Comparison of programming paradigms

From Wikipedia, the free encyclopedia

This article attempts to set out the various similarities and differences between the various programming paradigms as a summary in both graphical and tabular format with links to the separate discussions concerning these similarities and differences in extant Wikipedia articles.

Contents

Main paradigm approaches

The following are widely considered the main programming paradigms, as seen when measuring programming language popularity. There is some overlap between paradigms, inevitably, but the main features or identifiable differences are summarized in this table:

- Imperative programming – defines computation as statements that change a program state.
- Procedural programming, structured programming – specifies the steps a program must take to reach a desired state.
- Declarative programming – defines program logic, but not detailed control flow.
- Functional programming – treats programs as evaluating mathematical functions and avoids state and mutable data
- Object-oriented programming (OOP) – organizes programs as *objects*: data structures consisting of datafields and methods together with their interactions.
- Event-driven programming – program control flow is determined by events, such as sensor inputs or user actions (mouse clicks, key presses) or messages from other programs or threads.
- Automata-based programming – a program, or part, is treated as a model of a finite state machine or any other formal automaton.

None of the main programming paradigms have a precise, globally unanimous definition, nor official international standard. Nor is there any agreement on which paradigm constitutes the best method to developing software. The subroutines that implement OOP methods may be ultimately coded in an imperative, functional, or procedural style that may, or may not, directly alter state on behalf of the invoking program.

Paradigm	Description	Main traits	Related paradigm(s)	Critique	Examples
Imperative	Programs as statements that *directly* change computed state (datafields)	Direct assignments, common data structures, global variables		Edsger W. Dijkstra, Michael A. Jackson	C, C++, Java, PHP, Python, Ruby
Structured	A style of imperative programming with more logical program structure	Structograms, indentation, no or limited use of goto statements	Imperative		C, C++, Java, Python
Procedural	Derived from structured programming, based on the concept of modular programming or the *procedure call*	Local variables, sequence, selection, iteration, and modularization	Structured, imperative		C, C++, Lisp, PHP, Python
Functional	Treats computation as the evaluation of mathematical functions avoiding state and mutable data	Lambda calculus, compositionality, formula, recursion, referential transparency, no side effects	Declarative		C++,[1] Clojure, Coffeescript,[2] Elixir, Erlang, F#, Haskell, Lisp, Python, Ruby, Scala, SequenceL, Standard ML
Event-driven including time-driven	Control flow is determined mainly by events, such as mouse clicks or interrupts including timer	Main loop, event handlers, asynchronous processes	Procedural, dataflow		JavaScript, ActionScript, Visual Basic, Elm
Object-oriented	Treats datafields as *objects* manipulated through predefined methods only	Objects, methods, message passing, information hiding, data abstraction, encapsulation, polymorphism, inheritance, serialization-marshalling	Procedural	Here and[3][4][5]	Common Lisp, C++, C#, Eiffel, Java, PHP, Python, Ruby, Scala

Paradigm	Description	Main traits	Related paradigm(s)	Critique	Examples
Declarative	Defines program logic, but not detailed control flow	Fourth-generation languages, spreadsheets, report program generators			SQL, regular expressions, CSS, Prolog, OWL, SPARQL
Automata-based programming	Treats programs as a model of a finite state machine or any other formal automata	State enumeration, control variable, state changes, isomorphism, state transition table	Imperative, event-driven		Abstract State Machine Language

Differences in terminology

Despite multiple (types of) programming paradigms existing in parallel (with sometimes apparently conflicting definitions), many of the underlying *fundamental components* remain more or less the same (constants, variables, datafields, subroutines, calls etc.) and must somehow thus inevitably be incorporated into each separate paradigm with equally similar attributes or functions. The table above is not intended as a guide to precise similarities, but more of an index of where to look for more information, based on the different naming of these entities, within each paradigm. Further complicating matters are non-standardized implementations of each paradigm, in many programming languages, especially languages supporting multiple paradigms, each with its own jargon.

Language support

Syntactic sugar is the *sweetening* of program functionality by introducing language features that facilitate a given usage, even if the end result could be achieved without them. One example of syntactic sugar may arguably be the classes used in object-oriented programming languages. The imperative language C can support object-oriented programming via its facilities of function pointers, type casting, and structures. However, languages such as C++ aim to make object-oriented programming more convenient by introducing syntax specific to this coding style. Moreover, the specialized syntax works to emphasize the object-oriented approach. Similarly, functions and looping syntax in C (and other procedural and structured programming languages) could be considered syntactic sugar. Assembly language can support procedural or structured programming via its facilities for modifying register values and branching execution depending on program state. However, languages such as C introduced syntax specific to these coding styles to make procedural and structured programming more convenient. Features of the language C# (C Sharp), such as properties and interfaces, similarly enable no new functions, but are designed to make good programming practices more prominent and natural.

Some programmers feel that these features are unimportant or even frivolous. For example, Alan Perlis once quipped, in a reference to bracket-delimited languages, that "syntactic sugar causes cancer of the semicolon" (see Epigrams on Programming).

An extension of this is the syntactic saccharin, or gratuitous syntax that does not make programming easier.[6]

Performance comparison

In total instruction path length only, a program coded in an imperative style, using no subroutines, would have the lowest count. However, the binary size of such a program may be larger than the same program coded using subroutines (as in functional and procedural programming) and would reference more *non-local physical* instructions that may increase cache misses and instruction fetch overhead in modern processors.

The paradigms that use subroutines extensively (including functional, procedural, and object-oriented) and do not also use significant Inline expansion (inlining, via compiler optimizations) will, consequently, use a larger fraction of total resources on the subroutine linkages. Object-oriented programs that do not deliberately alter program state directly, instead using mutator methods (or *setters*) to encapsulate these state changes, will, as a direct consequence, have more overhead. This is because message passing is essentially a subroutine call, but with three added overheads: dynamic memory allocation, parameter copying, and dynamic dispatch. Obtaining memory from the heap and copying parameters for message passing may involve significant resources that far exceed those needed for the state change. Accessors (or *getters*) that merely return the values of private member variables also depend on similar message passing subroutines, instead of using a more direct assignment (or comparison), adding to total path length.

Managed code

For programs executing in a *managed code* environment, such as the .NET Framework, many issues affect performance that are significantly affected by the programming language paradigm and various language features used.[7]

Pseudocode examples comparing various paradigms

A pseudocode comparison of imperative, procedural, and object oriented approaches used to calculate the area of a circle (πr^2), assuming no subroutine inlining, no macro preprocessors, register arithmetic, and weighting each instruction 'step' as only 1 instruction – as a crude measure of instruction path length – is presented below. The instruction step that is conceptually performing the state change is highlighted in bold typeface in each case. The arithmetic operations used to compute the area of the circle are the same in all three paradigms, with the difference being that the procedural and object-oriented paradigms wrap those operations in a subroutine call that makes the computation general and reusable. The same effect could be achieved in a purely imperative program using a macro preprocessor at only the cost of increased program size (only at each macro invocation site) without a corresponding pro rata runtime cost (proportional to *n* invocations – that may be situated within an inner loop for instance). Conversely, subroutine inlining by a compiler could reduce procedural programs to something similar in size to the purely imperative code. However, for object-oriented programs, even with inlining, messages still must be built (from copies of the arguments) for processing by the object-oriented methods. The overhead of calls, virtual or otherwise, is not dominated by the control flow alteration – but by the surrounding calling convention costs, like prologue and epilogue code, stack setup and argument passing[8] (see here[9] for more realistic instruction path length, stack and other costs associated with calls on an x86 platform). See also here[10] for a slide presentation by Eric S. Roberts ("The Allocation of Memory to Variables", chapter 7)[11] – illustrating the use of stack and heap memory use when summing three rational numbers in the Java object-oriented language.

Imperative

```
load r;                          1
r2 = r * r;                      2
result = r2 * "3.142";           3
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.... storage .............
result variable
constant "3.142"
```

Procedural

```
area proc(r2,res):
   push stack                                 5
   load r2;                                   6
   r3 = r2 * r2;                              7
   res = r3 * "3.142";                        8
   pop stack                                  9
   return;                                   10
...............................................
main proc:
   load r;                                    1
   call area(r,result);
    +load p = address of parameter list;      2
    +load v = address of subroutine 'area';   3
    +goto v with return;                      4
.
.
.
.
.... storage .............
result variable
constant "3.142"
parameter list variable
function pointer (==>area)
stack storage
```

Object-oriented

```
circle.area method(r2):
   push stack                                     7
   load r2;                                       8
   r3 = r2 * r2;                                  9
   res = r3 * "3.142";                           10
   pop stack                                     11
   return(res);                               12,13
...............................................
main proc:
   load r;                                        1
   result = circle.area(r);
      +allocate heap storage;                     2[See 1]
      +copy r to message;                         3
      +load p = address of message;               4
      +load v = addr. of method 'circle.area'     5
      +goto v with return;                        6
.
.
.... storage .............
result variable (assumed pre-allocated)
immutable variable "3.142" (final)
(heap) message variable for circle method call
vtable(==>area)
stack storage
```

1. See section: *Allocation of dynamic memory for message and object storage*

The advantages of procedural abstraction and object-oriented-style polymorphism are poorly illustrated by a small example like the one above. This example is designed mainly to illustrate some intrinsic performance differences, not abstraction or code re-use.

Subroutine, method call overhead

The presence of a (called) subroutine in a program contributes nothing extra to the functionality of the program regardless of paradigm, but may contribute greatly to the structuring and generality of the program, making it much easier to write, modify, and extend.[12] The extent to which different paradigms use subroutines (and their consequent memory requirements) influences the overall performance of the complete algorithm, although as Guy Steele pointed out in a 1977 paper, a well-designed programming language implementation *can* have very low overheads for procedural abstraction (but laments, in most implementations, that they seldom achieve this in practice - being "rather thoughtless or careless in this regard"). In the same paper, Steele also makes a considered case for automata-based programming (using procedure calls with tail recursion) and concludes that "we should have a healthy respect for procedure calls" (because they are powerful) but suggested "use them sparingly"[12]

In the frequency of subroutine calls:

- For procedural programming, the granularity of the code is largely determined by the number of discrete procedures or modules.

- For functional programming, frequent calls to library subroutines are common, but may be often inlined by the optimizing compiler
- For object-oriented programming, the number of method calls invoked is also partly determined by the granularity of the data structures and may thus include many *read-only* accesses to low level objects that are encapsulated, and thus accessible in no other, more direct, way. Since increased granularity is a prerequisite for greater code reuse, the tendency is toward fine-grained data structures, and a corresponding increase in the number of discrete objects (and their methods) and, consequently, subroutine calls. The creation of *god objects* is actively discouraged. Constructors also add to the count as they are also subroutine calls (unless they are inlined). Performance problems caused by excessive granularity may not become apparent until scalability becomes an issue.
- For other paradigms, where a mix of the above paradigms may be employed, subroutine use is less predictable.

Allocation of dynamic memory for message and object storage

Uniquely, the object-oriented paradigm involves dynamic memory allocation from *heap storage* for both object creation and message passing. A 1994 benchmark - "Memory Allocation Costs in Large C and C++ Programs" conducted by Digital Equipment Corporation on a variety of software, using an instruction-level profiling tool, measured how many instructions were required per dynamic storage allocation. The results showed that the lowest absolute number of instructions executed averaged around 50 but others reached as high as 611.[13] See also "Heap:Pleasures and pains" by Murali R. Krishnan[14] that states "Heap implementations tend to stay general for all platforms, and hence have heavy overhead". The 1996 IBM paper "Scalability of Dynamic Storage Allocation Algorithms" by Arun Iyengar of IBM [15] demonstrates various dynamic storage algorithms and their respective instruction counts. Even the recommended MFLF I algorithm (H.S. Stone, RC 9674) shows instruction counts in a range between 200 and 400. The above pseudocode example does not include a realistic estimate of this memory allocation pathlength or the memory prefix overheads involved and the subsequent associated garbage collection overheads. Suggesting strongly that heap allocation is a nontrivial task, one open-source software microallocator, by game developer John W. Ratcliff, consists of nearly 1,000 lines of code.[16]

Dynamically dispatched message calls v. direct procedure call overheads

In their Abstract "*Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis*",[17] Jeffrey Dean, David Grove, and Craig Chambers of the Department of Computer Science and Engineering, at the University of Washington, claim that "Heavy use of inheritance and dynamically-bound messages is likely to make code more extensible and reusable, but it also imposes a significant performance overhead, relative to an equivalent but non-extensible program written in a non-object-oriented manner. In some domains, such as structured graphics packages, the performance cost of the extra flexibility provided by using a heavily object-oriented style is acceptable. However, in other domains, such as basic data structure libraries, numerical computing packages, rendering libraries, and trace-driven simulation frameworks, the cost of message passing can be too great, forcing the programmer to avoid object-oriented programming in the “hot spots” of their application."

Serializing objects

Serialization imposes large overheads when passing objects from one system to another, especially when the transfer is in human-readable formats such as Extensible Markup Language (XML) and JavaScript Object Notation (JSON). This contrasts with compact binary formats for non-object-oriented data. Both encoding and decoding of the objects data value and its attributes are involved in the serializing process, which also includes awareness of complex issues such as inheriting, encapsulating, and data hiding.

Parallel computing

Carnegie-Mellon University Professor Robert Harper in March 2011 wrote: "This semester Dan Licata and I are co-teaching a new course on functional programming for first-year prospective CS majors... Object-oriented programming is eliminated entirely from the introductory curriculum, because it is both anti-modular and anti-parallel by its very nature, and hence unsuitable for a modern CS curriculum. A proposed new course on object-oriented design methodology will be offered at the sophomore level for those students who wish to study this topic."[18]

See also

- Comparison of programming languages
- Comparison of programming languages (basic instructions)
- Granularity#Computing
- Message passing
- Subroutine

References

1. https://meetingcpp.com/tl_files/mcpp/slides/12/FunctionalProgrammingInC++11.pdf
2. Ruiz, Cedric (May 2014). "Functional CoffeeScript for the impatient". *Blog de Cedric Ruiz*. Cedric Ruiz. Retrieved 2015-08-09.
3. Shelly, Asaf (2008-08-22). "Flaws of Object-oriented Modeling". Intel Software Network. Retrieved 2010-07-04.
4. Yegge, Steve (2006-03-30). "Execution in the Kingdom of Nouns". steve-yegge.blogspot.com. Retrieved 2010-07-03.
5. [1] (http://gamesfromwithin.com/data-oriented-design)
6. "The Jargon File v4.4.7: "syntactic sugar" ".
7. Gray, Jan (June 2003). "Writing Faster Managed Code: Know What Things Cost". *MSDN*. Microsoft.
8. "The True Cost of Calls". wordpress.com. 2008-12-30.
9. http://en.wikibooks.org/wiki/X86_Disassembly/Functions_and_Stack_Frames
10. Roberts, Eric S. (2008). "Art and Science of Java; Chapter 7: Objects and Memory". Stanford University.
11. Roberts, Eric S. (2008). *Art and Science of Java*. Addison-Wesley. ISBN 978-0-321-48612-7.
12. Guy Lewis Steele, Jr. "Debunking the 'Expensive Procedure Call' Myth, or, Procedure Call Implementations Considered Harmful, or, Lambda: The Ultimate GOTO". MIT AI Lab. AI Lab Memo AIM-443. October 1977. [2] (http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-443.pdf)[3] (http://dspace.mit.edu/handle/1721.1/5753)[4] (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.72.4404&rep=rep1&type=pdf)
13. Detlefs, David; Dosser, Al; Zorn, Benjamin (June 1994). "Memory Allocation Costs in Large C and C++ Programs; Page 532". *Software—Practice and Experience*. **24** (6): 527–542. CiteSeerX 10.1.1.30.3073.
14. Krishnan, Murali R. (February 1999). "Heap: Pleasures and pains". microsoft.com.
15. "Scalability of Dynamic Storage Allocation Algorithms". CiteSeerX 10.1.1.3.3759.
16. "MicroAllocator.h". *Google Code*. Google. Retrieved 2012-01-29.

17. Dean, Jeffrey; Grove, David; Chambers, Craig. "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis". University of Washington. CiteSeerX 10.1.1.117.2420. doi:10.1007/3-540-49538-X_5.
18. Teaching FP to Freshmen (http://existentialtype.wordpress.com/2011/03/15/teaching-fp-to-freshmen/), from Harper's blog about teaching introductory computer science.[5] (http://existentialtype.wordpress.com/2011/03/15/getting-started/)

Further reading

- "A Memory Allocator" (http://g.oswego.edu/dl/html/malloc.html) by Doug Lea
- "Dynamic Memory Allocation and Linked Data Structures" (http://www.sqa.org.uk/e-learning/LinkedDS01CD/page_01.htm) by (Scottish Qualifications Authority)
- "Inside A Storage Allocator" (http://www.flounder.com/inside_storage_allocation.htm) by Dr. Newcomer Ph.D

External links

- Comparing Programming Paradigms (http://users.ecs.soton.ac.uk/mrd/research/prog.html) by Dr Rachel Harrison and Mr Lins Samaraweera
- Comparing Programming Paradigms: an Evaluation of Functional and Object-Oriented Programs (http://eprints.ecs.soton.ac.uk/597/) by Harrison, R., Samaraweera, L. G., Dobie, M. R. and Lewis, P. H. (1996) pp. 247–254. ISSN 0268-6961
- "The principal programming paradigms" (http://www.info.ucl.ac.be/~pvr/paradigmsDIAGRAMeng101.pdf) By Peter Van Roy
- "Concepts, Techniques, and Models of Computer Programming" (http://www.info.ucl.ac.be/~pvr/book.html) (2004) by Peter Van Roy & Seif Haridi, ISBN 0-262-22069-5
- The True Cost of Calls (http://hbfs.wordpress.com/2008/12/30/the-true-cost-of-calls/)- from "Harder, Better, Faster, Stronger" blog by computer scientist Steven Pigeon (http://www.stevenpigeon.org/Publications/)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Comparison_of_programming_paradigms&oldid=770515656"

Categories: Programming paradigms