Object-oriented programming

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 (such as C++, Object Pascal, Java, Python etc.) 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, JavaScript, Ruby, Perl, Object Pascal, Objective-C, Dart, Swift, Scala, 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

- 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 identifiers in one module will not be accidentally confused with a procedure or variable sharing the same 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 contain 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.

> Object-oriented programming is more than just classes and objects; it's a whole programming paradigm based around *objects* (data structures) that contain data fields and methods. It is essential to understand this; using classes to organize a bunch of unrelated methods together is not object orientation.
>
> Junade Ali, *Mastering PHP Design Patterns*[8]

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.

Class-based vs prototype-based

In class-based languages the *classes* are defined beforehand and the *objects* are instantiated based on the classes. If two objects *apple* and *orange* are instantiated from the class *Fruit*, they are inherently fruits and it is guaranteed that you may handle them in the same way; e.g. a programmer can expect the existence of the same attributes such as *color* or *sugar content* or *is ripe*.

In prototype-based languages the *objects* are the primary entities. No *classes* even exist. The *prototype* of an object is just another object to which the object is linked. Every object has one *prototype* link (and only one). New objects can be created based on already existing objects chosen as their prototype. You may call two different objects *apple* and *orange* a fruit, if the object *fruit* exists, and both *apple* and *orange* have *fruit* as their prototype. The idea of the *fruit* class doesn't exist explicitly, but as the equivalence class of the objects sharing the same prototype. The attributes and methods of the *prototype* are delegated to all the objects of the equivalence class defined by this prototype. The attributes and methods *owned* individually by the object may not be shared by other objects of the same equivalence class; e.g. the attributes *sugar content* may be unexpectedly not present in *apple*. Only single inheritance can be implemented through the prototype.

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 the call variability relies on more than the single type of the object on which it is called (i.e. at least one other parameter object is involved in the method choice), one speaks of 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 (either directly or through a pointer) 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 access to a place to store an Address object (either directly embedded within itself, or at a separate location addressed via a pointer).

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.[9]

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. Meanwhile, the same operation name among objects in an inheritance hierarchy may behave differently.

For example, objects of type Circle and Square are derived from a common class called Shape. The Draw function for each type of Shape implements what is necessary to draw itself while calling code can remain indifferent to the particular type of Shape is being drawn.

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.

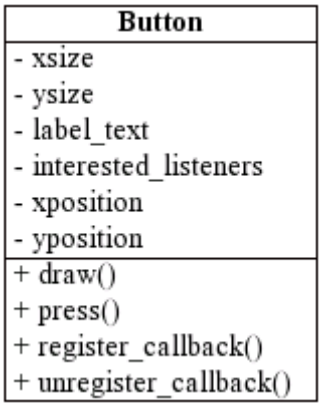
History

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);[10][11] Alan Kay was later to cite a detailed understanding of LISP internals as a strong influence on his thinking in 1966.[12]

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.[13] 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".[14][15]

In the 1960s, object-orientated programming was put into practice with the Simula language, which introduced important concepts that are today an essential part of object-orientated programming, such as class and object, inheritance, and dynamic binding.[16] Simula was also designed to take account of programming and data security. For programming security purposes a detection process was implemented so that through reference counts a last resort garbage collector deleted unused objects in the random-

access memory (RAM). But although the idea of data objects had already been established by 1965, data encapsulation through levels of scope for variables, such as private (-) and public (+), were not implemented in Simula because it would have required the accessing procedures to be also hidden.[17]



UML notation for a class. This Button class has variables for data, and functions. Through inheritance a subclass can be created as subset of the Button class. Objects are instances of a class.

In 1962, Kristen Nygaard initiated a project for a simulation language at the Norwegian Computing Center, based on his previous use of the Monte Carlo simulation and his work to conceptualise real-world systems. Ole-Johan Dahl formally joined the project and the Simula programming language was designed to run on the Universal Automatic Computer (UNIVAC) 1107. In the early stages Simula was supposed to be a procedure package for the programming language ALGOL 60. Dissatisfied with the restrictions imposed by ALGOL the researchers decided to develop Simula into a fully-fledged programming language, which used the UNIVAC ALGOL 60 compiler. Simula launched in 1964, and was promoted by Dahl and Nygaard throughout 1965 and 1966, leading to increasing use of the programming language in Sweden, Germany and the Soviet Union. In 1968, the language became widely available through the Burroughs B5500 computers, and was later also implemented on the URAL-16 computer. In 1966, Dahl and Nygaard wrote a Simula compiler. They became preoccupied with putting into practice Tony Hoare's record class concept, which had been implemented in the free-form, English-like general-purpose simulation language SIMSCRIPT. They settled for a generalised process concept with record class properties, and a second layer of prefixes. Through prefixing a process could reference its predecessor and have additional properties. Simula thus introduced the class and subclass hierarchy, and the possibility of generating objects from these classes. The Simula 1 compiler and a new version of the programming language, Simula 67, was introduced to the wider world through the research paper "Class and Subclass Declarations" at a 1967 conference.[18]

> I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages (so messaging came at the very beginning -- it took a while to see how to do messaging in a programming language efficiently enough to be useful).
>
> Alan Kay, [12]

A Simula 67 compiler was launched for the System/360 and System/370 IBM mainframe computers in 1972. In the same year a Simula 67 compiler was launched free of charge for the French CII 10070 and CII Iris 80 mainframe computers. By 1974, the Association of Simula Users had members in 23 different countries. Early 1975 a Simula 67 compiler was released free of charge for the DecSystem-10 mainframe family. By August the same year the DecSystem Simula 67 compiler had been installed at 28 sites, 22 of them in North America. The object-orientated Simula programming language was used mainly by researchers involved with physical modelling, such as models to study and improve the movement of ships and their content through cargo ports.[19]

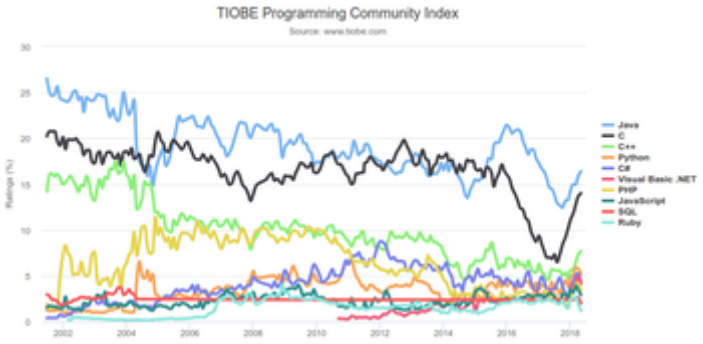
In the 1970s, the first version of the Smalltalk programming language was developed at Xerox PARC by Alan Kay, Dan Ingalls and Adele Goldberg. Smaltalk-72 included a programming environment and was dynamically typed, and at first was interpreted, not compiled. Smalltalk got noted for its application of object orientation at the language level and its graphical development environment. Smalltalk went through various versions and interest in the language grew.[20] While Smalltalk was influenced by the ideas introduced in Simula 67 it was designed to be a fully dynamic system in which classes could be created and modified dynamically.[21]

In the 1970s, Smalltalk 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 1981, Goldberg edited the August 1981 issue of Byte Magazine, introducing Smalltalk and object-orientated programming to a wider audience. In 1986, the Association for Computing Machinery organised the first *Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), which was unexpectedly attended by 1,000 people. In the mid-1980s Objective-C was developed by Brad Cox, who had used Smalltalk at ITT Inc., and Bjarne Stroustrup, who had used Simula for his PhD thesis, eventually went to create the object-orientated C++.[20] In 1985, Bertrand Meyer also produced the first design of the Eiffel language. Focused on software quality, Eiffel is a purely object-oriented programming language and 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.

In the early and mid-1990s object-oriented programming developed as the dominant programming paradigm 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).



The TIOBE programming language popularity index graph from 2002 to 2018. In the 2000s the object-orientated Java (blue) and the procedural C (black) competed for the top position.

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 another early example, and the one with which much of the theory of OOP 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, code point 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 approaches. 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 pattern 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 1994 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 have 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]

Leo Brodie has suggested a connection between the standalone nature of objects and a tendency to duplicate code[47] in violation of the don't repeat yourself principle[48] of software development.

Steve Yegge noted that, as opposed to functional programming:[49]

> 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.[50] Raymond compares this unfavourably to the approach taken with Unix and the C programming language.[50]

Rob Pike, a programmer involved in the creation of UTF-8 and Go, has called object-oriented programming "the Roman numerals of computing"[51] and has said that OOP languages frequently shift the focus from data structures and algorithms to types.[52] 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.[53]

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[54]
- 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;[55] 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)[55] 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 978-0-321-53205-3., 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 978-0-262-16209-8., 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* (https://en.wikiquote.org/wiki/Grady_Booch). 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. Ali, Junade (2016-09-28). *Mastering PHP Design Patterns | PACKT Books* (https://www.packtpub.com/application-development/mastering-php-design-patterns) (1 ed.). Birmingham, England, UK: Packt Publishing Limited. p. 11. ISBN 978-1-78588-713-0. Retrieved 11 December 2017.
9. Jacobsen, Ivar; Magnus Christerson; Patrik Jonsson; Gunnar Overgaard (1992). *Object Oriented Software Engineering*. Addison-Wesley ACM Press. pp. 43–69. ISBN 978-0-201-54435-0.
10. McCarthy, J.; Brayton, R.; Edwards, D.; Fox, P.; Hodes, L.; Luckham, D.; Maling, K.; Park, D.; Russell, S. (March 1960). "LISP I Programmers Manual" (https://web.archive.org/web/20100717111134/http://history.siam.org/sup/Fox_1960_LISP.pdf) (PDF). Boston, Massachusetts: Artificial Intelligence Group, M.I.T. Computation Center and Research Laboratory: 88f. Archived from the original (http://history.siam.org/sup/Fox_1960_LISP.pdf) (PDF) on 17 July 2010. "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"."
11. McCarthy, John; Abrahams, Paul W.; Edwards, Daniel J.; Hart, swapnil d.; Levin, Michael I. (1962). *LISP 1.5 Programmer's Manual* (https://web.archive.org/web/20080516210916/http://community.computerhistory.org/scc/projects/LISP/book/LISP%201.5%20Programmers%20Manual.pdf) (PDF). MIT Press. p. 105. ISBN 978-0-262-13011-0. Archived from the original (http://community.computerhistory.org/scc/projects/LISP/book/LISP%201.5%20Programmers%20Manual.pdf) (PDF) on 16 May 2008. "Object — a synonym for atomic symbol"
12. "Dr. Alan Kay on the Meaning of "Object-Oriented Programming"" (http://www.purl.org/stefan_ram/pub/doc_kay_oop_en). 2003. Retrieved 11 February 2010.
13. Sutherland, I. E. (30 January 1963). "Sketchpad: A Man-Machine Graphical Communication System" (http://handle.dtic.mil/100.2/AD404549) (PDF). Technical Report No. 296, Lincoln Laboratory, Massachusetts Institute of Technology via Defense Technical Information Center (stinet.dtic.mil). Retrieved 3 November 2007.
14. 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)
15. Ross, Doug. "The first software engineering language" (http://www.csail.mit.edu/timeline/timeline.php?query=event&id=19). *LCS/AI Lab Timeline:*. MIT Computer Science and Artificial Intelligence Laboratory. Retrieved 13 May 2010.
16. Holmevik, Jan Rune (1994). "Compiling Simula: A historical study of technological genesis" (http://www.idi.ntnu.no/grupper/su/publ/simula/holmevik-simula-ieeeannals94.pdf) (PDF). *IEEE Annals of the History of Computing*. **16** (4): 25–37. doi:10.1109/85.329756 (https://doi.org/10.1109%2F85.329756). Retrieved 3 March 2018.
17. Dahl, Ole Johan (2004). "The Birth of Object Orientation: The Simula Languages". *From Object-Orientation to Formal Methods* (http://www.mn.uio.no/ifi/english/about/ole-johan-dahl/bibliography/the-birth-of-object-orientation-the-simula-languages.pdf) (PDF). Lecture Notes in Computer Science. **2635**. pp. 15–25. CiteSeerX 10.1.1.133.6730 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.133.6730). doi:10.1007/978-3-540-39993-3_3 (https://doi.org/10.1007%2F978-3-540-39993-3_3). ISBN 978-3-540-21366-6. Retrieved 3 March 2018.
18. Holmevik, Jan Rune (1994). "Compiling Simula: A historical study of technological genesis" (http://www.idi.ntnu.no/grupper/su/publ/simula/holmevik-simula-ieeeannals94.pdf) (PDF). *IEEE Annals of the History of Computing*. **16** (4): 25–37. doi:10.1109/85.329756 (https://doi.org/10.1109%2F85.329756). Retrieved 3 March 2018.
19. Holmevik, Jan Rune (1994). "Compiling Simula: A historical study of technological genesis" (http://www.idi.ntnu.no/grupper/su/publ/simula/holmevik-simula-ieeeannals94.pdf) (PDF). *IEEE Annals of the History of Computing*. **16** (4): 25–37. doi:10.1109/85.329756 (https://doi.org/10.1109%2F85.329756). Retrieved 3 March 2018.
20. Bertrand Meyer (2009). *Touch of Class: Learning to Program Well with Objects and Contracts*. Springer Science & Business Media. p. 329. ISBN 9783540921448.

21. Kay, Alan. "The Early History of Smalltalk" (https://web.archive.org/web/20080710144930/http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html). Archived from the original (http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html) 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. Khurana, Rohit (2009-11-01). *Object Oriented Programming with C++, 1E* (https://books.google.co.uk/books?id=MHmqfSBTXsAC&pg=PA16&lpg=PA16). ISBN 9788125925323.
26. "The Emerald Programming Language" (http://www.emeraldprogramminglanguage.org/). 2011-02-26.
27. Neward, Ted (26 June 2006). "The Vietnam of Computer Science" (https://web.archive.org/web/20060704030226/http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx). Interoperability Happens. Archived from the original (http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx) on 4 July 2006. 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" (https://pdfs.semanticscholar.org/10bd/dc49b85196aaa6715dd46843d9dcffa38358.pdf) (PDF). *Computer*. **39** (1): 28–39. doi:10.1109/mc.2006.20 (https://doi.org/10.1109%2Fmc.2006.20). Retrieved 2016-10-02.
31. Yegge, Steve (30 March 2006). "Execution in the Kingdom of Nouns" (http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html). steve-yegge.blogspot.com. Retrieved 3 July 2010.
32. Boronczyk, Timothy (11 June 2009). "What's Wrong with OOP" (http://zaemis.blogspot.com/2009/06/whats-wrong-with-oop.html). zaemis.blogspot.com. Retrieved 3 July 2010.
33. Ambler, Scott (1 January 1998). "A Realistic Look at Object-Oriented Reuse" (http://www.drdobbs.com/184415594). www.drdobbs.com. Retrieved 4 July 2010.
34. Shelly, Asaf (22 August 2008). "Flaws of Object Oriented Modeling" (http://software.intel.com/en-us/blogs/2008/08/22/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" (https://web.archive.org/web/20071010105117/http://blogs.techrepublic.com.com/programming-and-development/?p=518). techrepublic.com. Archived from the original (http://blogs.techrepublic.com.com/programming-and-development/?p=518) on 10 October 2007. Retrieved 4 July 2010.
36. Shelly, Asaf (22 August 2008). "HOW TO: Multicore Programming (Multiprocessing) Visual C++ Class Design Guidelines, Member Functions" (http://support.microsoft.com/?scid=kb%3Ben-us%3B558117). support.microsoft.com. Retrieved 4 July 2010.
37. Robert Harper (17 April 2011). "Some thoughts on teaching FP" (http://existentialtype.wordpress.com/2011/04/17/some-advice-on-teaching-fp/). Existential Type Blog. Retrieved 5 December 2011.
38. Cardelli, Luca (1996). "Bad Engineering Properties of Object-Oriented Languages" (http://lucacardelli.name/Papers/BadPropertiesOfOO.html). *ACM Comput. Surv*. **28** (4es): 150–es. doi:10.1145/242224.242415 (https://doi.org/10.1145%2F242224.242415). ISSN 0360-0300 (https://www.worldcat.org/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" (http://www.stlport.org/resources/StepanovUSA.html). 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" (http://www.csm.ornl.gov/~v8q/Homepage/Papers%20Old/spetep-%20printable.pdf) (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 (https://doi.org/10.1002%2F%28SICI%291097-024X%28199908%2929%3A10%3C833%3A%3AAID-SPE258%3E3.0.CO%3B2-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" (http://www.smashcompany.com/technology/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" (http://www.paulgraham.com/noop.html). PaulGraham.com. Retrieved 13 November 2009.
47. Brodie, Leo (1984). *Thinking Forth* (http://thinking-forth.sourceforge.net/thinking-forth-ans.pdf) (PDF). pp. 92–93. Retrieved 4 May 2018.
48. Hunt, Andrew. "Don't Repeat Yourself" (http://wiki.c2.com/?DontRepeatYourself). *Category Extreme Programming*. Retrieved 4 May 2018.
49. Stevey's Blog Rants (http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html)
50. Eric S. Raymond (2003). "The Art of Unix Programming: Unix and Object-Oriented Languages" (http://www.catb.org/esr/writings/taoup/html/unix_and_oo.html). Retrieved 2014-08-06.
51. Pike, Rob (2004-03-02). "[9fans] Re: Threads: Sewing badges of honor onto a Kernel" (http://groups.google.com/group/comp.os.plan9/msg/006fec195aeeff15). *comp.os.plan9* (Mailing list). Retrieved 2016-11-17.
52. Pike, Rob (2012-06-25). "Less is exponentially more" (https://commandcenter.blogspot.com/2012/06/less-is-exponentially-more.html). Retrieved 2016-10-01.
53. Pike, Rob (2012-11-14). "A few years ago I saw this page" (https://plus.google.com/+RobPikeTheHuman/posts/hoJdanihKwb). Retrieved 2016-10-01.
54. Poll, Erik. "Subtyping and Inheritance for Categorical Datatypes" (http://www.cs.ru.nl/E.Poll/papers/kyoto97.pdf) (PDF). Retrieved 5 June 2011.
55. Abadi, Martin; Cardelli, Luca (1996). *A Theory of Objects* (http://portal.acm.org/citation.cfm?id=547964&dl=ACM&coll=portal). Springer-Verlag New York, Inc. ISBN 978-0-387-94775-4. Retrieved 21 April 2010.

Further reading

- Abadi, Martin; Luca Cardelli (1998). *A Theory of Objects*. Springer Verlag. ISBN 978-0-387-94775-4.
- Abelson, Harold; Gerald Jay Sussman (1997). *Structure and Interpretation of Computer Programs* (http://mitpress.mit.edu/sicp/). MIT Press. ISBN 978-0-262-01153-2.
- Armstrong, Deborah J. (February 2006). "The Quarks of Object-Oriented Development" (http://portal.acm.org/citation.cfm?id=1113040). *Communications of the ACM*. **49** (2): 123–128. Bibcode:1985CACM...28...22S (http://adsabs.harvard.edu/abs/1985CACM...28...22S). doi:10.1145/1113034.1113040 (https://doi.org/10.1145%2F1113034.1113040). ISSN 0001-0782 (https://www.worldcat.org/issn/0001-0782). Retrieved 8 August 2006.
- Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley. ISBN 978-0-8053-5340-2.
- Eeles, Peter; Oliver Sims (1998). *Building Business Objects*. John Wiley & Sons. ISBN 978-0-471-19176-6.
- Gamma, Erich; Richard Helm; Ralph Johnson; John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley. ISBN 978-0-201-63361-0.
- Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook – Lessons from Award-Winning Business Applications*. John Wiley & Sons. ISBN 978-0-471-14717-6.
- Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley. ISBN 978-0-201-54435-0.
- Kay, Alan. *The Early History of Smalltalk* (https://web.archive.org/web/20050404075821/http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html#). Archived from the original (http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html) on 4 April 2005. Retrieved 18 April 2005.
- Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall. ISBN 978-0-13-629155-8.
- Pecinovsky, Rudolf (2013). *OOP – Learn Object Oriented Thinking & Programming* (http://pub.bruckner.cz/titles/oop). 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 978-0-13-629841-0.
- Schach, Stephen (2006). *Object-Oriented and Classical Software Engineering, Seventh Edition*. McGraw-Hill. ISBN 978-0-07-319126-3.
- Schreiner, Axel-Tobias (1993). *Object oriented programming with ANSI-C*. Hanser. hdl:1850/8544 (https://hdl.handle.net/1850%2F8544). ISBN 978-3-446-17426-9.
- Taylor, David A. (1992). *Object-Oriented Information Systems – Planning and Implementation*. John Wiley & Sons. ISBN 978-0-471-54364-0.
- Weisfeld, Matt (2009). *The Object-Oriented Thought Process, Third Edition*. Addison-Wesley. ISBN 978-0-672-33016-2.
- West, David (2004). *Object Thinking (Developer Reference)*. Microsoft Press. ISBN 978-0735619654.

External links

- Object-oriented programming (https://curlie.org/Computers/Programming/Methodologies/Object-Oriented) at Curlie
- 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)
