

# PROGRAMMING PARADIGMS

**Rupinder Kaur and Inderdeep Kaur**

Department Of Computer Science & Engineering  
Institute of Engineering & Technology, Bhaddal (Ropar) Punjab  
Mandeep Kaur(Department GGS college of Modern technology (Kharar)

Email: - [talktorupinder@yahoo.co.in](mailto:talktorupinder@yahoo.co.in), [kaur.inderdeep@gmail.com](mailto:kaur.inderdeep@gmail.com),  
[er.deepvasantia@tgmail.com](mailto:er.deepvasantia@tgmail.com)

*Abstract-A programming paradigm is a fundamental style of programming regarding how solutions to problems are to be formulated in a programming language. A programming paradigm provides (and determines) the view that the programmer has of the execution of the program. For instance, in object-oriented programming, programmers can think of a program as a collection of interacting objects, while in functional programming a program can be thought of as a sequence of stateless function evaluations. When programming computers or systems with many processors, process oriented programming allows programmers to think about applications as sets of concurrent processes acting upon logically shared data structures. Just as different groups in software engineering advocate different methodologies, different programming languages advocate different programming paradigms. Some languages are designed to support one particular paradigm while other programming languages support multiple paradigms.*

## 1. INTRODUCTION

Many programming paradigms are as well known for what techniques they forbid as for what they enable. For instance, pure functional programming disallows the use

of side-effects; structured programming disallows the use of goto. Partly for this reason, new paradigms are often regarded as doctrinaire or overly rigid by those accustomed to earlier styles. However, this avoidance of certain techniques can make it easier to prove theorems about a program's correctness—or simply to understand its behavior—without limiting the generality of the programming language.

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++, one can write a purely object-oriented program in C++, or one can write a program that contains elements of both paradigms. A programming paradigm is a fundamental style of Computer programming. (Compare with a methodology, which is a style of solving specific software engineering problems).

A programming language can support multiple paradigms. For example programs written in C++ can be purely

procedural, or purely object-oriented, or contain elements of both paradigms. Software designers and programmers decide how to use those paradigm elements.

## 2. Categories of Programming paradigms

The various categories of programming paradigms are:

- Aspect-oriented programming
- Functional programming
- Generic programming
- Logic programming
- Object-oriented programming
- Service-oriented programming

### 2.1. Aspect-oriented programming

In software engineering, the programming paradigms of aspect-oriented programming (AOP), and aspect-oriented software development (AOSD) attempt to aid programmers in the separation of concerns, specifically cross-cutting concerns, as an advance in modularization. AOP does so using primarily language changes, while AOSD uses a combination of language, environment, and method. Separation of concerns entails breaking down a program into distinct parts that overlap in functionality as little as possible. All programming methodologies—including procedural programming and object-oriented programming—support some separation and encapsulation of concerns (or any area of interest or focus) into single entities. For example, procedures, packages, classes, and methods all help programmers encapsulate concerns into single entities. But some concerns defy these forms of encapsulation. Software engineers call these crosscutting concerns, because they cut across multiple modules in a program.

#### Terminology

The following are some standard terminology used in Aspect-oriented programming:

**Cross-cutting concerns:** Even though most classes in an OO model will perform a single, specific function, they often share common, secondary requirements with other classes. For example, we may want to add logging to classes within the data-access layer and also to classes in the UI layer whenever a thread enters or exits a method. Even though the primary functionality of each class is very different, the code needed to perform the secondary functionality is often identical.

**Advice:** This is the additional code that you want to apply to your existing model. In our example, this is the logging code that we want to apply whenever the thread enters or exits a method.

**Point-cut:** This is the term given to the point of execution in the application at which cross-cutting concern needs to be applied. In our example, a point-cut is reached when the thread enters a method, and another point-cut is reached when the thread exits the method.

**Aspect:** The combination of the point-cut and the advice is termed an aspect. In the example below, we add a logging aspect to our application by defining a point-cut and giving the correct advice.

### 2.2 Functional programming

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It emphasizes the application of functions, in contrast with the imperative programming style that emphasizes changes in state. Functional languages include APL, Erlang, Haskell, Lisp, ML, F# and Scheme. Functional

programming languages, especially purely functional ones, have largely been emphasized in academia rather than in commercial software development. However, notable functional programming languages used in industry and commercial applications include Erlang (concurrent applications), R (statistics), Mathematical (symbolic math), ML, J and K (financial analysis), and domain-specific programming languages like XSLT. The lambda calculus provides the model for functional programming. Modern functional languages can be viewed as embellishments to the lambda calculus.

### **Higher-order functions**

Functions are higher-order when they can take other functions as arguments, and return them as results. (The derivative and ant derivative in calculus are examples of this.) Higher-order functions are closely related to first-class functions, in those higher-order functions and first-class functions both allow functions as arguments and results of other functions. The distinction between the two is subtle: "higher-order" describes a mathematical concept of functions that operate on other functions, while "first-class" is a computer science term that describes programming language entities that have no restriction on their use.

### **Pure functions**

Purely functional programs have no side effects. This makes it easier to reason about their behavior. However, almost no programmers bother to write purely functional programs, since, by definition, a program with no side effects (one that accepts no input, produces no output, and interfaces with no external devices) is formally equivalent to a program that does

nothing; typically, purity is used to enforce a separation of concerns where one clearly-delineated section of the program does impure operations like I/O, and calls pure functions and libraries as needed to compute answer.

### **Recursion**

Iteration (looping) in functional languages is usually accomplished via recursion. Recursive functions invoke themselves, allowing an operation to be performed over and over. Recursion may require maintaining a stack, but tail recursion can be recognized and optimized by a compiler into the same code used to implement iteration in imperative languages. The Scheme programming language standard requires implementations to recognize and optimize tail recursion.

## **2.3 Generic Programming**

Generic programming is a programming paradigm for developing efficient, reusable software libraries. The Generic Programming process focuses on finding commonality among similar implementations of the same algorithm, then providing suitable abstractions so that a single, generic algorithm can cover many concrete implementations. This process, called lifting, is repeated until the generic algorithm has reached a suitable level of abstraction, where it provides maximal reusability while still yielding efficient, concrete implementations. The abstractions themselves are expressed as requirements on the parameters to the generic algorithm.

Once many algorithms within a given problem domain have been lifted, we start to see patterns among the requirements. It is common for the same set of

requirements to be required by several different algorithms. When this occurs, each set of requirements is bundled into a concept. Concepts describe a set of abstractions, each of which meets all of the requirements of a concept. When the Generic Programming process is carefully followed, the concepts that emerge tend to describe the abstractions within the problem domain in some logical way. A study of graph algorithms will produce Graph concepts that describe the behavior of graphs, whereas a study of linear algebra algorithms will produce Matrix and Vector concepts. Thus, the output of the Generic Programming process is not just a generic, reusable implementation, but a better understanding of the problem domain.

#### **2.4 Logic programming**

logic programming is, in its broadest sense, the use of mathematical logic for computer programming. In this view of logic programming, logic is used as a purely declarative representation language, and a theorem-prover or model-generator is used as the problem-solver. The problem-solving task is split between the programmers, who is responsible only for ensuring the truth of programs expressed in logical form, and the theorem-prover or model-generator, which is responsible for solving problems efficiently. However, logic programming, in the narrower sense in which it is more commonly understood, is the use of logic as both a declarative and procedural representation language.

In the simplified, propositional case in which a logic program and a top-level atomic goal contain no variables, backward reasoning determines an and-or tree, which constitutes the search space for solving the goal. The top-level goal is the root of the tree. Given any node in the tree and any clause whose head matches the

node, there exists a set of child nodes corresponding to the sub-goals in the body of the clause. These child nodes are grouped together by an "and". The alternative sets of children corresponding to alternative ways of solving the node are grouped together by an "or".

#### **Abdicative logic programming**

Abdicative Logic Programming is an extension of normal Logic Programming that allows some predicates, declared as abducible predicates, to be incompletely defined. Problem solving is achieved by deriving hypotheses expressed in terms of the abducible predicates as solutions of problems to be solved. These problems can be either observations that need to be explained (as in classical abductive reasoning) or goals to be achieved (as in normal logic programming). It has been used to solve problems in Diagnosis, Planning, Natural Language and Machine Learning. It has also been used to interpret Negation as Failure as a form of abductive reasoning.

#### **Higher-order logic programming**

Several researchers have extended logic programming with higher-order programming features derived from higher-order logic, such as predicate variables. Such languages include the Prolog extensions HiLog and  $\lambda$ Prolog

#### **2.5 Object-oriented programming**

Object-oriented programming (OOP) is a programming paradigm that uses "objects" and their interactions to design applications and computer programs. It is based on several techniques, including encapsulation, modularity, polymorphism, and inheritance. It was not commonly used in mainstream software application

development until the early 1990s. Many modern programming languages now support OOP.

The Simula programming language was the first to introduce the concepts underlying object-oriented programming (objects, classes, subclasses, virtual methods, coroutines, garbage collection, and discrete event simulation) as a superset of Algol. Simula was used for physical modeling, such as models to study and improve the movement of ships and their content through cargo ports. Smalltalk was the first programming language to be called "object-oriented". Object-oriented programming may be seen as a collection of cooperating objects, as opposed to a traditional view in which a program may be seen as a group of tasks to the computer ("subroutines"). In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent little machine with a distinct role or responsibility. The actions or "operators" on the objects are closely associated with the object. For example, in OOP, the data structures tend to carry their own operators around with them (or at least "inherit" them from a similar object or "class"). The traditional approach tends to view and consider data and behavior separately.

### **Fundamental concepts**

**Class:** A class defines the abstract characteristics of a thing (object), including the thing's characteristics (its attributes, fields or properties) and the thing's behaviors (the things it can do, or methods, operations or features). Classes provide modularity and structure in an object-oriented computer program. A class should typically be recognizable to a non-

programmer familiar with the problem domain, meaning that the characteristics of the class should make sense in context. Also, the code for a class should be relatively self-contained (generally using encapsulation). Collectively, the properties and methods defined by a class are called members.

**Object:** A particular instance of a class. The set of values of the attributes of a particular

Object is called its state. The object consists of state and the behaviors that's defined in the object's class.

Message passing :“The process by which an object sends data to another object or asks the other object to invoke a method.” On Java code level message passing corresponds to "method calling".

**Inheritance:** ‘Subclasses’ are more specialized versions of a class, which inherit attributes and behaviors from their parent classes, and can introduce their own.

Multiple inheritances is inheritance from more than one ancestor class, neither of these ancestors being an ancestor of the other.

**Encapsulation:** Encapsulation conceals the functional details of a class from objects that send messages to it. Members are often specified as public, protected or private, determining whether they are available to all classes, sub-classes or only the defining class. Some languages go further: Java uses the default access modifier to restrict access also to classes in the same package, C# and VB.NET reserve some members to classes in the same assembly using keywords internal (C#) or Friend (VB.NET), and Eiffel and C++ allows one to specify which classes may access any member.

**Abstraction:** Abstraction is simplifying complex reality by modelling classes appropriate to the problem, and working at

the most appropriate level of inheritance for a given aspect of the problem.

Polymorphism: Polymorphism allows you to treat derived class members just like their parent class' members. More precisely, Polymorphism in object-oriented programming is the ability of objects belonging to different data types to respond to method calls of methods of the same name, each one according to an appropriate type-specific behavior.

## 2.6 Service-orientation

Service-orientation is a design paradigm that specifies the creation of automation logic in the form of services. It is applied as a strategic goal in developing a service-oriented architecture (SOA). Like other design paradigms, service-orientation provides a means of achieving a separation of concerns. It is commonly acknowledged that several service-orientation principles have their roots in the object-oriented design paradigm. Some have claimed that service-orientation will ultimately replace object-orientation as the de facto design paradigm, while others state that the two are complementary paradigms and that there will always be a need for both.

Services inherit a number of features of software components, including:

could use domain-specific languages (DSL's), which are tailored to be highly productive in a specific problem domain, such as SQL for writing database queries. The strength of DSL's, domain specificity, is also their weakness, since any real-world program will involve many different domains.

In mainstream programming, most of the time spent "programming" is really just finding ways to express natural language concepts in terms of programming level

- Multiple-use
  - Non-context-specific
  - Composable
  - Encapsulated
- i.e., non-investigable through its interfaces
- A unit of independent deployment and versioning

## 3. Language Oriented Programming: The Next Programming Paradigm

It is time to begin the next technology revolution in software development, and the shape of this revolution is becoming more and clearer. It is not yet fully formed, different parts have different names: Intentional programming, MDA, generative programming, etc.

Programmers today have very restricted freedom. Programmers are restricted because they are heavily dependent on programming infrastructure which they cannot easily change, namely the languages and environments that they use. 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. Alternatively, we abstractions, which is difficult, not very creative, and more or less a waste of time.

A program in LOP is not a set of instructions. 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-of-instructions

domain, and so they will be much easier to write.

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. The idea of LOP is to make it easy to create special domain-specific languages, and those DSL's will make writing our programs easier. Language Oriented Programming can drastically improve software development.

### ***Language Oriented Programming: The Next Programming Paradigm***

#### **Understanding and Maintaining Existing Code**

The next problem we have is in understanding and maintaining existing code. Whether it is written by another programmer or by us, the problem is the same. Because general-purpose languages require me to translate high-level 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 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. 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 also an 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.

#### **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 combo-box should have a list of some data in it*

which will solve the problem. We don't need to explain the solution in terms of a programming language—it could be in almost any form. To explain how to lay out a GUI form, we 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.

#### **4. Comparison of all programming paradigms**

As with all immature technologies, widespread adoption of AOP is hindered by a lack of tool support, and widespread education. Some argue that slowing down is appropriate due to AOP's inherent ability to create unpredictable

and widespread errors in a system. Implementation issues of some AOP languages mean that something as simple as renaming a function can lead to an aspect no longer being applied leading to negative side effects.

Programmers need to be able to read code and understand what's happening in order to prevent errors<sup>1</sup>. While they have grown accustomed to ignoring the details of method dispatch or container-supplied behaviors, many are uncomfortable with the idea that an aspect can be injected later adding behavior to their code. There are also valid security questions that code weaving raises.

Functional programming is very different from imperative programming. The most significant differences stem from the fact that functional programming avoids side effects, which are used in imperative programming to implement state and I/O. Pure functional programming disallows side effects completely. Disallowing side effects provides for referential transparency, which makes it easier to verify, optimize, and parallelize programs, and easier to write automated tools to perform those tasks. Aspects emerged out of object-oriented programming and computational reflection. AOP languages have functionality similar to, but more restricted than met object protocols. Aspects relate closely to programming concepts like subjects, mixins, and delegation. Using AOP judiciously to develop your own code can result in powerful succinct expressiveness. Using AOP to add to code written by someone else (especially when you don't have the source code) is risky.

## Conclusion

The ideas underlying programming paradigm are not new, and have actually been around for more than 20 What is new is that these ideas have silently saturated the software development community, and their time has finally come. With this article, we hope to provide a seed around which these ideas can crystallize into new discussions, opinions, critiques, experiments, research, and real-life projects.