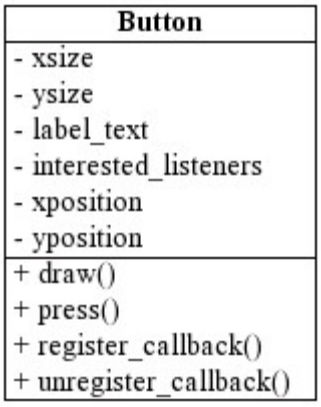
# SUBJECT-Object-oriented programming SYSTEM

- Bca –iind yr



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

**Object-oriented programming** (**OOP**) is a programming paradigm based on the concept of *objects*,[1] which can contain data and code: data in the form of fields (often known as attributes or properties), and code in the form of procedures (often known as methods). In OOP, computer programs are designed by making them out of objects that interact with one another.[2][3]

Many of the most widely used programming languages (such as C++, Java,[4] and Python) are multi-paradigm and support object-oriented programming to a greater or lesser degree, typically in combination with imperative programming, procedural programming and functional programming.

Significant object-oriented languages include Ada, ActionScript, C++, Common Lisp, C#, Dart, Eiffel, Fortran 2003, Haxe, Java,[4] JavaScript, Kotlin, Logo, MATLAB, Objective-C, Object Pascal, Perl, PHP, Python, R, Raku, Ruby, Scala, SIMSCRIPT, Simula, Smalltalk, Swift, Vala and Visual Basic.NET.

# History

Terminology invoking "objects" in the modern sense of object-oriented programming made its first appearance at the artificial intelligence group at MIT in the late 1950s and early 1960s. "Object" referred to LISP atoms with identified properties (attributes).[5][6] Another early MIT example was Sketchpad created by Ivan Sutherland in 1960–1961; 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.[7] Also, in 1968, 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".[8][9] Topics such as data abstraction and modular programming were common points of discussion at this time.

Independently of later MIT work such as AED, Simula was developed during the years 1961–1967.[8] Simula introduced important concepts that are today an essential part of object-oriented programming, such as class and object, inheritance, and dynamic binding.[10] The object-oriented 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.[10]

> 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, [1]

Influenced by the work at MIT and the Simula language, in November 1966 Alan Kay began working on ideas that would eventually be incorporated into the Smalltalk programming language. Kay used the term "object-oriented programming" in conversation as early as 1967.[1] Although sometimes called "the father of object-oriented programming",[11] Alan Kay has differentiated his notion of OO from the more conventional abstract data type notion of object, and has implied that the computer science establishment did not adopt his notion.[1] A 1976 MIT memo co-authored by Barbara Liskov lists Simula 67, CLU, and Alphard as object-oriented languages, but does not mention Smalltalk.[12]

In the 1970s, the first version of the Smalltalk programming language was developed at Xerox PARC by Alan Kay, Dan Ingalls and Adele Goldberg. Smalltalk-72 included a programming environment and was dynamically typed, and at first was interpreted, not compiled. Smalltalk became 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.[13] 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.[14]

During the late 1970s and 1980s, object-oriented programming rose to prominence. The Flavors object-oriented Lisp was developed starting 1979, introducing multiple inheritance and mixins.[15] In 1981, Goldberg edited the August issue of Byte Magazine, introducing Smalltalk and object-oriented programming to a wide audience.[16] LOOPS, the object system for Interlisp-D, was influenced by Smalltalk and Flavors, and a paper about it was published in 1982.[17] In 1986, the Association for Computing Machinery organized the first *Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), which was attended by 1,000 people. Among other developments was 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 the mid-1980s Objective-C was developed by Brad Cox, who had used Smalltalk at ITT Inc.. Bjarne Stroustrup, who had used Simula for his PhD thesis, created the object-oriented C++.[13] 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.[18] 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,[19][20] C++,[21] and Delphi[citation needed]. 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 investigated the concept of type checking across module boundaries. Modula-2 (1978) included this concept, and their succeeding design, Oberon (1987), included a distinctive approach to object orientation, classes, and such. Inheritance is not obvious in Wirth's design since his nomenclature looks in the opposite direction: It is called type extension and the viewpoint is from the parent down to the inheritor.

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, some 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 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.

# 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 common among languages considered to be strongly class- and object-oriented (or multi-paradigm with OOP support), with notable exceptions mentioned.[22][23][24][25] 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.[26]

## Shared with non-OOP 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 conflict with a procedure or variable sharing the same name in another file or module.

## Objects

An object is a data structure or abstract data type containing fields (state variables containing data) and methods (subroutines or procedures defining the object's behavior in code). Fields may also be known as members, attributes, or properties. Objects are typically stored as contiguous regions of memory. Objects are accessed somewhat like variables with complex internal structures, 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.

Objects sometimes correspond to things found in the real world.[27] For example, a graphics program may have objects such as "circle", "square", and "menu". An online shopping system might have objects such as "shopping cart", "customer", and "product". 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.

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). Date and Darwen have proposed a theoretical foundation that uses OOP as a kind of customizable type system to support RDBMS, but it forbids object pointers.[28]

The OOP paradigm has been criticized for overemphasizing the use of objects for software design and modeling at the expense of other important aspects (computation/algorithms).[29][30] For example, Rob Pike has said that OOP languages frequently shift the focus from data structures and algorithms to types.[31] Steve Yegge noted that, as opposed to functional programming:[32]

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.[30]

Alexander Stepanov compares object orientation unfavourably to generic programming:[29]

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.

## Inheritance

OOP languages are diverse, but typically OOP languages allow inheritance for code reuse and extensibility in the form of either classes or prototypes. These forms of inheritance are significantly different, but analogous terminology is used to define the concepts of *object* and *instance*.

### Class-based

In class-based programming, the most popular style, each object is required to be an instance of a particular *class*. The class defines the data format or type (including member variables and their types) and available procedures (class methods or member functions) for a given type or class of object. Objects are created by calling a special type of method in the class known as a constructor. Classes may inherit from other classes, so they are 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". It is guaranteed that all instances of class Employee will have the same variables, such as the name, position, and salary. Procedures and variables can be specific to either the class or the instance; this leads to the following terms:

- Class variables – belong to the *class as a whole*; there is only one copy of each variable, shared across all instances of the class
- Instance variables or attributes – data that belongs to individual *objects*; every object has its own copy of each one. All 4 variables mentioned above (first_name, position etc) are instance variables.
- 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 to only 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

Depending on the definition of the language, subclasses may or may not be able to 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 other concepts like traits and 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 do not share a common parent.

Abstract classes cannot be instantiated into objects; they exist only for inheritance into other "concrete" classes that can be instantiated. In Java, the `final` keyword can be used to prevent a class from being subclassed.[33]

### Prototype-based

In contrast, in prototype-based programming, *objects* are the primary entities. Generally, the concept of a "class" does not even exist. Rather, the *prototype* or *parent* of an object is just another object to which the object is linked. In Self, an object may have multiple or no parents,[34] but in the most popular prototype-based language, Javascript, 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 does not exist explicitly, but can be modeled as the equivalence class of the objects sharing the same prototype, or as the set of objects satisfying a certain interface (duck typing). Unlike class-based programming, it is typically possible in prototype-based languages to define attributes and methods not shared with other objects; for example, the attribute *sugar_content* may be defined in *apple* but not *orange*.

### Absence

Some languages like Go do not support inheritance at all. Go states that it is object-oriented,[35] and Bjarne Stroustrup, author of C++, has stated that it is possible to do OOP without inheritance.[36] 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. Delegation is another language feature that can be used as an alternative to inheritance.

Rob Pike has criticized the OO mindset for preferring a multilevel type hierarchy with layered abstractions to a three-line lookup table.[37] He has called object-oriented programming "the Roman numerals of computing".[38]

Bob Martin states that because they are software, related classes do not necessarily share the relationships of the things they represent.[39]

## 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. 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.

Dispatch interacts with inheritance; if a method is not present in a given object or class, the dispatch is delegated to its parent object or class, and so on, going up the chain of inheritance.

## Data abstraction and encapsulation

Data abstraction is a design pattern in which data are visible only to semantically related functions, to prevent misuse. The success of data abstraction leads to frequent incorporation of data hiding as a design principle in object-oriented and pure functional programming. Similarly, 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.

In object oriented programming, objects provide a layer which can be used to separate internal from external code and implement abstraction and encapsulation. External code can only use an object by calling a specific instance method with a certain set of input parameters, reading an instance variable, or writing to an instance variable. A program may create many instances of objects as it runs, which operate independently. This technique, it is claimed, allows easy re-use of the same procedures and data definitions for different sets of data, in addition to potentially mirroring real-world relationships intuitively. Rather than utilizing database tables and programming subroutines, the developer utilizes objects the user may be more familiar with: objects from their application domain.[40] These claims that the OOP paradigm enhances reusability and modularity have been criticized.[41][42]

The initial design is encouraged to use the most restrictive visibility possible, in order of local (or method) variables, private variables (in object oriented programming), and global (or public) variables, and only be expanded when and as much as necessary. This prevents changes to visibility from invalidating existing code.[43]

If a class does not allow calling code to access internal object data and permits access through methods only, this is also a form of information hiding. 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.[44] 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).[44] In other languages (like Python) this is enforced only by convention (for example, `private` methods may have names that start with an underscore). In C#, Swift & Kotlin languages, `internal` keyword permits access only to files present in the same assembly, package, or module as that of the class.[45]

In programming languages, particularly object-oriented ones, the emphasis on abstraction is vital. Object-oriented languages extend the notion of type to incorporate data abstraction, highlighting the significance of restricting access to internal data through methods.[46] Eric S. Raymond has written that object-oriented programming languages tend to encourage thickly layered programs that destroy transparency.[47] Raymond compares this unfavourably to the approach taken with Unix and the C programming language.[47]

The "open/closed principle" advocates that classes and functions "should be open for extension, but closed for modification". Luca Cardelli has claimed that OOP languages have "extremely poor modularity properties with respect to class extension and modification", and tend to be extremely complex.[41] The latter point is reiterated by Joe Armstrong, the principal inventor of Erlang, who is quoted as saying:[42]

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.

Leo Brodie has suggested a connection between the standalone nature of objects and a tendency to duplicate code[48] in violation of the don't repeat yourself principle[49] of software development.

### Polymorphism

Subtyping – a form of polymorphism – is when calling code can be independent of which class in the supported hierarchy it is operating on – the 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 the 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 being drawn.

This is another type of abstraction that simplifies code external to the class hierarchy and enables strong separation of concerns.

### Open recursion

A common feature of objects is that methods are attached to them and can access and modify the object's data fields. In this brand of OOP, there is usually a special name such as `this` or `self` used to refer to the current object. In languages that support open recursion, object methods can call other methods on the same object (including themselves) using this name. 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.

## 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: Ruby, Scala, Smalltalk, Eiffel, Emerald,[50] JADE, Self, Raku.
- Languages designed mainly for OO programming, but with some procedural elements.
  Examples: Java, Python, C++, C#, Delphi/Object Pascal, VB.NET.
- Languages that are historically procedural languages, but have been extended with some OO features. Examples: PHP, JavaScript, 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.

## Popularity and reception


The TIOBE programming language popularity index graph from 2002 to 2023. In the 2000s the object-oriented Java (orange) and the procedural C (dark blue) competed for the top position.

Many widely used languages, such as C++, Java, and Python, provide object-oriented features. Although in the past object-oriented programming was widely accepted,[51] more recently essays criticizing object-oriented programming and recommending the avoidance of these features (generally in favor of functional programming) have been very popular in the developer community.[52] 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".[53] 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".[47]

Richard Feldman argues that these languages may have improved their modularity by adding OO features, but they became popular for reasons other than being object-oriented.[54] 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.[55] A study by Potok et al. has shown no significant difference in productivity between OOP and procedural approaches.[56] Luca Cardelli has claimed that OOP code is "intrinsically less efficient" than procedural code and that OOP can take longer to compile.[41]

## 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 can 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 IBM i 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

One way to address challenges of object-oriented design is via design patterns which are solution patterns to commonly occurring problems in software design. Some of these commonly occurring problems have implications and solutions particular to object-oriented development.

## Object patterns

The following are notable software design patterns for OOP objects.[57]

- Function object: with a single method (in C++, the function operator, `operator()`) it acts much like a function
- Immutable object: does not change state after creation
- First-class object: can be used without restriction
- Container object: contains other objects
- Factory object: creates other objects
- Metaobject: from which other objects can be created (compare with a class, which is not necessarily an object)
- Prototype object: a specialized metaobject from which other objects can be created by copying
- Singleton object: only instance of its class for the lifetime of the program
- Filter object: receives a stream of data as its input and transforms it into the object's output

As an example of an object anti-pattern, the God object knows or does too much.

## 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

*Main article: Design pattern (computer science)*

*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.

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

*Main articles: Object-relational impedance mismatch, Object-relational mapping, and Object database*

Both object-oriented programming and relational database management systems (RDBMSs) are extremely common in software today. Since relational databases do not 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 some approaches to cope with this problem, but no general solution without downsides.[58] 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 or that real-world mapping is even a worthy goal; Bertrand Meyer argues in *Object-Oriented Software Construction* 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".[59] At the same time, some principal limitations of OOP have been noted.[60] 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 is therefore well suited to model complex systems with complex behavior"[61] (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).[62] This problem may cause OOP to suffer more convoluted solutions than procedural programming.[63]

### OOP and control flow

OOP was developed to increase the reusability and maintainability of source code.[64] 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.

### 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 which spells out five software engineering design principles:

- 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.

## 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[69]
- recursive types
- encapsulated state
- inheritance
- records are the 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;[70] 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*[70] 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").

## Systems

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

## Modeling languages

- IDEF4
- Interface description language
- UML