Object-oriented programming

From Wikipedia, the free encyclopedia

Object-oriented programming (**OOP**) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as *attributes;* and code, in the form of procedures, often known as *methods*. A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self"). In OOP, computer programs are designed by making them out of objects that interact with one another.[1][2] There is significant diversity of OOP languages, but the most popular ones are class-based, meaning that objects are instances of classes, which typically also determine their type.

Many of the most widely used programming languages are multi-paradigm programming languages that support object-oriented programming to a greater or lesser degree, typically in combination with imperative, procedural programming. Significant object-oriented languages include Java, C++, C#, Python, PHP, Ruby, Perl, Delphi, Objective-C, Swift, Common Lisp, and Smalltalk.

Contents

Features

Object-oriented Programming uses objects, but not all of the associated techniques and structures are supported directly in languages that claim to support OOP. The features listed below are, however, common among languages considered strongly class- and object-oriented (or multi-paradigm with OOP support), with notable exceptions mentioned.[3][4][5][6]

Shared with non-OOP predecessor languages

Object-oriented programming languages typically share low-level features with high-level procedural programming languages (which were invented first). The fundamental tools that can be used to construct a program include:

- Variables that can store information formatted in a small number of built-in data types like integers and alphanumeric characters. This may include data structures like strings, lists, and hash tables that are either built-in or result from combining variables using memory pointers
- Procedures – also known as functions, methods, routines, or subroutines – that take input, generate output, and manipulate data. Modern languages include structured programming constructs like loops and conditionals.

Modular programming support provides the ability to group procedures into files and modules for organizational purposes. Modules are namespaced so code in one module will not be accidentally confused with the same procedure or variable name in another file or module.

Objects and classes

Languages that support object-oriented programming typically use inheritance for code reuse and extensibility in the form of either classes or prototypes. Those that use classes support two main concepts:

- Classes – the definitions for the data format and available procedures for a given type or class of object; may also contain data and procedures (known as class methods) themselves, i.e. classes contains the data members and member functions
- Objects – instances of classes

Objects sometimes correspond to things found in the real world. For example, a graphics program may have objects such as "circle", "square", "menu". An online shopping system might have objects such as "shopping cart", "customer", and "product".[7] Sometimes objects represent more abstract entities, like an object that represents an open file, or an object that provides the service of translating measurements from U.S. customary to metric.

Each object is said to be an instance of a particular class (for example, an object with its name field set to "Mary" might be an instance of class Employee). Procedures in object-oriented programming are known as methods; variables are also known as fields, members, attributes, or properties. This leads to the following terms:

- Class variables – belong to the *class as a whole*; there is only one copy of each one
- Instance variables or attributes – data that belongs to individual *objects*; every object has its own copy of each one
- Member variables – refers to both the class and instance variables that are defined by a particular class
- Class methods – belong to the *class as a whole* and have access only to class variables and inputs from the procedure call
- Instance methods – belong to *individual objects*, and have access to instance variables for the specific object they are called on, inputs, and class variables

Objects are accessed somewhat like variables with complex internal structure, and in many languages are effectively pointers, serving as actual references to a single instance of said object in memory within a heap or stack. They provide a layer of abstraction which can be used to separate internal from external code. External code

can use an object by calling a specific instance method with a certain set of input parameters, read an instance variable, or write to an instance variable. Objects are created by calling a special type of method in the class known as a constructor. A program may create many instances of the same class as it runs, which operate independently. This is an easy way for the same procedures to be used on different sets of data.

Object-oriented programming that uses classes is sometimes called class-based programming, while prototype-based programming does not typically use classes. As a result, a significantly different yet analogous terminology is used to define the concepts of *object* and *instance*.

In some languages classes and objects can be composed using other concepts like traits and mixins.

Dynamic dispatch/message passing

It is the responsibility of the object, not any external code, to select the procedural code to execute in response to a method call, typically by looking up the method at run time in a table associated with the object. This feature is known as dynamic dispatch, and distinguishes an object from an abstract data type (or module), which has a fixed (static) implementation of the operations for all instances. If there are multiple methods that might be run for a given name, it is known as multiple dispatch.

A method call is also known as *message passing*. It is conceptualized as a message (the name of the method and its input parameters) being passed to the object for dispatch.

Encapsulation

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of data hiding.

If a class does not allow calling code to access internal object data and permits access through methods only, this is a strong form of abstraction or information hiding known as encapsulation. Some languages (Java, for example) let classes enforce access restrictions explicitly, for example denoting internal data with the `private` keyword and designating methods intended for use by code outside the class with the `public` keyword. Methods may also be designed public, private, or intermediate levels such as `protected` (which allows access from the same class and its subclasses, but not objects of a different class). In other languages (like Python) this is enforced only by convention (for example, `private` methods may have names that start with an underscore). Encapsulation prevents external code from being concerned with the internal workings of an object. This facilitates code refactoring, for example allowing the author of the class to change how objects of that class represent their data internally without changing any external code (as long as "public" method calls work the same way). It also encourages programmers to put all the code that is concerned with a certain set of data in the same class, which organizes it for easy comprehension by other programmers. Encapsulation is a technique that encourages decoupling.

Composition, inheritance, and delegation

Objects can contain other objects in their instance variables; this is known as object composition. For example, an object in the Employee class might contain (point to) an object in the Address class, in addition to its own instance variables like "first_name" and "position". Object composition is used to represent "has-a" relationships: every employee has an address, so every Employee object has a place to store an Address object.

Languages that support classes almost always support inheritance. This allows classes to be arranged in a hierarchy that represents "is-a-type-of" relationships. For example, class Employee might inherit from class Person. All the data and methods available to the parent class also appear in the child class with the same names. For example, class Person might define variables "first_name" and "last_name" with method "make_full_name()". These will

also be available in class Employee, which might add the variables "position" and "salary". This technique allows easy re-use of the same procedures and data definitions, in addition to potentially mirroring real-world relationships in an intuitive way. Rather than utilizing database tables and programming subroutines, the developer utilizes objects the user may be more familiar with: objects from their application domain.[8]

Subclasses can override the methods defined by superclasses. Multiple inheritance is allowed in some languages, though this can make resolving overrides complicated. Some languages have special support for mixins, though in any language with multiple inheritance, a mixin is simply a class that does not represent an is-a-type-of relationship. Mixins are typically used to add the same methods to multiple classes. For example, class UnicodeConversionMixin might provide a method unicode_to_ascii() when included in class FileReader and class WebPageScraper, which don't share a common parent.

Abstract classes cannot be instantiated into objects; they exist only for the purpose of inheritance into other "concrete" classes which can be instantiated. In Java, the `final` keyword can be used to prevent a class from being subclassed.

The doctrine of composition over inheritance advocates implementing has-a relationships using composition instead of inheritance. For example, instead of inheriting from class Person, class Employee could give each Employee object an internal Person object, which it then has the opportunity to hide from external code even if class Person has many public attributes or methods. Some languages, like Go do not support inheritance at all.

The "open/closed principle" advocates that classes and functions "should be open for extension, but closed for modification".

Delegation is another language feature that can be used as an alternative to inheritance.

Polymorphism

Subtyping, a form of polymorphism, is when calling code can be agnostic as to whether an object belongs to a parent class or one of its descendants. For example, a function might call "make_full_name()" on an object, which will work whether the object is of class Person or class Employee. This is another type of abstraction which simplifies code external to the class hierarchy and enables strong separation of concerns.

Open recursion

In languages that support open recursion, object methods can call other methods on the same object (including themselves), typically using a special variable or keyword called `this` or `self`. This variable is *late-bound*; it allows a method defined in one class to invoke another method that is defined later, in some subclass thereof.

History

Terminology invoking "objects" and "oriented" in the modern sense of object-oriented programming made its first appearance at MIT in the late 1950s and early 1960s. In the environment of the artificial intelligence group, as early as 1960, "object" could refer to identified items (LISP atoms) with properties (attributes);[9][10] Alan Kay was later to cite a detailed understanding of LISP internals as a strong influence on his thinking in 1966.[11] Another early MIT example was Sketchpad created by Ivan Sutherland in 1960–61; in the glossary of the 1963 technical report based on his dissertation about Sketchpad, Sutherland defined notions of "object" and "instance" (with the class concept covered by "master" or "definition"), albeit specialized to graphical interaction.[12] Also, an MIT ALGOL version, AED-0, established a direct link between data structures ("plexes", in that dialect) and procedures, prefiguring what were later termed "messages", "methods", and "member functions".[13][14]

The formal programming concept of objects was introduced in the mid-1960s with Simula 67, a major revision of Simula I, a programming language designed for discrete event simulation, created by Ole-Johan Dahl and Kristen Nygaard of the Norwegian Computing Center in Oslo.[15] [16] [17] [18] [19]

Simula 67 was influenced by SIMSCRIPT and C.A.R. "Tony" Hoare's proposed "record classes".[13][20] Simula introduced the notion of classes and instances or objects (as well as subclasses, virtual procedures, coroutines, and discrete event simulation) as part of an explicit programming paradigm. The language also used automatic garbage collection that had been invented earlier for the functional programming language Lisp. Simula was used for physical modeling, such as models to study and improve the movement of ships and their content through cargo ports. The ideas of Simula 67 influenced many later languages, including Smalltalk, derivatives of LISP (CLOS), Object Pascal, and C++.

The Smalltalk language, which was developed at Xerox PARC (by Alan Kay and others) in the 1970s, introduced the term *object-oriented programming* to represent the pervasive use of objects and messages as the basis for computation. Smalltalk creators were influenced by the ideas introduced in Simula 67, but Smalltalk was designed to be a fully dynamic system in which classes could be created and modified dynamically rather than statically as in Simula 67.[21] Smalltalk and with it OOP were introduced to a wider audience by the August 1981 issue of *Byte Magazine*.

In the 1970s, Kay's Smalltalk work had influenced the Lisp community to incorporate object-based techniques that were introduced to developers via the Lisp machine. Experimentation with various extensions to Lisp (such as LOOPS and Flavors introducing multiple inheritance and mixins) eventually led to the Common Lisp Object System, which integrates functional programming and object-oriented programming and allows extension via a Meta-object protocol. In the 1980s, there were a few attempts to design processor architectures that included hardware support for objects in memory but these were not successful. Examples include the Intel iAPX 432 and the Linn Smart Rekursiv.

In 1985, Bertrand Meyer produced the first design of the Eiffel language. Focused on software quality, Eiffel is among the purely object-oriented languages, but differs in the sense that the language itself is not only a programming language, but a notation supporting the entire software lifecycle. Meyer described the Eiffel software development method, based on a small number of key ideas from software engineering and computer science, in Object-Oriented Software Construction. Essential to the quality focus of Eiffel is Meyer's reliability mechanism, Design by Contract, which is an integral part of both the method and language.

Object-oriented programming developed as the dominant programming methodology in the early and mid 1990s when programming languages supporting the techniques became widely available. These included Visual FoxPro 3.0,[22][23][24] C++,[25] and Delphi. Its dominance was further enhanced by the rising popularity of graphical user interfaces, which rely heavily upon object-oriented programming techniques. An example of a closely related dynamic GUI library and OOP language can be found in the Cocoa frameworks on Mac OS X, written in Objective-C, an object-oriented, dynamic messaging extension to C based on Smalltalk. OOP toolkits also enhanced the popularity of event-driven programming (although this concept is not limited to OOP).

At ETH Zürich, Niklaus Wirth and his colleagues had also been investigating such topics as data abstraction and modular programming (although this had been in common use in the 1960s or earlier). Modula-2 (1978) included both, and their succeeding design, Oberon, included a distinctive approach to object orientation, classes, and such.

Object-oriented features have been added to many previously existing languages, including Ada, BASIC, Fortran, Pascal, and COBOL. Adding these features to languages that were not initially designed for them often led to problems with compatibility and maintainability of code.

More recently, a number of languages have emerged that are primarily object-oriented, but that are also compatible with procedural methodology. Two such languages are Python and Ruby. Probably the most commercially important recent object-oriented languages are Java, developed by Sun Microsystems, as well as C# and Visual Basic.NET (VB.NET), both designed for Microsoft's .NET platform. Each of these two frameworks shows, in its own way, the benefit of using OOP by creating an abstraction from implementation. VB.NET and C# support cross-language inheritance, allowing classes defined in one language to subclass classes defined in the other language.

OOP languages

Simula (1967) is generally accepted as being the first language with the primary features of an object-oriented language. It was created for making simulation programs, in which what came to be called objects were the most important information representation. Smalltalk (1972 to 1980) is an early example, and the one with which much of the theory of object-oriented programming was developed. Concerning the degree of object orientation, the following distinctions can be made:

- Languages called "pure" OO languages, because everything in them is treated consistently as an object, from primitives such as characters and punctuation, all the way up to whole classes, prototypes, blocks, modules, etc. They were designed specifically to facilitate, even enforce, OO methods. Examples: Python, Ruby, Scala, Smalltalk, Eiffel, Emerald,[26] JADE, Self.
- Languages designed mainly for OO programming, but with some procedural elements. Examples: Java, C++, C#, Delphi/Object Pascal, VB.NET.
- Languages that are historically procedural languages, but have been extended with some OO features. Examples: PHP, Perl, Visual Basic (derived from BASIC), MATLAB, COBOL 2002, Fortran 2003, ABAP, Ada 95, Pascal.
- Languages with most of the features of objects (classes, methods, inheritance), but in a distinctly original form. Examples: Oberon (Oberon-1 or Oberon-2).
- Languages with abstract data type support which may be used to resemble OO programming, but without all features of object-orientation. This includes object-*based* and prototype-based languages. Examples: JavaScript, Lua, Modula-2, CLU.
- Chameleon languages that support multiple paradigms, including OO. Tcl stands out among these for TclOO, a hybrid object system that supports both prototype-based programming and class-based OO.

OOP in dynamic languages

In recent years, object-oriented programming has become especially popular in dynamic programming languages. Python, PowerShell, Ruby and Groovy are dynamic languages built on OOP principles, while Perl and PHP have been adding object-oriented features since Perl 5 and PHP 4, and ColdFusion since version 6.

The Document Object Model of HTML, XHTML, and XML documents on the Internet has bindings to the popular JavaScript/ECMAScript language. JavaScript is perhaps the best known prototype-based programming language, which employs cloning from prototypes rather than inheriting from a class (contrast to class-based programming). Another scripting language that takes this approach is Lua.

OOP in a network protocol

The messages that flow between computers to request services in a client-server environment can be designed as the linearizations of objects defined by class objects known to both the client and the server. For example, a simple linearized object would consist of a length field, a code point identifying the class, and a data value. A more complex example would be a command consisting of the length and code point of the command and values consisting of linearized objects representing the command's parameters. Each such command must be directed by

the server to an object whose class (or superclass) recognizes the command and is able to provide the requested service. Clients and servers are best modeled as complex object-oriented structures. Distributed Data Management Architecture (DDM) took this approach and used class objects to define objects at four levels of a formal hierarchy:

- Fields defining the data values that form messages, such as their length, codepoint and data values.
- Objects and collections of objects similar to what would be found in a Smalltalk program for messages and parameters.
- Managers similar to AS/400 objects, such as a directory to files and files consisting of metadata and records. Managers conceptually provide memory and processing resources for their contained objects.
- A client or server consisting of all the managers necessary to implement a full processing environment, supporting such aspects as directory services, security and concurrency control.

The initial version of DDM defined distributed file services. It was later extended to be the foundation of Distributed Relational Database Architecture (DRDA).

Design patterns

Challenges of object-oriented design are addressed by several methodologies. Most common is known as the design patterns codified by Gamma *et al*.. More broadly, the term "design patterns" can be used to refer to any general, repeatable solution to a commonly occurring problem in software design. Some of these commonly occurring problems have implications and solutions particular to object-oriented development.

Inheritance and behavioral subtyping

It is intuitive to assume that inheritance creates a semantic "is a" relationship, and thus to infer that objects instantiated from subclasses can always be *safely* used instead of those instantiated from the superclass. This intuition is unfortunately false in most OOP languages, in particular in all those that allow mutable objects. Subtype polymorphism as enforced by the type checker in OOP languages (with mutable objects) cannot guarantee behavioral subtyping in any context. Behavioral subtyping is undecidable in general, so it cannot be implemented by a program (compiler). Class or object hierarchies must be carefully designed, considering possible incorrect uses that cannot be detected syntactically. This issue is known as the Liskov substitution principle.

Gang of Four design patterns

Design Patterns: Elements of Reusable Object-Oriented Software is an influential book published in 1995 by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, often referred to humorously as the "Gang of Four". Along with exploring the capabilities and pitfalls of object-oriented programming, it describes 23 common programming problems and patterns for solving them. As of April 2007, the book was in its 36th printing.

The book describes the following patterns:

- *Creational patterns* (5): Factory method pattern, Abstract factory pattern, Singleton pattern, Builder pattern, Prototype pattern
- *Structural patterns* (7): Adapter pattern, Bridge pattern, Composite pattern, Decorator pattern, Facade pattern, Flyweight pattern, Proxy pattern
- *Behavioral patterns* (11): Chain-of-responsibility pattern, Command pattern, Interpreter pattern, Iterator pattern, Mediator pattern, Memento pattern, Observer pattern, State pattern, Strategy pattern, Template method pattern, Visitor pattern

Object-orientation and databases

Both object-oriented programming and relational database management systems (RDBMSs) are extremely common in software today. Since relational databases don't store objects directly (though some RDBMSs have object-oriented features to approximate this), there is a general need to bridge the two worlds. The problem of bridging object-oriented programming accesses and data patterns with relational databases is known as object-relational impedance mismatch. There are a number of approaches to cope with this problem, but no general solution without downsides.[27] One of the most common approaches is object-relational mapping, as found in IDE languages such as Visual FoxPro and libraries such as Java Data Objects and Ruby on Rails' ActiveRecord.

There are also object databases that can be used to replace RDBMSs, but these have not been as technically and commercially successful as RDBMSs.

Real-world modeling and relationships

OOP can be used to associate real-world objects and processes with digital counterparts. However, not everyone agrees that OOP facilitates direct real-world mapping (see Criticism section) or that real-world mapping is even a worthy goal; Bertrand Meyer argues in *Object-Oriented Software Construction*[28] that a program is not a model of the world but a model of some part of the world; "Reality is a cousin twice removed". At the same time, some principal limitations of OOP had been noted.[29] For example, the circle-ellipse problem is difficult to handle using OOP's concept of inheritance.

However, Niklaus Wirth (who popularized the adage now known as Wirth's law: "Software is getting slower more rapidly than hardware becomes faster") said of OOP in his paper, "Good Ideas through the Looking Glass", "This paradigm closely reflects the structure of systems 'in the real world', and it is therefore well suited to model complex systems with complex behaviours"[30] (contrast KISS principle).

Steve Yegge and others noted that natural languages lack the OOP approach of strictly prioritizing *things* (objects/nouns) before *actions* (methods/verbs).[31] This problem may cause OOP to suffer more convoluted solutions than procedural programming.[32]

OOP and control flow

OOP was developed to increase the reusability and maintainability of source code.[33] Transparent representation of the control flow had no priority and was meant to be handled by a compiler. With the increasing relevance of parallel hardware and multithreaded coding, developing transparent control flow becomes more important, something hard to achieve with OOP.[34][35][36][37]

Responsibility- vs. data-driven design

Responsibility-driven design defines classes in terms of a contract, that is, a class should be defined around a responsibility and the information that it shares. This is contrasted by Wirfs-Brock and Wilkerson with data-driven design, where classes are defined around the data-structures that must be held. The authors hold that responsibility-driven design is preferable.

SOLID and GRASP guidelines

SOLID is a mnemonic invented by Michael Feathers that stands for and advocates five programming practices:

- Single responsibility principle
- Open/closed principle

- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

GRASP (General Responsibility Assignment Software Patterns) is another set of guidelines advocated by Craig Larman.

Criticism

The OOP paradigm has been criticised for a number of reasons, including not meeting its stated goals of reusability and modularity,[38][39] and for overemphasizing one aspect of software design and modeling (data/objects) at the expense of other important aspects (computation/algorithms).[40][41]

Luca Cardelli has claimed that OOP code is "intrinsically less efficient" than procedural code, that OOP can take longer to compile, and that OOP languages have "extremely poor modularity properties with respect to class extension and modification", and tend to be extremely complex.[38] The latter point is reiterated by Joe Armstrong, the principal inventor of Erlang, who is quoted as saying:[39]

> The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

A study by Potok et al. has shown no significant difference in productivity between OOP and procedural approaches.[42]

Christopher J. Date stated that critical comparison of OOP to other technologies, relational in particular, is difficult because of lack of an agreed-upon and rigorous definition of OOP;[43] however, Date and Darwen have proposed a theoretical foundation on OOP that uses OOP as a kind of customizable type system to support RDBMS.[44]

In an article Lawrence Krubner claimed that compared to other languages (LISP dialects, functional languages, etc.) OOP languages have no unique strengths, and inflict a heavy burden of unneeded complexity.[45]

Alexander Stepanov compares object orientation unfavourably to generic programming:[40]

> I find OOP technically unsound. It attempts to decompose the world in terms of interfaces that vary on a single type. To deal with the real problems you need multisorted algebras — families of interfaces that span multiple types. I find OOP philosophically unsound. It claims that everything is an object. Even if it is true it is not very interesting — saying that everything is an object is saying nothing at all.

Paul Graham has suggested that OOP's popularity within large companies is due to "large (and frequently changing) groups of mediocre programmers". According to Graham, the discipline imposed by OOP prevents any one programmer from "doing too much damage".[46]

Steve Yegge noted that, as opposed to functional programming:[47]

> Object Oriented Programming puts the Nouns first and foremost. Why would you go to such lengths to put one part of speech on a pedestal? Why should one kind of concept take precedence over another? It's not as if OOP has suddenly made verbs less important in the way we actually think. It's a strangely skewed perspective.

Rich Hickey, creator of Clojure, described object systems as overly simplistic models of the real world. He emphasized the inability of OOP to model time properly, which is getting increasingly problematic as software systems become more concurrent.[41]

Eric S. Raymond, a Unix programmer and open-source software advocate, has been critical of claims that present object-oriented programming as the "One True Solution", and has written that object-oriented programming languages tend to encourage thickly layered programs that destroy transparency.[48] Raymond compares this unfavourably to the approach taken with Unix and the C programming language.[48]

Rob Pike, a programmer involved in the creation of UTF-8 and Go, has said that OOP languages frequently favor inheritance over composition, and the paradigm as a whole shifts the focus from data structures and algorithms to types.[49] Furthermore, he cites an instance of a Java professor whose "idiomatic" solution to a problem was to create six new classes, rather than to simply use a lookup table.[50]

Formal semantics

Objects are the run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data, or any item that the program has to handle.

There have been several attempts at formalizing the concepts used in object-oriented programming. The following concepts and constructs have been used as interpretations of OOP concepts:

- co algebraic data types[51]
- abstract data types (which have existential types) allow the definition of modules but these do not support dynamic dispatch
- recursive types
- encapsulated state
- inheritance
- records are basis for understanding objects if function literals can be stored in fields (like in functional programming languages), but the actual calculi need be considerably more complex to incorporate essential features of OOP. Several extensions of System $F_{<:}$ that deal with mutable objects have been studied;[52] these allow both subtype polymorphism and parametric polymorphism (generics)

Attempts to find a consensus definition or theory behind objects have not proven very successful (however, see Abadi & Cardelli, *A Theory of Objects* (http://portal.acm.org/citation.cfm?id=547964&dl=ACM&coll=portal)[52] for formal definitions of many OOP concepts and constructs), and often diverge widely. For example, some definitions focus on mental activities, and some on program structuring. One of the simpler definitions is that OOP is the act of using "map" data structures or arrays that can contain functions and pointers to other maps, all with some syntactic and scoping sugar on top. Inheritance can be performed by cloning the maps (sometimes called "prototyping").

See also

- Comparison of programming languages (object-oriented programming)
- Comparison of programming paradigms
- Component-based software engineering
- Design by contract
- Object association
- Object database
- Object modeling language
- Object-oriented analysis and design
- Object-relational impedance mismatch (and The Third Manifesto)
- Object-relational mapping

Systems

- CADES
- Common Object Request Broker Architecture (CORBA)
- Distributed Component Object Model
- Distributed Data Management Architecture
- Jeroo

Modeling languages

- IDEF4
- Interface description language
- Lepus3
- UML

References

1. Kindler, E.; Krivy, I. (2011). "Object-Oriented Simulation of systems with sophisticated control". International Journal of General Systems: 313–343.
2. Lewis, John; Loftus, William (2008). *Java Software Solutions Foundations of Programming Design 6th ed*. Pearson Education Inc. ISBN 0-321-53205-8., section 1.6 "Object-Oriented Programming"
3. Deborah J. Armstrong. *The Quarks of Object-Oriented Development*. A survey of nearly 40 years of computing literature which identified a number of fundamental concepts found in the large majority of definitions of OOP, in descending order of popularity: Inheritance, Object, Class, Encapsulation, Method, Message Passing, Polymorphism, and Abstraction.
4. John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, ISBN 0-521-78098-5, p.278. Lists: Dynamic dispatch, abstraction, subtype polymorphism, and inheritance.
5. Michael Lee Scott, *Programming language pragmatics*, Edition 2, Morgan Kaufmann, 2006, ISBN 0-12-633951-1, p. 470. Lists encapsulation, inheritance, and dynamic dispatch.
6. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press. ISBN 0-262-16209-1., section 18.1 "What is Object-Oriented Programming?" Lists: Dynamic dispatch, encapsulation or multi-methods (multiple dispatch), subtype polymorphism, inheritance or delegation, open recursion ("this"/"self")
7. Booch, Grady (1986). *Software Engineering with Ada*. Addison Wesley. p. 220. ISBN 978-0805306088. "Perhaps the greatest strength of an object-oriented approach to development is that it offers a mechanism that captures a model of the real world."
8. Jacobsen, Ivar; Magnus Christerson; Patrik Jonsson; Gunnar Overgaard (1992). *Object Oriented Software Engineering*. Addison-Wesley ACM Press. pp. 43–69. ISBN 0-201-54435-0.
9. McCarthy, J.; Brayton, R.; Edwards, D.; Fox, P.; Hodes, L.; Luckham, D.; Maling, K.; Park, D.; Russell, S. (March 1960). "LISP I Programmers Manual" (PDF). Boston, Massachusetts: Artificial Intelligence Group, M.I.T. Computation Center and Research Laboratory: 88f. "In the local M.I.T. patois, association lists [of atomic symbols] are also referred to as "property lists", and atomic symbols are sometimes called "objects"."

10. McCarthy, John; Abrahams, Paul W.; Edwards, Daniel J.; Hart, swapnil d.; Levin, Michael I. (1962). *LISP 1.5 Programmer's Manual* (PDF). MIT Press. p. 105. ISBN 0-262-13011-4. "Object — a synonym for atomic symbol"
11. "Dr. Alan Kay on the Meaning of "Object-Oriented Programming" ". 2003. Retrieved 11 February 2010.
12. Sutherland, I. E. (30 January 1963). "Sketchpad: A Man-Machine Graphical Communication System" (PDF). Technical Report No. 296, Lincoln Laboratory, Massachusetts Institute of Technology via Defense Technical Information Center (stinet.dtic.mil). Retrieved 3 November 2007.
13. The Development of the Simula Languages, Kristen Nygaard, Ole-Johan Dahl, p.254 Uni-kl.ac.at (http://cs-exhibitions.uni-klu.ac.at/fileadmin/template/documents/text/The_development_of_the_simula_languages.pdf)
14. Ross, Doug. "The first software engineering language". *LCS/AI Lab Timeline:*. MIT Computer Science and Artificial Intelligence Laboratory. Retrieved 13 May 2010.
15. Dahl, Ole Johan (2004). "The Birth of Object Orientation: the Simula Languages" (PDF). doi:10.1007/978-3-540-39993-3_3. Retrieved 9 June 2016.
16. Holmevik, Jan Rune (1994). "Compiling Simula: A historical study of technological genesis" (PDF). *IEEE Annals of the History of Computing*. **16** (4): 25–37. doi:10.1109/85.329756. Retrieved 12 May 2010.
17. Wilson, Leslie B.; Robert G. Clark (2001). *Comparative Programming Languages* (3 ed.). Addison-Wesley. p. 35. ISBN 978-0-201-71012-0. "Simula was based on Algol 60 with one very important addition – the class concept. It is possible to declare a class, generate objects of that class, name these objects and form a hierarchical structure of class declarations."
18. Wilson, Leslie B.; Robert G. Clark (2001). *Comparative Programming Languages* (3 ed.). Addison-Wesley. p. 35. ISBN 978-0-201-71012-0. "The impact of Simula on the design of programming languages is large as it is the original object-oriented language. The class concept has been taken over and used in many later languages, such as C++, Ada, Smalltalk, Eiffel and Java."
19. Bal, Henri E.; Dick Grune (1994). *Programming Language Essentials*. Addison-Wesley. p. 134. ISBN 0-201-63179-2. "Simula was the first language to use a class concept."
20. Hoare, C. A. (Nov 1965). "Record Handling". *ALGOL Bulletin* (21): 39–69. doi:10.1145/1061032.1061041.
21. Kay, Alan. "The Early History of Smalltalk". Archived from the original on 10 July 2008. Retrieved 13 September 2007.
22. 1995 (June) Visual FoxPro 3.0, FoxPro evolves from a procedural language to an object-oriented language. Visual FoxPro 3.0 introduces a database container, seamless client/server capabilities, support for ActiveX technologies, and OLE Automation and null support. Summary of Fox releases (http://www.foxprohistory.org/foxprotimeline.htm#summary_of_fox_releases)
23. FoxPro History web site: Foxprohistory.org (http://www.foxprohistory.org/tableofcontents.htm)
24. 1995 Reviewers Guide to Visual FoxPro 3.0: DFpug.de (http://www.dfpug.de/loseblattsammlung/migration/whitepapers/vfp_rg.htm)
25. https://books.google.co.uk/books?id=MHmqfSBTXsAC&pg=PA16&lpg=PA16
26. "The Emerald Programming Language". 2011-02-26.
27. Neward, Ted (26 June 2006). "The Vietnam of Computer Science". Interoperability Happens. Retrieved 2 June 2010.
28. Meyer, Second Edition, p. 230
29. M.Trofimov, *OOOP – The Third "O" Solution: Open OOP*. First Class, OMG, 1993, Vol. 3, issue 3, p.14.
30. Wirth, Nicklaus (2006). "Good Ideas, Through the Looking Glass" (PDF). *Computer (magazine)*. **39** (1): 28–39. Retrieved 2016-10-02.
31. Yegge, Steve (30 March 2006). "Execution in the Kingdom of Nouns". steve-yegge.blogspot.com. Retrieved 3 July 2010.
32. Boronczyk, Timothy (11 June 2009). "What's Wrong with OOP". zaemis.blogspot.com. Retrieved 3 July 2010.
33. Ambler, Scott (1 January 1998). "A Realistic Look at Object-Oriented Reuse". www.drdobbs.com. Retrieved 4 July 2010.
34. Shelly, Asaf (22 August 2008). "Flaws of Object Oriented Modeling". Intel Software Network. Retrieved 4 July 2010.
35. James, Justin (1 October 2007). "Multithreading is a verb not a noun". techrepublic.com. Retrieved 4 July 2010.
36. Shelly, Asaf (22 August 2008). "HOW TO: Multicore Programming (Multiprocessing) Visual C++ Class Design Guidelines, Member Functions". support.microsoft.com. Retrieved 4 July 2010.
37. Robert Harper (17 April 2011). "Some thoughts on teaching FP". Existential Type Blog. Retrieved 5 December 2011.
38. Cardelli, Luca (1996). "Bad Engineering Properties of Object-Oriented Languages". *ACM Comput. Surv.* ACM. **28** (4es): 150. doi:10.1145/242224.242415. ISSN 0360-0300. Retrieved 21 April 2010.
39. Armstrong, Joe. In *Coders at Work: Reflections on the Craft of Programming*. Peter Seibel, ed. Codersatwork.com (http://www.codersatwork.com/), Accessed 13 November 2009.
40. Stepanov, Alexander. "STLport: An Interview with A. Stepanov". Retrieved 21 April 2010.

41. Rich Hickey, JVM Languages Summit 2009 keynote, Are We There Yet? (http://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey) November 2009.
42. Potok, Thomas; Mladen Vouk; Andy Rindos (1999). "Productivity Analysis of Object-Oriented Software Developed in a Commercial Environment" (PDF). *Software – Practice and Experience*. **29** (10): 833–847. doi:10.1002/(SICI)1097-024X(199908)29:10<833::AID-SPE258>3.0.CO;2-P. Retrieved 21 April 2010.
43. C. J. Date, Introduction to Database Systems, 6th-ed., Page 650
44. C. J. Date, Hugh Darwen. *Foundation for Future Database Systems: The Third Manifesto* (2nd Edition)
45. Krubner, Lawrence. "Object Oriented Programming is an expensive disaster which must end". smashcompany.com. Retrieved 14 October 2014.
46. Graham, Paul. "Why ARC isn't especially Object-Oriented.". PaulGraham.com. Retrieved 13 November 2009.
47. Stevey's Blog Rants (http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html)
48. Eric S. Raymond (2003). "The Art of Unix Programming: Unix and Object-Oriented Languages". Retrieved 2014-08-06.
49. Pike, Rob (2012-06-25). "Less is exponentially more". Retrieved 2016-10-01.
50. Pike, Rob (2012-11-14). "A few years ago I saw this page". Retrieved 2016-10-01.
51. Poll, Erik. "Subtyping and Inheritance for Categorical Datatypes" (PDF). Retrieved 5 June 2011.
52. Abadi, Martin; Cardelli, Luca (1996). *A Theory of Objects*. Springer-Verlag New York, Inc. ISBN 0-387-94775-2. Retrieved 21 April 2010.

Further reading

- Abadi, Martin; Luca Cardelli (1998). *A Theory of Objects*. Springer Verlag. ISBN 0-387-94775-2.
- Abelson, Harold; Gerald Jay Sussman (1997). *Structure and Interpretation of Computer Programs*. MIT Press. ISBN 0-262-01153-0.
- Armstrong, Deborah J. (February 2006). "The Quarks of Object-Oriented Development". *Communications of the ACM*. **49** (2): 123–128. doi:10.1145/1113034.1113040. ISSN 0001-0782. Retrieved 8 August 2006.
- Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley. ISBN 0-8053-5340-2.
- Eeles, Peter; Oliver Sims (1998). *Building Business Objects*. John Wiley & Sons. ISBN 0-471-19176-0.
- Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2. Cite uses deprecated parameter `|coauthors=` (help)
- Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook – Lessons from Award-Winning Business Applications*. John Wiley & Sons. ISBN 0-471-14717-6.
- Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley. ISBN 0-201-54435-0.
- Kay, Alan. *The Early History of Smalltalk*.
- Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall. ISBN 0-13-629155-4.
- Pecinovsky, Rudolf (2013). *OOP – Learn Object Oriented Thinking & Programming*. Bruckner Publishing. ISBN 978-80-904661-8-0.
- Rumbaugh, James; Michael Blaha; William Premerlani; Frederick Eddy; William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall. ISBN 0-13-629841-9.
- Schach, Stephen (2006). *Object-Oriented and Classical Software Engineering, Seventh Edition*. McGraw-Hill. ISBN 0-07-319126-4.
- Schreiner, Axel-Tobias (1993). *Object oriented programming with ANSI-C*. Hanser. ISBN 3-446-17426-5. hdl:1850/8544.
- Taylor, David A. (1992). *Object-Oriented Information Systems – Planning and Implementation*. John Wiley & Sons. ISBN 0-471-54364-0.
- Weisfeld, Matt (2009). *The Object-Oriented Thought Process, Third Edition*. Addison-Wesley. ISBN 0-672-33016-4.
- West, David (2004). *Object Thinking (Developer Reference)*. Microsoft Press. ISBN 0735619654.

External links

- Object-oriented programming (https://www.dmoz.org/Computers/Programming/Methodologies/Object-Oriented) at DMOZ
- Real life and real world example of object oriented programming (http://www.objectorienteddesign.org)
- Introduction to Object Oriented Programming Concepts (OOP) and More (http://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep) by L.W.C. Nirosh
- Discussion about the flaws of OOD (http://software.intel.com/en-us/blogs/2008/08/22/flaws-of-object-oriented-modeling/)
- OOP Concepts (Java Tutorials) (http://java.sun.com/docs/books/tutorial/java/concepts/index.html)
- Science or Snake Oil: Empirical Software engineering (https://se9book.wordpress.com/2011/08/29/science-or-snake-oil-empirical-software-engineering/) Thoughts on software and systems engineering, by Ian Sommerville (2011-8-29)

Wikiquote has quotations related to: ***Object-orientation***

Wikiversity has learning materials about ***Object-oriented programming*** at Topic:Object-Oriented Programming

Wikibooks has a book on the topic of: ***Object Oriented Programming***

Retrieved from "https://en.wikipedia.org/w/index.php?title=Object-oriented_programming&oldid=748997044"

Categories: Object-oriented programming | Programming paradigms | Norwegian inventions

---