the simplest programming domain is required, which is a general-purpose imperative programming. This simple language would support 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.

For example, in order to add two numbers, programmer should be able to say ‘a + b’ as simple as that. 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:

1. It can be extended to create programmer’s own language based on it,
2. Its concepts may be referenced in programs, and

3. The code to the Base Language could be generated. There will be various generators available to transform the Base Language into other languages like Java, C++, etc.

#### **(b) Collection Language**

The next most important language required is a language for working with collections. Every major mainstream language has some sort of support for collections. For example, `java.util` in Java, `STL` in C++.

Everybody needs collections. So 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.

#### **(c) User Interface Language**

The Editor Language can possibly 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. Many environments today include GUI builders to simplify user-interface creation.

### **Conclusion**

The ideas underlying LOP and MPS are not new, and have actually been around for more than 20 years. The term 'Language Oriented Programming' itself has been around for at least 10 years. There are some doubtful responses to LOP, such as it is too risky to start a real-life project with an untested method like LOP, or, it is not yet ready for prime time.

It is true that MPS is currently not ready for the real world and there is a little documentation, but it is getting there. MPS and LOP need to be explored in more depth. We should remember that even OOP took 20 years to become mainstream. The ideas of LOP have silently saturated the software development community, and their time has finally come. It gives an easy way to create, reuse, and modify languages and environments.

Today, programmers are actually doing the work of designing and using little specialized languages cobbled together with classes and methods. It is time to begin the next technology revolution in software development. To start with, programmers can try just a little bit of LOP on the project to see if it provides a practical advantage, and then try a bit more if he like it. The concept of LOP is already being used to develop MPS itself. Language Oriented Programming can drastically improve software development.

### **References**

- [1] Donald E. Knuth. *Literate programming*. The Computer Journal, May 1984.
- [2] M. Ward. *Language Oriented Programming*. Software - Concepts and Tools
- [3] Charles Simonyi. *The Death of Computer Languages ,The Birth of Intentional Programming*.
- [4] John Brockman. *Intentional Programming*:
- [5] Microsoft Research. *Intentional Programming*.
- [6] Charles Simonyi. *Intentional Programming: Asymptotic Fun*
- [7] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley.
- [8] Jack Herrington. *Code Generation in Action*. Manning
- [9] Xactium. *Applied Metamodelling: A Foundation for Language Driven Development*
- [10] Matt Quail. *Totally Gridbag*.
- [11] Jack Herrington. *Code Generation Network*.
- [12] <http://www.xactium.com>
- [13] [http://codegeneration.net/tiki-read\\_article.php?articleId=60](http://codegeneration.net/tiki-read_article.php?articleId=60)
- [14] [http://codegeneration.net/tiki-read\\_article.php?articleId=61](http://codegeneration.net/tiki-read_article.php?articleId=61)
- [15] [http://codegeneration.net/tiki-read\\_article.php?articleId=64](http://codegeneration.net/tiki-read_article.php?articleId=64)
- [16] [http://codegeneration.net/tiki-read\\_article.php?articleId=68](http://codegeneration.net/tiki-read_article.php?articleId=68)