Programming paradigm

From Wikipedia, the free encyclopedia

Programming paradigms are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms.

Some paradigms are concerned mainly with implications for the execution model of the language, such as allowing side effects, or whether the sequence of operations is defined by the execution model. Other paradigms are concerned mainly with the way that code is organized, such as grouping a code into units along with the state that is modified by the code. Yet others are concerned mainly with the style of syntax and grammar.

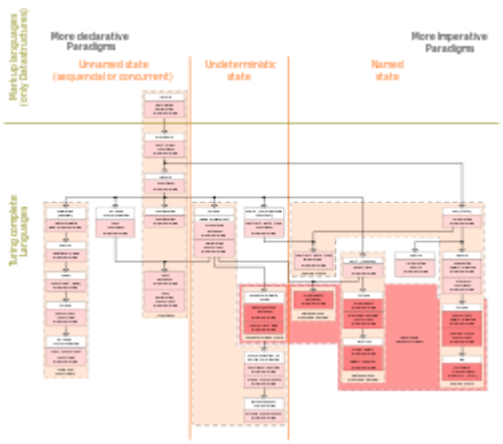
Common programming paradigms include:

- imperative which allows side effects,
- functional which disallows side effects,
- declarative which does not state the order in which operations execute,
- object-oriented which groups code together with the state the code modifies,
- procedural which groups code into functions,
- logic which has a particular style of execution model coupled to a particular style of syntax and grammar, and
- symbolic programming which has a particular style of syntax and grammar.[1][2][3]

For example, languages that fall into the **imperative paradigm** have two main features: they state the order in which operations occur, with constructs that explicitly control that order, and they allow side effects, in which state can be modified at one point in time, within one unit of code, and then later read at a different point in time inside a different unit of code. The communication between the units of code is not explicit. Meanwhile, in **object-oriented** programming, code is organized into objects that contain state that is only modified by the code that is part of the object. Most object-oriented languages are also imperative languages. In contrast, languages that fit the **declarative paradigm** do not state the order in which to execute operations. Instead, they supply a number of operations that are available in the system, along with the conditions under which each is allowed to execute. The implementation of the language's execution model tracks which operations are free to execute and chooses the order on its own. More at Comparison of multi-paradigm programming languages.

Contents

Overview



Overview of the various programming paradigms according to Peter Van Roy[4]:5

Just as software engineering (as a process) is defined by differing *methodologies*, so the programming languages (as models of computation) are defined by differing *paradigms*. Some languages are designed to support one paradigm (Smalltalk supports object-oriented programming, Haskell supports functional programming), while other programming languages support multiple paradigms (such as Object Pascal, C++, Java, C#, Scala, Visual Basic, Common Lisp, Scheme, Perl, PHP, Python, Ruby, Oz, and F#). For example, programs written in C++, Object Pascal or PHP can be purely procedural, purely object-oriented, or can contain elements of both or other paradigms. Software designers and programmers decide how to use those paradigm elements.

In object-oriented programming, programs are treated as a set of interacting objects. In functional programming, programs are treated as a sequence of stateless function evaluations. When programming computers or systems with many processors, in process-oriented programming, programs are treated as sets of concurrent processes acting on logically shared data structures.

Many programming paradigms are as well known for the techniques they *forbid* as for those they *enable*. For instance, pure functional programming disallows use of side-effects, while structured programming disallows use of the goto statement. Partly for this reason, new paradigms are often regarded as doctrinaire or overly rigid by those accustomed to earlier styles.[5] Yet, avoiding certain techniques can make it easier to understand program behavior, and to prove theorems about program correctness.

Programming paradigms can also be compared with *programming models* which allow invoking an execution model by using only an API. Programming models can also be classified into paradigms, based on features of the execution model.

For parallel computing, using a programming model instead of a language is common. The reason is that details of the parallel hardware leak into the abstractions used to program the hardware. This causes the programmer to have to map patterns in the algorithm onto patterns in the execution model (which have been inserted due to leakage of hardware into the abstraction). As a consequence, no one parallel programming language maps well to all computation problems. It is thus more convenient to use a base sequential language and insert API calls to parallel execution models, via a programming model. Such parallel programming models can be classified according to abstractions that reflect the hardware, such as shared memory, distributed memory with message passing, notions of *place* visible in the code, and so forth. These can be considered flavors of programming paradigm that apply to only parallel languages and programming models.

History

Different approaches to programming have developed over time, being identified as such either at the time or retrospectively. An early approach consciously identified as such is structured programming, advocated since the mid 1960s. The concept of a "programming paradigm" as such dates at least to 1978, in the Turing Award lecture of Robert W. Floyd, entitled *The Paradigms of Programming*, which cites the notion of paradigm as used by Thomas Kuhn in his *The Structure of Scientific Revolutions* (1962).[6]

Machine code

The lowest-level programming paradigms are machine code, which directly represents the instructions (the contents of program memory) as a sequence of numbers, and assembly language where the machine instructions are represented by mnemonics and memory addresses can be given symbolic labels. These are sometimes called first- and second-generation languages.

In the 1960s, assembly languages were developed to support library COPY and quite sophisticated conditional macro generation and preprocessing abilities, CALL to (subroutines), external variables and common sections (globals), enabling significant code re-use and isolation from hardware specifics via use of logical operators such as READ/WRITE/GET/PUT. Assembly was, and still is, used for time critical systems and often in embedded systems as it gives the most direct control of what the machine does.

Procedural languages

The next advance was the development of procedural languages. These third-generation languages (the first described as high-level languages) use vocabulary related to the problem being solved. For example,

- COmmon Business Oriented Language (COBOL) – uses terms like file, move and copy.
- FORmula TRANslation (FORTRAN) – using mathematical language terminology, it was developed mainly for scientific and engineering problems.
- ALGOrithmic Language (ALGOL) – focused on being an appropriate language to define algorithms, while using mathematical language terminology and targeting scientific and engineering problems just like FORTRAN.
- Programming Language One (PL/I) – a hybrid commercial-scientific general purpose language supporting pointers.
- Beginners All purpose Symbolic Instruction Code (BASIC) – it was developed to enable more people to write programs.
- C – a general-purpose programming language, initially developed by Dennis Ritchie between 1969 and 1973 at AT&T Bell Labs.

All these languages follow the procedural paradigm. That is, they describe, step by step, exactly the procedure that should, according to the particular programmer at least, be followed to solve a specific problem. The efficacy and efficiency of any such solution are both therefore entirely subjective and highly dependent on that programmer's experience, inventiveness, and ability.

Object-oriented programming

Following the widespread use of procedural languages, object-oriented programming (OOP) languages were created, such as Simula, Smalltalk, C++, C#, Eiffel, PHP, and Java. In these languages, data and methods to manipulate it are kept as one unit called an object. The only way that another object or user can access the data is via the object's *methods*. Thus, the inner workings of an object may be changed without affecting any code that uses the object. There is still some controversy raised by Alexander Stepanov, Richard Stallman[7] and other programmers, concerning the efficacy of the OOP paradigm versus the procedural paradigm. The need for every object to have associative methods leads some skeptics to associate OOP with software bloat; an attempt to resolve this dilemma came through polymorphism.

Because object-oriented programming is considered a paradigm, not a language, it is possible to create even an object-oriented assembler language. High Level Assembly (HLA) is an example of this that fully supports advanced data types and object-oriented assembly language programming – despite its early origins. Thus, differing programming paradigms can be seen rather like *motivational memes* of their advocates, rather than necessarily representing progress from one level to the next. Precise comparisons of the efficacy of competing paradigms are frequently made more difficult because of new and differing terminology applied to similar entities and processes together with numerous implementation distinctions across languages.

Further paradigms

Literate programming, as a form of imperative programming, structures programs as a human-centered web, as in a hypertext essay: documentation is integral to the program, and the program is structured following the logic of prose exposition, rather than compiler convenience.

Independent of the imperative branch, declarative programming paradigms were developed. In these languages, the computer is told what the problem is, not how to solve the problem – the program is structured as a set of properties to find in the expected result, not as a procedure to follow. Given a database or a set of rules, the computer tries to find a solution matching all the desired properties. An archetype of a declarative language is the fourth generation language SQL, and the family of functional languages and logic programming.

Functional programming is a subset of declarative programming. Programs written using this paradigm use functions, blocks of code intended to behave like mathematical functions. Functional languages discourage changes in the value of variables through assignment, making a great deal of use of recursion instead.

The logic programming paradigm views computation as automated reasoning over a body of knowledge. Facts about the problem domain are expressed as logic formulae, and programs are executed by applying inference rules over them until an answer to the problem is found, or the set of formulae is proved inconsistent.

Symbolic programming is a paradigm that describes programs able to manipulate formulas and program components as data.[3] Programs can thus effectively modify themselves, and appear to "learn", making them suited for applications such as artificial intelligence, expert systems, natural language processing and computer games. Languages that support this paradigm include Lisp and Prolog.[8]

Multi-paradigm

A *multi-paradigm programming language* is a programming language that supports more than one programming paradigm.[9] The design goal of such languages is to allow programmers to use the most suitable programming style and associated language constructs for a given job, considering that no single paradigm solves all problems in the easiest or most efficient way.

One example is C#, which includes imperative and object-oriented paradigms, together with a certain level of support for functional programming with features like delegates (allowing functions to be treated as first-order objects), type inference, anonymous functions and Language Integrated Query. Other examples are F# and Scala, which provide similar functionality to C# but also include full support for functional programming (including currying, pattern matching, algebraic data types, lazy evaluation, tail recursion, immutability, etc.). Perhaps the most extreme example is Oz, which has subsets that adhere to logic (Oz descends from logic programming), functional, object-oriented, dataflow concurrent, and other paradigms. Oz was designed over a ten-year period to combine in a harmonious way concepts that are traditionally associated with different programming paradigms. Lisp, while often taught as a functional language, is known for its malleability and thus its ability to engulf many paradigms.

See also

- Architecture description language
- Comparison of programming languages
- Comparison of programming paradigms
- Domain-specific language
- Mindset
- Modeling language
- Programming domain
- Type system
- Turing completeness
- Von Neumann programming languages

References

1. Nørmark, Kurt. *Overview of the four main programming paradigms (http://people.cs.aau.dk/~normark/prog 3-03/html/notes/paradigms_themes-paradigm-overview-section.html)*. Aalborg University, 9 May 2011. Retrieved 22 September 2012.
2. Frans Coenen (1999-10-11). "Characteristics of declarative programming languages" (http://cgi.csc.liv.ac.uk/~frans/OldLectures/2CS24/declarative.html#detail). *cgi.csc.liv.ac.uk*. Retrieved 2014-02-20.
3. Michael A. Covington (2010-08-23). "CSCI/ARTI 4540/6540: First Lecture on Symbolic Programming and LISP" (http://www.ai.uga.edu/mc/LispNotes/FirstLectureOnSymbolicProgramming.pdf) (PDF). University of Georgia. Retrieved 2013-11-20.
4. Peter Van Roy (2009-05-12). "Programming Paradigms for Dummies: What Every Programmer Should Know" (http://www.info.ucl.ac.be/~pvr/VanRoyChapter.pdf) (PDF). info.ucl.ac.be. Retrieved 2014-01-27.
5. Frank Rubin (March 1987). " 'GOTO Considered Harmful' Considered Harmful" (https://web.archive.org/web/20090320002214/http://www.ecn.purdue.edu/ParaMount/papers/rubin87goto.pdf) (PDF). *Communications of the ACM*. **30** (3): 195–196. doi:10.1145/214748.315722 (https://doi.org/10.1145%2F214748.315722). Archived from the original (http://www.ecn.purdue.edu/ParaMount/papers/rubin87goto.pdf) (PDF) on March 20, 2009.
6. Floyd, R. W. (1979). "The paradigms of programming" (http://dl.acm.org/ft_gateway.cfm?id=359140&ftid=289772&dwn=1&CFID=285645736&CFTOKEN=55009136). *Communications of the ACM*. **22** (8): 455. doi:10.1145/359138.359140 (https://doi.org/10.1145%2F359138.359140).
7. "Mode inheritance, cloning, hooks & OOP (Google Groups Discussion)" (http://groups.google.com/group/comp.emacs.xemacs/browse_thread/thread/d0af257a2837640c/37f251537fafbb03?lnk=st&q=%22Richard+Stallman%22+oop&rnum=5&hl=en#37f251537fafbb03).
8. "Business glossary: Symbolic programming definition" (http://www.allbusiness.com/glossaries/symbolic-programming/4950308-1.html). *allbusiness.com*. Retrieved 2014-07-30.
9. "Multi-Paradigm Programming Language" (https://developer.mozilla.org/en-US/docs/multiparadigmlanguage.html). developer.mozilla.org. Retrieved 21 October 2013.

External links

- Classification of the principal programming paradigms (http://www.info.ucl.ac.be/~pvr/paradigms.html)
- How programming paradigms evolve and get adopted? (http://www.janeve.me/articles/understanding-programming-paradigms)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Programming_paradigm&oldid=799268854"
