

# Design Patterns

## Thanks to WikiPedia

# Contents

## Articles

Design pattern (computer science)	1
Abstract factory pattern	9
Builder pattern	13
Factory method pattern	17
Lazy initialization	23
Multiton pattern	32
Object pool pattern	38
Prototype pattern	40
Singleton pattern	43
Structural pattern	48
Adapter pattern	49
Bridge pattern	53
Composite pattern	56
Decorator pattern	60
Facade pattern	69
Flyweight pattern	72
Proxy pattern	75
Chain-of-responsibility pattern	78
Command pattern	84
Interpreter pattern	90
Iterator pattern	94
Mediator pattern	97
Memento pattern	101
Null Object pattern	103
Observer pattern	107
Design pattern Servant	112
Specification pattern	116
State pattern	121
Strategy pattern	125
Template method pattern	128
Visitor pattern	132
Active object	138
Balking pattern	138
Messaging pattern	140

Double-checked locking	141
Asynchronous method invocation	146
Guarded suspension	148
Lock (computer science)	149
Monitor (synchronization)	153
Reactor pattern	164
Readers-writer lock	165
Scheduler pattern	167
Thread pool pattern	167
Thread-local storage	169

## References

Article Sources and Contributors	173
Image Sources, Licenses and Contributors	176

## Article Licenses

License	177
---------	-----

# Design pattern (computer science)

---

In software engineering, a **design pattern** is a general reusable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Many patterns imply object-orientation or more generally mutable state, and so may not be as applicable in functional programming languages, in which data is immutable or treated as such.

Design patterns reside in the domain of modules and interconnections. At a higher level there are architectural patterns that are larger in scope, usually describing an overall pattern followed by an entire system.<sup>[1]</sup>

There are many types of design patterns including: structural design patterns, computational design patterns, algorithm strategy patterns, implementation strategy patterns and execution patterns. Structural patterns address concerns related to the high level structure of an application being developed. Computational patterns address concerns related to the identification of key computations. Algorithm strategy patterns address concerns related to high level strategies that describe how to exploit application characteristic on a computation platform. Implementation strategy patterns address concerns related to the realization of the source code to support (i) how the program itself is organized and (ii) the common data structures specific to parallel programming. Execution patterns address concerns related to the support of the execution of an application, including the strategies in executing streams of tasks and building blocks to support the synchronization between tasks.

## History

Patterns originated as an architectural concept by Christopher Alexander (1977/79). In 1987, Kent Beck and Ward Cunningham began experimenting with the idea of applying patterns to programming and presented their results at the OOPSLA conference that year.<sup>[2] [3]</sup> In the following years, Beck, Cunningham and others followed up on this work.

Design patterns gained popularity in computer science after the book *Design Patterns: Elements of Reusable Object-Oriented Software* was published in 1994 by the so-called "Gang of Four" (Gamma et al.). That same year, the first Pattern Languages of Programming Conference was held and the following year, the Portland Pattern Repository was set up for documentation of design patterns. The scope of the term remains a matter of dispute. Notable books in the design pattern genre include:

- Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.
- Buschmann, Frank; Regine Meunier, Hans Rohnert, Peter Sommerlad (1996). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons. ISBN 0-471-95869-7.
- Schmidt, Douglas C.; Michael Stal, Hans Rohnert, Frank Buschmann (2000). *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons. ISBN 0-471-60695-2.
- Fowler, Martin (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley. ISBN 978-0321127426.
- Hohpe, Gregor; Bobby Woolf (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley. ISBN 0-321-20068-3.
- Freeman, Eric T; Elisabeth Robson, Bert Bates, Kathy Sierra (2004). *Head First Design Patterns*. O'Reilly Media. ISBN 0-596-00712-4.

Although design patterns have been applied practically for a long time, formalization of the concept of design patterns languished for several years.<sup>[4]</sup>

In 2009 over 30 contributors collaborated with Thomas Erl on his book, *SOA Design Patterns*<sup>[5]</sup>. The goal of this book was to establish a de facto catalog of design patterns for SOA and service-orientation<sup>[6]</sup>. (Over 200+ IT professionals participated world-wide in reviewing Erl's book and patterns.) These patterns are also published and discussed on the community research site [soapatterns.org](http://soapatterns.org)<sup>[7]</sup>

## Practice

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems, and it also improves code readability for coders and architects who are familiar with the patterns.

In order to achieve flexibility, design patterns usually introduce additional levels of indirection, which in some cases may complicate the resulting designs and hurt application performance.

By definition, a pattern must be programmed anew into each application that uses it. Since some authors see this as a step backward from software reuse as provided by components, researchers have worked to turn patterns into components. Meyer and Arnout were able to provide full or partial componentization of two-thirds of the patterns they attempted.<sup>[8]</sup>

Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that does not require specifics tied to a particular problem.

## Structure

Design patterns are composed of several sections (see Documentation below). Of particular interest are the Structure, Participants, and Collaboration sections. These sections describe a *design motif*: a prototypical *micro-architecture* that developers copy and adapt to their particular designs to solve the recurrent problem described by the design pattern. A micro-architecture is a set of program constituents (e.g., classes, methods...) and their relationships. Developers use the design pattern by introducing in their designs this prototypical micro-architecture, which means that micro-architectures in their designs will have structure and organization similar to the chosen design motif.

In addition to this, patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than *ad-hoc* designs.

## Domain-specific patterns

Efforts have also been made to codify design patterns in particular domains, including use of existing design patterns as well as domain specific design patterns. Examples include user interface design patterns,<sup>[9]</sup> information visualization<sup>[10]</sup>, secure design<sup>[11]</sup>, "secure usability"<sup>[12]</sup>, web design<sup>[13]</sup> and business model design.<sup>[14]</sup>

The annual Pattern Languages of Programming Conference proceedings<sup>[15]</sup> include many examples of domain specific patterns.

---

## Classification and list

Design patterns were originally grouped into the categories: creational patterns, structural patterns, and behavioral patterns, and described using the concepts of delegation, aggregation, and consultation. For further background on object-oriented design, see coupling and cohesion, inheritance, interface, and polymorphism. Another classification has also introduced the notion of architectural design pattern that may be applied at the architecture level of the software such as the Model-View-Controller pattern.

Name	Description	In Design Patterns	In Code Complete <sup>[16]</sup>	Other
<b>Creational patterns</b>				
Abstract factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.	Yes	Yes	N/A
Builder	Separate the construction of a complex object from its representation allowing the same construction process to create various representations.	Yes	No	N/A
Factory method	Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.	Yes	Yes	N/A
Lazy initialization	Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed.	No	No	PoEAA <sup>[17]</sup>
Multiton	Ensure a class has only named instances, and provide global point of access to them.	No	No	N/A
Object pool	Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalisation of connection pool and thread pool patterns.	No	No	N/A
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.	Yes	No	N/A
Resource acquisition is initialization	Ensure that resources are properly released by tying them to the lifespan of suitable objects.	No	No	N/A
Singleton	Ensure a class has only one instance, and provide a global point of access to it.	Yes	Yes	N/A
<b>Structural patterns</b>				
Adapter or Wrapper	Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.	Yes	Yes	N/A
Bridge	Decouple an abstraction from its implementation allowing the two to vary independently.	Yes	Yes	N/A
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.	Yes	Yes	N/A
Decorator	Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.	Yes	Yes	N/A
Facade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.	Yes	Yes	N/A
Front Controller	Provide a unified interface to a set of interfaces in a subsystem. Front Controller defines a higher-level interface that makes the subsystem easier to use.	No	Yes	N/A
Flyweight	Use sharing to support large numbers of fine-grained objects efficiently.	Yes	No	N/A

Proxy	Provide a surrogate or placeholder for another object to control access to it.	Yes	No	N/A
<b>Behavioral patterns</b>				
Blackboard	Generalized observer, which allows multiple readers and writers. Communicates information system-wide.	No	No	N/A
Chain of responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.	Yes	No	N/A
Command	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.	Yes	No	N/A
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.	Yes	No	N/A
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.	Yes	Yes	N/A
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.	Yes	No	N/A
Memento	Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later.	Yes	No	N/A
Null object	Avoid null references by providing a default object.	No	No	N/A
Observer or Publish/subscribe	Define a one-to-many dependency between objects where a state change in one object results with all its dependents being notified and updated automatically.	Yes	Yes	N/A
Servant	Define common functionality for a group of classes	No	No	N/A
Specification	Recombinable business logic in a boolean fashion	No	No	N/A
State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.	Yes	No	N/A
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.	Yes	Yes	N/A
Template method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.	Yes	Yes	N/A
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.	Yes	No	N/A

Name	Description	In <i>POSA2</i> <sup>[18]</sup>	Other
<b>Concurrency patterns</b>			
Active Object	Decouples method execution from method invocation that reside in their own thread of control. The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests.	Yes	N/A
Balking	Only execute an action on an object when the object is in a particular state.	No	N/A
Binding Properties	Combining multiple observers to force properties in different objects to be synchronized or coordinated in some way. <sup>[19]</sup>	No	N/A
Messaging pattern	The messaging design pattern (MDP) allows the interchange of information (i.e. messages) between components and applications.	No	N/A
Double-checked locking	Reduce the overhead of acquiring a lock by first testing the locking criterion (the 'lock hint') in an unsafe manner; only if that succeeds does the actual lock proceed. Can be unsafe when implemented in some language/hardware combinations. It can therefore sometimes be considered an anti-pattern.	Yes	N/A
Event-based asynchronous	Addresses problems with the Asynchronous pattern that occur in multithreaded programs. <sup>[20]</sup>	No	N/A
Guarded suspension	Manages operations that require both a lock to be acquired and a precondition to be satisfied before the operation can be executed.	No	N/A
Lock	One thread puts a "lock" on a resource, preventing other threads from accessing or modifying it. <sup>[21]</sup>	No	<i>PoEAA</i> <sup>[17]</sup>
Monitor object	An object whose methods are subject to mutual exclusion, thus preventing multiple objects from erroneously trying to use it at the same time.	Yes	N/A
Reactor	A reactor object provides an asynchronous interface to resources that must be handled synchronously.	Yes	N/A
Read-write lock	Allows concurrent read access to an object but requires exclusive access for write operations.	No	N/A
Scheduler	Explicitly control when threads may execute single-threaded code.	No	N/A
Thread pool	A number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads. Can be considered a special case of the object pool pattern.	No	N/A
Thread-specific storage	Static or "global" memory local to a thread.	Yes	N/A

## Documentation

The documentation for a design pattern describes the context in which the pattern is used, the forces within the context that the pattern seeks to resolve, and the suggested solution.<sup>[22]</sup> There is no single, standard format for documenting design patterns. Rather, a variety of different formats have been used by different pattern authors. However, according to Martin Fowler certain pattern forms have become more well-known than others, and consequently become common starting points for new pattern writing efforts.<sup>[23]</sup> One example of a commonly used documentation format is the one used by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (collectively known as the "Gang of Four", or GoF for short) in their book *Design Patterns*. It contains the following sections:

- **Pattern Name and Classification:** A descriptive and unique name that helps in identifying and referring to the pattern.
- **Intent:** A description of the goal behind the pattern and the reason for using it.



- **Also Known As:** Other names for the pattern.
- **Motivation (Forces):** A scenario consisting of a problem and a context in which this pattern can be used.
- **Applicability:** Situations in which this pattern is usable; the context for the pattern.
- **Structure:** A graphical representation of the pattern. Class diagrams and Interaction diagrams may be used for this purpose.
- **Participants:** A listing of the classes and objects used in the pattern and their roles in the design.
- **Collaboration:** A description of how classes and objects used in the pattern interact with each other.
- **Consequences:** A description of the results, side effects, and trade offs caused by using the pattern.
- **Implementation:** A description of an implementation of the pattern; the solution part of the pattern.
- **Sample Code:** An illustration of how the pattern can be used in a programming language.
- **Known Uses:** Examples of real usages of the pattern.
- **Related Patterns:** Other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

## References

- [1] Martin, Robert C.. "Design Principles and Design Patterns" ([http://www.objectmentor.com/resources/articles/Principles\\_and\\_Patterns.pdf](http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf)). . Retrieved 2000.
- [2] Smith, Reid (October 1987). "Panel on design methodology". *OOPSLA '87 Addendum to the Proceedings*. OOPSLA '87. doi:10.1145/62138.62151. . *"Ward cautioned against requiring too much programming at, what he termed, 'the high level of wizards.' He pointed out that a written 'pattern language' can significantly improve the selection and application of abstractions. He proposed a 'radical shift in the burden of design and implementation' basing the new methodology on an adaptation of Christopher Alexander's work in pattern languages and that programming-oriented pattern languages developed at Tektronix has significantly aided their software development efforts."*
- [3] Beck, Kent; Ward Cunningham (September 1987). "Using Pattern Languages for Object-Oriented Program" (<http://c2.com/doc/oopsla87.html>). *OOPSLA '87 workshop on Specification and Design for Object-Oriented Programming*. OOPSLA '87. . Retrieved 2006-05-26.
- [4] Baroni, Aline Lúcia; Yann-Gaël Guéhéneuc and Hervé Albin-Amiot (June 2003). "Design Patterns Formalization" (<http://www.iro.umontreal.ca/~ptidej/Publications/Documents/Research+report+Metamodeling+June03.doc.pdf>) (PDF). Nantes: École Nationale Supérieure des Techniques Industrielles et des Mines de Nantes. . Retrieved 2007-12-29.
- [5] Erl, Thomas (2009). *SOA Design Patterns*. New York: Prentice Hall/PearsonPTR. pp. 864. ISBN 0-13-613516-1.
- [6] <http://soa.sys-con.com/node/809800>
- [7] <http://www.soapatterns.org>
- [8] Meyer, Bertrand; Karine Arnout (July 2006). "Componentization: The Visitor Example" (<http://se.ethz.ch/~meyer/publications/computer/visitor.pdf>). *IEEE Computer* (IEEE) **39** (7): 23–30. .
- [9] Laakso, Sari A. (2003-09-16). "Collection of User Interface Design Patterns" (<http://www.cs.helsinki.fi/u/salaakso/patterns/index.html>). University of Helsinki, Dept. of Computer Science. . Retrieved 2008-01-31.
- [10] Heer, J.; M. Agrawala (2006). "Software Design Patterns for Information Visualization" ([http://vis.berkeley.edu/papers/infovis\\_design\\_patterns/](http://vis.berkeley.edu/papers/infovis_design_patterns/)). *IEEE Transactions on Visualization and Computer Graphics* **12** (5): 853. doi:10.1109/TVCG.2006.178. PMID 17080809. .
- [11] Chad Dougherty et al (2009). *Secure Design Patterns* (<http://www.cert.org/archive/pdf/09tr010.pdf>). .
- [12] Simson L. Garfinkel (2005). *Design Principles and Patterns for Computer Systems That Are Simultaneously Secure and Usable* (<http://www.simson.net/thesis/>). .
- [13] "Yahoo! Design Pattern Library" (<http://developer.yahoo.com/ypatterns/>). . Retrieved 2008-01-31.
- [14] "How to design your Business Model as a Lean Startup?" (<http://torgronsund.wordpress.com/2010/01/06/lean-startup-business-model-pattern/>). . Retrieved 2010-01-06.
- [15] Pattern Languages of Programming, Conference proceedings (annual, 1994—) (<http://hillside.net/plop/pastconferences.html>)
- [16] McConnell, Steve (June 2004). "Design in Construction". *Code Complete* (2nd ed.). Microsoft Press. pp. 104. ISBN 978-0735619678. "Table 5.1 Popular Design Patterns"
- [17] Fowler, Martin (2002). *Patterns of Enterprise Application Architecture* (<http://martinfowler.com/>). Addison-Wesley. ISBN 978-0321127426. .
- [18] Schmidt, Douglas C.; Michael Stal, Hans Rohnert, Frank Buschmann (2000). *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons. ISBN 0-471-60695-2.
- [19] <http://c2.com/cgi/wiki?BindingProperties>
- [20] Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, and Morgan Skinner (2008). "Event-based Asynchronous Pattern". *Professional C# 2008*. Wiley. pp. 570–571. ISBN 0470191376.
- [21] <http://c2.com/cgi/wiki?LockPattern>

- [22] Gabriel, Dick. "A Pattern Definition" (<http://web.archive.org/web/20070209224120/http://hillside.net/patterns/definition.html>). Archived from the original (<http://hillside.net/patterns/definition.html>) on 2007-02-09. . Retrieved 2007-03-06.
- [23] Fowler, Martin (2006-08-01). "Writing Software Patterns" (<http://www.martinfowler.com/articles/writingPatterns.html>). . Retrieved 2007-03-06.

## Further reading

### Books

- Alexander, Christopher; Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel (1977). *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press. ISBN 978-0195019193.
- Alur, Deepak; John Crupi, Dan Malks (May 2003). *Core J2EE Patterns: Best Practices and Design Strategies (2nd Edition)*. Prentice Hall. ISBN 0131422464.
- Beck, Kent (October 2007). *Implementation Patterns*. Addison-Wesley. ISBN 978-0321413093.
- Beck, Kent; R. Crocker, G. Meszaros, J.O. Coplien, L. Dominick, F. Paulisch, and J. Vlissides (March 1996). *Proceedings of the 18th International Conference on Software Engineering*. pp. 25–30.
- Borchers, Jan (2001). *A Pattern Approach to Interaction Design*. John Wiley & Sons. ISBN 0-471-49828-9.
- Coplien, James O.; Douglas C. Schmidt (1995). *Pattern Languages of Program Design*. Addison-Wesley. ISBN 0-201-60734-4.
- Coplien, James O.; John M. Vlissides, and Norman L. Kerth (1996). *Pattern Languages of Program Design 2*. Addison-Wesley. ISBN 0-201-89527-7.
- Fowler, Martin (1997). *Analysis Patterns: Reusable Object Models*. Addison-Wesley. ISBN 0-201-89542-0.
- Fowler, Martin (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley. ISBN 978-0321127426.
- Freeman, Eric; Elisabeth Freeman, Kathy Sierra, and Bert Bates (2004). *Head First Design Patterns*. O'Reilly Media. ISBN 0-596-00712-4.
- Hohmann, Luke; Martin Fowler and Guy Kawasaki (2003). *Beyond Software Architecture*. Addison-Wesley. ISBN 0-201-77594-8.
- Alur, Deepak; Elisabeth Freeman, Kathy Sierra, and Bert Bates (2004). *Head First Design Patterns*. O'Reilly Media. ISBN 0-596-00712-4.
- Gabriel, Richard (1996) (PDF). *Patterns of Software: Tales From The Software Community* (<http://www.dreamsongs.com/NewFiles/PatternsOfSoftware.pdf>). Oxford University Press. pp. 235. ISBN 0-19-512123-6.
- Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.
- Hohpe, Gregor; Bobby Woolf (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley. ISBN 0-321-20068-3.
- Holub, Allen (2004). *Holub on Patterns*. Apress. ISBN 1-59059-388-X.
- Kircher, Michael; Markus Völter and Uwe Zdun (2005). *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*. John Wiley & Sons. ISBN 0-470-85662-9.
- Larman, Craig (2005). *Applying UML and Patterns*. Prentice Hall. ISBN 0-13-148906-2.
- Liskov, Barbara; John Guttag (2000). *Program Development in Java: Abstraction, Specification, and Object-Oriented Design..* Addison-Wesley. ISBN 0-201-65768-6.
- Manolescu, Dragos; Markus Voelter and James Noble (2006). *Pattern Languages of Program Design 5*. Addison-Wesley. ISBN 0-321-32194-4.
- Marinescu, Floyd (2002). *EJB Design Patterns: Advanced Patterns, Processes and Idioms*. John Wiley & Sons. ISBN 0-471-20831-0.
- Martin, Robert Cecil; Dirk Riehle and Frank Buschmann (1997). *Pattern Languages of Program Design 3*. Addison-Wesley. ISBN 0-201-31011-2.

- Mattson, Timothy G; Beverly A. Sanders and Berna L. Massingill (2005). *Patterns for Parallel Programming*. Addison-Wesley. ISBN 0-321-22811-1.
- Shalloway, Alan; James R. Trott (2001). *Design Patterns Explained, Second Edition: A New Perspective on Object-Oriented Design*. Addison-Wesley. ISBN 0-321-24714-0.
- Vlissides, John M. (1998). *Pattern Hatching: Design Patterns Applied*. Addison-Wesley. ISBN 0-201-43293-5.
- Weir, Charles; James Noble (2000). *Small Memory Software: Patterns for systems with limited memory* (<http://www.cix.co.uk/~smallmemory/>). Addison-Wesley. ISBN 0201596075.

#### Web sites

- "History of Patterns" (<http://www.c2.com/cgi-bin/wiki?HistoryOfPatterns>). *Portland Pattern Repository*. Retrieved 2005-07-28.
- "Are Design Patterns Missing Language Features?" (<http://www.c2.com/cgi/wiki?AreDesignPatternsMissingLanguageFeatures>). Cunningham & Cunningham, Inc.. Retrieved 2006-01-20.
- "Show Trial of the Gang of Four" (<http://www.c2.com/cgi/wiki?ShowTrialOfTheGangOfFour>). Cunningham & Cunningham, Inc.. Retrieved 2006-01-20.
- "Design Patterns in Modern Day Software Factories (WCSF)" (<http://www.xoftware.co.uk/Articles/WCSFDesignPatterns/>). XO Software, Ltd. Retrieved 2010-02-22.

## External links

- Directory of websites that provide pattern catalogs (<http://hillside.net/patterns/onlinepatterncatalog.htm>) at hillside.net.
- Ward Cunningham's *Portland Pattern Repository*.
- Messaging Design Pattern (<http://jt.dev.java.net/files/documents/5553/150311/designPatterns.pdf>) Published in the 17th conference on Pattern Languages of Programs (PLoP 2010).
- Patterns and Anti-Patterns ([http://www.dmoz.org/Computers/Programming/Methodologies/Patterns\\_and\\_Anti-Patterns/](http://www.dmoz.org/Computers/Programming/Methodologies/Patterns_and_Anti-Patterns/)) at the Open Directory Project
- PerfectJPattern Open Source Project (<http://perfectjpattern.sourceforge.net>) Design Patterns library that aims to provide full or partial componentized version of all known Patterns in Java.
- Lean Startup Business Model Pattern (<http://torgronsund.wordpress.com/2010/01/06/lean-startup-business-model-pattern/>) Example of design pattern thinking applied to business models
- Jt (<http://jt.dev.java.net>) J2EE Pattern Oriented Framework
- Printable Design Patterns Quick Reference Cards (<http://www.mcdonaldland.info/2007/11/28/40/>)
- 101 Design Patterns & Tips for Developers (<http://sourcemaking.com/design-patterns-and-tips>)
- On Patterns and Pattern Languages ([http://media.wiley.com/product\\_data/excerpt/28/04700590/0470059028.pdf](http://media.wiley.com/product_data/excerpt/28/04700590/0470059028.pdf)) by Buschmann, Henney, and Schmidt
- Patterns for Scripted Applications (<http://www.doc.ic.ac.uk/~np2/patterns/scripting/>)
- Design Patterns Reference (<http://www.oodeesign.com/>) at oodeesign.com
- Design Patterns for 70% programmers in the world (<http://www.slideshare.net/saurabh.net/design-patterns-for-70-of-programmers-in-the-world>) by Saurabh Verma at slideshare.com

# Abstract factory pattern

---

The **abstract factory pattern** is a software design pattern that provides a way to encapsulate a group of individual factories that have a common theme. In normal usage, the client software creates a concrete implementation of the abstract factory and then uses the generic interfaces to create the concrete objects that are part of the theme. The client does not know (or care) which concrete objects it gets from each of these internal factories since it uses only the generic interfaces of their products. This pattern separates the details of implementation of a set of objects from their general usage.

An example of this would be an abstract factory class *DocumentCreator* that provides interfaces to create a number of products (e.g. *createLetter()* and *createResume()*). The system would have any number of derived concrete versions of the *DocumentCreator* class like *FancyDocumentCreator* or *ModernDocumentCreator*, each with a different implementation of *createLetter()* and *createResume()* that would create a corresponding object like *FancyLetter* or *ModernResume*. Each of these products is derived from a simple abstract class like *Letter* or *Resume* of which the client is aware. The client code would get an appropriate instance of the *DocumentCreator* and call its factory methods. Each of the resulting objects would be created from the same *DocumentCreator* implementation and would share a common theme (they would all be fancy or modern objects). The client would need to know how to handle only the abstract *Letter* or *Resume* class, not the specific version that it got from the concrete factory.

In software development, a **Factory** is the location in the code at which objects are constructed. The intent in employing the pattern is to insulate the creation of objects from their usage. This allows for new derived types to be introduced with no change to the code that uses the base class.

Use of this pattern makes it possible to interchange concrete classes without changing the code that uses them, even at runtime. However, employment of this pattern, as with similar design patterns, may result in unnecessary complexity and extra work in the initial writing of code. Used correctly the "extra work" pays off in the second instance of using the Factory.

## Usage

The *factory* determines the actual *concrete* type of object to be created, and it is here that the object is actually created (in C++, for instance, by the **new** operator). However, the factory only returns an *abstract* pointer to the created concrete object.

This insulates client code from object creation by having clients ask a factory object to create an object of the desired abstract type and to return an abstract pointer to the object.

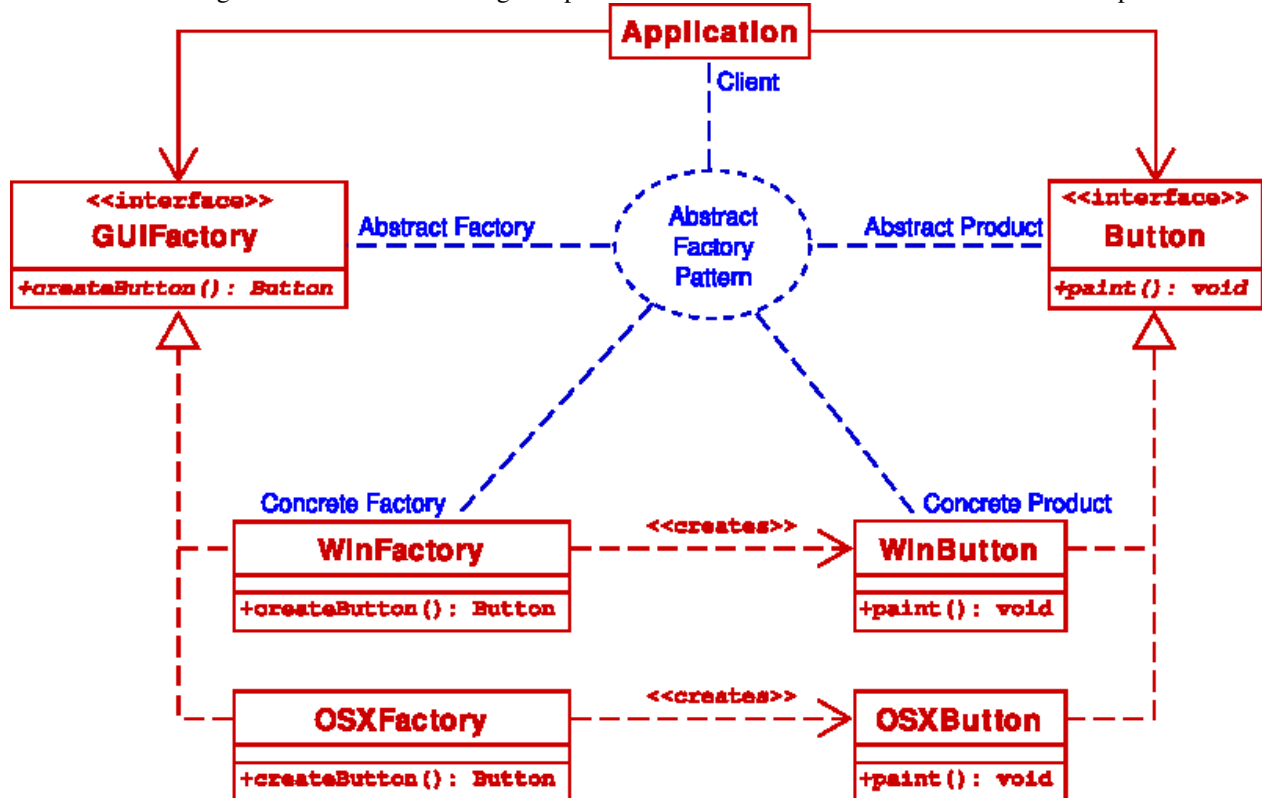
As the factory only returns an abstract pointer, the client code (which requested the object from the factory) does not know – and is not burdened by – the actual concrete type of the object which was just created. However, the type of a concrete object (and hence a concrete factory) is known by the abstract factory; for instance, the factory may read it from a configuration file. The client has no need to specify the type, since it has already been specified in the configuration file. In particular, this means:

- The client code has no knowledge whatsoever of the concrete type, not needing to include any header files or class declarations relating to the concrete type. The client code deals only with the abstract type. Objects of a concrete type are indeed created by the factory, but the client code accesses such objects only through their abstract interface.
- Adding new concrete types is done by modifying the client code to use a different factory, a modification which is typically one line in one file. (The different factory then creates objects of a *different* concrete type, but still returns a pointer of the *same* abstract type as before – thus insulating the client code from change.) This is significantly easier than modifying the client code to instantiate a new type, which would require changing *every* location in the code where a new object is created (as well as making sure that all such code locations also have

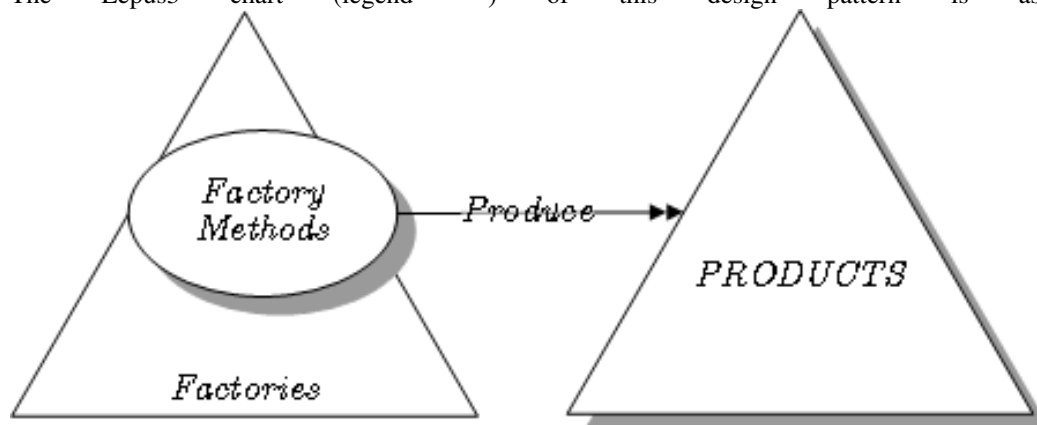
knowledge of the new concrete type, by including for instance a concrete class header file). If all factory objects are stored globally in a singleton object, and all client code goes through the singleton to access the proper factory for object creation, then changing factories is as easy as changing the singleton object.

## Structure

The class diagram of this design pattern as demonstrated in the example below:



The Lepus3 chart (legend <sup>[1]</sup>) of this design pattern is as shown here:



This class diagram does not emphasize the usage of abstract factory pattern in creating families of related or non related objects.

## Example

The output should be either "I'm a WinButton" or "I'm an OSXButton" depending on which kind of factory was used. Note that the Application has no idea what kind of GUIFactory it is given or even what kind of Button that factory creates.

### Java

```
/* GUIFactory example -- */

interface GUIFactory {
    public Button createButton();
}

class WinFactory implements GUIFactory {
    public Button createButton() {
        return new WinButton();
    }
}

class OSXFactory implements GUIFactory {
    public Button createButton() {
        return new OSXButton();
    }
}

interface Button {
    public void paint();
}

class WinButton implements Button {
    public void paint() {
        System.out.println("I'm a WinButton");
    }
}

class OSXButton implements Button {
    public void paint() {
        System.out.println("I'm an OSXButton");
    }
}
```

```
class Application {
    public Application(GUIFactory factory) {
        Button button = factory.createButton();
        button.paint();
    }
}

public class ApplicationRunner {
    public static void main(String[] args) {
        new Application(createOsSpecificFactory());
    }

    public static GUIFactory createOsSpecificFactory() {
        int sys = readFromConfigFile("OS_TYPE");
        if (sys == 0) {
            return new WinFactory();
        } else {
            return new OSXFactory();
        }
    }
}
```

## External links

- Abstract Factory <sup>[2]</sup> in Source Making
- Abstract Factory <sup>[3]</sup> in the C2 wiki
- Abstract Factory <sup>[4]</sup> on the Net Objectives Repository
- Abstract Factory <sup>[5]</sup> UML diagram + formal specification in LePUS3 and Class-Z (a Design Description Language)
- C++ Abstract Factory <sup>[6]</sup>, design and tutorial
- Abstract Factory <sup>[7]</sup> in data & object factory

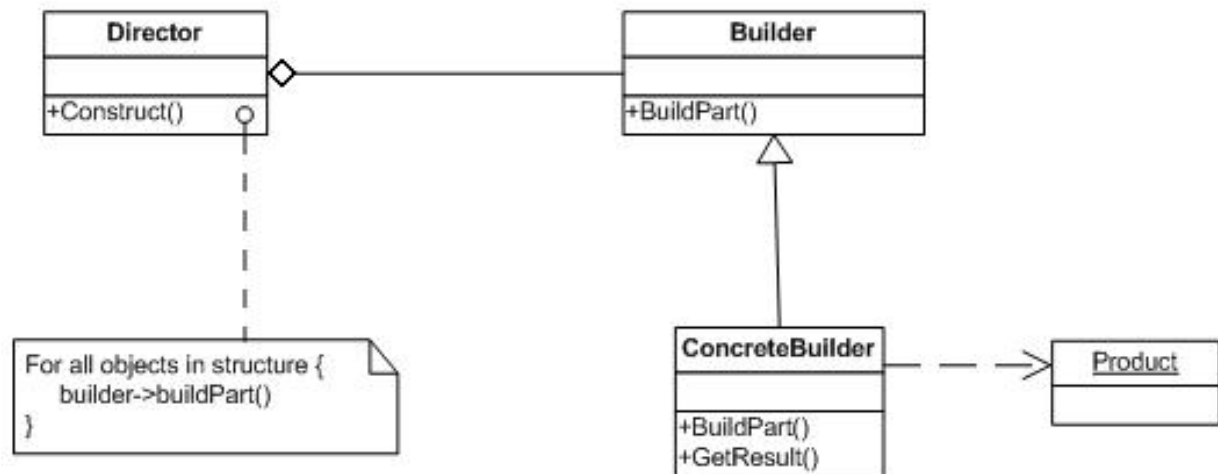
## References

- [1] <http://www.lepus.org.uk/ref/legend/legend.xml>
- [2] [http://sourcemaking.com/design\\_patterns/abstract\\_factory](http://sourcemaking.com/design_patterns/abstract_factory)
- [3] <http://c2.com/cgi-bin/wiki?AbstractFactory>
- [4] <http://www.netobjectivesrepository.com/TheAbstractFactoryPattern>
- [5] <http://www.lepus.org.uk/ref/companion/AbstractFactory.xml>
- [6] <http://kent.dl.sourceforge.net/sourceforge/papafactory/PapaFactory-9.06.pdf>
- [7] <http://www.dofactory.com/Patterns/PatternAbstract.aspx>

# Builder pattern

The **builder pattern** is a software design pattern. The intention is to abstract steps of construction of objects so that different implementations of these steps can construct different representations of objects. Often, the builder pattern is used to build products in accordance to the composite pattern, a structural pattern.

## Class diagram



### Builder

Abstract interface for creating objects (product).

### Concrete Builder

Provides implementation for Builder. It is an object able to construct other objects. Constructs and assembles parts to build the objects.

### Director

The Director class is responsible for managing the correct sequence of object creation. It receives a Concrete Builder as a parameter and executes the necessary operations on it.

### Product

The final object that will be created by the Director using Builder.

## Useful tips

- Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.
- Builder often builds a Composite.
- Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.
- Sometimes creational patterns are complementary: Builder can use develop building like an object and also using senglal and cement one of the other patterns to implement which components are built. Abstract Factory, Builder, and Prototype can use Singleton in their implementations.



## Example

An example in Java:

```
/** "Product" */
class Pizza {
    private String dough = "";
    private String sauce = "";
    private String topping = "";

    public void setDough(String dough) {
        this.dough = dough;
    }

    public void setSauce(String sauce) {
        this.sauce = sauce;
    }

    public void setTopping(String topping) {
        this.topping = topping;
    }
}

/** "Abstract Builder" */
abstract class PizzaBuilder {
    protected Pizza pizza;

    public Pizza getPizza() {
        return pizza;
    }

    public void createNewPizzaProduct() {
        pizza = new Pizza();
    }

    public abstract void buildDough();

    public abstract void buildSauce();

    public abstract void buildTopping();
}

/** "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder {
    public void buildDough() {
        pizza.setDough("cross");
    }

    public void buildSauce() {
```

```
        pizza.setSauce("mild");
    }

    public void buildTopping() {
        pizza.setTopping("ham+pineapple");
    }
}

/** "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder {
    public void buildDough() {
        pizza.setDough("pan baked");
    }

    public void buildSauce() {
        pizza.setSauce("hot");
    }

    public void buildTopping() {
        pizza.setTopping("pepperoni+salami");
    }
}

/** "Director" */
class Cook {
    private PizzaBuilder pizzaBuilder;

    public void setPizzaBuilder(PizzaBuilder pb) {
        pizzaBuilder = pb;
    }

    public Pizza getPizza() {
        return pizzaBuilder.getPizza();
    }

    public void constructPizza() {
        pizzaBuilder.createNewPizzaProduct();
        pizzaBuilder.buildDough();
        pizzaBuilder.buildSauce();
        pizzaBuilder.buildTopping();
    }
}

/** A given type of pizza being constructed. */
public class BuilderExample {
    public static void main(String[] args) {
        Cook cook = new Cook();
        PizzaBuilder hawaiianPizzaBuilder = new
```

```
HawaiianPizzaBuilder();

    PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();

    cook.setPizzaBuilder(hawaiianPizzaBuilder);
    cook.constructPizza();

    Pizza hawaiian = cook.getPizza();

    cook.setPizzaBuilder(spicyPizzaBuilder);
    cook.constructPizza();

    Pizza spicy = cook.getPizza();

}
}
```

## External links

- The JavaWorld article Build user interfaces without getters and setters <sup>[1]</sup> (Allen Holub) shows the complete Java source code for a Builder.
- Item 2: Consider a builder <sup>[2]</sup> by Joshua Bloch

## References

[1] <http://www.javaworld.com/javaworld/jw-01-2004/jw-0102-toolbox.html>

[2] <http://www.ddj.com/java/208403883?pgno=2>

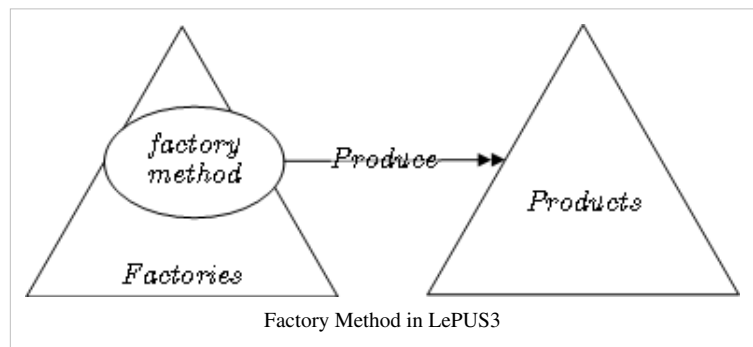
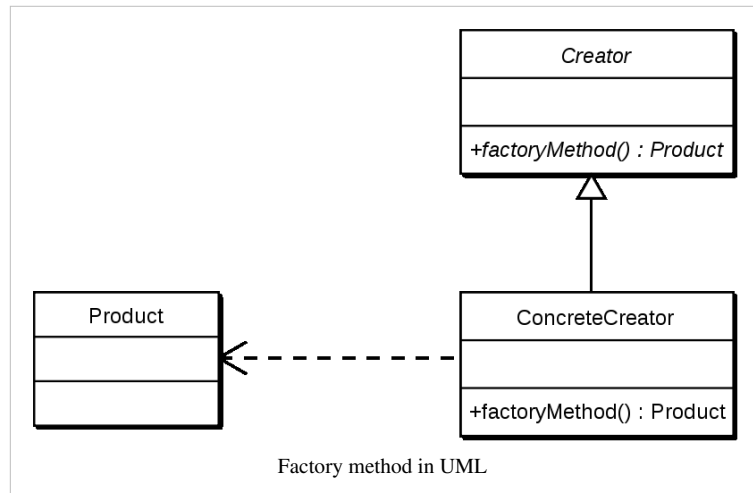
# Factory method pattern

The **factory method pattern** is an object-oriented design pattern to implement the concept of factories.

Like other creational patterns, it deals with the problem of creating objects (products) without specifying the exact class of object that will be created. The creation of an object often requires complex processes not appropriate to include within a composing object. The object's creation may lead to a significant duplication of code, may require information not accessible to the composing object, may not provide a sufficient level of abstraction, or may otherwise not be part of the composing object's concerns. The factory method design pattern handles these problems by defining a separate method for creating the objects, which subclasses can then override to specify the derived type of product that will be created.

Some of the processes required in the creation of an object include determining which object to create, managing the lifetime of the object, and managing specialized build-up and tear-down concerns of the object.

Outside the scope of design patterns, the term *factory method* can also refer to a method of a factory whose main purpose is creation of objects.



## Definition

The essence of the Factory method Pattern is to "Define an interface for creating an object, but let the subclasses decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses."<sup>[1]</sup>

## Concept

In object-oriented computer programming, a **factory** is an object for creating other objects. It is an abstraction of a constructor, and can be used to implement various allocation schemes. For example, using this definition, singletons implemented by the singleton pattern are formal factories.

A factory object typically has a method for every kind of object it is capable of creating. These methods optionally accept parameters defining how the object is created, and then return the created object.

Factory objects are used in situations where getting hold of an object of a particular kind is a more complex process than simply creating a new object. The factory object might decide to create the object's class (if applicable) dynamically, return it from an object pool, do complex configuration on the object, or other things.

These kinds of objects have proven useful and several design patterns have been developed to implement them in any given language. For example, several "GoF patterns", like the "Factory method pattern", the "Builder" or even

the "Singleton" are implementations of this concept. The "Abstract factory pattern" instead is a method to build collections of factories.

## Common usage

Factory methods are common in toolkits and frameworks where library code needs to create objects of types which may be subclassed by applications using the framework.

Parallel class hierarchies often require objects from one hierarchy to be able to create appropriate objects from another.

Factory methods are used in test-driven development to allow classes to be put under test.<sup>[2]</sup> If such a class `Foo` creates another object `Dangerous` that can't be put under automated unit tests (perhaps it communicates with a production database that isn't always available), then the creation of `Dangerous` objects is placed in the virtual factory method `createDangerous` in class `Foo`. For testing, `TestFoo` (a subclass of `Foo`) is then created, with the virtual factory method `createDangerous` overridden to create and return `FakeDangerous`, a fake object. Unit tests then use `TestFoo` to test the functionality of `Foo` without incurring the side effect of using a real `Dangerous` object.

## Applicability

Use the factory pattern when:

- The creation of the object precludes reuse without significantly duplicating code.
- The creation of the object requires access to information or resources not appropriate to contain within the composing object.
- The lifetime management of created objects needs to be centralised to ensure consistent behavior.

## Other benefits and variants

Although the motivation behind the factory method pattern is to allow subclasses to choose which type of object to create, there are other benefits to using factory methods, many of which do not depend on subclassing. Therefore, it is common to define "factory methods" that are not polymorphic to create objects in order to gain these other benefits. Such methods are often static.

## Descriptive names

A factory method has a distinct name. In many object-oriented languages, constructors must have the same name as the class they are in, which can lead to ambiguity if there is more than one way to create an object (see overloading). Factory methods have no such constraint and can have descriptive names. As an example, when complex numbers are created from two real numbers the real numbers can be interpreted as Cartesian or polar coordinates, but using factory methods, the meaning is clear (the following examples are in Java):

```
class Complex
{
    public static Complex fromCartesian(double real, double imaginary)
    {
        return new Complex(real, imaginary);
    }

    public static Complex fromPolar(double modulus, double angle)
    {
        return new Complex(modulus * cos(angle), modulus * sin(angle));
    }
}
```

```
    }

    private Complex(double a, double b)
    {
        //...
    }
}

Complex c = Complex.fromPolar(1, pi);
```

When factory methods are used for disambiguation like this, the constructor is often made private to force clients to use the factory methods.

## Encapsulation

Factory methods encapsulate the creation of objects. This can be useful if the creation process is very complex, for example if it depends on settings in configuration files or on user input.

Consider as an example a program to read image files and make thumbnails out of them. The program supports different image formats, represented by a reader class for each format:

```
public interface ImageReader
{
    public DecodedImage getDecodedImage();
}

public class GifReader implements ImageReader
{
    public DecodedImage getDecodedImage()
    {
        // ...
        return decodedImage;
    }
}

public class JpegReader implements ImageReader
{
    public DecodedImage getDecodedImage()
    {
        // ...
        return decodedImage;
    }
}
```

Each time the program reads an image it needs to create a reader of the appropriate type based on some information in the file. This logic can be encapsulated in a factory method:

```
public class ImageReaderFactory
{
    public static ImageReader getImageReader(InputStream is)
    {
```

```
int imageType = determineImageType(is);

switch(imageType)
{
    case ImageReaderFactory.GIF:
        return new GifReader(is);
    case ImageReaderFactory.JPEG:
        return new JpegReader(is);
    // etc.
}
}
```

The code fragment in the previous example uses a switch statement to associate an imageType with a specific factory object. Alternatively, this association could also be implemented as a mapping. This would allow the switch statement to be replaced with an associative array lookup.

## Example Implementations

### Java

A maze game may be played in two modes, one with regular rooms that are only connected with adjacent rooms, and one with magic rooms that allow players to be transported at random (this Java example is similar to one in the book *Design Patterns*). The regular game mode could use this template method:

```
public class MazeGame {
    public MazeGame() {
        Room room1 = makeRoom();
        Room room2 = makeRoom();
        room1.connect(room2);
        this.addRoom(room1);
        this.addRoom(room2);
    }

    protected Room makeRoom() {
        return new OrdinaryRoom();
    }
}
```

In the above snippet, makeRoom is a template method. It encapsulates the creation of rooms such that other rooms can be used in a subclass. To implement the other game mode that has magic rooms, it suffices to override the makeRoom method:

```
public class MagicMazeGame extends MazeGame {
    protected Room makeRoom() {
        return new MagicRoom();
    }
}
```

## PHP

```
class Factory
{
    public static function build($type)
    {
        $class = 'Format' . $type;
        if (!class_exists($class)) {
            throw new Exception('Missing format class.');
```

```
        }
        return new $class;
    }
}

class FormatString {}
class FormatNumber {}

try {
    $string = Factory::build('String');
}
catch (Exception $e) {
    echo $e->getMessage();
}

try {
    $number = Factory::build('Number');
}
catch (Exception $e) {
    echo $e->getMessage();
}
```

## Limitations

There are three limitations associated with the use of the factory method. The first relates to refactoring existing code; the other two relate to extending a class.

- The first limitation is that refactoring an existing class to use factories breaks existing clients. For example, if class `Complex` was a standard class, it might have numerous clients with code like:

```
Complex c = new Complex(-1, 0);
```

Once we realize that two different factories are needed, we change the class (to the code shown earlier). But since the constructor is now private, the existing client code no longer compiles.

- The second limitation is that, since the pattern relies on using a private constructor, the class cannot be extended. Any subclass must invoke the inherited constructor, but this cannot be done if that constructor is private.
- The third limitation is that, if we do extend the class (e.g., by making the constructor protected—this is risky but feasible), the subclass must provide its own re-implementation of all factory methods with exactly the same signatures. For example, if class `StrangeComplex` extends `Complex`, then unless `StrangeComplex` provides its own version of all factory methods, the call `StrangeComplex.fromPolar(1, pi)` will yield an instance of *Complex* (the superclass) rather than the expected instance of the subclass.



All three problems could be alleviated by altering the underlying programming language to make factories first-class class members (see also Virtual class).<sup>[3]</sup>

## Uses

- In ADO.NET, IDbCommand.CreateCommand<sup>[4]</sup> is an example of the use of factory method to connect parallel class hierarchies.
- In Qt, QMainWindow::createPopupMenu<sup>[5]</sup> is a factory method declared in a framework which can be overridden in application code.
- In Java, several factories are used in the javax.xml.parsers<sup>[6]</sup> package. e.g. javax.xml.parsers.DocumentBuilderFactory or javax.xml.parsers.SAXParserFactory.

## References

- [1] Gang Of Four
- [2] Feathers, Michael (October 2004), *Working Effectively with Legacy Code*, Upper Saddle River, NJ: Prentice Hall Professional Technical Reference, ISBN 978-0131177055
- [3] Agerbo, Aino; Agerbo, Cornils (1998). "How to preserve the benefits of design patterns". *Conference on Object Oriented Programming Systems Languages and Applications* (Vancouver, British Columbia, Canada: ACM): 134–143. ISBN 1-58113-005-8.
- [4] <http://msdn2.microsoft.com/en-us/library/system.data.idbcommand.createparameter.aspx>
- [5] <http://doc.trolltech.com/4.0/qmainwindow.html#createPopupMenu>
- [6] <http://java.sun.com/j2se/1.4.2/docs/api/javax/xml/parsers/package-summary.html>
- Fowler, Martin; Kent Beck, John Brant, William Opdyke, and Don Roberts (June 1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley. ISBN 0-201-48567-2.
- Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.
- Cox, Brad J.; (1986). *Object-oriented programming: an evolutionary approach*. Addison-Wesley. ISBN 9780201103939.
- Cohen, Tal; Gil, Joseph (2007). "Better Construction with Factories" (<http://tal.forum2.org/static/cv/Factories.pdf>) (PDF). *Journal of Object Technology* (Bertrand Meyer). Retrieved 2007-03-12.

## External links

- Factory Method Design Pattern ([http://sourcemaking.com/design\\_patterns/factory\\_method](http://sourcemaking.com/design_patterns/factory_method)) in Source Making
- Factory method Example in PHP (<http://www.labelmedia.co.uk/blog/posts/design-patterns-factory-method.html>)
- Factory method in UML and in LePUS3 (<http://www.lepus.org.uk/ref/companion/FactoryMethod.xml>) (a Design Description Language)
- Description from the Portland Pattern Repository (<http://c2.com/cgi/wiki?FactoryMethodPattern>)
- Jt (<http://www.fsw.com/Jt/Jt.htm>) J2EE Pattern Oriented Framework

# Lazy initialization

---

In computer programming, **lazy initialization** is the tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed.

This is typically accomplished by maintaining a flag indicating whether the process has taken place. Each time the desired object is summoned, the flag is tested. If it is ready, it is returned. If not, it is initialized on the spot.

See lazy evaluation for a general treatment of this idea. In heavily imperative languages this pattern carries hidden dangers, as does any programming habit that relies on shared state.

## The "lazy factory"

In a software design pattern view, lazy initialization is often used together with a factory method pattern. This combines three ideas:

- using a factory method to get instances of a class (factory method pattern)
- storing the instances in a map, so you get the *same* instance the next time you ask for an instance with *same* parameter (Multiton pattern, similar to the singleton pattern)
- using lazy initialization to instantiate the object the first time it is requested (lazy initialization pattern).

## Examples

### C#

Here is a dummy example in C#.

The Fruit class itself doesn't do anything here, The class variable `_typesDictionary` is a Dictionary/Map used to store Fruit instances by typeName.

```
using System;
using System.Collections;
using System.Collections.Generic;

public class Fruit
{
    private string _typeName;

    private static Dictionary<string, Fruit> _typesDictionary = new Dictionary<string, Fruit>();

    private Fruit (String typeName)
    {
        this._typeName = typeName;
    }

    public static Fruit GetFruitByTypeName (string type)
    {
        Fruit fruit;

        if (!_typesDictionary.TryGetValue(type, out fruit))
        {
            // Lazy initialization
        }
    }
}
```

```
        fruit = new Fruit(type);

        _typesDictionary.Add(type, fruit);
    }

    return fruit;
}

public static void ShowAll()
{
    if (_typesDictionary.Count > 0)
    {
        Console.WriteLine("Number of instances made = {0}",
            _typesDictionary.Count);

        foreach (KeyValuePair<string, Fruit> kvp in _typesDictionary)
        {
            Console.WriteLine(kvp.Key);
        }

        Console.WriteLine();
    }
}

class Program
{
    static void Main(string[] args)
    {
        Fruit.GetFruitByTypeName("Banana");
        Fruit.ShowAll();

        Fruit.GetFruitByTypeName("Apple");
        Fruit.ShowAll();

        // returns pre-existing instance from first
        // time Fruit with "Banana" was created
        Fruit.GetFruitByTypeName("Banana");
        Fruit.ShowAll();

        Console.ReadLine();
    }
}
```

## Java

Here is an example in Java.

```
public class Fruit {

    private static final Map<String,Fruit> types = new HashMap<String, Fruit>();
    private final String type;

    // using a private constructor to force use of the factory method.
    private Fruit(String type) {
        this.type = type;
    }

    /**
     * Lazy Factory method, gets the Fruit instance associated with a
     * certain type. Instantiates new ones as needed.
     * @param type Any string that describes a fruit type, e.g. "apple"
     * @return The Fruit instance associated with that type.
     */
    public static synchronized Fruit getFruit(String type) {
        if(!types.containsKey(type)) {
            types.put(type, new Fruit(type)); // Lazy initialization
        }
        return types.get(type);
    }
}
```

## JavaScript

Here is an example in JavaScript.

```
var Fruit = (function () {

    var types = {};

    function Fruit() {};

    // counts own properties in object
    function count(obj) {
        var i = 0;
        for (var key in obj) {
            if (obj.hasOwnProperty(key)) {
                i++;
            }
        }
    }

    return i;
})
```

```
}

var _static = {
  getFruit: function (type) {
    if (types[type] === undefined) {
      types[type] = new Fruit;
    }

    return types[type];
  },
  printCurrentTypes: function () {
    console.log('Number of instances made: ' + count(types));
    for (var type in types) {
      console.log(type);
    }
  }
};

return _static;

})();

Fruit.getFruit('Apple');
Fruit.printCurrentTypes();
Fruit.getFruit('Banana');
Fruit.printCurrentTypes();
Fruit.getFruit('Apple');
Fruit.printCurrentTypes();
```

### Output

```
Number of instances made: 1
Apple

Number of instances made: 2
Apple
Banana

Number of instances made: 2
Apple
Banana
```

## C++

Here is an example in C++.

```
#include <iostream>
#include <string>
#include <map>

using namespace std;

class Fruit {
public:
    static Fruit* getFruit(const string& type);
    static void printCurrentTypes();

private:
    static map<string,Fruit*> types;
    string type;

    // note: constructor private forcing one to use static
getFruit()
    Fruit(const string& t) : type( t ) {}
};

//definition needed for using any static member variable
map<string,Fruit*> Fruit::types;

/*
 * Lazy Factory method, gets the Fruit instance associated with a
 * certain type. Instantiates new ones as needed.
 * precondition: type. Any string that describes a fruit type, e.g.
 "apple"
 * postcondition: The Fruit instance associated with that type.
 */
Fruit* Fruit::getFruit(const string& type) {
    Fruit *f = types[type]; // try to find an existing instance; if
not found std::map will return NULL

    if (!f) {                // if no instance with the proper type was
found, make one
        f = new Fruit(type); // lazy initialization part
        types[type] = f;     // adding the newly created Fruit to the
types map for later lookup
    }
    return f;
}

/*
 * For example purposes to see pattern in action
```

```
*/  
void Fruit::printCurrentTypes() {  
    if (!types.empty()) {  
        cout << "Number of instances made = " << types.size() << endl;  
        for (map<string,Fruit*>::iterator iter = types.begin(); iter != types.end();  
++iter) {  
            cout << (*iter).first << endl;  
        }  
        cout << endl;  
    }  
}  
  
int main(void) {  
    Fruit::getFruit("Banana");  
    Fruit::printCurrentTypes();  
  
    Fruit::getFruit("Apple");  
    Fruit::printCurrentTypes();  
  
    // returns pre-existing instance from first  
    // time Fruit with "Banana" was created  
    Fruit::getFruit("Banana");  
    Fruit::printCurrentTypes();  
  
    return 0;  
}  
  
/*  
OUTPUT:  
Number of instances made = 1  
Banana  
  
Number of instances made = 2  
Apple  
Banana  
  
Number of instances made = 2  
Apple  
Banana  
*/
```

## Smalltalk

Here is an example in Smalltalk, of a typical accessor method to return the value of a variable using lazy initialization.

```
height
  height ifNil: [height := 2.0].
  ^height
```

The 'non-lazy' alternative is to use an initialization method that is run when the object is created and then use a simpler accessor method to fetch the value.

```
initialize
  height := 2.0

height
  ^height
```

Note that lazy initialization can also be used in non-object-oriented languages.

## Ruby

Here is an example in Ruby, of lazily initializing an authentication token from a remote service like Google. The way that `@auth_token` is cached is also an example of memoization.

```
require 'net/http'
class Blogger
  def auth_token
    @auth_token ||=
      (res = Net::HTTP.post_form(uri, params)) &&
      get_token_from_http_response(res)
  end

  # get_token_from_http_response, uri and params are defined later in
the class
end

b = Blogger.new
b.instance_variable_get(:@auth_token) # returns nil
b.auth_token # returns token
b.instance_variable_get(:@auth_token) # returns token
```

## Python

Here is an example in Python.

```
class Fruit:
    def __init__(self, sort):
        self.sort = sort

class Fruits:
    def __init__(self):
        self.sorts = {}
```



```
def get_fruit(self, sort):
    if sort not in self.sorts:
        self.sorts[sort] = Fruit(sort)

    return self.sorts[sort]

if __name__ == '__main__':
    fruits = Fruits()
    print fruits.get_fruit('Apple')
    print fruits.get_fruit('Lime')
```

## PHP

Here is an example of lazy initialization in PHP 5:

```
<?php
header('Content-type:text/plain; charset=utf-8');

class Fruit
{
    private $type;
    private static $types = array();

    private function __construct($type)
    {
        $this->type = $type;
    }

    public static function getFruit($type)
    {
        // Lazy initialization takes place here
        if(!array_key_exists($type, self::$types)) {
            self::$types[$type] = new Fruit($type);
        }

        return self::$types[$type];
    }

    public static function printCurrentTypes()
    {
        echo 'Number of instances made: ' . count(self::$types) . "\n";

        foreach(array_keys(self::$types) as $key) echo "$key\n";

        echo "\n";
    }
}
```

```
Fruit::getFruit('Apple');
Fruit::printCurrentTypes();

Fruit::getFruit('Banana');
Fruit::printCurrentTypes();

Fruit::getFruit('Apple');
Fruit::printCurrentTypes();

/*
OUTPUT:

Number of instances made: 1
Apple

Number of instances made: 2
Apple
Banana

Number of instances made: 2
Apple
Banana
*/
?>
```

## External links

- Article "Java Tip 67: Lazy instantiation <sup>[1]</sup> - Balancing performance and resource usage" by Philip Bishop and Nigel Warren
- Java code examples <sup>[2]</sup>
- Use Lazy Initialization to Conserve Resources <sup>[3]</sup>
- Description from the Portland Pattern Repository <sup>[4]</sup>
- Lazy Initialization of Application Server Services <sup>[5]</sup>
- Lazy Inheritance in JavaScript <sup>[6]</sup>

## References

- [1] <http://www.javaworld.com/javaworld/jvatips/jw-javatip67.html>
- [2] <http://javapractices.com/Topic34.cjp>
- [3] <http://devx.com/tips/Tip/18007>
- [4] <http://c2.com/cgi/wiki?LazyInitialization>
- [5] [http://weblogs.java.net/blog/binod/archive/2005/09/lazy\\_initializa.html](http://weblogs.java.net/blog/binod/archive/2005/09/lazy_initializa.html)
- [6] <http://sourceforge.net/projects/jsiner>

# Multiton pattern

---

In software engineering, the **multiton pattern** is a design pattern similar to the singleton, which allows only one instance of a class to be created. The multiton pattern expands on the singleton concept to manage a map of named instances as key-value pairs.

Rather than have a single instance *per application* (e.g. the `java.lang.Runtime` object in the Java programming language) the multiton pattern instead ensures a single instance *per key*.

Most people and textbooks consider this a singleton pattern. For example, multiton does not explicitly appear in the highly-regarded object-oriented programming text book *Design Patterns* (it appears as a more flexible approach named **Registry of singletons**).

## Example

An example thread-safe Java implementation follows:

### Java

The first example synchronizes the whole `getInstance()`-method (which may be expensive in a highly concurrent environment).

```
public class FooMultiton {
    private static final Map<Object, FooMultiton> instances = new HashMap<Object, FooMultiton>();

    private FooMultiton() /* also acceptable: protected, {default} */ {
        /* no explicit implementation */
    }

    public static FooMultiton getInstance(Object key) {
        synchronized (instances) {

            // Our "per key" singleton
            FooMultiton instance = instances.get(key);

            if (instance == null) {

                // Lazily create instance
                instance = new FooMultiton();

                // Add it to map
                instances.put(key, instance);
            }

            return instance;
        }
    }

    // other fields and methods ...
}
```

```
}
```

To avoid this (expensive) synchronization for many reader-threads in a highly concurrent environment one may also combine the multiton pattern with double-checked locking:

```
public class FooMultiton {
    private static final Map<Object, FooMultiton> instances = new HashMap<Object, FooMultiton>();

    private FooMultiton() /* also acceptable: protected, {default}
*/ {
    /* no explicit implementation */
}

    public static FooMultiton getInstance(Object key) {
        // Our "per key" singleton
        FooMultiton instance = instances.get(key);

        // if the instance has never been created ...
        if (instance == null) {
            synchronized (instances) {

                // Check again, after having acquired the lock to
make sure
                // the instance was not created meanwhile by
another thread

                instance = instances.get(key);

                if (instance == null) {
                    // Lazily create instance
                    instance = new FooMultiton();

                    // Add it to map
                    instances.put(key, instance);
                }
            }

            return instance;
        }

        // other fields and methods ...
    }
}
```

## C#

```
using System.Collections.Generic;

namespace MyApplication
{
    class FooMultiton
    {
        private static readonly Dictionary<object, FooMultiton> _instances = new
Dictionary<object, FooMultiton>();

        private FooMultiton() {}

        public static FooMultiton GetInstance(object key)
        {
            FooMultiton instance;
            if (!_instances.TryGetValue(key, out instance))
            {
                lock (_instances)
                {
                    if (!_instances.TryGetValue(key, out instance))
                    {
                        instance = new FooMultiton();
                        _instances.Add(key, instance);
                    }
                }
            }
            return instance;
        }
    }
}
```

## Python

```
class Multiton(object):
    def __init__(self):
        self.instances = {}

    def __call__(self, key, instance):
        self.instances[key] = instance
        return instance

    def get_instance(self, key):
        return self.instances[key]

class A(object):
    def __init__(self, *args, **kw):
        pass
```

```

m = Multiton()
a0 = m('a0', A())
a1 = m('a1', A())
print m.get_instance('a0')
print m.get_instance('a1')

```

### Python (using decorators)

```

def multiton(cls):
    instances = {}
    def getinstance(name):
        if name not in instances:
            instances[name] = cls()
        return instances[name]
    return getinstance

@multiton
class MyClass:
    ...

a=MyClass("MyClass0")
b=MyClass("MyClass0")
c=MyClass("MyClass1")
print a is b #True
print a is c #False

```

### PHP

```

<?php
//orochoi
// This example requires php 5.3+
abstract class Multiton {
    private static $instances = array();
    public static function getInstance() {
        // For non-complex construction arguments, you can just use the
        $arg as the key
        $key = serialize(func_get_args());
        if (!isset(self::$instances[$key])) {
            // You can do this without the reflection class if you want
            to hard code the class constructor arguments
            $rc = new ReflectionClass(get_called_class());
            self::$instances[$key] =
            $rc->newInstanceArgs(func_get_args());
        }
        return self::$instances[$key];
    }
}

class Hello extends Multiton {

```

```

        public function __construct($string = 'world') {
            echo "Hello $string\n";
        }
    }

    class GoodBye extends Multiton {
        public function __construct($string = 'my', $string2 = 'darling')
        {
            echo "Goodbye $string $string2\n";
        }
    }

    $a = Hello::getInstance('world');

    $b = Hello::getInstance('bob');
    // $a !== $b

    $c = Hello::getInstance('world');
    // $a === $c

    $d = GoodBye::getInstance();

    $e = GoodBye::getInstance();
    // $d === $e

    $f = GoodBye::getInstance('your');
    // $d !== $f

```

### Action Script 3.0/ Flex

```

import flash.utils.Dictionary;

public class InternalModelLocator
{
    private static var instances:Dictionary= new
Dictionary();//Dictionary is treated as Java Hash Map

    public function InternalModelLocator()
    {
        /* Only One Instance created with
GetInstanceMethod*/
    }

    /* module_uuid can be a String -----
        In case of PureMVC "multitonKey" (this.multitonKey)
        can be used as unique key for multiple modules
        */

```

```
public static function
getInstance(module_uuid:String):InternalModelLocator
{
    var instance:InternalModelLocator =
instances[module_uuid];

    if(instance == null)
    {
        instance = new InternalModelLocator();
        instances[module_uuid] = instance;
    }
    return instance;
}
}
```

## Clarification of example code

While it may appear that the multiton is no more than a simple hash table with synchronized access there are two important distinctions. First, the multiton does not allow clients to add mappings. Secondly, the multiton never returns a null or empty reference; instead, it creates and stores a multiton instance on the first request with the associated key. Subsequent requests with the same key return the original instance. A hash table is merely an implementation detail and not the only possible approach. The pattern simplifies retrieval of shared objects in an application.

Since the object pool is created only once, being a member associated with the class (instead of the instance), the multiton retains its flat behavior rather than evolving into a tree structure.

The multiton is unique in that it provides centralized access to a single directory (i.e. all keys are in the same namespace, *per se*) of multitons, where each multiton instance in the pool may exist having its own state. In this manner, the pattern advocates indexed storage of essential objects for the system (such as would be provided by an LDAP system, for example). However, a multiton is limited to wide use by a single system rather than a myriad of distributed systems.

## Drawbacks

This pattern, like the Singleton pattern, makes unit testing far more difficult<sup>[1]</sup>, as it introduces global state into an application.

## References

[1] <http://googletesting.blogspot.com/2008/11/clean-code-talks-global-state-and.html>

## External links

- Multiton implementation in Ruby language (<http://raa.ruby-lang.org/project/multiton/>)
- Multiton usage in PureMVC Framework for ActionScript 3 ([http://trac.puremvc.org/PureMVC\\_AS3\\_MultiCore/browser/tags/1.0.4/src/org/puremvc/as3/multicore/patterns/facade/Facade.as](http://trac.puremvc.org/PureMVC_AS3_MultiCore/browser/tags/1.0.4/src/org/puremvc/as3/multicore/patterns/facade/Facade.as))
- Article with a C# Multiton implementation, example of use, and discussion of memory issues (<http://gen5.info/q/2008/07/25/the-multiton-design-pattern/>)



# Object pool pattern

---

For the article about a general pool see [Pool \(Computer science\)](#)

In computer programming, an **object pool** is a software design pattern. An **object pool** is a set of initialised objects that are kept ready to use, rather than allocated and destroyed on demand. A client of the pool will request an object from the pool and perform operations on the returned object. When the client has finished with an object, it returns it to the pool, rather than destroying it. It is a specific type of factory object.

Object pooling can offer a significant performance boost in situations where the cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instances in use at any one time is low. The pooled object is obtained in predictable time when creation of the new objects (especially over network) may take variable time.

However these benefits are mostly true for objects which are expensive with respect to time, such as database connections, socket connections, threads and large graphic objects like fonts or bitmaps. In certain situations, simple object pooling (which hold no external resources, but only occupy memory) may not be efficient and could decrease performance [1].

## Handling of empty pools

Object pools employ one of three strategies to handle a request when there are no spare objects in the pool.

1. Fail to provide an object (and return an error to the client).
2. Allocate a new object, thus increasing the size of the pool. Pools that do this usually allow you to set the high water mark (the maximum number of objects ever used).
3. In a multithreaded environment, a pool may block the client until another thread returns an object to the pool.

## Pitfalls

When writing an object pool, the programmer has to be careful to make sure the state of the objects returned to the pool is reset back to a sensible state for the next use of the object. If this is not observed, the object will often be in some state that was unexpected by the client program and may cause the client program to fail. The pool is responsible for resetting the objects, not the clients. Object pools full of objects with dangerously stale state are sometimes called object cesspools and regarded as an anti-pattern.

The presence of stale state is not always an issue; it becomes dangerous when the presence of stale state causes the object to behave differently. For example, an object that represents authentication details may break if the "successfully authenticated" flag is not reset before it is passed out, since it will indicate that a user is correctly authenticated (possibly as someone else) when they haven't yet attempted to authenticate. However, it will work just fine if you fail to reset some value only used for debugging, such as the identity of the last authentication server used.

Inadequate resetting of objects may also cause an information leak. If an object contains confidential data (e.g. a user's credit card numbers) that isn't cleared before the object is passed to a new client, a malicious or buggy client may disclose the data to an unauthorized party.

If the pool is used by multiple threads, it may need means to prevent parallel threads from grabbing and trying to reuse the same object in parallel. This is not necessary if the pooled objects are immutable or otherwise thread-safe.

---

## Class diagram

## Criticism

Some publications do not recommend using object pooling with certain languages, such as Java, especially for objects that only use memory and hold no external resources [2]. Opponents usually say that object allocation is relatively fast in modern languages with garbage collectors; while the operator "new" needs only 10 instructions, the classic "new" - "delete" pair found in pooling designs requires hundreds of them as it does more complex work. Also, most garbage collectors scan "live" object references, and not the memory that these objects use for their content. This means that any number of "dead" objects without references can be discarded with little cost. In contrast, keeping a large number of "live" but unused objects increases the duration of garbage collection [1]. In some cases, programs that use garbage collection instead of directly managing memory may run faster. [3]

## Examples

In the .NET Base Class Library there are a few objects that implement this pattern. `System.Threading.ThreadPool` is configured to have a predefined number of threads to allocate. When the threads are returned, they are available for another computation. Thus, one can use threads without paying the cost of creation and disposal of threads.

Java supports thread pooling via `java.util.concurrent.ExecutorService` and other related classes. The executor service has a certain number of "basic" threads that are never discarded. If all threads are busy, the service allocates the allowed number of extra threads that are later discarded if not used for the certain expiration time. If no more threads are allowed, the tasks can be placed in the queue. Finally, if this queue may get too long, it can be configured to suspend the requesting thread.

## References

- Kircher, Michael; Prashant Jain; (2002-07-04). "Pooling Pattern" <sup>[4]</sup>. *EuroPLoP 2002*. Germany. Retrieved 2007-06-09.

## External links

- OODesign article <sup>[5]</sup>
  - Improving Performance with Object Pooling (Microsoft Developer Network ) <sup>[6]</sup>
  - Developer.com article <sup>[7]</sup>
  - Portland Pattern Repository entry <sup>[8]</sup>
  - Apache Commons Pool: A mini-framework to correctly implement object pooling in Java <sup>[9]</sup>
  - Game Programming Patterns: Object Pool <sup>[10]</sup>
-

## References

- [1] <http://www-128.ibm.com/developerworks/java/library/j-jtp09275.html?ca=dgr-jw22JavaUrbanLegends>
- [2] <http://www.ibm.com/developerworks/java/library/j-jtp11253/>
- [3] <http://www.cs.ubc.ca/local/reading/proceedings/spe91-95/spe/vol23/issue7/spe836.pdf>
- [4] <http://www.kircher-schwanninger.de/michael/publications/Pooling.pdf>
- [5] <http://www.oodeesign.com/object-pool-pattern.html>
- [6] <http://msdn2.microsoft.com/en-us/library/ms682822.aspx>
- [7] <http://www.developer.com/java/ent/article.php/626171>
- [8] <http://c2.com/cgi-bin/wiki?ObjectPoolPattern>
- [9] <http://commons.apache.org/pool/>
- [10] <http://gameprogrammingpatterns.com/object-pool.html>

## Prototype pattern

The **prototype pattern** is a creational design pattern used in software development when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects. This pattern is used to:

- avoid subclasses of an object creator in the client application, like the abstract factory pattern does.
- avoid the inherent cost of creating a new object in the standard way (e.g., using the 'new' keyword) when it is prohibitively expensive for a given application.

To implement the pattern, declare an abstract base class that specifies a pure virtual *clone()* method. Any class that needs a "polymorphic constructor" capability derives itself from the abstract base class, and implements the *clone()* operation.

The client, instead of writing code that invokes the "new" operator on a hard-coded class name, calls the *clone()* method on the prototype, calls a factory method with a parameter designating the particular concrete derived class desired, or invokes the *clone()* method through some mechanism provided by another design pattern.

## Example

The Prototype pattern specifies the kind of objects to create using a prototypical instance. Prototypes of new products are often built prior to full production, but in this example, the prototype is passive and does not participate in copying itself. The mitotic division of a cell - resulting in two identical cells - is an example of a prototype that plays an active role in copying itself and thus, demonstrates the Prototype pattern. When a cell splits, two cells of identical genotype result. In other words, the cell clones itself.<sup>[1]</sup>

## Java

```
/**
 * Prototype Class
 */
abstract class PrototypeFactory implements Cloneable {
    public Object clone() throws CloneNotSupportedException {
        // call Object.clone()
        PrototypeFactory copy = (PrototypeFactory) super.clone();
        //In an actual implementation of this pattern you might now
change references to
        //the expensive to produce parts from the copies that are held
inside the prototype.
        return copy;
    }
}
```

```

    }

    abstract void prototypeFactory(int x);

    abstract void printValue();
}

/**
 * Concrete Prototypes to clone
 */
class PrototypeImpl extends PrototypeFactory {
    int x;

    public PrototypeImpl(int x) {
        this.x = x;
    }

    @Override
    void prototypeFactory(int x) {
        this.x = x;
    }

    public void printValue() {
        System.out.println("Value :" + x);
    }
}

/**
 * Client Class
 */
public class PrototypeExample {

    private PrototypeFactory example; // Could have been a private
    Cloneable example.

    public PrototypeExample(PrototypeFactory example) {
        this.example = example;
    }

    public PrototypeFactory makeCopy() throws
    CloneNotSupportedException {
        return (PrototypeFactory) this.example.clone();
    }

    public static void main(String args[]) {
        try {
            PrototypeFactory tempExample = null;

```

```

        int num = 1000;
        PrototypeFactory prot = new PrototypeImpl(1000);
        PrototypeExample cm = new PrototypeExample(prot);
        for (int i = 0; i < 10; i++) {
            tempExample = cm.makeCopy();
            tempExample.prototypeFactory(i * num);
            tempExample.printValue();
        }
    } catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }
}

/*
**Code output**

Value :0
Value :1000
Value :2000
Value :3000
Value :4000
Value :5000
Value :6000
Value :7000
Value :8000
Value :9000
*/

```

## Rules of thumb

Sometimes creational patterns overlap - there are cases when either Prototype or Abstract Factory would be appropriate. At other times they complement each other: Abstract Factory might store a set of Prototypes from which to clone and return product objects (GoF, p126). Abstract Factory, Builder, and Prototype can use Singleton in their implementations. (GoF, p81, 134). Abstract Factory classes are often implemented with Factory Methods (creation through inheritance), but they can be implemented using Prototype (creation through delegation). (GoF, p95)

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. (GoF, p136)

Prototype doesn't require subclassing, but it does require an "initialize" operation. Factory Method requires subclassing, but doesn't require initialization. (GoF, p116)

Designs that make heavy use of the Composite and Decorator patterns often can benefit from Prototype as well. (GoF, p126)

The rule of thumb could be that you would need to clone() an *Object* when you want to create another *Object* *at runtime* which is a *true copy* of the *Object* you are cloning. *True copy* means all the attributes of the newly created *Object* should be the same as the *Object* you are cloning. If you could have *instantiated* the class by using *new* instead, you would get an *Object* with all attributes as their initial values. For example, if you are designing a system

for performing bank account transactions, then you would want to make a copy of the Object which holds your account information, perform transactions on it, and then replace the original Object with the modified one. In such cases, you would want to use `clone()` instead of `new`.

## References

[1] Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p. 54

- Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.

## External links

- Prototype ([http://sourcemaking.com/design\\_patterns/prototype](http://sourcemaking.com/design_patterns/prototype)) in Source Making

# Singleton pattern

---

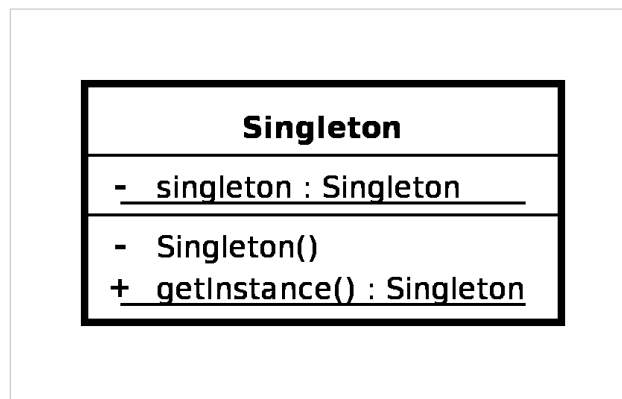
In computer engineering, the **singleton pattern** is a design pattern used to implement the mathematical concept of a singleton, by restricting the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects (say, five). Some consider it an anti-pattern, judging that it is overused, introduces unnecessary limitations in situations where a sole instance of a class is not actually required, and introduces global state into an application.<sup>[1] [2] [3] [4] [5] [6]</sup>

In C++ it also serves to isolate from the unpredictability of the order of dynamic initialization, returning control to the programmer.

## Common uses

- The Abstract Factory, Builder, and Prototype patterns can use Singletons in their implementation.
- Facade objects are often Singletons because only one Facade object **is required**.
- State objects are often Singletons.
- Singletons are often preferred to global variables because:
  - They **don't pollute the global name space (or, in languages with namespaces, their containing namespace) with unnecessary variables**.<sup>[7]</sup>
  - They permit lazy allocation and initialization, **whereas global variables in many languages will always consume resources**.

## Class diagram



## Implementation

**Implementation** of a singleton pattern must satisfy the single instance and global access principles. It requires a mechanism to access the singleton class member without creating a class object and a mechanism to persist the value of class members among class objects. The singleton pattern is implemented by creating a class with a method that creates a new instance of the class if one does not exist. If an instance already exists, it simply returns a reference to that object. To make sure that the object cannot be instantiated any other way, the constructor is made protected (not private, because reuse and unit test could need to access the constructor). Note the **distinction** between a simple static instance of a class and a singleton: although a singleton can be implemented as a static instance, it can also be lazily constructed, requiring no memory or resources until needed. Another notable difference is that static member classes cannot implement an interface, unless that interface is simply a marker. So if the class has to realize a contract expressed by an interface, it really has to be a singleton.

The singleton pattern must be carefully constructed in multi-threaded applications. If two threads are to execute the creation method at the same time when a singleton does not yet exist, they both must check for an instance of the singleton and then only one should create the new one. If the programming language has concurrent processing capabilities the method should be constructed to execute as a mutually exclusive operation.

The classic solution to this problem is to use mutual exclusion on the class that indicates that the object is being **instantiated**.

## Example

The Java programming language solutions provided here are all thread-safe but differ in supported language versions and lazy-loading. The newest most effective way to create a Singleton in Java, since 5.0 is the enum type approach, given at the end of this section.

### Traditional simple way

This solution is thread-safe without requiring special language constructs, but it may lack the laziness of the one below. The `INSTANCE` is created as soon as the `Singleton` class is initialized<sup>[8]</sup>. That might even be long before `getInstance()` is called. It might be (for example) when some static method of the class is used. If laziness is not needed or the instance needs to be created early in the application's execution, or your class has no other static members or methods that could prompt early initialization (and thus creation of the instance), this (slightly) simpler solution can be used:

```
public class Singleton {
```

```
private static final Singleton INSTANCE = new Singleton();

// Private constructor prevents instantiation from other classes
private Singleton() {
}

public static Singleton getInstance() {
    return INSTANCE;
}
}
```

## The solution of Bill Pugh

University of Maryland Computer Science researcher Bill Pugh has written about the code issues underlying the Singleton pattern when implemented in Java.<sup>[9]</sup> Pugh's efforts on the "Double-checked locking" idiom led to changes in the Java memory model in Java 5 and to what is generally regarded as the standard method to implement Singletons in Java. The technique known as the initialization on demand holder idiom, is as lazy as possible, and works in all known versions of Java. It takes advantage of language guarantees about class initialization, and will therefore work correctly in all Java-compliant compilers and virtual machines.

The nested class is referenced no earlier (and therefore loaded no earlier by the class loader) than the moment that `getInstance()` is called. Thus, this solution is thread-safe without requiring special language constructs (*i.e.* `volatile` or `synchronized`).

```
public class Singleton {

    // Private constructor prevents instantiation from other classes
    private Singleton() {
    }

    /**
     * SingletonHolder is loaded on the first execution of
     * Singleton.getInstance()
     * or the first access to SingletonHolder.INSTANCE, not before.
     */
    private static class SingletonHolder {
        public static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```



## The Enum-way

In the second edition of his book "Effective Java" Joshua Bloch claims that "a single-element enum type is the best way to implement a singleton"<sup>[10]</sup> for any Java that supports enums. The use of an enum is very easy to implement and has no drawbacks regarding serializable objects, which have to be circumvented in the other ways.

```
public enum Singleton {  
    INSTANCE;  
}
```

## Prototype-based singleton

In a prototype-based programming language, where objects but not classes are used, a "singleton" simply refers to an object without copies or that is not used as the prototype for any other object. Example in Io:

```
Foo := Object clone  
Foo clone := Foo
```

## Example of use with the factory method pattern

The singleton pattern is often used in conjunction with the factory method pattern to create a system-wide resource whose specific type is not known to the code that uses it. An example of using these two patterns together is the Java Abstract Window Toolkit (AWT).

`java.awt.Toolkit` is an abstract class that binds the various AWT components to particular native toolkit implementations. The `Toolkit` class has a `Toolkit.getDefaultToolkit()` factory method that returns the platform-specific subclass of `Toolkit`. The `Toolkit` object is a singleton because the AWT needs only a single object to perform the binding and the object is relatively expensive to create. The toolkit methods must be implemented in an object and not as static methods of a class because the specific implementation is not known by the platform-independent components. The name of the specific `Toolkit` subclass used is specified by the `"awt.toolkit"` environment property accessed through `System.getProperties()`.

The binding performed by the toolkit allows, for example, the backing implementation of a `java.awt.Window` to bind to the platform-specific `java.awt.peer.WindowPeer` implementation. Neither the `Window` class nor the application using the window needs to be aware of which platform-specific subclass of the peer is used.

## Drawbacks

This pattern makes unit testing far more difficult<sup>[6]</sup>, as it introduces global state into an application.

It should also be noted that this pattern reduces the potential for parallelism within a program, because access to the singleton in a multi-threaded context must be serialised, e.g., by locking.

Advocates of dependency injection would regard this as an anti-pattern, mainly due to its use of private and static methods.

Some have suggested ways to break down the singleton pattern using methods such as reflection in languages such as Java or PHP.<sup>[11]</sup>

## References

- [1] Alex Miller. Patterns I hate #1: Singleton (<http://tech.puredanger.com/2007/07/03/pattern-hate-singleton/>), July 2007
- [2] Scott Densmore. Why singletons are evil (<http://blogs.msdn.com/scottdensmore/archive/2004/05/25/140827.aspx>), May 2004
- [3] Steve Yegge. Singletons considered stupid (<http://steve.yegge.googlepages.com/singleton-considered-stupid>), September 2004
- [4] J.B. Rainsberger, IBM. Use your singletons wisely (<http://www-128.ibm.com/developerworks/webservices/library/co-single.html>), July 2001
- [5] Chris Reath. Singleton I love you, but you're bringing me down (<http://www.codingwithoutcomments.com/2008/10/08/singleton-i-love-you-but-youre-bringing-me-down/>), October 2008
- [6] <http://googletesting.blogspot.com/2008/11/clean-code-talks-global-state-and.html>
- [7] Gamma, E, Helm, R, Johnson, R, Vlissides, J: "Design Patterns", page 128. Addison-Wesley, 1995
- [8] [http://java.sun.com/docs/books/jls/third\\_edition/html/execution.html#12.4](http://java.sun.com/docs/books/jls/third_edition/html/execution.html#12.4)
- [9] Pugh, Bill (November 16, 2008). "The Java Memory Model" (<http://www.cs.umd.edu/~pugh/java/memoryModel/>). . Retrieved April 27, 2009.
- [10] Joshua Bloch: Effective Java 2nd edition, ISBN 978-0-321-35668-0, 2008, p. 18
- [11] Singleton and Multiton with a different approach (<http://phpgoodness.wordpress.com/2010/07/21/singleton-and-multiton-with-a-different-approach/>), July 21, 2010
- "C++ and the Perils of Double-Checked Locking" ([http://www.aristeia.com/Papers/DDJ\\_Jul\\_Aug\\_2004\\_revised.pdf#search="meyers double checked locking"](http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf#search=)) Meyers, Scott and Alexandrescu, Andrei, September 2004.
- "The Boost.Threads Library" (<http://www.ddj.com/dept/cpp/184401518>) Kempf, B., Dr. Dobb's Portal, April 2003.

## External links

- How to make a class singleton (<http://thekiransblog.blogspot.com/2010/02/how-to-make-class-singleton.html>)
- The "Double-Checked Locking is Broken" Declaration (<http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>) (Java)
- A Pattern Enforcing Compiler (<http://pec.dev.java.net/>) that enforces the Singleton pattern amongst other patterns
- Description from the Portland Pattern Repository (<http://c2.com/cgi/wiki?SingletonPattern>)
- Implementing the Singleton Pattern in C# (<http://csharpindepth.com/Articles/General/Singleton.aspx>) by Jon Skeet
- A Threadsafe C++ Template Singleton Pattern for Windows Platforms (<http://www.opbarnes.com/blog/Programming/OPB/Snippets/Singleton.html>) by O. Patrick Barnes
- Implementing the Inheritable Singleton Pattern in PHP5 (<http://svn.shadanakar.org/filedetails.php?repname=onPHP&path=/trunk/core/Base/Singleton.class.php&rev=0&sc=0>)
- Singleton Pattern and Thread Safety (<http://www.oaklib.org/docs/oak/singleton.html>)
- PHP patterns (<http://www.php.net/manual/en/language.oop5.patterns.php>)
- Javascript implementation of a Singleton Pattern (<http://prototyp.ical.ly/index.php/2007/03/01/javascript-design-patterns-1-the-singleton/>) by Christian Schaefer
- Singletons Cause Cancer (<http://www.prestonlee.com/2006/07/11/singletons-cause-cancer/>) by Preston Lee
- Singleton examples ([http://www.oodeesign.com/oo\\_design\\_patterns/creational\\_patterns/singleton.html](http://www.oodeesign.com/oo_design_patterns/creational_patterns/singleton.html))
- Article " Double-checked locking and the Singleton pattern (<http://www-128.ibm.com/developerworks/java/library/j-dcl.html?loc=j>)" by Peter Haggar
- Article " Use your singletons wisely (<http://www-106.ibm.com/developerworks/library/co-single.html>)" by J. B. Rainsberger
- Article " Simply Singleton (<http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns.html>)" by David Geary
- Article " Description of Singleton (<http://www.dofactory.com/Patterns/PatternSingleton.aspx>)" by Aruna

- Article " Why Singletons Are Controversial (<http://code.google.com/p/google-singleton-detector/wiki/WhySingletonsAreControversial>)"
- The Google Singleton Detector (<http://code.google.com/p/google-singleton-detector/>) analyzes Java bytecode to detect singletons, so that their usefulness can be evaluated.
- Serialization of Singleton in Java (<http://www.exampledepot.com/egs/java.io/DeserSingle.html?l=rel>)
- Breaking the Singleton Pattern using Reflection in Java (<http://yohanliyanage.blogspot.com/2009/09/breaking-singleton.html>)
- Singleton at Microsoft patterns & practices Developer Center (<http://msdn.microsoft.com/en-us/library/ms998426.aspx>)
- Singleton Class in ActionScript 3 (<http://www.how-to-code.com/as3-actionscript3/as3-design-patterns/as3-singleton-class.html>)
- Standard way of implementing Singletons in ActionScript 3 (<http://code.google.com/p/maashaack/wiki/Singleton>)
- singleton: Ruby Standard Library Documentation (<http://ruby-doc.org/stdlib/libdoc/singleton/rdoc/index.html>)
- Polymorphic\_Singletons C++ ([http://developer.novell.com/wiki/index.php/Polymorphic\\_Singletons](http://developer.novell.com/wiki/index.php/Polymorphic_Singletons))

## Structural pattern

---

In software engineering, **structural design patterns** are design patterns that ease the design by identifying a simple way to realize relationships between entities.

Examples of Structural Patterns include:

- Adapter pattern: 'adapts' one interface for a class into one that a client expects
  - Retrofit Interface Pattern<sup>[1] [2]</sup> : An adapter used as a new interface for multiple classes at the same time.
  - Adapter pipeline: Use multiple adapters for debugging purposes.<sup>[3]</sup>
- Aggregate pattern: a version of the Composite pattern with methods for aggregation of children
- Bridge pattern: decouple an abstraction from its implementation so that the two can vary independently
  - Tombstone: An intermediate "lookup" object contains the real location of an object.<sup>[4]</sup>
- Composite pattern: a tree structure of objects where every object has the same interface
- Decorator pattern: add additional functionality to a class at runtime where subclassing would result in an exponential rise of new classes
- Extensibility pattern: aka. Framework - hide complex code behind a simple interface
- Facade pattern: create a simplified interface of an existing interface to ease usage for common tasks
- Flyweight pattern: a high quantity of objects share a common properties object to save space
- Proxy pattern: a class functioning as an interface to another thing
- Pipes and filters: a chain of processes where the output of each process is the input of the next
- Private class data pattern: restrict accessor/mutator access

## References

- [1] <http://c2.com/cgi/wiki?RetrofitInterfacePattern>
- [2] <http://c2.com/cgi/wiki?ExternalPolymorphism>
- [3] <http://c2.com/cgi/wiki?AdapterPipeline>
- [4] <http://c2.com/cgi/wiki?TombStone>

# Adapter pattern

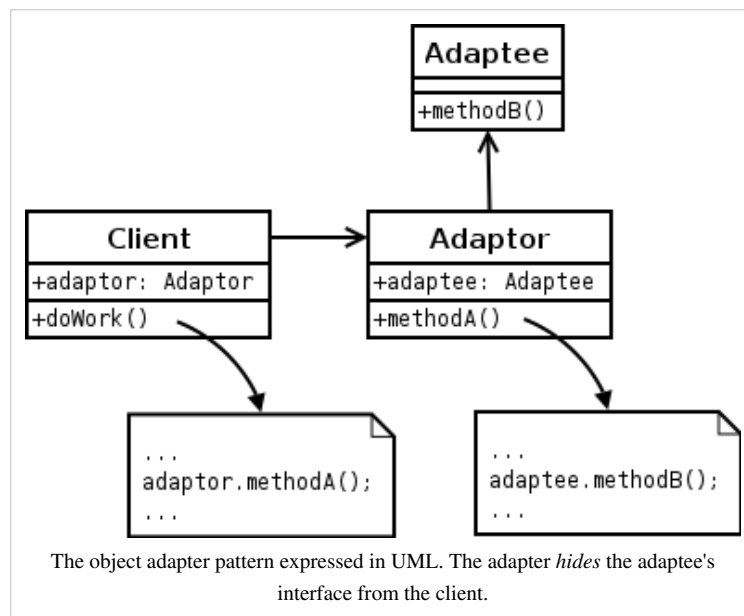
In computer programming, the **adapter pattern** (often referred to as the **wrapper pattern** or simply a **wrapper**) is a design pattern that translates one interface for a class into a compatible interface. An *adapter* allows classes to work together that normally could not because of incompatible interfaces, by providing its interface to clients while using the original interface. The adapter translates calls to its interface into calls to the original interface, and the amount of code necessary to do this is typically small. The adapter is also responsible for transforming data into appropriate forms. For instance, if multiple boolean values are stored as a single integer (i.e. flags) but your consumer requires a 'true'/'false', the adapter would be responsible for extracting the appropriate values from the integer value. Another example is transforming the format of dates (e.g. YYYYMMDD to MM/DD/YYYY or DD/MM/YYYY).

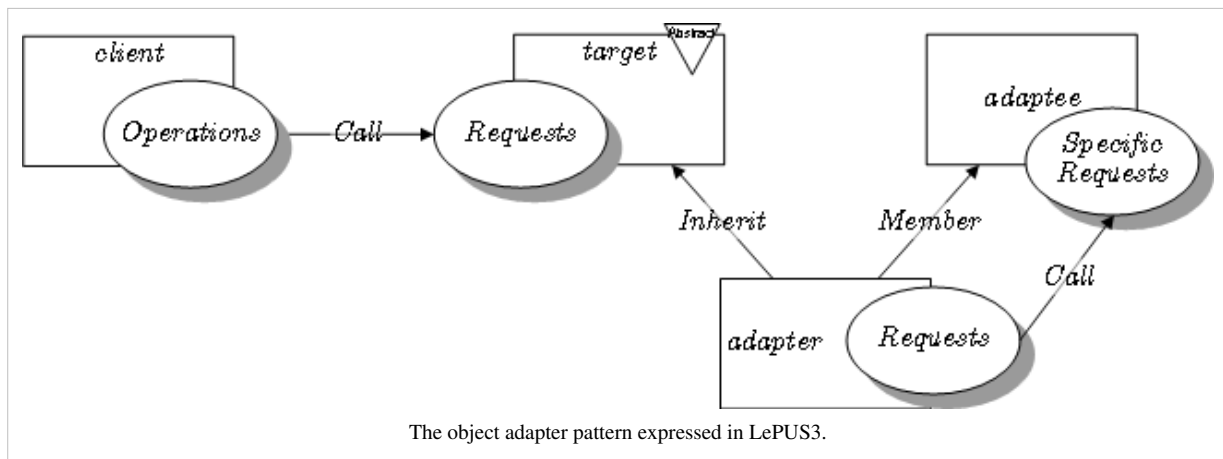
## Structure

There are two types of adapter pattern:

### Object Adapter pattern

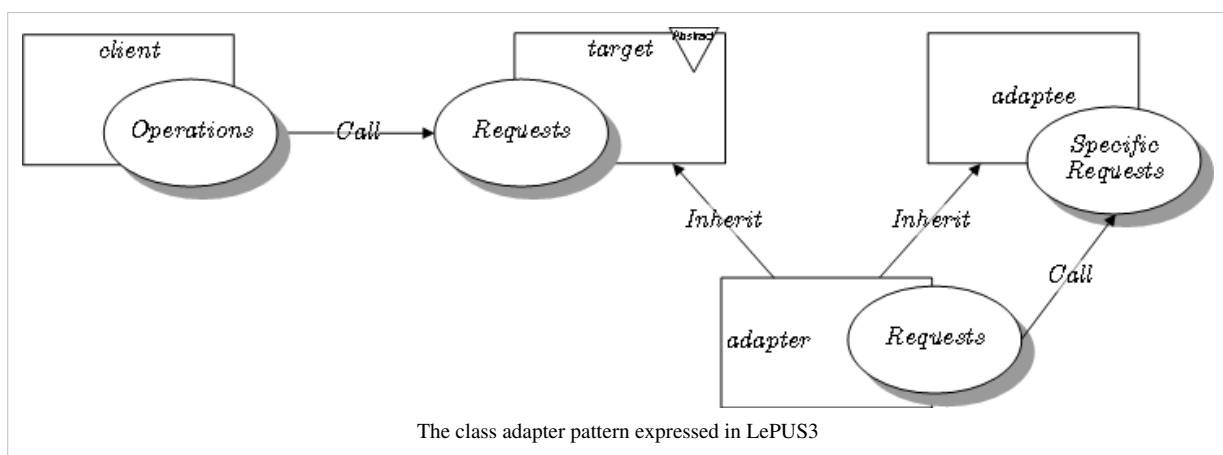
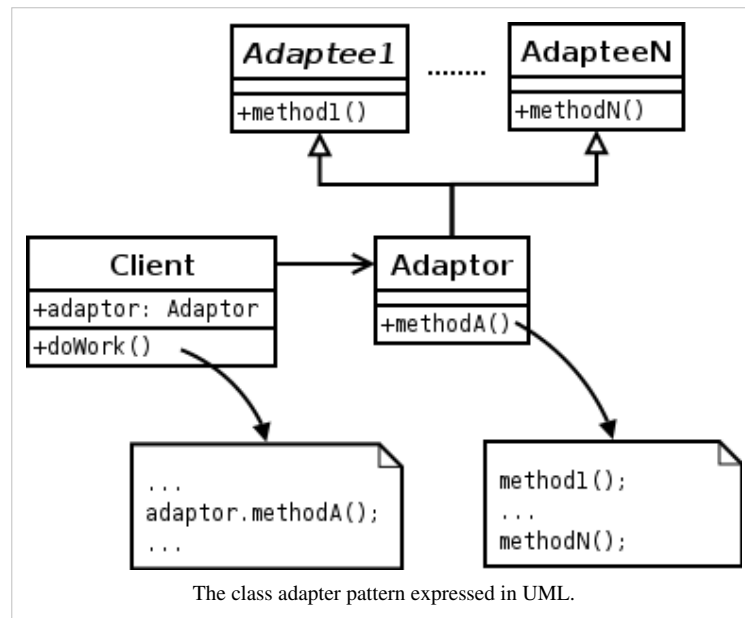
In this type of adapter pattern, the adapter contains an instance of the class it wraps. In this situation, the adapter makes calls to the instance of the wrapped object.





### Class Adapter pattern

This type of adapter uses multiple inheritance to achieve its goal. The adapter is created inheriting from both the interface that is expected and the interface that is pre-existing. It is typical for the expected interface to be created as a pure interface class, especially in languages such as Java that do not support multiple inheritance.



The adapter pattern is useful in situations where an already existing class provides some or all of the services you need but does not use the interface you need. A good real life example is an adapter that converts the interface of a Document Object Model of an XML document into a tree structure that can be displayed. A link to a tutorial that uses the adapter design pattern is listed in the links below.

### A further form of runtime Adapter pattern

There is a further form of runtime Adapter pattern as follows:

It is desired for classA to supply classB with some data, let us suppose some String data. A compile time solution is:

```
classB.setStringData(classA.getStringData());
```

However, suppose that the format of the string data must be varied. A compile time solution is to use inheritance:

```
Format1ClassA extends ClassA {
    public String getStringData() {
        return format(toString());
    }
}
```

and perhaps create the correctly "formatting" object at runtime by means of the Factory pattern.

A solution using "adapters" proceeds as follows:

(i) define an intermediary "Provider" interface, and write an implementation of that Provider interface which wraps the source of the data, ClassA in this example, and outputs the data formatted as appropriate:

```
public interface StringProvider {
    public String getStringData();
}

public class ClassAFormat1 implements StringProvider {
    ClassA classA;

    public ClassAFormat1(ClassA classA) {
        this.classA = classA;
    }

    public String getStringData() {
        return format(classA.toString());
    }
}
```

(ii) Write an Adapter class which returns the specific implementation of the Provider:

```
public class ClassAFormat1Adapter extends Adapter {
    public Object adapt(Object o) {
        return new ClassAFormat1((ClassA) o);
    }

    public boolean isAdapterFor(Class c) {

        return c.equals(StringProvider.class);
    }
}
```

```
}  
  
}
```

(iii) Register the Adapter with a global registry, so that the Adapter can be looked up at runtime:

```
AdapterFactory.getInstance().registerAdapter(ClassA.class,  
ClassAFormat1Adapter.class, "format1");
```

(iv) In your code, when you wish to transfer data from ClassA to ClassB, write:

```
Adapter adapter =  
AdapterFactory.getInstance().getAdapterFromTo(ClassA.class,  
StringProvider.class, "format1");  
  
StringProvider provider = (StringProvider) adapter.adapt(classA);  
  
String string = provider.getStringData();  
  
classB.setStringData(string);
```

or more concisely:

```
classB.setStringData(((StringProvider)  
AdapterFactory.getInstance().getAdapterFromTo(ClassA.class,  
StringProvider.class, "format1").adapt(classA)).getStringData());
```

(v) The advantage can be seen in that, if it is desired to transfer the data in a second format, then look up the different adapter/provider:

```
Adapter adapter =  
AdapterFactory.getInstance().getAdapterFromTo(ClassA.class,  
StringProvider.class, "format2");
```

(vi) And if it is desired to output the data from ClassA as, say, image data in Class C:

```
Adapter adapter =  
AdapterFactory.getInstance().getAdapterFromTo(ClassA.class,  
ImageProvider.class, "format1");  
  
ImageProvider provider = (ImageProvider) adapter.adapt(classA);  
  
classC.setImage(provider.getImage());
```

(vii) In this way, the use of adapters and providers allows multiple "views" by ClassB and ClassC into ClassA without having to alter the class hierarchy. In general, it permits a mechanism for arbitrary data flows between objects which can be retrofitted to an existing object hierarchy.

## External links

- Adapter in UML and in LePUS3 (a formal modelling language) <sup>[1]</sup>
- Description in Portland Pattern Repository's Wiki <sup>[2]</sup>
- Java Tutorial on the Document Object Model <sup>[3]</sup> (uses the adapter pattern)
- A generic implementation of The Adapter Pattern in C++ <sup>[4]</sup>
- PerfectJPattern Open Source Project <sup>[5]</sup>, Provides componentized implementation of the Adapter Pattern in Java
- The Adapter Pattern in Python <sup>[6]</sup> (detailed tutorial)
- The Adapter Pattern in Eclipse RCP <sup>[7]</sup>
- Adapter Design Pattern <sup>[8]</sup>

## References

- [1] [http://www.lepus.org.uk/ref/companion/Adapter\(Class\).xml](http://www.lepus.org.uk/ref/companion/Adapter(Class).xml)  
 [2] <http://www.c2.com/cgi/wiki?AdapterPattern>  
 [3] <http://java.sun.com/webservices/jaxp/dist/1.1/docs/tutorial/dom/>  
 [4] <http://www.ddj.com/architect/199204099>  
 [5] <http://perfectjpattern.sourceforge.net/dp-adapter.html>  
 [6] <http://ginstrom.com/scribbles/2009/03/27/the-adapter-pattern-in-python/>  
 [7] <http://www.eclipse.org/articles/article.php?file=Article-Adapters/index.html>  
 [8] [http://sourcemaking.com/design\\_patterns/adapter](http://sourcemaking.com/design_patterns/adapter)

## Bridge pattern

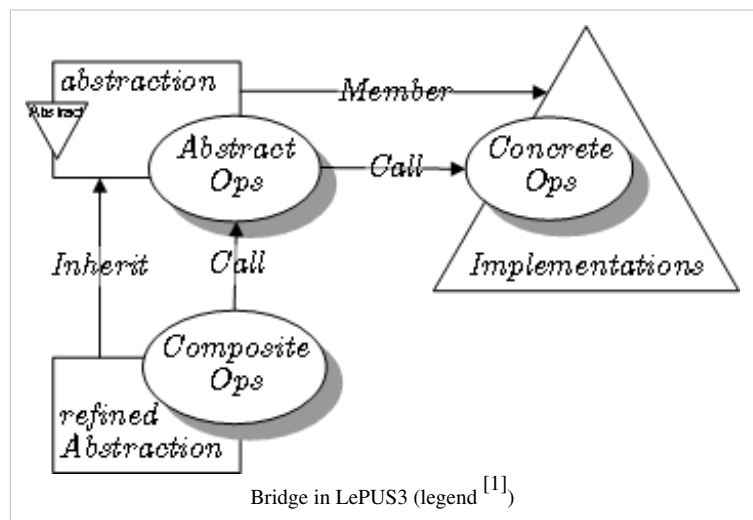
The **bridge pattern** is a design pattern used in software engineering which is meant to "decouple an abstraction from its implementation so that the two can vary independently" <sup>[2]</sup>. The bridge uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes.

When a class varies often, the features of object-oriented programming become very useful because changes to a program's code can be made easily with minimal prior knowledge about the program. The bridge pattern is useful when both the class as well

as what it does vary often. The class itself can be thought of as the *implementation* and what the class can do as the *abstraction*. The bridge pattern can also be thought of as two layers of abstraction.

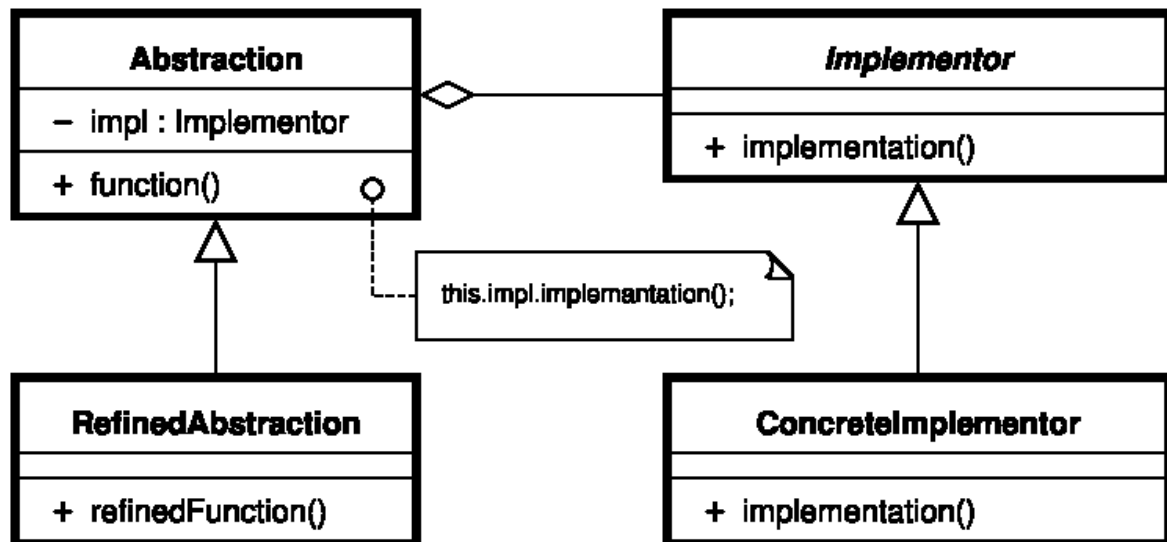
The **bridge pattern** is often confused with the adapter pattern. In fact, the **bridge pattern** is often implemented using the **class adapter pattern**, e.g. in the Java code below.

**Variant:** The implementation can be decoupled even more by deferring the presence of the implementation to the point where the abstraction is utilized.





## Structure



### Abstraction

- defines the abstract interface
- maintains the **Implementor** reference

### RefinedAbstraction

- extends the interface defined by **Abstraction**

### Implementor

- defines the interface for implementation classes

### ConcreteImplementor

- implements the **Implementor** interface

## Example

The following Java (SE 6) program illustrates the 'shape' example given above and will output:

```
API1.circle at 1.000000:2.000000 radius 7.500000
API2.circle at 5.000000:7.000000 radius 27.500000
```

```
/** "Implementor" */
interface DrawingAPI {
    public void drawCircle(double x, double y, double radius);
}

/** "ConcreteImplementor" 1/2 */
class DrawingAPI1 implements DrawingAPI {
    public void drawCircle(double x, double y, double radius) {
        System.out.printf("API1.circle at %f:%f radius %f\n", x, y,
radius);
    }
}
```

```

/** "ConcreteImplementor" 2/2 */
class DrawingAPI2 implements DrawingAPI {
    public void drawCircle(double x, double y, double radius) {
        System.out.printf("API2.circle at %f:%f radius %f\n", x, y,
radius);
    }
}

/** "Abstraction" */
interface Shape {
    public void draw(); //
low-level
    public void resizeByPercentage(double pct); // high-level
}

/** "Refined Abstraction" */
class CircleShape implements Shape {
    private double x, y, radius;
    private DrawingAPI drawingAPI;
    public CircleShape(double x, double y, double radius, DrawingAPI
drawingAPI) {
        this.x = x; this.y = y; this.radius = radius;
        this.drawingAPI = drawingAPI;
    }

    // low-level i.e. Implementation specific
    public void draw() {
        drawingAPI.drawCircle(x, y, radius);
    }
    // high-level i.e. Abstraction specific
    public void resizeByPercentage(double pct) {
        radius *= pct;
    }
}

/** "Client" */
class BridgePattern {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[] {
            new CircleShape(1, 2, 3, new DrawingAPI1()),
            new CircleShape(5, 7, 11, new DrawingAPI2()),
        };

        for (Shape shape : shapes) {
            shape.resizeByPercentage(2.5);
            shape.draw();
        }
    }
}

```

```
}  
  
}
```

## References

- [1] <http://lepus.org.uk/ref/legend/legend.xml>
- [2] Gamma, E, Helm, R, Johnson, R, Vlissides, J: *Design Patterns*, page 151. Addison-Wesley, 1995

## External links

- Bridge in UML and in LePUS3 (<http://www.lepus.org.uk/ref/companion/Bridge.xml>) (a formal modelling language)
- "C# Design Patterns: The Bridge Pattern" (<http://www.informit.com/articles/article.aspx?p=30297>). *Sample Chapter*. From: James W. Cooper. *C# Design Patterns: A Tutorial* (<http://www.informit.com/store/product.aspx?isbn=0201844532>). Addison-Wesley. ISBN 0201844532.

# Composite pattern

---

In software engineering, the **composite pattern** is a partitioning design pattern. The composite pattern describes that a group of objects are to be treated in the same way as a single instance of an object. The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies. Implementing the composite pattern lets clients treat individual objects and compositions uniformly.<sup>[1]</sup>

## Motivation

When dealing with tree-structured data, programmers often have to discriminate between a leaf-node and a branch. This makes code more complex, and therefore, error prone. The solution is an interface that allows treating complex and primitive objects uniformly. In object-oriented programming, a composite is an object designed as a composition of one-or-more similar objects, all exhibiting similar functionality. This is known as a "has-a" relationship between objects<sup>[2]</sup>. The key concept is that you can manipulate a single instance of the object just as you would manipulate a group of them. The operations you can perform on all the composite objects often have a least common denominator relationship. For example, if defining a system to portray grouped shapes on a screen, it would be useful to define resizing a group of shapes to have the same effect (in some sense) as resizing a single shape.

## When to use

Composite can be used when clients should ignore the difference between compositions of objects and individual objects.<sup>[1]</sup> If programmers find that they are using multiple objects in the same way, and often have nearly identical code to handle each of them, then composite is a good choice; it is less complex in this situation to treat primitives and composites as homogeneous.

## Structure

### Component

- is the abstraction for all components, including composite ones
- declares the interface for objects in the composition
- (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate

### Leaf

- represents leaf objects in the composition
- implements all Component methods

### Composite

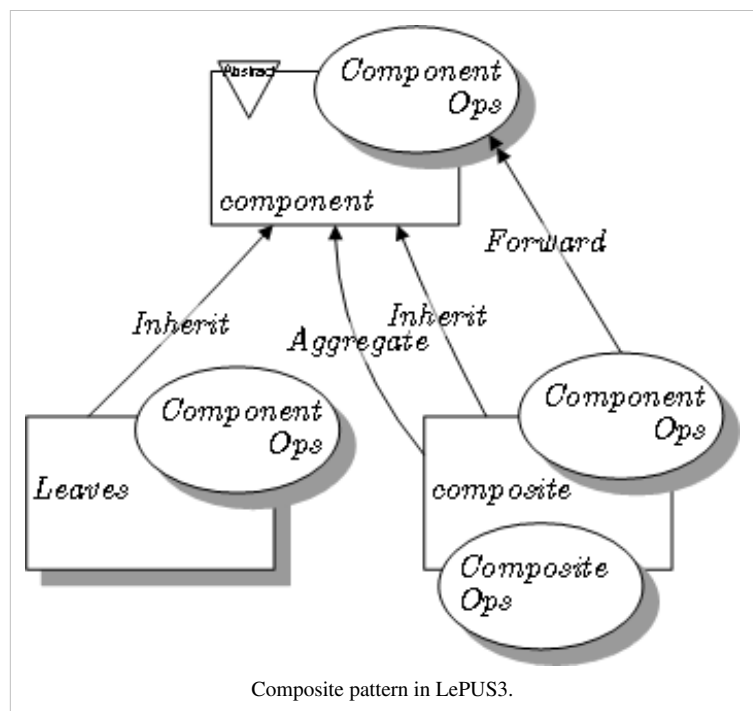
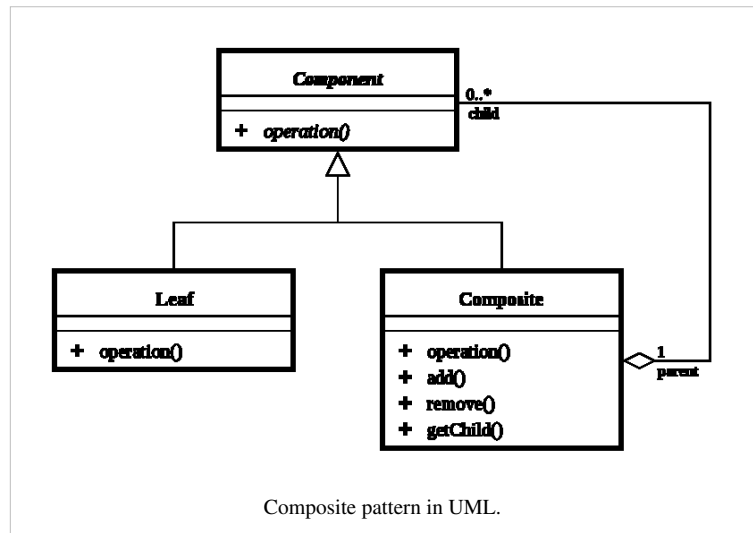
- represents a composite Component (component having children)
- implements methods to manipulate children
- implements all Component methods, generally by delegating them to its children

## Variation

As it is described in Design Patterns, the pattern also involves including the child-manipulation methods in the main Component interface, not just the Composite subclass. More recent descriptions sometimes omit these methods.<sup>[3]</sup>

## Example

The following example, written in Java, implements a graphic class, which can be either an ellipse or a composition of several graphics. Every graphic can be printed. In algebraic form,



```
Graphic = ellipse | GraphicList
GraphicList = empty | Graphic GraphicList
```

It could be extended to implement several other shapes (rectangle, etc.) and methods (translate, etc.).

```
import java.util.List;
import java.util.ArrayList;

/** "Component" */
interface Graphic {

    //Prints the graphic.
    public void print();
}

/** "Composite" */
class CompositeGraphic implements Graphic {

    //Collection of child graphics.
    private List<Graphic> mChildGraphics = new ArrayList<Graphic>();

    //Prints the graphic.
    public void print() {
        for (Graphic graphic : mChildGraphics) {
            graphic.print();
        }
    }

    //Adds the graphic to the composition.
    public void add(Graphic graphic) {
        mChildGraphics.add(graphic);
    }

    //Removes the graphic from the composition.
    public void remove(Graphic graphic) {
        mChildGraphics.remove(graphic);
    }
}

/** "Leaf" */
class Ellipse implements Graphic {

    //Prints the graphic.
    public void print() {
        System.out.println("Ellipse");
    }
}
```

```
/** Client */
public class Program {

    public static void main(String[] args) {
        //Initialize four ellipses
        Ellipse ellipse1 = new Ellipse();
        Ellipse ellipse2 = new Ellipse();
        Ellipse ellipse3 = new Ellipse();
        Ellipse ellipse4 = new Ellipse();

        //Initialize three composite graphics
        CompositeGraphic graphic = new CompositeGraphic();
        CompositeGraphic graphic1 = new CompositeGraphic();
        CompositeGraphic graphic2 = new CompositeGraphic();

        //Composes the graphics
        graphic1.add(ellipse1);
        graphic1.add(ellipse2);
        graphic1.add(ellipse3);

        graphic2.add(ellipse4);

        graphic.add(graphic1);
        graphic.add(graphic2);

        //Prints the complete graphic (four times the string "Ellipse").
        graphic.print();
    }
}
```

## External links

- Composite pattern description from the Portland Pattern Repository <sup>[4]</sup>
- Composite pattern in UML and in LePUS3, a formal modelling language <sup>[5]</sup>
- Class::Delegation on CPAN <sup>[6]</sup>
- "The End of Inheritance: Automatic Run-time Interface Building for Aggregated Objects" <sup>[7]</sup> by Paul Baranowski
- PerfectJPattern Open Source Project <sup>[8]</sup>, Provides componentized implementation of the Composite Pattern in Java
- [9] A persistent Java-based implementation
- Composite Design Pattern <sup>[10]</sup>

## References

- [1] Gamma, Erich; Richard Helm, Ralph Johnson, John M. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. pp. 395. ISBN 0201633612.
- [2] Scott Walters (2004). *Perl Design Patterns Book* (<http://perldesignpatterns.com/?CompositePattern>). .
- [3] Geary, David (13 Sep 2002). "A look at the Composite design pattern" (<http://www.javaworld.com/javaworld/jw-09-2002/jw-0913-designpatterns.html>). .
- [4] <http://c2.com/cgi/wiki?CompositePattern>
- [5] <http://www.lepus.org.uk/ref/companion/Composite.xml>
- [6] <http://search.cpan.org/dist/Class-Delegation/lib/Class/Delegation.pm>
- [7] <http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/149878>
- [8] <http://perfectjpattern.sourceforge.net/dp-composite.html>
- [9] <http://www.theresearchkitchen.com/blog/archives/57>
- [10] [http://sourcemaking.com/design\\_patterns/composite](http://sourcemaking.com/design_patterns/composite)

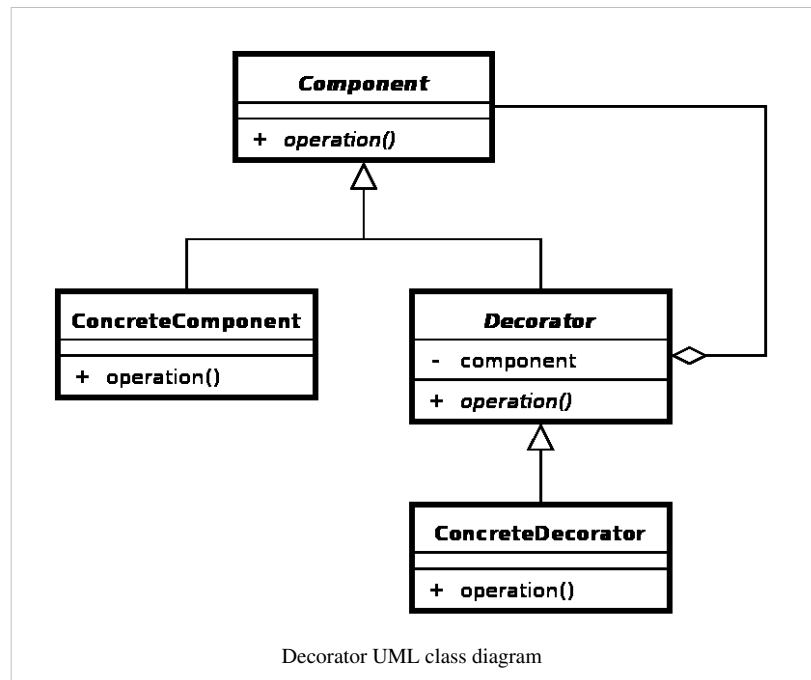
## Decorator pattern

In object-oriented programming, the **decorator pattern** is a design pattern that allows new/additional behaviour to be added to an existing object dynamically.

### Introduction

The decorator pattern can be used to make it possible to extend (decorate) the functionality of a certain object at runtime, independently of other instances of the same class, provided some groundwork is done at design time. This is achieved by designing a new *decorator* class that wraps the original class. This wrapping could be achieved by the following sequence of steps:

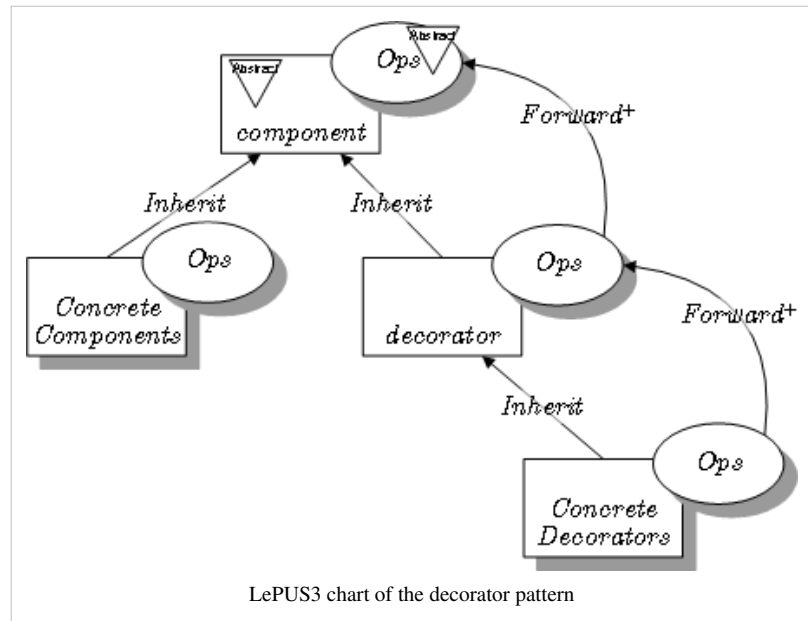
1. Subclass the original "Component" class into a "Decorator" class (see UML diagram);
2. In the Decorator class, add a Component pointer as a field;
3. Pass a Component to the Decorator constructor to initialize the Component pointer;
4. In the Decorator class, redirect all "Component" methods to the "Component" pointer; and



5. In the ConcreteDecorator class, override any Component method(s) whose behavior needs to be modified.

This pattern is designed so that multiple decorators can be stacked on top of each other, each time adding a new functionality to the overridden method(s).

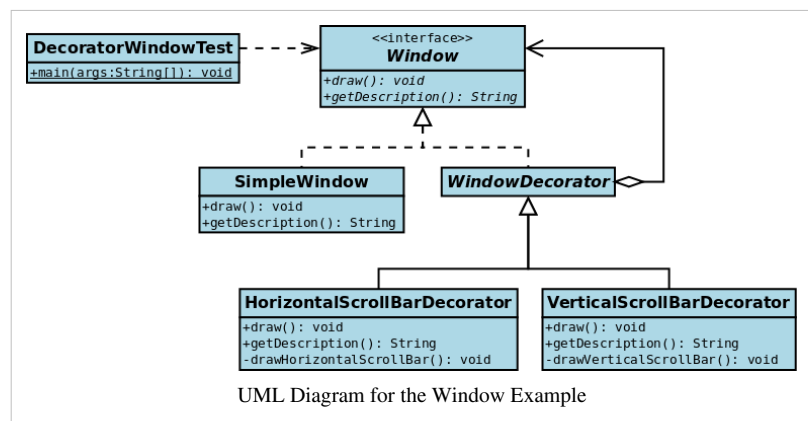
The decorator pattern is an alternative to subclassing. Subclassing adds behavior at compile time, and the change affects all instances of the original class; decorating can provide new behavior at runtime for individual objects.



This difference becomes most important when there are several *independent* ways of extending functionality. In some object-oriented programming languages, classes cannot be created at runtime, and it is typically not possible to predict, at design time, what combinations of extensions will be needed. This would mean that a new class would have to be made for every possible combination. By contrast, decorators are objects, created at runtime, and can be combined on a per-use basis. The I/O Streams implementations of both Java and the .NET Framework incorporate the decorator pattern.

## Motivation

As an example, consider a window in a windowing system. To allow scrolling of the window's contents, we may wish to add horizontal or vertical scrollbars to it, as appropriate. Assume windows are represented by instances of the *Window* class, and assume this class has no functionality for adding scrollbars. We could create a subclass *ScrollingWindow* that provides them, or we could create a



*ScrollingWindowDecorator* that adds this functionality to existing *Window* objects. At this point, either solution would be fine.

Now let's assume we also desire the ability to add borders to our windows. Again, our original *Window* class has no support. The *ScrollingWindow* subclass now poses a problem, because it has effectively created a new kind of window. If we wish to add border support to *all* windows, we must create subclasses *WindowWithBorder* and *ScrollingWindowWithBorder*. Obviously, this problem gets worse with every new feature to be added. For the decorator solution, we simply create a new *BorderedWindowDecorator*—at runtime, we can decorate existing windows with the *ScrollingWindowDecorator* or the *BorderedWindowDecorator* or both, as we see fit.

Another good example of where a decorator can be desired is when there is a need to restrict access to an object's properties or methods according to some set of rules or perhaps several parallel sets of rules (different user



credentials, etc.) In this case instead of implementing the access control in the original object it is left unchanged and unaware of any restrictions on its use, and it is wrapped in an access control decorator object, which can then serve only the permitted subset of the original object's interface.

## Examples

### Java

#### First Example (window/scrolling scenario)

The following Java example illustrates the use of decorators using the window/scrolling scenario.

```
// the Window interface
interface Window {
    public void draw(); // draws the Window
    public String getDescription(); // returns a description of the
Window
}

// implementation of a simple Window without any scrollbars
class SimpleWindow implements Window {
    public void draw() {
        // draw window
    }

    public String getDescription() {
        return "simple window";
    }
}
```

The following classes contain the decorators for all Window classes, including the decorator classes themselves..

```
// abstract decorator class - note that it implements Window
abstract class WindowDecorator implements Window {
    protected Window decoratedWindow; // the Window being decorated

    public WindowDecorator (Window decoratedWindow) {
        this.decoratedWindow = decoratedWindow;
    }

    public void draw() {
        decoratedWindow.draw();
    }
}

// the first concrete decorator which adds vertical scrollbar
functionality
class VerticalScrollBarDecorator extends WindowDecorator {
    public VerticalScrollBarDecorator (Window decoratedWindow) {
        super(decoratedWindow);
    }
}
```

```

    public void draw() {
        decoratedWindow.draw();
        drawVerticalScrollBar();
    }

    private void drawVerticalScrollBar() {
        // draw the vertical scrollbar
    }

    public String getDescription() {
        return decoratedWindow.getDescription() + ", including vertical
scrollbars";
    }
}

// the second concrete decorator which adds horizontal scrollbar
functionality
class HorizontalScrollBarDecorator extends WindowDecorator {
    public HorizontalScrollBarDecorator (Window decoratedWindow) {
        super(decoratedWindow);
    }

    public void draw() {
        decoratedWindow.draw();
        drawHorizontalScrollBar();
    }

    private void drawHorizontalScrollBar() {
        // draw the horizontal scrollbar
    }

    public String getDescription() {
        return decoratedWindow.getDescription() + ", including
horizontal scrollbars";
    }
}

```

Here's a test program that creates a `Window` instance which is fully decorated (i.e., with vertical and horizontal scrollbars), and prints its description:

```

public class DecoratedWindowTest {
    public static void main(String[] args) {
        // create a decorated Window with horizontal and vertical
scrollbars
        Window decoratedWindow = new HorizontalScrollBarDecorator (
            new VerticalScrollBarDecorator(new SimpleWindow()));
    }
}

```

```
        // print the Window's description
        System.out.println(decoratedWindow.getDescription());
    }
}
```

The output of this program is "simple window, including vertical scrollbars, including horizontal scrollbars". Notice how the `getDescription` method of the two decorators first retrieve the decorated Window's description and *decorates* it with a suffix.

### Second Example (coffee making scenario)

The next Java example illustrates the use of decorators using coffee making scenario. In this example, the scenario only includes cost and ingredients.

```
//The Coffee Interface defines the functionality of Coffee implemented
by decorator
public interface Coffee
{
    public double getCost(); // returns the cost of coffee

    public String getIngredient(); //returns the ingredients mixed with
    coffee
}

//implementation of simple coffee without any extra ingredients
public class SimpleCoffee implements Coffee
{
    double cost;
    String ingredient;

    public SimpleCoffee()
    {
        cost = 1;
        ingredient = "Coffee";
    }

    public double getCost()
    {
        return cost;
    }

    public String getIngredient()
    {
        return ingredient;
    }
}
```

The following classes contain the decorators for all `Coffee` classes, including the decorator classes themselves..

```
//abstract decorator class - note that it implements coffee interface
abstract public class CoffeeDecorator implements Coffee
{

    protected Coffee decoratedCoffee;
    protected String ingredientSeparator;

    public CoffeeDecorator(Coffee decoratedCoffee)
    {
        this.decoratedCoffee = decoratedCoffee;
        ingredientSeparator = ", ";
    }

    public double getCost() //note it implements the getCost function
defined in interface Coffee
    {
        return decoratedCoffee.getCost();
    }

    public String getIngredient()
    {
        return decoratedCoffee.getIngredient();
    }
}

//Decorator Milk that mixes milk with coffee
//note it extends CoffeeDecorator
public class Milk extends CoffeeDecorator
{

    double cost;
    String ingredient;

    public Milk(Coffee decoratedCoffee)
    {
        super(decoratedCoffee);
        cost = 0.5;
        ingredient = "Milk";
    }

    public double getCost()
    {
        return super.getCost() + cost;
    }

    public String getIngredient()
    {

```

```
        return super.getIngredient() + super.ingredientSeparator +
ingredient;
    }
}

//Decorator Whip that mixes whip with coffee
//note it extends CoffeeDecorator
public class Whip extends CoffeeDecorator
{

    double cost;
    String ingredient;

    public Whip(Coffee decoratedCoffee)
    {
        super(decoratedCoffee);
        cost = 0.7;
        ingredient = "Whip";
    }

    public double getCost()
    {
        return super.getCost() + cost;
    }

    public String getIngredient()
    {
        return super.getIngredient() + super.ingredientSeparator +
ingredient;
    }
}

//Decorator Sprinkles that mixes sprinkles with coffee
//note it extends CoffeeDecorator
public class Sprinkles extends CoffeeDecorator
{

    double cost;
    String ingredient;

    public Sprinkles(Coffee decoratedCoffee)
    {
        super(decoratedCoffee);
        cost = 0.2;
        ingredient = "Sprinkles";
    }
}
```

```
public double getCost()
{
    return super.getCost() + cost;
}

public String getIngredient()
{
    return super.getIngredient() + super.ingredientSeparator +
ingredient;
}
}
```

Here's a test program that creates a `Coffee` instance which is fully decorated (i.e., with milk, whip, sprinkles), and calculate cost of coffee and prints its ingredients:

```
public class Main
{
    public static void main(String[] args)
    {
        Coffee sampleCoffee = new SimpleCoffee();
        System.out.println("Cost: " + sampleCoffee.getCost() + "
Ingredient: " + sampleCoffee.getIngredient());

        sampleCoffee = new Milk(sampleCoffee);
        System.out.println("Cost: " + sampleCoffee.getCost() + "
Ingredient: " + sampleCoffee.getIngredient());

        sampleCoffee = new Sprinkles(sampleCoffee);
        System.out.println("Cost: " + sampleCoffee.getCost() + "
Ingredient: " + sampleCoffee.getIngredient());

        sampleCoffee = new Whip(sampleCoffee);
        System.out.println("Cost: " + sampleCoffee.getCost() + "
Ingredient: " + sampleCoffee.getIngredient());
    }
}
```

The output of this program is given below:

Cost: 1.0 Ingredient: Coffee

Cost: 1.5 Ingredient: Coffee, Milk

Cost: 1.7 Ingredient: Coffee, Milk, Sprinkles

Cost: 2.4 Ingredient: Coffee, Milk, Sprinkles, Whip

## Dynamic languages

The decorator pattern can also be implemented in dynamic languages with neither interfaces nor traditional OOP inheritance.

### JavaScript (coffee making scenario)

```
// Class to be decorated
function Coffee() {
    this.cost = function() {
        return 1;
    };
}

// Decorator A
function Milk(coffee) {
    this.cost = function() {
        return coffee.cost() + 0.5;
    };
}

// Decorator B
function Whip(coffee) {
    this.cost = function() {
        return coffee.cost() + 0.7;
    };
}

// Decorator C
function Sprinkles(coffee) {
    this.cost = function() {
        return coffee.cost() + 0.2;
    };
}

// Here's one way of using it
var coffee = new Milk(new Whip(new Sprinkles(new Coffee())));
alert( coffee.cost() );

// Here's another
var coffee = new Coffee();
coffee = new Sprinkles(coffee);
coffee = new Whip(coffee);
coffee = new Milk(coffee);
alert(coffee.cost());
```

## External links

- Decorator pattern description from the Portland Pattern Repository <sup>[1]</sup>
- Article "The Decorator Design Pattern <sup>[2]</sup>" (Java) by Antonio García and Stephen Wong
- Article "Using the Decorator Pattern <sup>[3]</sup>" (Java) by Budi Kurniawan
- Decorator in UML and in LePUS3 (a formal modelling language) <sup>[4]</sup>
- A Delphi approach <sup>[5]</sup> (Delphi)
- A JavaScript implementation <sup>[6]</sup>
- Decorator Pattern with Ruby in 8 Lines <sup>[7]</sup> (Ruby)
- PerfectJPattern Open Source Project <sup>[8]</sup>, Provides componentized implementation of the Decorator Pattern in Java

## References

- [1] <http://c2.com/cgi/wiki?DecoratorPattern>
- [2] <http://exciton.cs.rice.edu/JavaResources/DesignPatterns/DecoratorPattern.htm>
- [3] <http://www.onjava.com/pub/a/onjava/2003/02/05/decorator.html>
- [4] <http://www.lepus.org.uk/ref/companion/Decorator.xml>
- [5] <http://www.castle-cadenza.demon.co.uk/decorate.htm>
- [6] <http://www.phpied.com/a-javascript-implementation-of-the-decorator-pattern/>
- [7] <http://www.lukeredpath.co.uk/2006/9/6/decorator-pattern-with-ruby-in-8-lines>
- [8] <http://perfectjpattern.sourceforge.net/dp-decorator.html>

# Facade pattern

---

The **facade pattern** is a software engineering design pattern commonly used with Object-oriented programming. The name is by analogy to an architectural facade.

A facade is an object that provides a simplified interface to a larger body of code, such as a class library. A facade can:

- make a software library easier to use, understand and test, since the facade has convenient methods for common tasks;
- make code that uses the library more readable, for the same reason;
- reduce dependencies of outside code on the inner workings of a library, since most code uses the facade, thus allowing more flexibility in developing the system;
- wrap a poorly-designed collection of APIs with a single well-designed API (as per task needs).

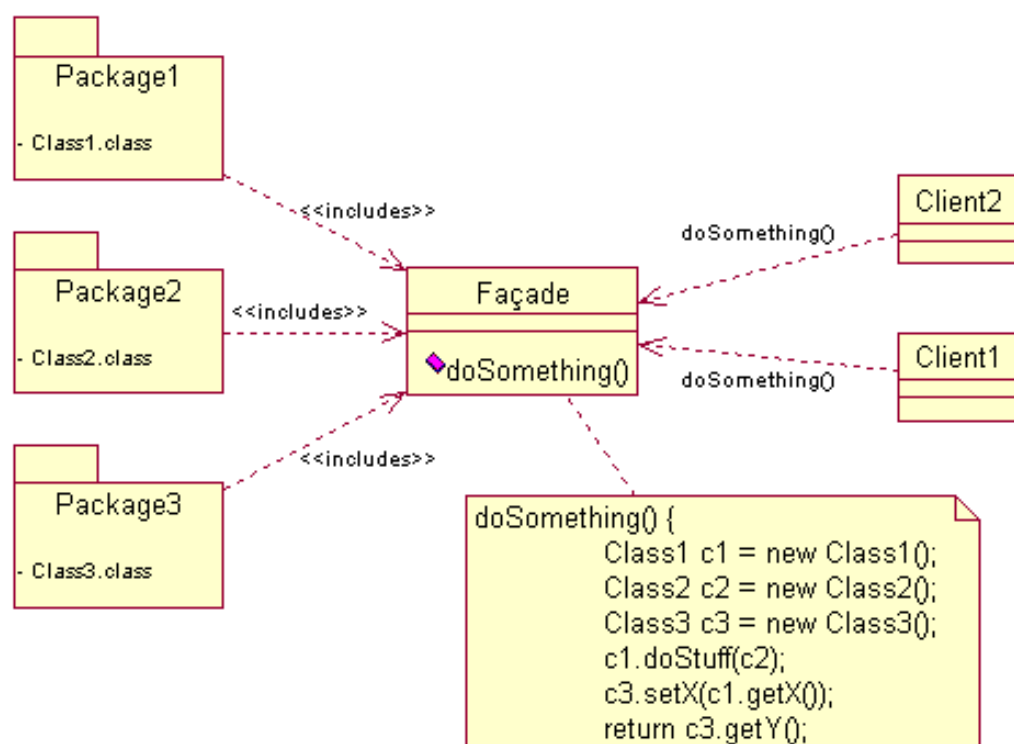
An Adapter is used when the wrapper must respect a particular interface and must support a polymorphic behavior. On the other hand, a facade is used when one wants an easier or simpler interface to work with.

From the French "façade", pronounced /fə'sɑːd/, wherein the "ç" diacritical (pronounced "sɑ") has been dropped according to Anglicization conventions.

---



## Structure



### Facade

The facade class abstracts Packages 1, 2, and 3 from the rest of the application.

### Clients

The objects using the Facade Pattern to access resources from the Packages.

## Example

This is an abstract example of how a client ("you") interacts with a facade (the "computer") to a complex system (internal computer parts, like CPU and HardDrive).

```

/* Complex parts */

class CPU {

    public void freeze() { ... }
    public void jump(long position) { ... }
    public void execute() { ... }

}

class Memory {

    public void load(long position, byte[] data) { ... }

}
  
```

```
class HardDrive {

    public byte[] read(long lba, int size) { ... }

}

/* Facade */

class Computer {

    private CPU cpu;
    private Memory memory;
    private HardDrive hardDrive;

    public Computer() {
        this.cpu = new CPU();
        this.memory = new Memory();
        this.hardDrive = new HardDrive();
    }

    public void startComputer() {
        cpu.freeze();
        memory.load(BOOT_ADDRESS, hardDrive.read(BOOT_SECTOR,
SECTOR_SIZE));
        cpu.jump(BOOT_ADDRESS);
        cpu.execute();
    }

}

/* Client */

class You {

    public static void main(String[] args) {
        Computer facade = new Computer();
        facade.startComputer();
    }

}
```

## External links

- Description from the Portland Pattern Repository <sup>[1]</sup>
- Description from the Net Objectives Repository <sup>[2]</sup>

## References

[1] <http://c2.com/cgi/wiki?FacadePattern>

[2] <http://www.netobjectivesrepository.com/TheFacadePattern>

# Flyweight pattern

---

*Flyweight* is a software design pattern. A flyweight is an object that minimizes memory use by sharing as much data as possible with other similar objects; it is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory. The term is named after the boxing weight class. Often some parts of the object state can be shared and it's common to put them in external data structures and pass them to the flyweight objects temporarily when they are used.

A classic example usage of the flyweight pattern is the data structures for graphical representation of characters in a word processor. It might be desirable to have, for each character in a document, a glyph object containing its font outline, font metrics, and other formatting data, but this would amount to hundreds or thousands of bytes for each character. Instead, for every character there might be a reference to a flyweight glyph object shared by every instance of the same character in the document; only the position of each character (in the document and/or the page) would need to be stored internally.

In other contexts the idea of sharing identical data structures is called hash consing.

## Example

The following programs illustrate the document example given above: the flyweights are called `FontData` in the Java example.

The examples illustrate the flyweight pattern used to reduce memory by loading only the data necessary to perform some immediate task from a large `Font` object into a much smaller `FontData` (flyweight) object.

```
import java.lang.ref.WeakReference;
import java.util.WeakHashMap;
import java.util.Collections;
import java.util.EnumSet;
import java.util.Set;
import java.awt.Color;

public final class FontData {

    enum FontEffect {
        BOLD, ITALIC, SUPERScript, SUBSCRIPT, STRIKETHROUGH
    }

    /**
     * A weak hash map will drop unused references to FontData.
     * Values have to be wrapped in WeakReferences,
```

```

    * because value objects in weak hash map are held by strong
    references.
    */
    private static final WeakHashMap<FontData, WeakReference<FontData>> FLY_WEIGHT_DATA =
        new WeakHashMap<FontData, WeakReference<FontData>>();
    private final int pointSize;
    private final String fontFace;
    private final Color color;
    private final Set<FontEffect> effects;

    private FontData(int pointSize, String fontFace, Color color,
EnumSet<FontEffect> effects) {
        this.pointSize = pointSize;
        this.fontFace = fontFace;
        this.color = color;
        this.effects = Collections.unmodifiableSet(effects);
    }

    public static FontData create(int pointSize, String fontFace, Color
color,
        FontEffect... effects) {
        EnumSet<FontEffect> effectsSet = EnumSet.noneOf(FontEffect.class);
        for (FontEffect fontEffect : effects) {
            effectsSet.add(fontEffect);
        }
        // We are unconcerned with object creation cost, we are
reducing overall memory consumption
        FontData data = new FontData(pointSize, fontFace, color,
effectsSet);

        FontData result = null;
        // Retrieve previously created instance with the given values
if it (still) exists
        WeakReference<FontData> ref = FLY_WEIGHT_DATA.get(data);
        if (ref != null) {
            result = ref.get();
        }

        // Store new font data instance if no matching instance exists
        if (result == null) {
            FLY_WEIGHT_DATA.put(data, new WeakReference<FontData> (data));
            result = data;
        }

        // return the single immutable copy with the given values
        return result;
    }

```

```

@Override
public boolean equals(Object obj) {
    if (obj instanceof FontData) {
        if (obj == this) {
            return true;
        }
        FontData other = (FontData) obj;
        return other.pointSize == pointSize &&
other.fontFace.equals(fontFace)
            && other.color.equals(color) &&
other.effects.equals(effects);
    }
    return false;
}

@Override
public int hashCode() {
    return (pointSize * 37 + effects.hashCode() * 13) *
fontFace.hashCode();
}

// Getters for the font data, but no setters. FontData is
immutable.
}

```

## External links

- Flyweight in UML and in LePUS3 (a formal modelling language) <sup>[1]</sup>
- Article "Make your apps fly - Implement Flyweight to improve performance <sup>[2]</sup>" by David Geary
- Article "Enhancing Web Application Performance with Caching <sup>[3]</sup>" by Neal Ford
- Sample Chapter "C# Design Patterns: The Flyweight Pattern <sup>[4]</sup>" by James W. Cooper
- Section "Flyweight Text Entry Fields (archive.org) <sup>[5]</sup>" from the RIDES Reference Manual by Allen Munro and Quentin A. Pizzini
- Description <sup>[6]</sup> from Portland's Pattern Repository
- Sourdough Design <sup>[7]</sup>
- Structural Patterns - Flyweight Pattern <sup>[8]</sup>
- Class::Flyweight - implement the flyweight pattern in OO perl <sup>[9]</sup>
- Boost.Flyweight - A generic C++ implementation <sup>[10]</sup>

## References

- [1] <http://www.lepus.org.uk/ref/companion/Flyweight.xml>
- [2] <http://www.javaworld.com/javaworld/jw-07-2003/jw-0725-designpatterns.html>
- [3] <http://theserverside.com/articles/article.tss?l=Caching>
- [4] <http://www.informit.com/articles/article.aspx?p=31563>
- [5] [http://web.archive.org/web/20070404160614/http://btl.usc.edu/rides/documentn/refMan/rf21\\_d.html](http://web.archive.org/web/20070404160614/http://btl.usc.edu/rides/documentn/refMan/rf21_d.html)
- [6] <http://c2.com/cgi/wiki?FlyweightPattern>
- [7] <http://sourdough.phpee.com/index.php?node=18>
- [8] [http://www.allapllabs.com/java\\_design\\_patterns/flyweight\\_pattern.htm](http://www.allapllabs.com/java_design_patterns/flyweight_pattern.htm)
- [9] [http://www.perlmonks.org/?node\\_id=94783](http://www.perlmonks.org/?node_id=94783)
- [10] <http://boost.org/libs/flyweight/index.html>

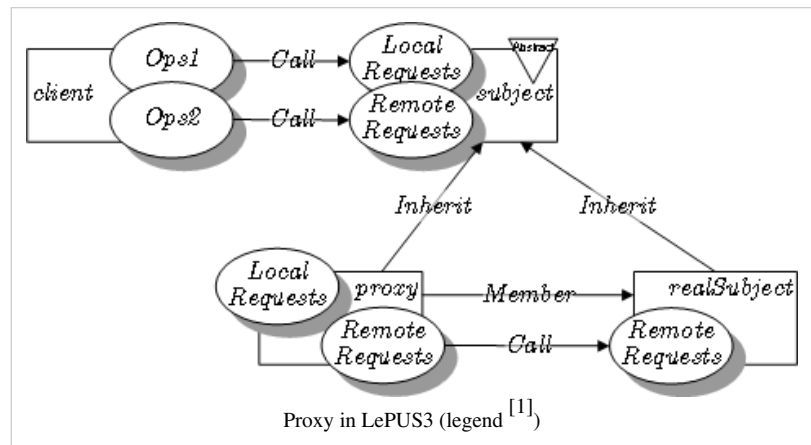
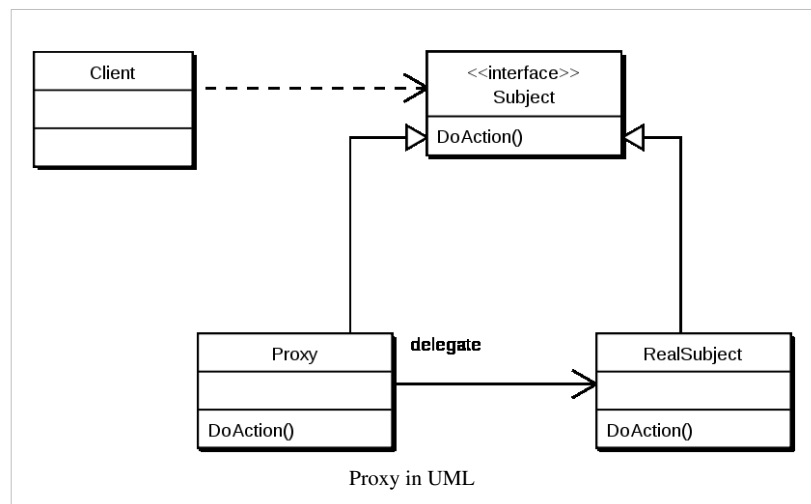
## Proxy pattern

In computer programming, the **proxy pattern** is a software design pattern.

A proxy, in its most general form, is a class functioning as an interface to something else. The proxy could interface to anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate.

A well-known example of the proxy pattern is a reference counting pointer object.

In situations where multiple copies of a complex object must exist the proxy pattern can be adapted to incorporate the flyweight pattern in order to reduce the application's memory footprint. Typically one instance of the complex object is created, and multiple proxy objects are created, all of which contain a reference to the single original complex object. Any operations performed on the proxies are forwarded to the original object. Once all instances of the proxy are out of scope, the complex object's memory may be deallocated.



## Example

The following Java example illustrates the "virtual proxy" pattern. The ProxyImage class is used to access a remote method.

```
interface Image
{
    public abstract void displayImage();
}

//on System A
class RealImage implements Image
{
    private String filename;

    public RealImage(String filename)
    {
        this.filename = filename;
        loadImageFromDisk();
    }

    private void loadImageFromDisk()
    {
        System.out.println("Loading " + filename);
    }

    public void displayImage()
    {
        System.out.println("Displaying " + filename);
    }
}

//on System B
class ProxyImage implements Image
{
    private String filename;
    private RealImage image;

    public ProxyImage(String filename)
    {
        this.filename = filename;
    }

    public void displayImage()
    {
        if (image == null)
        {
            image = new RealImage(filename);
        }
        image.displayImage();
    }
}
```

```
}

class ProxyExample
{
    public static void main(String[] args)
    {
        Image image1 = new ProxyImage("HiRes_10MB_Photo1");
        Image image2 = new ProxyImage("HiRes_10MB_Photo2");

        image1.displayImage(); // loading necessary
        image2.displayImage(); // loading necessary
        image1.displayImage(); // loading unnecessary
    }
}
```

The program's output is:

```
Loading      HiRes_10MB_Photo1
Displaying    HiRes_10MB_Photo1
Loading      HiRes_10MB_Photo2
Displaying    HiRes_10MB_Photo2
Displaying    HiRes_10MB_Photo1
```

## External links

- Proxy pattern in Java <sup>[1]</sup>
- Proxy pattern in UML and in LePUS3 (a formal modelling language) <sup>[2]</sup>
- Take control with the Proxy design pattern <sup>[3]</sup> by David Geary, JavaWorld.com
- PerfectJPattern Open Source Project <sup>[4]</sup>, Provides componentized implementation of the Proxy Pattern in Java
- Adapter vs. Proxy vs. Facade Pattern Comparison <sup>[5]</sup>
- Proxy Design Pattern <sup>[6]</sup>

## References

- [1] <http://wiki.java.net/bin/view/Javapedia/ProxyPattern>
- [2] <http://www.lepus.org.uk/ref/companion/Proxy.xml>
- [3] <http://www.javaworld.com/javaworld/jw-02-2002/jw-0222-designpatterns.html>
- [4] <http://perfectjpattern.sourceforge.net/dp-proxy.html>
- [5] <http://www.netobjectives.com/PatternRepository/index.php?title=AdapterVersusProxyVersusFacadePatternComparison>
- [6] [http://sourcemaking.com/design\\_patterns/proxy](http://sourcemaking.com/design_patterns/proxy)



# Chain-of-responsibility pattern

In Object Oriented Design, the **chain-of-responsibility pattern** is a design pattern consisting of a source of command objects and a series of **processing objects**. Each processing object contains a set of logic that describes the types of command objects that it can handle, and how to pass off those that it cannot handle to the next processing object in the chain. A mechanism also exists for adding new processing objects to the end of this chain.

In a variation of the standard chain-of-responsibility model, some handlers may act as dispatchers, capable of sending commands out in a variety of directions, forming a *tree of responsibility*. In some cases, this can occur recursively, with processing objects calling higher-up processing objects with commands that attempt to solve some smaller part of the problem; in this case recursion continues until the command is processed, or the entire tree has been explored. An XML interpreter (parsed, but not yet executed) might be a fitting example.

This pattern promotes the idea of loose coupling, which is considered a programming best practice.

## Example

The following Java code illustrates the pattern with the example of a logging class. Each logging handler decides if any action is to be taken at this log level and then passes the message on to the next logging handler. The output is:

```
Writing to stdout:   Entering function y.
Writing to stdout:   Step1 completed.
Sending via e-mail:  Step1 completed.
Writing to stdout:   An error has occurred.
Sending via e-mail:  An error has occurred.
Writing to stderr:   An error has occurred.
```

Note that this example should not be seen as a recommendation on how to write logging classes.

Also, note that in a 'pure' implementation of the chain of responsibility pattern, a logger would not pass responsibility further down the chain after handling a message. In this example, a message will be passed down the chain whether it is handled or not.

```
abstract class Logger {

    public static int ERR = 3;
    public static int NOTICE = 5;
    public static int DEBUG = 7;
    protected int mask;

    // The next element in the chain of responsibility
    protected Logger next;

    public Logger setNext(Logger l) {
        next = l;
        return l;
    }

    public void message(String msg, int priority) {
        if (priority <= mask) {
            writeMessage(msg);
        }
    }
}
```

```
    }
    if (next != null) {
        next.message(msg, priority);
    }
}

abstract protected void writeMessage(String msg);
}

class StdoutLogger extends Logger {

    public StdoutLogger(int mask) {
        this.mask = mask;
    }

    protected void writeMessage(String msg) {
        System.out.println("Writing to stdout: " + msg);
    }
}

class EmailLogger extends Logger {

    public EmailLogger(int mask) {
        this.mask = mask;
    }

    protected void writeMessage(String msg) {
        System.out.println("Sending via email: " + msg);
    }
}

class StderrLogger extends Logger {

    public StderrLogger(int mask) {
        this.mask = mask;
    }

    protected void writeMessage(String msg) {
        System.err.println("Sending to stderr: " + msg);
    }
}
```

```

public class ChainOfResponsibilityExample {

    public static void main(String[] args) {
        // Build the chain of responsibility
        Logger l, ll;
        ll = l = new StdoutLogger(Logger.DEBUG);
        ll = ll.setNext(new EmailLogger(Logger.NOTICE));
        ll = ll.setNext(new StderrLogger(Logger.ERR));

        // Handled by StdoutLogger
        l.message("Entering function y.", Logger.DEBUG);

        // Handled by StdoutLogger and EmailLogger
        l.message("Step1 completed.", Logger.NOTICE);

        // Handled by all three loggers
        l.message("An error has occurred.", Logger.ERR);
    }
}

```

Below is another example of this pattern in Java. In this example we have different roles, each having a fix purchase power limit and a successor. Every time a user in a role receives a purchase request, when it's over his limit, he just passes that request to his successor.

The PurchasePower abstract class with the abstract method processRequest.

```

abstract class PurchasePower {

    protected final double base = 500;
    protected PurchasePower successor;

    public void setSuccessor(PurchasePower successor) {
        this.successor = successor;
    }

    abstract public void processRequest(PurchaseRequest request);
}

```

Four implementations of the abstract class above: Manager, Director, Vice President, President

```

class ManagerPPower extends PurchasePower {

    private final double ALLOWABLE = 10 * base;

    public void processRequest(PurchaseRequest request) {
        if(request.getAmount() < ALLOWABLE) {
            System.out.println("Manager will approve $" +
request.getAmount());

```

```
    }  
    else if(successor != null) {  
        successor.processRequest(request);  
    }  
}  
  
class DirectorPPower extends PurchasePower {  
  
    private final double ALLOWABLE = 20 * base;  
  
    public void processRequest(PurchaseRequest request) {  
        if(request.getAmount() < ALLOWABLE) {  
            System.out.println("Director will approve $" +  
request.getAmount());  
        }  
        else if(successor != null) {  
            successor.processRequest(request);  
        }  
    }  
}  
  
class VicePresidentPPower extends PurchasePower {  
  
    private final double ALLOWABLE = 40 * base;  
  
    public void processRequest(PurchaseRequest request) {  
        if(request.getAmount() < ALLOWABLE) {  
            System.out.println("Vice President will approve $" +  
request.getAmount());  
        }  
        else if(successor != null) {  
            successor.processRequest(request);  
        }  
    }  
}  
  
class PresidentPPower extends PurchasePower {  
  
    private final double ALLOWABLE = 60 * base;  
  
    public void processRequest(PurchaseRequest request) {  
        if(request.getAmount() < ALLOWABLE) {  
            System.out.println("President will approve $" +  
request.getAmount());  
        }  
    }  
}
```

```
    }  
    else {  
        System.out.println( "Your request for $" +  
request.getAmount() + " needs a board meeting!");  
    }  
}  
  
}
```

The PurchaseRequest class with its Getter methods which keeps the request data in this example.

```
class PurchaseRequest {  
  
    private int number;  
    private double amount;  
    private String purpose;  
  
    public PurchaseRequest(int number, double amount, String purpose) {  
        this.number = number;  
        this.amount = amount;  
        this.purpose = purpose;  
    }  
  
    public double getAmount() {  
        return amount;  
    }  
  
    public void setAmount(double amt) {  
        amount = amt;  
    }  
  
    public String getPurpose() {  
        return purpose;  
    }  
  
    public void setPurpose(String reason) {  
        purpose = reason;  
    }  
  
    public int getNumber() {  
        return number;  
    }  
  
    public void setNumber(int num) {  
        number = num;  
    }  
  
}
```

And here a usage example, the successors are set like this: Manager -> Director -> Vice President -> President

```

class CheckAuthority {

    public static void main(String[] args) {
        ManagerPPower manager = new ManagerPPower();
        DirectorPPower director = new DirectorPPower();
        VicePresidentPPower vp = new VicePresidentPPower();
        PresidentPPower president = new PresidentPPower();
        manager.setSuccessor(director);
        director.setSuccessor(vp);
        vp.setSuccessor(president);

        //enter ctrl+c to kill.
        try{
            while (true) {
                System.out.println("Enter the amount to check who
should approve your expenditure.");
                System.out.print(">");
                double d = Double.parseDouble(new BufferedReader(new
InputStreamReader(System.in)).readLine());
                manager.processRequest(new PurchaseRequest(0, d,
"General"));
            }
        }
        catch(Exception e){
            System.exit(1);
        }
    }
}

```

## External links

- Article "The Chain of Responsibility pattern's pitfalls and improvements <sup>[1]</sup>" by Michael Xinsheng Huang
- Article "Follow the Chain of Responsibility <sup>[2]</sup>" by David Geary
- Article "Pattern Summaries: Chain of Responsibility <sup>[3]</sup>" by Mark Grand
- Behavioral Patterns - Chain of Responsibility Pattern <sup>[4]</sup>
- Descriptions from Portland Pattern Repository <sup>[5]</sup>
- Apache Jakarta Commons Chain <sup>[6]</sup>
- PerfectJPattern Open Source Project <sup>[7]</sup>, Provides a context-free and type-safe implementation of the Chain of Responsibility Pattern in Java
- Chain.NET(NChain) <sup>[8]</sup> - Ready-to-use, generic and lightweight implementation of the Chain of Responsibility pattern for .NET and Mono
- Chain of Responsibility Design Pattern <sup>[9]</sup> - An Example
- Sourcemaking Tutorial <sup>[10]</sup>

## References

- [1] <http://www.javaworld.com/javaworld/jw-08-2004/jw-0816-chain.html>
- [2] <http://www.javaworld.com/javaworld/jw-08-2003/jw-0829-designpatterns.html>
- [3] <http://developer.com/java/other/article.php/631261>
- [4] [http://allaplabs.com/java\\_design\\_patterns/chain\\_of\\_responsibility\\_pattern.htm](http://allaplabs.com/java_design_patterns/chain_of_responsibility_pattern.htm)
- [5] <http://c2.com/cgi/wiki?ChainOfResponsibilityPattern>
- [6] <http://jakarta.apache.org/commons/chain/>
- [7] <http://perfectjpattern.sourceforge.net/dp-chainofresponsibility.html>
- [8] <http://nchain.sourceforge.net>
- [9] <http://patrickmcguirk.net/papers/chainofresponsibility.html>
- [10] [http://sourcemaking.com/design\\_patterns/chain\\_of\\_responsibility](http://sourcemaking.com/design_patterns/chain_of_responsibility)

## Command pattern

---

In object-oriented programming, the **command pattern** is a design pattern in which an object is used to represent and encapsulate all the information needed to call a method at a later time. This information includes the method name, the object that owns the method and values for the method parameters.

Three terms always associated with the command pattern are *client*, *invoker* and *receiver*. The *client* instantiates the command object and provides the information required to call the method at a later time. The *invoker* decides when the method should be called. The *receiver* is an instance of the class that contains the method's code.

Using command objects makes it easier to construct general components that need to delegate, sequence or execute method calls at a time of their choosing without the need to know the owner of the method or the method parameters.

## Uses

Command objects are useful for implementing:

### Multi-level undo

If all user actions in a program are implemented as command objects, the program can keep a stack of the most recently executed commands. When the user wants to undo a command, the program simply pops the most recent command object and executes its `undo()` method.

### Transactional behavior

Similar to undo, a database engine or software installer may keep a list of operations that have been or will be performed. Should one of them fail, all others can be reverted or discarded (usually called *rollback*). For example, if two database tables which refer to each other must be updated, and the second update fails, the transaction can be rolled back, so that the first table does not now contain an invalid reference.

### Progress bars

Suppose a program has a sequence of commands that it executes in order. If each command object has a `getEstimatedDuration()` method, the program can easily estimate the total duration. It can show a progress bar that meaningfully reflects how close the program is to completing all the tasks.

### Wizards

Often a wizard presents several pages of configuration for a single action that happens only when the user clicks the "Finish" button on the last page. In these cases, a natural way to separate user interface code from application code is to implement the wizard using a command object. The command object is created when the wizard is first displayed. Each wizard page stores its GUI changes in the command object, so the object is populated as the user progresses. "Finish" simply triggers a call to `execute()`. This way, the command class contains no user interface code.

---

### GUI buttons and menu items

In Swing and Borland Delphi programming, an Action is a command object. In addition to the ability to perform the desired command, an Action may have an associated icon, keyboard shortcut, tooltip text, and so on. A toolbar button or menu item component may be completely initialized using only the Action object.

### Thread pools

A typical, general-purpose thread pool class might have a public `addTask()` method that adds a work item to an internal queue of tasks waiting to be done. It maintains a pool of threads that execute commands from the queue. The items in the queue are command objects. Typically these objects implement a common interface such as `java.lang.Runnable` that allows the thread pool to execute the command even though the thread pool class itself was written without any knowledge of the specific tasks for which it would be used.

### Macro recording

If all user actions are represented by command objects, a program can record a sequence of actions simply by keeping a list of the command objects as they are executed. It can then "play back" the same actions by executing the same command objects again in sequence. If the program embeds a scripting engine, each command object can implement a `toScript()` method, and user actions can then be easily recorded as scripts.

### Networking

It is possible to send whole command objects across the network to be executed on the other machines, for example player actions in computer games.

### Parallel Processing

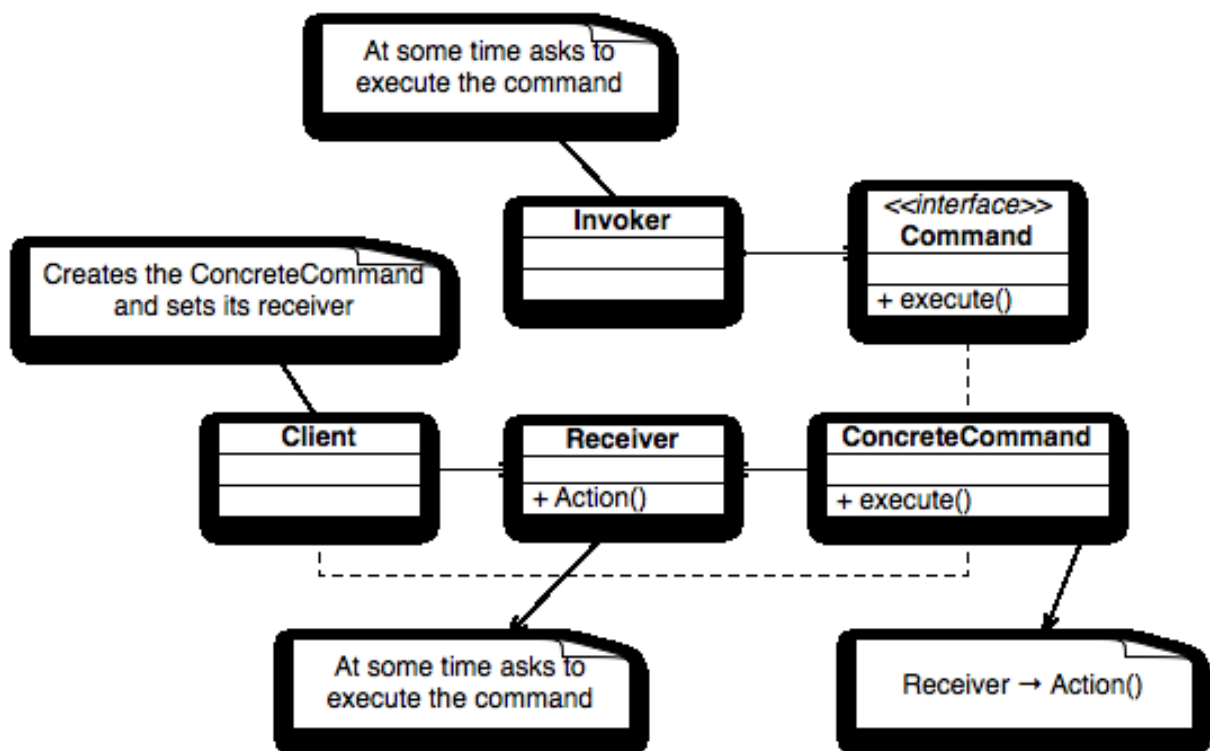
Where the commands are written as tasks to a shared resource and executed by many threads in parallel (possibly on remote machines -this variant is often referred to as the Master/Worker pattern)

### Mobile Code

Using languages such as Java where code can be streamed/slurped from one location to another via `URLClassloaders` and `Codebases` the commands can enable new behavior to be delivered to remote locations (EJB Command, Master Worker)



## Structure



- The explanation for the Receiver block above should be "The actual work to be done by the command (action)"

## Terminology

The terminology used to describe command pattern implementations is not consistent and can therefore be confusing. This is the result of ambiguity, the use of synonyms, and implementations that may obscure the original pattern by going well beyond it.

### 1. Ambiguity.

1. The term **command** is ambiguous. For example, *move up*, *move up* may refer to a single (move up) command that should be executed twice, or it may refer to two commands, each of which happens to do the same thing (move up). If the former command is added twice to an undo stack, both items on the stack refer to the same command instance. This may be appropriate when a command can always be undone the same way (e.g. move down). Both the Gang of Four and the Java example below use this interpretation of the term *command*. On the other hand, if the latter commands are added to an undo stack, the stack refers to two separate objects. This may be appropriate when each object on the stack must contain information that allows the command to be undone. For example, to undo a *delete selection* command, the object may contain a copy of the deleted text so that it can be re-inserted if the *delete selection* command must be undone. Note that using a separate object for each invocation of a command is also an example of the chain of responsibility pattern.
2. The term **execute** is also ambiguous. It may refer to running the code identified by the command object's *execute* method. However, in Microsoft's Windows Presentation Foundation a command is considered to have been executed when the command's *execute* method has been invoked, but that does not necessarily mean that the application code has run. That occurs only after some further event processing.

### 2. Synonyms and homonyms.

1. **Client, Source, Invoker**: the button, toolbar button, or menu item clicked, the shortcut key pressed by the user.

2. **Command Object, Routed Command Object, Action Object:** a singleton object (e.g. there is only one CopyCommand object), which knows about shortcut keys, button images, command text, etc. related to the command. A source/invoke object calls the Command/Action object's *execute/performAction* method. The Command/Action object notifies the appropriate source/invoke objects when the availability of a command/action has changed. This allows buttons and menu items to become inactive (grayed out) when a command/action cannot be executed/performed.
  3. **Receiver, Target Object:** the object that is about to be copied, pasted, moved, etc. The receiver object owns the method that is called by the command's *execute* method. The receiver is typically also the target object. For example, if the receiver object is a *cursor* and the method is called *moveUp*, then one would expect that the cursor is the target of the moveUp action. On the other hand, if the code is defined by the command object itself, the target object will be a different object entirely.
  4. **Command Object, routed event args, event object:** the object that is passed from the source to the Command/Action object, to the Target object to the code that does the work. Each button click or shortcut key results in a new command/event object. Some implementations add more information to the command/event object as it is being passed from one object (e.g. CopyCommand) to another (e.g. document section). Other implementations put command/event objects in other event objects (like a box inside a bigger box) as they move along the line, to avoid naming conflicts. (See also chain of responsibility pattern).
  5. **Handler, ExecutedRoutedEventHandler, method, function:** the actual code that does the copying, pasting, moving, etc. In some implementations the handler code is part of the command/action object. In other implementations the code is part of the Receiver/Target Object, and in yet other implementations the handler code is kept separate from the other objects.
  6. **Command Manager, Undo Manager, Scheduler, Queue, Dispatcher, Invoker:** an object that puts command/event objects on an undo stack or redo stack, or that holds on to command/event objects until other objects are ready to act on them, or that routes the command/event objects to the appropriate receiver/target object or handler code.
3. Implementations that go well beyond the original command pattern.
1. Microsoft's Windows Presentation Foundation <sup>[1]</sup> (WPF), introduces routed commands, which combine the command pattern with event processing. As a result the command object no longer contains a reference to the target object nor a reference to the application code. Instead, invoking the command object's *execute* command results in a so called *Executed Routed Event* which during the event's tunneling or bubbling may encounter a so called *binding* object which identifies the target and the application code, which is executed at that point.

## Examples

Consider a "simple" switch. In this example we configure the Switch with 2 commands: to turn the light on and to turn the light off.

A benefit of this particular implementation of the command pattern is that the switch can be used with any device, not just a light - the Switch in the following example turns a light on and off, but the Switch's constructor is able to accept any subclasses of Command for its 2 parameters. For example, you could configure the Switch to start an engine.

Note: A criticism of the sample application below is that it doesn't truly model an electrical circuit. An electrical switch is dumb. A true binary switch knows only that it is either on or off. It does not know about or have any direct relationship with the various loads that might be attached to a circuit (ie. you hook up a switch to a circuit, not directly to a load). The switch should simply publish an event of its current state (either an ON or OFF). The circuit then should contain a State Engine which manages circuit states at various points along it (by listening to switch events) in order to properly accommodate complex circuits with multiple loads and switches. Each load is then conditional to a specific circuit's state (not directly to any specific switch). In conclusion, the switch itself should not

be aware of any lamp details.

```
/* The Invoker class */
public class Switch {

    private Command flipUpCommand;
    private Command flipDownCommand;

    public Switch(Command flipUpCmd, Command flipDownCmd) {
        this.flipUpCommand = flipUpCmd;
        this.flipDownCommand = flipDownCmd;
    }

    public void flipUp() {
        flipUpCommand.execute();
    }

    public void flipDown() {
        flipDownCommand.execute();
    }
}

/* The Receiver class */
public class Light {

    public Light() { }

    public void turnOn() {
        System.out.println("The light is on");
    }

    public void turnOff() {
        System.out.println("The light is off");
    }
}

/* The Command interface */
public interface Command {
    void execute();
}

/* The Command for turning the light on in North America, or turning
the light off in most other places */
public class FlipUpCommand implements Command {

    private Light theLight;

    public FlipUpCommand(Light light) {
```

```
        this.theLight=light;
    }

    public void execute() {
        theLight.turnOn();
    }
}

/* The Command for turning the light off in North America, or turning
the light on in most other places */
public class FlipDownCommand implements Command {

    private Light theLight;

    public FlipDownCommand(Light light) {
        this.theLight=light;
    }

    public void execute() {
        theLight.turnOff();
    }
}

/* The test class or client */
public class PressSwitch {

    public static void main(String[] args) {
        Light lamp = new Light();
        Command switchUp = new FlipUpCommand(lamp);
        Command switchDown = new FlipDownCommand(lamp);

        // See criticism of this model above:
        // The switch itself should not be aware of lamp details
        (switchUp, switchDown)
        // either directly or indirectly
        Switch s = new Switch(switchUp, switchDown);

        try {
            if (args[0].equalsIgnoreCase("ON")) {
                s.flipUp();
            } else if (args[0].equalsIgnoreCase("OFF")) {
                s.flipDown();
            } else {
                System.out.println("Argument \"ON\" or \"OFF\" is
required.");
            }
        } catch (Exception e) {
```

```
        System.out.println("Arguments required.");
    }
}
}
```

## References

- Freeman, E; Sierra, K; Bates, B (2004). Head First Design Patterns. O'Reilly.
- GoF - Design Patterns

## External links

- <http://c2.com/cgi/wiki?CommandPattern>
- <http://www.javaworld.com/javaworld/jvatips/jw-jvatip68.html>
- PerfectJPattern Open Source Project <sup>[2]</sup>, Provides a componentized i.e. context-free and type-safe implementation of the Command Pattern in Java
- Command Pattern Video Tutorial <sup>[3]</sup>, Teaches the Command Pattern using StarCraft references with a free downloadable video tutorial

## References

- [1] <http://msdn.microsoft.com/en-us/library/ms752308.aspx>  
[2] <http://perfectjpattern.sourceforge.net/dp-command.html>  
[3] <http://johnlindquist.com/2010/09/09/patterncraft-command-pattern>

# Interpreter pattern

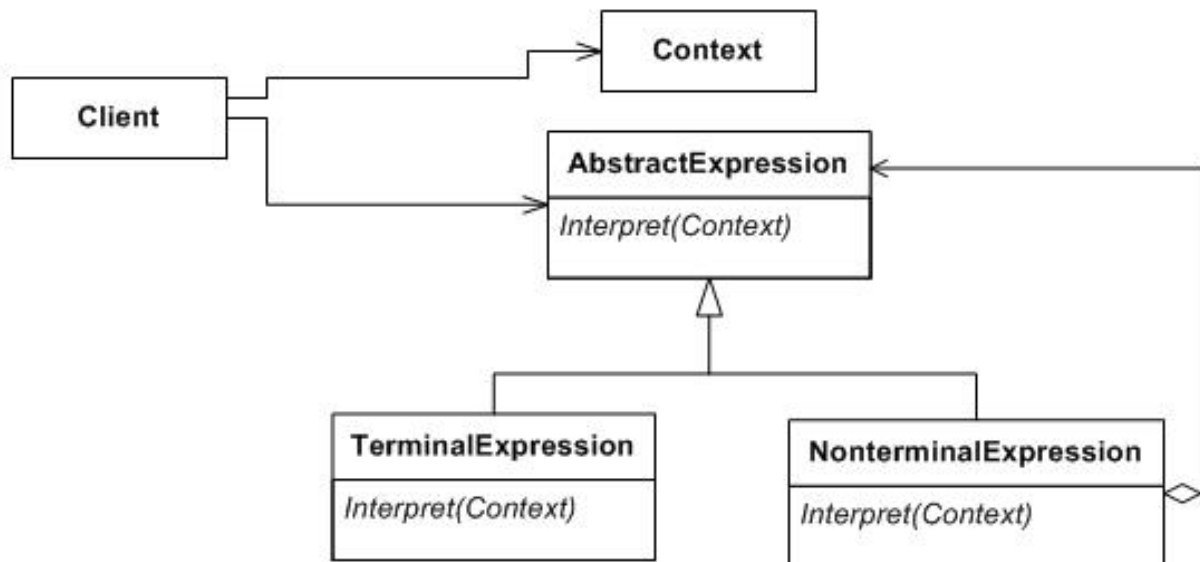
---

In computer programming, the **interpreter pattern** is a particular design pattern. The interpreter pattern specifies how to evaluate sentences in a language. The basic idea is to have a class for each symbol (terminal or nonterminal) in a specialized computer language. The syntax tree of a sentence in the language is an instance of the composite pattern and is used to evaluate (interpret) the sentence.<sup>[1]</sup>

## Uses for the Interpreter pattern

- Specialized database query languages such as SQL.
- Specialized computer languages which are often used to describe communication protocols.
- Most general-purpose computer languages actually incorporate several specialized languages.

## Structure



## Example

The following Reverse Polish notation example illustrates the interpreter pattern. The grammar expression `::= plus | minus | variable | number`

`plus ::= expression expression '+'`

`minus ::= expression expression '-'`

`variable ::= 'a' | 'b' | 'c' | ... | 'z'`

`digit = '0' | '1' | ... | '9'`

`number ::= digit | digit number`

defines a language which contains reverse polish expressions like:

`a b +`

`a b c + -`

`a b + c a - -`

Following the interpreter pattern there is a class for each grammar rule.

```

import java.util.HashMap;

interface Expression {
    public int interpret (HashMap<String,Expression> variables);
}

class Number implements Expression {
    private int number;
    public Number(int number) { this.number = number; }
    public int interpret (HashMap<String,Expression> variables) { return number; }
}
  
```

```

}

class Plus implements Expression {
    Expression leftOperand;
    Expression rightOperand;
    public Plus(Expression left, Expression right) {
        leftOperand = left;
        rightOperand = right;
    }

    public int interpret(HashMap<String,Expression> variables) {
        return leftOperand.interpret(variables) +
rightOperand.interpret(variables);
    }
}

class Minus implements Expression {
    Expression leftOperand;
    Expression rightOperand;
    public Minus(Expression left, Expression right) {
        leftOperand = left;
        rightOperand = right;
    }

    public int interpret(HashMap<String,Expression> variables) {
        return leftOperand.interpret(variables) -
rightOperand.interpret(variables);
    }
}

class Variable implements Expression {
    private String name;
    public Variable(String name) { this.name = name; }
    public int interpret(HashMap<String,Expression> variables) {
        if(null==variables.get(name)) return 0; //Either return new
Number(0) .
        return variables.get(name).interpret(variables);
    }
}

```

While the interpreter pattern does not address parsing<sup>[2]</sup> a parser is provided for completeness.

```

import java.util.HashMap;
import java.util.Stack;

class Evaluator implements Expression {
    private Expression syntaxTree;

```

```

public Evaluator(String expression) {
    Stack<Expression> expressionStack = new Stack<Expression>();
    for (String token : expression.split(" ")) {
        if (token.equals("+")) {
            Expression subExpression = new
Plus(expressionStack.pop(), expressionStack.pop());
            expressionStack.push( subExpression );
        }
        else if (token.equals("-")) {
            // it's necessary remove first the right operand from
the stack

            Expression right = expressionStack.pop();
            // ..and after the left one
            Expression left = expressionStack.pop();
            Expression subExpression = new Minus(left, right);
            expressionStack.push( subExpression );
        }
        else
            expressionStack.push( new Variable(token) );
    }
    syntaxTree = expressionStack.pop();
}

public int interpret(HashMap<String,Expression> context) {
    return syntaxTree.interpret(context);
}
}

```

Finally evaluating the expression "w x z - +" with w = 5, x = 10, and z = 42.

```

import java.util.HashMap;

public class InterpreterExample {
    public static void main(String[] args) {
        String expression = "w x z - +";
        Evaluator sentence = new Evaluator(expression);
        HashMap<String,Expression> variables = new HashMap<String,Expression>();
        variables.put("w", new Number(5));
        variables.put("x", new Number(10));
        variables.put("z", new Number(42));
        int result = sentence.interpret(variables);
        System.out.println(result);
    }
}

```



## References

- [1] Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. pp. 243 ISBN 0-201-63361-2
- [2] Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. pp. 247 ISBN 0-201-63361-2

## External links

- Interpreter implementation (<http://lukaszwoebel.pl/blog/interpreter-design-pattern>) in Ruby
- SourceMaking tutorial ([http://sourcemaking.com/design\\_patterns/interpreter](http://sourcemaking.com/design_patterns/interpreter))

# Iterator pattern

---

In object-oriented programming, the **Iterator pattern** is a design pattern in which iterators are used to access the elements of an aggregate object sequentially without exposing its underlying implementation. An **Iterator object** encapsulates the internal structure of how the iteration occurs.

For example, a tree, linked list, hash table, and an array all need to be iterated with the methods search, sort, and next. Rather than having 12 different methods to manage (one implementation for each of the previous three methods in each structure), using the iterator pattern yields just seven: one for each class using the iterator to obtain the iterator and one for each of the three methods. Therefore, to run the search method on the array, you would call `array.search()`, which hides the call to `array.iterator.search()`.

## Examples

```
public class BitSetIterator implements Iterator<Boolean> {

    private final BitSet bitset;
    private int index = 0;

    public BitSetIterator(BitSet bitset) {
        this.bitset = bitset;
    }

    public boolean hasNext() {
        return index < bitset.length();
    }

    public Boolean next() {
        if (index >= bitset.length()) {
            throw new NoSuchElementException();
        }
        return bitset.get(index++);
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

```
}
```

Two different usage examples:

```
public class TestClientBitSet {

    public static void main(String[] args) {
        // create BitSet and set some bits
        BitSet bitset = new BitSet();
        bitset.set(1);
        bitset.set(3400);
        bitset.set(20);
        bitset.set(47);
        for (BitSetIterator iter = new BitSetIterator(bitset);
iter.hasNext(); ) {
            Boolean b = iter.next();
            String tf = (b.booleanValue() ? "T" : "F");
            System.out.print(tf);
        }
        System.out.println();
    }
}

public class TestClientIterator {

    public static void main(String[] args) {
        Collection<Object> al = new ArrayList<Object>();
        al.add(new Integer(42));
        al.add("test");
        al.add(new Double("-12.34"));
        for (Iterator<Object> iter = al.iterator(); iter.hasNext(); ) {
            System.out.println(iter.next());
        }
        for (Object o : al) {
            System.out.println(o);
        }
    }
}
```

## Language-specific syntax

Due to the utility of the iterator pattern for orthogonalizing traversal and traversal-based algorithms, some languages standardize syntax. C++ and Python are notable examples.

### C++

The designers of the C++ standard template library observed that C already had a syntax for iterators: the syntax of pointers. In C++, a class can overload all of the pointer operations, so an iterator can be implemented that acts more or less like a pointer, complete with dereference, increment, and decrement. This has the advantage that C++ algorithms such as `std::sort` can immediately be applied to plain old memory buffers, and that there is no new syntax to learn. However, it requires an "end" iterator to test for equality, rather than allowing an iterator to know that it has reached the end. In C++ language, we say that an iterator models the iterator concept.

### Python

Python prescribes a syntax for iterators as part of the language itself, so that language keywords such as `for` work with what Python calls sequences. A sequence has an `__iter__()` method that returns an iterator object. The "iterator protocol" which requires `next()` return the next element or raise a `StopIteration` exception upon reaching the end of the sequence. Iterators also provide an `__iter__()` method returning themselves so that they can also be iterated over e.g., using a `for` loop. (In Python 3, `next()` was replaced with `__next__()`.)<sup>[1]</sup>

## References

[1] <http://docs.python.org/library/stdtypes.html>

## External links

- Object iteration (<http://us3.php.net/manual/en/language.oop5.iterations.php>) in PHP
- Iterator Pattern (<http://www.dofactory.com/Patterns/PatternIterator.aspx>) in C#
- Iterator pattern in UML and in LePUS3 (a formal modelling language) (<http://www.lepus.org.uk/ref/companion/Iterator.xml>)
- SourceMaking tutorial ([http://sourcemaking.com/design\\_patterns/iterator](http://sourcemaking.com/design_patterns/iterator))

# Mediator pattern

The **mediator pattern**, one of the 23 design patterns described in *Design Patterns: Elements of Reusable Object-Oriented Software*, provides a unified interface to a set of interfaces in a subsystem. This pattern is considered to be a behavioral pattern due to the way it can alter the program's running behavior.

Usually a program is made up of a (sometimes large) number of classes. So the logic and computation is distributed among these classes. However, as more classes are developed in a program, especially during maintenance and/or refactoring, the problem of communication between these classes may become more complex. This makes the program harder to read and maintain. Furthermore, it can become difficult to change the program, since any change may affect code in several other classes.

With the **mediator pattern** communication between objects is encapsulated with a **mediator** object. Objects no longer communicate directly with each other, but instead communicate through the mediator. This reduces the dependencies between communicating objects, thereby lowering the coupling.

## Example

```
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

//Colleague interface
interface Command {
    void execute();
}

//Concrete mediator
class Mediator {

    BtnView btnView;
    BtnSearch btnSearch;
    BtnBook btnBook;
    LblDisplay show;

    //....
    void registerView(BtnView v) {
        btnView = v;
    }

    void registerSearch(BtnSearch s) {
        btnSearch = s;
    }
}
```

```
void registerBook(BtnBook b) {
    btnBook = b;
}

void registerDisplay(LblDisplay d) {
    show = d;
}

void book() {
    btnBook.setEnabled(false);
    btnView.setEnabled(true);
    btnSearch.setEnabled(true);
    show.setText("booking...");
}

void view() {
    btnView.setEnabled(false);
    btnSearch.setEnabled(true);
    btnBook.setEnabled(true);
    show.setText("viewing...");
}

void search() {
    btnSearch.setEnabled(false);
    btnView.setEnabled(true);
    btnBook.setEnabled(true);
    show.setText("searching...");
}
}

//A concrete colleague
class BtnView extends JButton implements Command {

    Mediator med;

    BtnView(ActionListener al, Mediator m) {
        super("View");
        addActionListener(al);
        med = m;
        med.registerView(this);
    }

    public void execute() {
        med.view();
    }
}
```

```
}

//A concrete colleague
class BtnSearch extends JButton implements Command {

    Mediator med;

    BtnSearch(ActionListener al, Mediator m) {
        super("Search");
        addActionListener(al);
        med = m;
        med.registerSearch(this);
    }

    public void execute() {
        med.search();
    }

}

//A concrete colleague
class BtnBook extends JButton implements Command {

    Mediator med;

    BtnBook(ActionListener al, Mediator m) {
        super("Book");
        addActionListener(al);
        med = m;
        med.registerBook(this);
    }

    public void execute() {
        med.book();
    }

}

class LblDisplay extends JLabel {

    Mediator med;

    LblDisplay(Mediator m) {
        super("Just start...");
        med = m;
        med.registerDisplay(this);
        setFont(new Font("Arial", Font.BOLD, 24));
    }

}
```

```
    }

}

class MediatorDemo extends JFrame implements ActionListener {

    Mediator med = new Mediator();

    MediatorDemo() {
        JPanel p = new JPanel();
        p.add(new BtnView(this, med));
        p.add(new BtnBook(this, med));
        p.add(new BtnSearch(this, med));
        getContentPane().add(new LblDisplay(med), "North");
        getContentPane().add(p, "South");
        setSize(400, 200);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent ae) {
        Command cmd = (Command) ae.getSource();
        cmd.execute();
    }

    public static void main(String[] args) {
        new MediatorDemo();
    }

}
```

## Participants

**Mediator** - defines the interface for communication between *Colleague* objects

**ConcreteMediator** - implements the Mediator interface and coordinates communication between *Colleague* objects. It is aware of all the *Colleagues* and their purpose with regards to inter communication.

**ConcreteColleague** - communicates with other *Colleagues* through its *Mediator*

## External links

- [Screencast showing how to implement the pattern in C#](#) <sup>[1]</sup>
- [Mediator Design Pattern](#) <sup>[2]</sup>

## References

- [1] <http://www.dimecasts.net/Casts/CastFeedDetails/124>
- [2] [http://sourcemaking.com/design\\_patterns/mediator](http://sourcemaking.com/design_patterns/mediator)

# Memento pattern

---

The **memento pattern** is a software design pattern that provides the ability to restore an object to its previous state (undo via rollback).

The memento pattern is used by two objects: the *originator* and a *caretaker*. The originator is some object that has an internal state. The caretaker is going to do something to the originator, but wants to be able to undo the change. The caretaker first asks the originator for a memento object. Then it does whatever operation (or sequence of operations) it was going to do. To roll back to the state before the operations, it returns the memento object to the originator. The memento object itself is an opaque object (one which the caretaker cannot, or should not, change). When using this pattern, care should be taken if the originator may change other objects or resources - the memento pattern operates on a single object.

Classic examples of the memento pattern include the seed of a pseudorandom number generator (it will always produce the same sequence thereafter when initialized with the seed state) and the state in a finite state machine.

## Example

The following Java program illustrates the "undo" usage of the Memento Pattern.

```
import java.util.List;
import java.util.ArrayList;
class Originator {
    private String state;
    // The class could also contain additional data that is not part of
the
    // state saved in the memento.

    public void set(String state) {
        System.out.println("Originator: Setting state to " + state);
        this.state = state;
    }

    public Memento saveToMemento() {
        System.out.println("Originator: Saving to Memento.");
        return new Memento(state);
    }

    public void restoreFromMemento(Memento memento) {
        state = memento.getSavedState();
        System.out.println("Originator: State after restoring from
Memento: " + state);
    }

    public static class Memento {
```



```
    private final String state;

    private Memento(String stateToSave) {
        state = stateToSave;
    }

    private String getSavedState() {
        return state;
    }
}

class Caretaker {
    public static void main(String[] args) {
        List<Originator.Memento> savedStates = new ArrayList<Originator.Memento>();

        Originator originator = new Originator();
        originator.set("State1");
        originator.set("State2");
        savedStates.add(originator.saveToMemento());
        originator.set("State3");
        // We can request multiple mementos, and choose which one to
roll back to.
        savedStates.add(originator.saveToMemento());
        originator.set("State4");

        originator.restoreFromMemento(savedStates.get(1));
    }
}
```

The output is:

```
Originator: Setting state to State1
Originator: Setting state to State2
Originator: Saving to Memento.
Originator: Setting state to State3
Originator: Saving to Memento.
Originator: Setting state to State4
Originator: State after restoring from Memento: State3
```

## External links

- Description of Memento Pattern <sup>[1]</sup> in Ada
- Memento UML Class Diagram <sup>[2]</sup> with C# and .NET code samples
- SourceMaking Tutorial <sup>[3]</sup>

## References

[1] <http://adapower.com/index.php?Command=Class&ClassID=Patterns&CID=271>

[2] <http://dofactory.com/Patterns/PatternMemento.aspx>

[3] [http://sourcemaking.com/design\\_patterns/memento](http://sourcemaking.com/design_patterns/memento)

# Null Object pattern

---

In object-oriented computer programming, a **Null Object** is an object with defined neutral ("null") behavior. The Null Object design pattern describes the uses of such objects and their behavior (or lack thereof). It was first published in the *Pattern Languages of Program Design* book series<sup>[1]</sup>.

## Motivation

In most object-oriented languages, such as Java or C#, references may be null. These references need to be checked to ensure they are not null before invoking any methods, because one can't invoke anything on a null reference.

## Description

Instead of using a null reference to convey absence of an object (for instance, a non-existent customer), one uses an object which implements the expected interface, but whose method body is empty. The advantage of this approach over a working default implementation is that a Null Object is very predictable and has no side effects: it does *nothing*.

For example, a function may retrieve a list of files in a directory and perform some action on each. In the case of an empty directory, one response may be to throw an exception or return a null reference rather than a list. Thus, the code which expects a list must verify that it in fact has one before continuing, which can complicate the design.

By returning a null object (i.e. an empty list) instead, there is no need to verify that the return value is in fact a list. The calling function may simply iterate the list as normal, effectively doing nothing. It is, however, still possible to check whether the return value is a null object (e.g. an empty list) and react differently if desired.

The null object pattern can also be used to act as a stub for testing if a certain feature, such as a database, is not available for testing.

## Relation to other patterns

It can be regarded as a special case of the State pattern and the Strategy pattern.

It is not a pattern from Design Patterns, but is mentioned in Martin Fowler's *Refactoring*<sup>[2]</sup> and Joshua Kerievsky's<sup>[3]</sup> book on refactoring in the *Insert Null Object* refactoring.

Chapter 17 is dedicated to the pattern in Robert Cecil Martin's *Agile Software Development: Principles, Patterns and Practices*<sup>[4]</sup>

---

## In various languages

### C

In C, functions can be written such that they accept a null pointer without failing. For instance the standard function `free` may be called with a null argument. This allows code like

```
free(p);  
p = NULL;
```

to be safely executed two or more times.

### C++

A language with statically typed references to objects illustrates how the null object becomes a more complicated pattern:

```
class animal {  
public:  
    virtual void make_sound() = 0;  
};  
  
class dog : public animal {  
    void make_sound() { cout << "woof!" << endl; }  
};  
  
class null_animal : public animal {  
    void make_sound() { }  
};
```

Here, the idea is that there are situations where a pointer or reference to an animal object is required, but there is no appropriate object available. A null reference is impossible in standard-conforming C++. A null `animal *` pointer is possible, and could be useful as a place-holder, but may not be used for direct dispatch: `a->make_sound()` is undefined behavior if `a` is a null pointer.

The null object pattern solves this problem by providing a code special `null_animal` class which can be instantiated bound to an animal pointer or reference.

The special null class must be created for each class hierarchy that is to have a null object, since a `null_animal` is of no use when what is needed is a null object with regard to some widget base class that is not related to the animal hierarchy.

### C#

In C#, arrays are first class objects with methods and properties that are available as long as you have an array instance, no matter how many elements the array has, whether it be 0, 1, or 100 items. Zero-length arrays are an example of the null object pattern.

```
public class NullObjectExample  
{  
    public static void Main()  
    {  
        string[] list;
```

```

    list = new string[1];
    list[0] = "This contains a keyword";
    FindSubstring(list);

    // These statements are legal and do not cause exceptions. C#
    // allows array objects to be created that have zero elements.
    // 'list' contains a real object, so FindSubString can call
Length
    // on it all the same. It's just that length is zero, so the
loop
    // never executes in FindSubString.
    list = new string[0];
    FindSubstring(list);
}

// Searches the given string array for a keyword,
// and upon finding it, prints the entire string.
public static void FindSubstring(string[] documentLines)
{
    for (int i = 0; i < documentLines.Length; i++)
    {
        string line = documentLines[i];

        if (line.Contains("keyword"))
        {
            System.Console.WriteLine(line);
        }
    }
}
}

```

## Smalltalk

Following the Smalltalk principle, *everything is an object*, the absence of an object is itself modeled by an object, called nil. In the GNU Smalltalk for example, the class of nil is UndefinedObject, a direct descendant of Object.

Any operation that fails to return a sensible object for its purpose may return nil instead, thus avoiding the special case of returning "no object". This method has the advantage of simplicity (no need for a special case) over the classical "null" or "no object" or "null reference" approach. Especially useful messages to be used with nil are isNil or ifNil:, which make it practical and safe to deal with possible references to nil in Smalltalk programs.

## Common Lisp

In Lisp, functions can gracefully accept the special object `nil`, which reduces the amount of special case testing in application code. For instance although `nil` is an atom and does not have any fields, the functions `car` and `cdr` accept `nil` and just return it, which is very useful and results in shorter code.

Since `nil` is the empty list in Lisp, the situation described in the introduction above doesn't exist. Code which returns `nil` is returning what is in fact the empty list (and not anything resembling a null reference to a list type), so the caller does not need to test the value to see whether or not it has a list.

The null object pattern is also supported in multiple value processing. If the program attempts to extract a value from an expression which returns no values, the behavior is that the null object `nil` is substituted. Thus `(list (values))` returns `(nil)` (a one-element list containing `nil`). The `(values)` expression returns no values at all, but since the function call to `list` needs to reduce its argument expression to a value, the null object is automatically substituted.

## CLOS

In Common Lisp, the object `nil` is the one and only instance of the special class `null`. What this means is that a method can be specialized to the null class, thereby implementing the null design pattern. Which is to say, it is essentially built into the object system:

```
;; empty dog class

(defclass dog () ())

;; a dog object makes a sound by barking: woof! is printed on standard
output
;; when (make-sound x) is called, if x is an instance of the dog class.

(defmethod make-sound ((obj dog))
  (format t "woof!~%"))

;; allow (make-sound nil) to work via specialization to null class.
;; innocuous empty body: nil makes no sound.
(defmethod make-sound ((obj null)))
```

The class `null` is a subclass of the symbol class, because `nil` is a symbol. Since `nil` also represents the empty list, `null` is a subclass of the list class, too. Methods parameters specialized to symbol or list will thus take a `nil` argument. Of course, a null specialization can still be defined which is a more specific match for `nil`.

## Scheme

Unlike Common Lisp, and many dialects of Lisp, the Scheme dialect does not have a `nil` value which works this way; the functions `car` and `cdr` may not be applied to an empty list; Scheme application code therefore has to use the `empty?` or `pair?` predicate functions to sidestep this situation, even in situations where very similar Lisp would not need to distinguish the empty and non-empty cases thanks to the behavior of `nil`.

## Criticism

This pattern should be used carefully as it can make errors/bugs appear as normal program execution.<sup>[5]</sup>

## External links

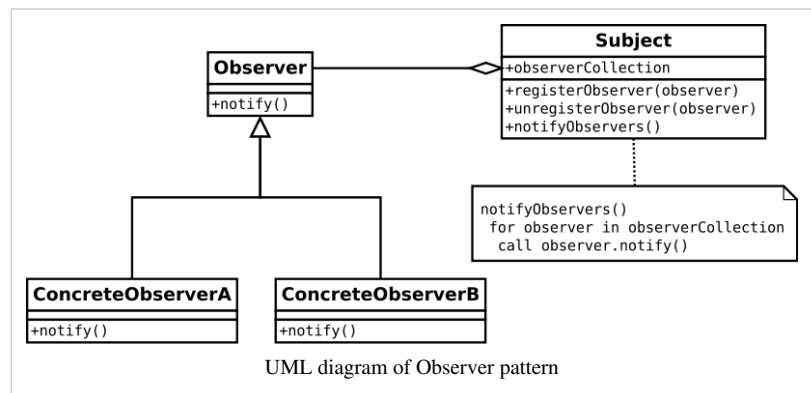
- Jeffrey Walkers' account of the Null Object Pattern <sup>[6]</sup>
- Antonio Garcias' account of the Null Object Pattern <sup>[7]</sup>
- Martin Fowlers' description of Special Case, a slightly more general pattern <sup>[8]</sup>
- Null Object Pattern Revisited <sup>[9]</sup>
- Introduce Null Object refactoring <sup>[10]</sup>
- SourceMaking Tutorial <sup>[11]</sup>

## References

- [1] Woolf, Bobby (1998). "Null Object". In Martin, Robert; Riehle, Dirk; Buschmann, Frank. *Pattern Languages of Program Design 3*. Addison-Wesley
- [2] Fowler, Martin (1999). *Refactoring. Improving the Design of Existing Code*. Addison-Wesley. ISBN 0-201-48567-2.
- [3] Kerievsky, Joshua (2004). *Refactoring To Patterns*. Addison-Wesley. ISBN 0-321-21335-1.
- [4] Martin, Robert (2002). *Agile Software Development: Principles, Patterns and Practices*. Pearson Education. ISBN 0-13-597444-5.
- [5] Fowler, Martin (1999). Refactoring pp. 261
- [6] <http://www.cs.oberlin.edu/~jwalker/nullObjPattern/>
- [7] <http://exciton.cs.rice.edu/javaresources/DesignPatterns/NullPattern.htm>
- [8] <http://martinfowler.com/eaCatalog/specialCase.html>
- [9] [http://www.owl.net.rice.edu/~comp212/00-spring/handouts/week06/null\\_object\\_revisited.htm](http://www.owl.net.rice.edu/~comp212/00-spring/handouts/week06/null_object_revisited.htm)
- [10] <http://www.refactoring.com/catalog/introduceNullObject.html>
- [11] [http://sourcemaking.com/design\\_patterns/null\\_object](http://sourcemaking.com/design_patterns/null_object)

# Observer pattern

The **observer pattern** (a subset of the publish/subscribe pattern) is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems.



## Example

Below is an example that takes keyboard input and treats each input line as an event. The example is built upon the library classes `java.util.Observer` <sup>[1]</sup> and `java.util.Observable` <sup>[2]</sup>. When a string is supplied from `System.in`, the method `notifyObservers` is then called, in order to notify all observers of the event's occurrence, in the form of an invocation of their 'update' methods - in our example, `ResponseHandler.update(...)`.

The file `MyApp.java` contains a `main()` method that might be used in order to run the code.

```

/* File Name : EventSource.java */

package obs;
  
```

```

import java.util.Observable;           //Observable is here
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class EventSource extends Observable implements Runnable {
    public void run() {
        try {
            final InputStreamReader isr = new InputStreamReader(
System.in );
            final BufferedReader br = new BufferedReader( isr );
            while( true ) {
                String response = br.readLine();
                setChanged();
                notifyObservers( response );
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

*/\* File Name: ResponseHandler.java \*/*

```

package obs;

import java.util.Observable;
import java.util.Observer; /* this is Event Handler */

public class ResponseHandler implements Observer {
    private String resp;
    public void update (Observable obj, Object arg) {
        if (arg instanceof String) {
            resp = (String) arg;
            System.out.println("\nReceived Response: "+ resp );
        }
    }
}

```

*/\* Filename : MyApp.java \*/*

*/\* This is the main program \*/*

```

package obs;

public class MyApp {
    public static void main(String args[]) {
        System.out.println("Enter Text >");
    }
}

```

```
// create an event source - reads from stdin
final EventSource evSrc = new EventSource();

// create an observer
final ResponseHandler respHandler = new ResponseHandler();

// subscribe the observer to the event source
evSrc.addObserver( respHandler );

// starts the event thread
Thread thread = new Thread(evSrc);
thread.start();
    }
}
```

## Implementations

The observer pattern is implemented in numerous programming libraries and systems, including almost all GUI toolkits.

Some of the most notable implementations of this pattern:

### ActionScript

- `flash.events`<sup>[3]</sup>, a package in ActionScript 3.0 (following from the `mx.events` package in ActionScript 2.0).

### C

- `GObject`, in `GLib` - an implementation of objects and signals/callbacks in C. (This library has many bindings to other programming languages.)

### C++

- `libsigc++`<sup>[4]</sup> - the C++ signalling template library.
- `sigslot`<sup>[5]</sup> - C++ Signal/Slot Library
- `Cpp::Events`<sup>[6]</sup> - Template-based C++ implementation that introduces separation of connection management interface of the event object from the invocation interface.
- `XLObject`<sup>[7]</sup> - Template-based C++ signal/slot model patterned after Qt.
- `Signals`<sup>[8]</sup> - A lightweight and non-intrusive C++ signal/slot model implementation.
- `libevent`<sup>[9]</sup> - Multi-threaded Crossplatform Signal/Slot C++ Library
- `Boost.Signals`<sup>[10]</sup>, an extension of the C++ STL providing a signal/slot model
- MFC's `CDocument-CView`-framework
- The Qt C++ framework's signal/slot model
- The MRPT robotics C++ framework's observer/observable<sup>[11]</sup> model.



## C#

- The `IObserver<T>` Interface <sup>[12]</sup> - The .NET Framework supported way of implementing the observer pattern.
- Exploring the Observer Design Pattern <sup>[13]</sup> - the C# and Visual Basic .NET implementation, using delegates and the Event pattern

## ColdFusion

- <http://www.cfdesignpatterns.com/behavioral-patterns/observer-design-pattern-in-coldfusion/>

## Delphi

- Delphi Observer Pattern <sup>[14]</sup>, a Delphi implementation

## Java

- The Java Swing library makes extensive use of the observer pattern for event management
- The Java Servlet API is heavily using observer pattern to notify application entities about significant events, observed by web container. Listeners are provided for servlet and session lifecycle, session migration, changes to scoped attributes etc.
- PerfectJPattern Open Source Project <sup>[15]</sup>, Provides a context-free and type-safe implementation of the Observer Pattern in Java.

## JavaScript

- EventDispatcher singleton <sup>[16]</sup>, a JavaScript core API based Signals and slots implementation - an observer concept different from Publish/subscribe - pretty lightweighted but still type-safety enforcing.

## Lisp

- Cells <sup>[17]</sup>, a dataflow extension to Common Lisp that uses meta-programming to hide some of the details of Observer pattern implementation.

## PHP

- Event\_Dispatcher <sup>[18]</sup>, a PHP implementation
- SPL <sup>[19]</sup>, the Standard PHP Library
- Symfony Event Dispatcher <sup>[20]</sup>, a standalone library by the Symfony team

## Python

- Louie <sup>[21]</sup>, an implementation by Patrick K. O'Brien.
  - PyDispatcher <sup>[22]</sup>, the implementation on which the Django <sup>[23]</sup> web framework's signals are based.
  - Py-notify <sup>[24]</sup>, a Python implementation
  - Observer Pattern using Weak References <sup>[25]</sup> implementation by Michael Kent
  - PyPubSub <sup>[26]</sup> an in-application Pub/Sub library for Observer behavior
  - NotificationFramework <sup>[27]</sup> classes directly implementing Observer patterns
-

## Ruby

- Observer<sup>[28]</sup>, from the Ruby Standard Library.

## Other/Misc

- CSP<sup>[29]</sup> - *Observer Pattern* using *CSP-like Rendezvous* (each actor is a process, communication is via rendezvous).
- YUI Event utility<sup>[30]</sup> implements custom events through the observer pattern
- Publish/Subscribe with LabVIEW<sup>[31]</sup>, Implementation example of Observer or Publish/Subscribe using G.

## References

- <http://www.research.ibm.com/designpatterns/example.htm>
- <http://msdn.microsoft.com/en-us/library/ms954621.aspx>
- "Speaking on the Observer pattern"<sup>[32]</sup> - JavaWorld

## External links

- Observer Pattern implementation in JDK 6<sup>[2]</sup>
- Observer Pattern in Java<sup>[32]</sup>
- Definition, C# example & UML diagram<sup>[33]</sup>
- Jt<sup>[34]</sup> J2EE Pattern Oriented Framework
- Subject Observer example in C++<sup>[35]</sup>
- Observer Pattern recipe in Python<sup>[36]</sup>
- SourceMaking Tutorial<sup>[37]</sup>
- Observer Pattern in Objective-C<sup>[38]</sup>

## References

- [1] <http://java.sun.com/javase/6/docs/api/java/util/Observer.html>
- [2] <http://java.sun.com/javase/6/docs/api/java/util/Observable.html>
- [3] <http://livedocs.macromedia.com/flex/2/langref/flash/events/package-detail.html>
- [4] <http://libsigc.sourceforge.net>
- [5] <http://sigslot.sourceforge.net/>
- [6] <http://code.google.com/p/cpp-events>
- [7] <http://xlobject.sourceforge.net/>
- [8] <http://github.com/pbhogan/Signals>
- [9] <http://www.monkey.org/~provos/libevent/>
- [10] <http://www.boost.org/doc/html/signals.html>
- [11] [http://reference.mrpt.org/svn/classmrpt\\_1\\_1utils\\_1\\_1\\_c\\_observable.html](http://reference.mrpt.org/svn/classmrpt_1_1utils_1_1_c_observable.html)
- [12] <http://msdn.microsoft.com/en-us/library/dd783449.aspx>
- [13] <http://msdn.microsoft.com/en-us/library/ee817669.aspx>
- [14] <http://blogs.teamb.com/joannacarter/2004/06/30/690>
- [15] <http://perfectjpattern.sourceforge.net/dp-observer.html>
- [16] [http://www.refactory.org/s/eventdispatcher\\_singleton\\_a\\_core\\_api\\_based\\_signals\\_and\\_slots\\_implementation/view/latest](http://www.refactory.org/s/eventdispatcher_singleton_a_core_api_based_signals_and_slots_implementation/view/latest)
- [17] <http://common-lisp.net/project/cells/>
- [18] [http://pear.php.net/package/Event\\_Dispatcher](http://pear.php.net/package/Event_Dispatcher)
- [19] <http://www.php.net/~helly/php/ext/spl/main.html>
- [20] <http://components.symfony-project.org/event-dispatcher/>
- [21] <http://github.com/11craft/louie>
- [22] <http://pydispatcher.sourceforge.net/>
- [23] <http://www.djangoproject.com/>
- [24] <http://home.gna.org/py-notify/>
- [25] <http://radio-weblogs.com/0124960/2004/06/15.html>

- [26] <http://sourceforge.net/projects/pubsub/>
- [27] <http://pypi.python.org/pypi/NotificationFramework/>
- [28] <http://ruby-doc.org/stdlib/libdoc/observer/rdoc/index.html>
- [29] [http://ptolemy.eecs.berkeley.edu/presentations/06/FutureOfEmbeddedSoftware\\_Lee\\_Graz.ppt](http://ptolemy.eecs.berkeley.edu/presentations/06/FutureOfEmbeddedSoftware_Lee_Graz.ppt)
- [30] <http://developer.yahoo.com/yui/event/>
- [31] <http://www.labviewportal.eu/viewtopic.php?f=19&t=9>
- [32] <http://www.javaworld.com/javaworld/javaqa/2001-05/04-qa-0525-observer.html>
- [33] <http://www.dofactory.com/Patterns/PatternObserver.aspx>
- [34] <http://www.fsw.com/Jt/Jt.htm>
- [35] <http://rtmatheson.com/2010/03/working-on-the-subject-observer-pattern/>
- [36] <http://code.activestate.com/recipes/131499-observer-pattern/>
- [37] [http://sourcemaking.com/design\\_patterns/observer](http://sourcemaking.com/design_patterns/observer)
- [38] <http://www.a-coding.com/2010/10/observer-pattern-in-objective-c.html>

## Design pattern Servant

---

**Servant** is a design pattern used to offer some functionality to a group of classes without defining that functionality in each of them. A Servant is a class whose instance (or even just class) provides a methods that takes care of a desired service, while objects for which (or with whom) the servant does something, takes as parameters.

### Description and simple example

Servant is used for providing some behavior to group of classes. Instead of defining that behavior in each class (also when we cannot factor out this behavior in the common parent class), it is defined once in servant.

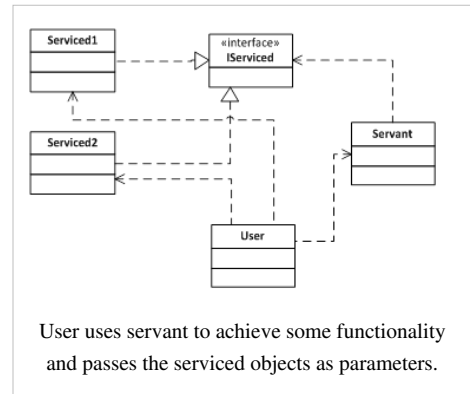
For example we have a few classes representing geometric objects (rectangle, ellipse, and triangle). We can draw these objects on some canvas. When we need to provide a “move” method for these objects we could implement this method in each class, or we can define an interface they implement and then offer the “move” functionality in a servant. An interface is defined to ensure that serviced classes have methods, that servant needs to provide desired behavior. If we continue in our example, we define an Interface “IMovable” specifying that, every class implementing this interface needs to implement method “getPosition” and “setPosition”. First method gets the position of object on canvas and second one sets position of object and draws it on canvas. Then we define a servant class “MoveServant”, which has two methods “moveTo(IMovable movedObject, Position where)” and “moveby(IMovable movedObject, int dx, int dy)”. Servant class can now move every object, which implements the IMovable and we have that “moving” code in only one class, applying “Separation of concerns” rule.

### Two ways of implementation

There are two ways to implement this design pattern.

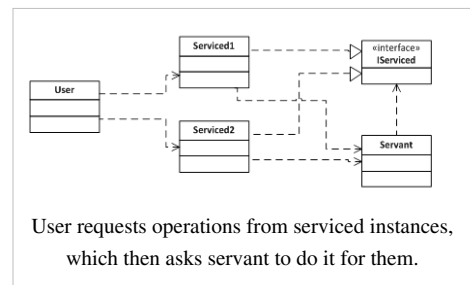
1. User knows the servant (in which case he doesn't need to know the serviced classes) and sends messages with his requests to the servant instances, passing the serviced objects as parameters.
2. Serviced instances know the servant and the user sends them messages with his requests (in which case she doesn't have to know the servant). The serviced instances then send messages to the instances of servant, asking for service.
1. The serviced classes (geometric objects from our example) don't know about servant, but they implement the “IServiced” interface. The user class just calls the method of servant and passes serviced objects as parameters. This situation is shown on figure 1.

1. On figure 2 is shown opposite situation, where user don't know about servant class and calls directly serviced classes. Serviced classes then asks servant themselves to achieve desired functionality.



## How to implement Servant

1. Analyze what behavior servant should take care of. State what methods servant will define and what these methods will need from serviced parameter. By other words, what serviced instance must provide, so that servants methods can achieve their goals.
1. Analyze what abilities serviced classes must have, so they can be properly serviced.
1. We define an interface, which will enforce implementation of declared methods.
1. Define an interface specifying requested behavior of serviced objects. If some instance wants to be served by servant, it must implement this interface.
1. Define (or acquire somehow) specified servant (his class).
1. Implement defined interface with serviced classes.



## Example

This simple example shows the situation described above. This example is only illustrative and will not offer any actual drawing of geometric objects, nor specifying how they look like.

```

// Servant class, offering its functionality to classes implementing
// IMovable Interface
public class MoveServant{
    // Method, which will move IMovable implementing class to position
    where
    public void moveTo(IMovable serviced, Position where){
        // Do some other stuff to ensure it moves smoothly and nicely,
        this is the place to offer the functionality
        serviced.setPosition(where);
    }
    // Method, which will move IMovable implementing class by dx and dy
    public void moveBy(IMovable serviced, int dx, int dy){
        // this is the place to offer the functionality
        dx += serviced.getPosition().xPosition;
        dy += serviced.getPosition().yPosition;
        serviced.setPosition(new Position(dx, dy));
    }
  
```

```
    }  
}  
  
// Interface specifying what serviced classes needs to implement, to be  
// serviced by servant.  
public interface IMovable{  
    public void setPosition(Position p);  
    public Position getPosition();  
}  
  
// One of geometric classes  
public class Triangle implements IMovable{  
    // Position of the geometric object on some canvas  
    private Position p;  
    // Method, which returns position of geometric object  
    public void setPosition(Position p){  
        this.p = p;  
    }  
    // Method, which sets position of geometric object  
    public Position getPosition(){  
        return this.p;  
    }  
}  
  
// One of geometric classes  
public class Ellipse implements IMovable{  
    // Position of the geometric object on some canvas  
    private Position p;  
    // Method, which returns position of geometric object  
    public void setPosition(Position p){  
        this.p = p;  
    }  
    // Method, which sets position of geometric object  
    public Position getPosition(){  
        return this.p;  
    }  
}  
  
// One of geometric classes  
public class Rectangle implements IMovable{  
    // Position of the geometric object on some canvas  
    private Position p;  
    // Method, which returns position of geometric object  
    public void setPosition(Position p){  
        this.p = p;  
    }  
    // Method, which sets position of geometric object
```

```
    public Position getPosition() {
        return this.p;
    }
}

// Just a very simple container class for position.
public class Position{
    public int xPosition;
    public int yPosition;
    public Position(int dx, int dy){
        xPosition = dx;
        yPosition = dy;
    }
}
```

## Similar design pattern: Command

Design patterns Command and Servant are indeed very similar and implementation is often virtually the same. Difference between them is the approach to the problem, which programmer chose.

- In case of pattern Servant we have some objects, to which we want offer, some functionality. So we create class, whose instances offer that requested functionality and which defines an interface, which serviced objects must implement. Serviced instances are then passed as parameters to the servant.
- In case of pattern Command we have some objects that we want to modify with some functionality (we want to add something to them). So we define an interface, which classes with desired functionality must implement. Instances of those classes are then passed to original objects as parameters of their methods.

Even though design patterns Command and Servant are similar it doesn't mean it's always like that. There are a number of situations where use of design pattern Command doesn't relate to with design pattern Servant. In these situations we usually need to pass to called methods just reference to another method, which she will need in accomplishing her goal. Because we can't pass reference to method in many languages (Java), we have to pass object implementing interface which declares signature of passed method.

## Resources

Pecinovský, Rudolf; Jarmila Pavlíčková, Luboš Pavlíček (6 2006). "Let's Modify the Objects First Approach into Design Patterns First" <sup>[1]</sup>. Eleventh Annual Conference on Innovation and Technology in Computer Science Education, University of Bologna <sup>[2]</sup>.

## References

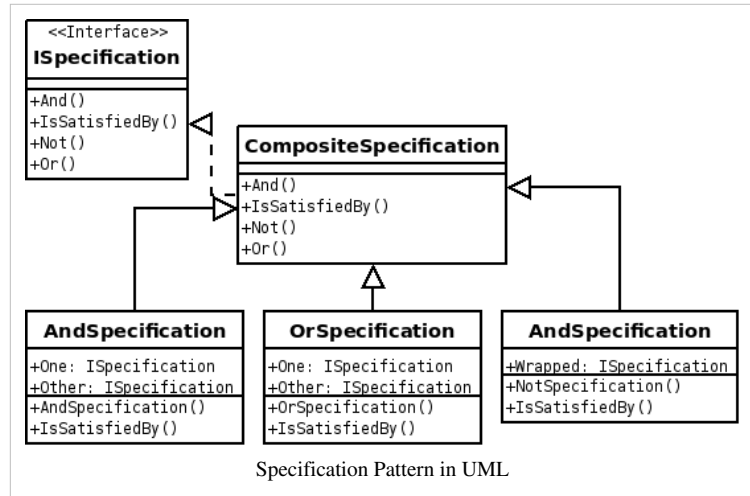
- [1] [http://edu.pecinovsky.cz/papers/2006\\_ITiCSE\\_Design\\_Patterns\\_First.pdf](http://edu.pecinovsky.cz/papers/2006_ITiCSE_Design_Patterns_First.pdf)  
[2] <http://www.iticse06.cs.unibo.it/>

# Specification pattern

In computer programming, the **specification pattern** is a particular software design pattern, whereby business logic can be recombined by chaining the business logic together using boolean logic.

A specification pattern outlines a unit of business logic that is combinable with other business logic units. In this pattern, a unit of business logic inherits its functionality from the abstract aggregate Composite Specification class. The Composite Specification class has one function called `IsSatisfiedBy` that returns a boolean value.

After instantiation, the specification is "chained" with other specifications, making new specifications easily maintainable, yet highly customizable business logic. Furthermore upon instantiation the business logic may, through method invocation or inversion of control, have its state altered in order to become a delegate of other classes such as a persistence repository.



## Code examples

### C#

```

public interface ISpecification
{
    bool IsSatisfiedBy(object candidate);

    ISpecification And(ISpecification other);

    ISpecification Or(ISpecification other);

    ISpecification Not();
}

public abstract class CompositeSpecification : ISpecification
{
    public abstract bool IsSatisfiedBy(object candidate);

    public ISpecification And(ISpecification other)
    {
        return new AndSpecification(this, other);
    }

    public ISpecification Or(ISpecification other)
    {

```

```
        return new OrSpecification(this, other);
    }

    public ISpecification Not()
    {
        return new NotSpecification(this);
    }
}

public class AndSpecification : CompositeSpecification
{
    private ISpecification One;
    private ISpecification Other;

    public AndSpecification(ISpecification x, ISpecification y)
    {
        One = x;
        Other = y;
    }

    public override bool IsSatisfiedBy(object candidate)
    {
        return One.IsSatisfiedBy(candidate) &&
Other.IsSatisfiedBy(candidate);
    }
}

public class OrSpecification : CompositeSpecification
{
    private ISpecification One;
    private ISpecification Other;

    public OrSpecification(ISpecification x, ISpecification y)
    {
        One = x;
        Other = y;
    }

    public override bool IsSatisfiedBy(object candidate)
    {
        return One.IsSatisfiedBy(candidate) ||
Other.IsSatisfiedBy(candidate);
    }
}

public class NotSpecification : CompositeSpecification
{

```



```
private ISpecification Wrapped;

public NotSpecification(ISpecification x)
{
    Wrapped = x;
}

public override bool IsSatisfiedBy(object candidate)
{
    return !Wrapped.IsSatisfiedBy(candidate);
}
}
```

### C# 3.0, simplified with generics and extension methods

```
public interface ISpecification<TEntity>
{
    bool IsSatisfiedBy(TEntity entity);
}

internal class AndSpecification<TEntity> : ISpecification<TEntity>
{
    private ISpecification<TEntity> Spec1;
    private ISpecification<TEntity> Spec2;

    internal AndSpecification(ISpecification<TEntity> s1, ISpecification<TEntity> s2)
    {
        Spec1 = s1;
        Spec2 = s2;
    }

    public bool IsSatisfiedBy(TEntity candidate)
    {
        return Spec1.IsSatisfiedBy(candidate) &&
Spec2.IsSatisfiedBy(candidate);
    }
}

internal class OrSpecification<TEntity> : ISpecification<TEntity>
{
    private ISpecification<TEntity> Spec1;
    private ISpecification<TEntity> Spec2;

    internal OrSpecification(ISpecification<TEntity> s1, ISpecification<TEntity> s2)
    {
        Spec1 = s1;
        Spec2 = s2;
    }
}
```

```
    public bool IsSatisfiedBy(TEntity candidate)
    {
        return Spec1.IsSatisfiedBy(candidate) ||
Spec2.IsSatisfiedBy(candidate);
    }
}

internal class NotSpecification<TEntity> : ISpecification<TEntity>
{
    private ISpecification<TEntity> Wrapped;

    internal NotSpecification(ISpecification<TEntity> x)
    {
        Wrapped = x;
    }

    public bool IsSatisfiedBy(TEntity candidate)
    {
        return !Wrapped.IsSatisfiedBy(candidate);
    }
}

public static class ExtensionMethods
{
    public static ISpecification<TEntity> And<TEntity>(this ISpecification<TEntity> s1,
ISpecification<TEntity> s2)
    {
        return new AndSpecification<TEntity>(s1, s2);
    }

    public static ISpecification<TEntity> Or<TEntity>(this ISpecification<TEntity> s1,
ISpecification<TEntity> s2)
    {
        return new OrSpecification<TEntity>(s1, s2);
    }

    public static ISpecification<TEntity> Not<TEntity>(this ISpecification<TEntity> s)
    {
        return new NotSpecification<TEntity>(s);
    }
}
```

## Example of use

In this example, we are retrieving invoices and sending them to a collection agency if they are overdue, notices have been sent and they are not already with the collection agency.

We previously defined an `OverdueSpecification` class that it is satisfied when an invoice's due date is 30 days or older, a `NoticeSentSpecification` class that is satisfied when three notices have been sent to the customer, and an `InCollectionSpecification` class that is satisfied when an invoice has already been sent to the collection agency.

Using these three specifications, we created a new specification called `SendToCollection` which will be satisfied when an invoice is overdue, when notices have been sent to the customer, and are not already with the collection agency.

```
OverDueSpecification OverDue = new OverDueSpecification();
NoticeSentSpecification NoticeSent = new NoticeSentSpecification();
InCollectionSpecification InCollection = new
InCollectionSpecification();

ISpecification SendToCollection =
OverDue.And(NoticeSent).And(InCollection.Not());

InvoiceCollection = Service.GetInvoices();

foreach (Invoice currentInvoice in InvoiceCollection)
{
    if (SendToCollection.IsSatisfiedBy(currentInvoice))
    {
        currentInvoice.SendToCollection();
    }
}
```

## References

- Evans, E: "Domain-Driven Design.", page 224. Addison-Wesley, 2004.

## External links

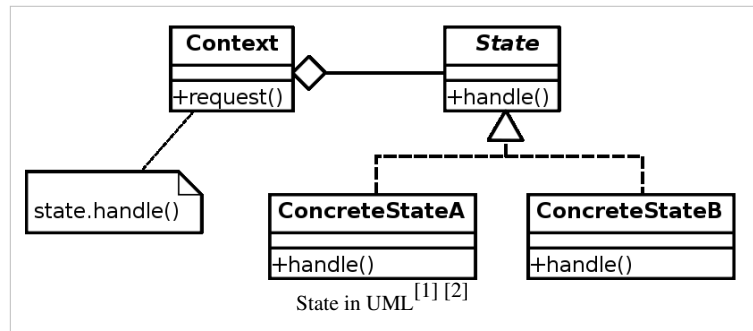
- Specifications <sup>[1]</sup> by Eric Evans and Martin Fowler
- The Specification Pattern: A Primer <sup>[2]</sup> by Matt Berther
- The Specification Pattern: A Four Part Introduction using VB.Net <sup>[3]</sup> by Richard Dalton
- specification pattern in flash actionscript 3 <sup>[4]</sup> by Rolf Vreijdenberger

## References

- [1] <http://www.martinfowler.com/apSUPP/spec.pdf>
- [2] <http://www.mattberther.com/2005/03/25/the-specification-pattern-a-primer/>
- [3] <http://www.codeproject.com/KB/architecture/SpecificationPart1.aspx>
- [4] <http://www.dpdn.nl/opensource/specification-pattern-for-selection-on-lists>

# State pattern

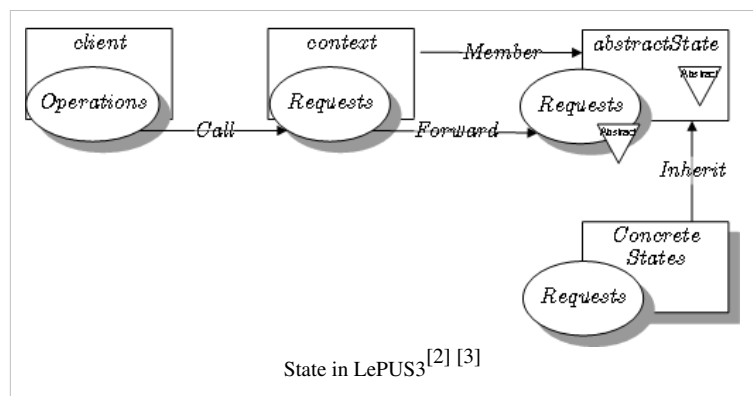
The **state pattern** is a behavioral software design pattern, also known as the **objects for states pattern**. This pattern is used in computer programming to represent the state of an object. This is a clean way for an object to partially change its type at runtime<sup>[1]:395</sup>.



## Pseudocode example

Take, for example, a drawing program. The program has a mouse cursor, which at any point in time can act as one of several tools. Instead of switching between multiple cursor objects, the cursor maintains an internal state representing the tool currently in use. When a tool-dependent method is called (say, as a result of a mouse click), the method call is passed on to the cursor's state.

Each tool corresponds to a state. The shared abstract state class is AbstractTool:



```

class AbstractTool is
    function moveTo(point) is
        input:  the location point the mouse moved to
        (this function must be implemented by subclasses)

    function mouseDown(point) is
        input:  the location point the mouse is at
        (this function must be implemented by subclasses)

    function mouseUp(point) is
        input:  the location point the mouse is at
        (this function must be implemented by subclasses)
  
```

According to this definition, each tool must handle movement of the mouse cursor and also the start and end of any click or drag.

Using that base class, simple pen and selection tools could look like this:

```

subclass PenTool of AbstractTool is
    last_mouse_position := invalid
    mouse_button := up

    function moveTo(point) is
        input:  the location point the mouse moved to
        if mouse_button = down
  
```

```

        (draw a line from the last_mouse_position to point)
        last_mouse_position := point

function mouseDown(point) is
    input:  the location point the mouse is at
    mouse_button := down
    last_mouse_position := point

function mouseUp(point) is
    input:  the location point the mouse is at
    mouse_button := up

subclass SelectionTool of AbstractTool is
    selection_start := invalid
    mouse_button := up

function moveTo(point) is
    input:  the location point the mouse moved to
    if mouse_button = down
        (select the rectangle between selection_start and point)

function mouseDown(point) is
    input:  the location point the mouse is at
    mouse_button := down
    selection_start := point

function mouseUp(point) is
    input:  the location point the mouse is at
    mouse_button := up

```

For this example, the class for the context is called `Cursor`. The methods named in the abstract state class (`AbstractTool` in this case) are also implemented in the context. In the context class, these methods invoke the corresponding method of the current state, represented by `current_tool`.

```

class Cursor is
    current_tool := new PenTool

function moveTo(point) is
    input:  the location point the mouse moved to
    current_tool.moveTo(point)

function mouseDown(point) is
    input:  the location point the mouse is at
    current_tool.mouseDown(point)

function mouseUp(point) is
    input:  the location point the mouse is at
    current_tool.mouseUp(point)

```

```

function usePenTool() is
    current_tool := new PenTool

function useSelectionTool() is
    current_tool := new SelectionTool

```

Notice how one Cursor object can act both as a PenTool and a SelectionTool at different points, by passing the appropriate method calls on to whichever tool is active. That is the essence of the **state pattern**. In this case, we could have combined state and object by creating PenCursor and SelectCursor classes, thus reducing the solution to simple inheritance, but in practice, Cursor may carry data which is expensive or inelegant to copy to a new object whenever a new tool is selected.

## Example

The state interface and two implementations. The state method has a reference to the context object and is able to change its state.

```

interface State {
    public void writeName(StateContext stateContext, String name);
}

class StateA implements State {
    public void writeName(StateContext stateContext, String name) {
        System.out.println(name.toLowerCase());
        stateContext.setState(new StateB());
    }
}

class StateB implements State {
    private int count=0;
    public void writeName(StateContext stateContext, String name){
        System.out.println(name.toUpperCase());
        // change state after StateB's writeName() gets invoked
        if(++count>1) {
            stateContext.setState(new StateA());
        }
    }
}

```

The context class has a state variable which it instantiates in an initial state, in this case StateA. In its method, it uses the corresponding methods of the state object.

```

public class StateContext {
    private State myState;
    public StateContext() {
        setState(new StateA());
    }
}

```

```
// normally only called by classes implementing the State
interface
{
    public void setState(State newState) {
        this.myState = newState;
    }

    public void writeName(String name) {
        this.myState.writeName(this, name);
    }
}
```

And the usage:

```
public class TestClientState {
    public static void main(String[] args) {
        StateContext sc = new StateContext();
        sc.writeName("Monday");
        sc.writeName("Tuesday");
        sc.writeName("Wednesday");
        sc.writeName("Thursday");
        sc.writeName("Saturday");
        sc.writeName("Sunday");
    }
}
```

According to the above code, the output of main() from TestClientState should be:

```
monday
TUESDAY
WEDNESDAY
thursday
SATURDAY
SUNDAY
```

## External links

- State Design Pattern <sup>[4]</sup>

## References

- [1] Erich Gamma; Richard Helm, Ralph Johnson, John M. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0201633612.
- [2] State pattern in UML and in LePUS3 (<http://www.lepus.org.uk/ref/companion/State.xml>)
- [3] legend (<http://lepus.org.uk/ref/legend/legend.xml>)
- [4] [http://sourcemaking.com/design\\_patterns/state](http://sourcemaking.com/design_patterns/state)

# Strategy pattern

In computer programming, the **strategy pattern** (also known as the **policy pattern**) is a particular software design pattern, whereby algorithms can be selected at runtime.

The strategy pattern is intended to provide a means to define a family of algorithms, encapsulate each one as an object, and make them interchangeable. The strategy pattern lets the algorithms vary independently from clients that use them.

For instance, a class that performs validation on incoming data may use a strategy pattern to select a validation algorithm based on the type of data, the source of the data, or other discriminating factors. These factors are not known for each case until run-time, and may require radically different validation to be performed. The validation strategies, encapsulated separately from the validating object, may be used by other validating objects in different areas of the system (or even different systems) without code duplication.

The essential requirement in the programming language is the ability to store a reference to some code in a data structure and retrieve it. This can be achieved by mechanisms such as the native function pointer, the first-class function, classes or class instances in object-oriented programming languages, or accessing the language implementation's internal storage of code via reflection.

## Formal Definition

The strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.<sup>[1]</sup>

## Example

The following example is in Java.

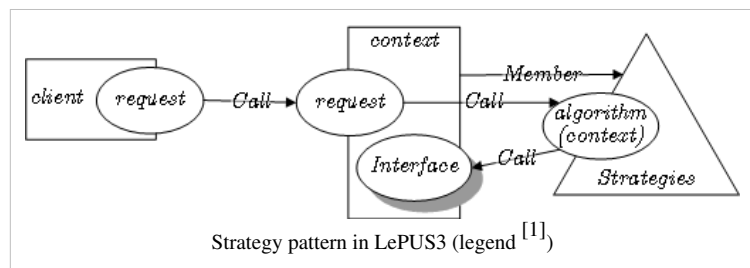
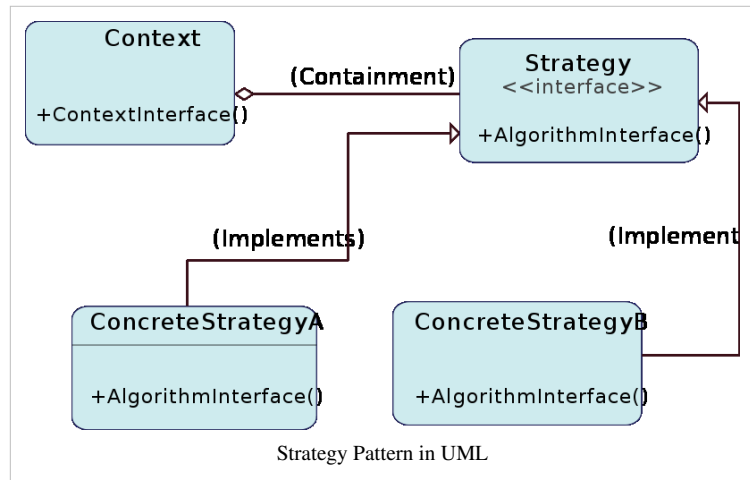
```
//StrategyExample test application

class StrategyExample {

    public static void main(String[] args) {

        Context context;

        // Three contexts following different strategies
```





```
        context = new Context(new ConcreteStrategyAdd());
        int resultA = context.executeStrategy(3,4);

        context = new Context(new ConcreteStrategySubtract());
        int resultB = context.executeStrategy(3,4);

        context = new Context(new ConcreteStrategyMultiply());
        int resultC = context.executeStrategy(3,4);
    }
}

// The classes that implement a concrete strategy should implement this

// The context class uses this to call the concrete strategy
interface Strategy {
    int execute(int a, int b);
}

// Implements the algorithm using the strategy interface
class ConcreteStrategyAdd implements Strategy {

    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategyAdd's execute()");
        return a + b;    // Do an addition with a and b
    }
}

class ConcreteStrategySubtract implements Strategy {

    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategySubtract's
execute()");
        return a - b;    // Do a subtraction with a and b
    }
}

class ConcreteStrategyMultiply implements Strategy {

    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategyMultiply's
execute()");
        return a * b;    // Do a multiplication with a and b
    }
}

// Configured with a ConcreteStrategy object and maintains a reference
to a Strategy object
```

```

class Context {

    private Strategy strategy;

    // Constructor
    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    public int executeStrategy(int a, int b) {
        return strategy.execute(a, b);
    }
}

```

## Strategy versus Bridge

The UML class diagram for the Strategy pattern is the same as the diagram for the Bridge pattern. However, these two design patterns aren't the same in their *intent*. While the Strategy pattern is meant for *behavior*, the Bridge pattern is meant for *structure*.

The coupling between the context and the strategies is tighter than the coupling between the abstraction and the implementation in the Bridge pattern.

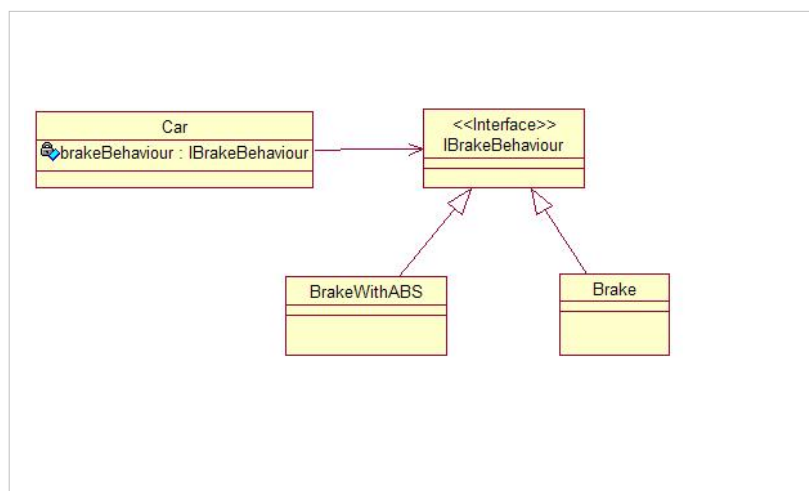
## Strategy and open closed principle

According to Strategy pattern, the behaviors of a class should not be inherited, instead they should be encapsulated using interfaces. As an example, consider a car class. Two possible functionalities for car are brake and accelerate.

Since accelerate and brake behaviors change frequently between models, a common approach is to implement these behaviors in subclasses. This approach has significant drawbacks: accelerate and brake behaviors must be

declared in each new Car model. The work of managing these behaviors increases greatly as the number of models increases, and requires code to be duplicated across models. Additionally, it is not easy to determine the exact nature of the behavior for each model without investigating the code in each.

The strategy pattern uses composition instead of inheritance. In the strategy pattern behaviors are defined as separate interfaces and specific classes that implement these interfaces. Specific classes encapsulate these interfaces. This allows better decoupling between the behavior and the class that uses the behavior. The behavior can be changed without breaking the classes that use it, and the classes can switch between behaviors by changing the specific implementation used without requiring any significant code changes. Behaviors can also be changed at run-time as well as at design-time. For instance, a car object's brake behavior can be changed from BrakeWithABS() to Brake() by changing the brakeBehavior member to:



```
brakeBehavior = new Brake();
```

This gives greater flexibility in design and is in harmony with the Open/closed principle (OCP) that states classes should be open for extension but closed for modification.

## References

- [1] Eric Freeman, Elisabeth Freeman, Kathy Sierra and Bert Bates, *Head First Design Patterns*, First Edition, Chapter 1, Page 24, O'Reilly Media, Inc, 2004. ISBN 978-0-596-00712-6

## External links

- The Strategy Pattern from the Net Objectives Repository (<http://www.netobjectivesrepository.com/TheStrategyPattern>)
- Strategy Pattern for Java article (<http://www.javaworld.com/javaworld/jw-04-2002/jw-0426-designpatterns.html>)
- Strategy pattern in UML and in LePUS3 (<http://www.lepus.org.uk/ref/companion/Strategy.xml>) (a formal modelling notation)
- Data & object factory (<http://www.dofactory.com/Patterns/PatternStrategy.aspx>)
- Refactoring: Replace Type Code with State/Strategy (<http://www.refactoring.com/catalog/replaceTypeCodeWithStateStrategy.html>)
- Strategy Pattern with a twist! (<http://anirudhvyas.com/root/2008/04/02/a-much-better-strategy-pattern/>)
- Strategy Pattern example in PHP (<http://www.labelmedia.co.uk/blog/posts/design-patterns-strategy-pattern.html>)

## Template method pattern

---

In software engineering, the **template method pattern** is a design pattern. It is a behavioral pattern, and is unrelated to C++ templates.

## Introduction

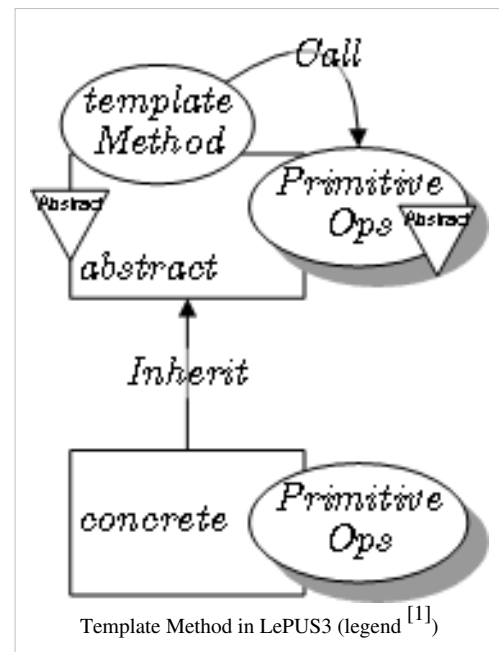
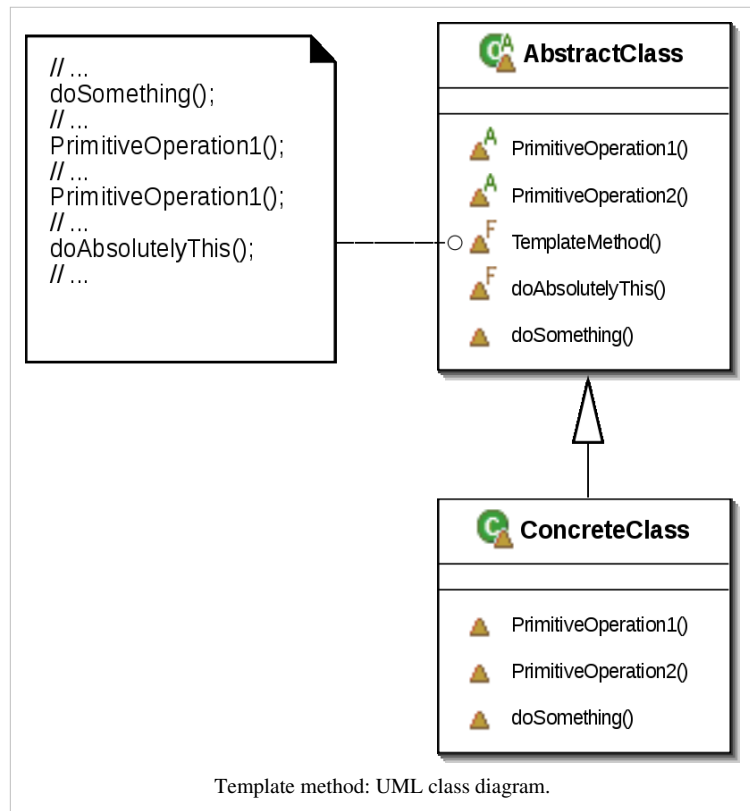
A *template method* defines the program skeleton of an algorithm. One or more of the algorithm steps can be overridden by subclasses to allow differing behaviors while ensuring that the overarching algorithm is still followed.

In object-oriented programming, first a class is created that provides the basic steps of an algorithm design. These steps are implemented using abstract methods. Later on, subclasses change the abstract methods to implement real actions. Thus the general algorithm is saved in one place but the concrete steps may be changed by the subclasses.

The template method thus manages the larger picture of task semantics, and more refined implementation details of selection and sequence of methods. This larger picture calls abstract and non-abstract methods for the task at hand. The non-abstract methods are completely controlled by the template method but the abstract methods, implemented in subclasses, provide the pattern's expressive power and degree of freedom. Some or all of the abstract methods can be specialized in a subclass, allowing the writer of the subclass to provide particular behavior with minimal modifications to the larger semantics. The template method (which is

non-abstract) remains unchanged in this pattern, ensuring that the subordinate non-abstract methods and abstract methods are called in the originally-intended sequence.

The template method occurs frequently, at least in its simplest case, where a method calls only one abstract method, with object oriented languages. If a software writer uses a polymorphic method at all, this design pattern may be a rather natural consequence. This is because a method calling an abstract or polymorphic function is simply the reason for being of the abstract or polymorphic method. The template method may be used to add immediate present value to the software or with a vision to enhancements in the future.



The Template method pattern is strongly related to the Non-Virtual Interface (NVI) pattern. The NVI pattern recognizes the benefits of a non-abstract method invoking the subordinate abstract methods. This level of indirection allows for pre and post operations relative to the abstract operations both immediately and with future unforeseen changes. The NVI pattern can be deployed with very little software production and runtime cost. Many commercial software frameworks employ the NVI pattern.

Template method implements the Protected variations GRASP principle, like the Adapter pattern does. The difference is that Adapter gives the same interface for several operations while Template Method does so only for one.

## Usage

The template method is used to:

- let subclasses implement (through method overriding) behavior that can vary
- avoid duplication in the code: the general workflow structure is implemented once in the abstract class's algorithm, and necessary variations are implemented in each of the subclasses.
- control at what point(s) subclassing is allowed. As opposed to a simple polymorphic override, where the base method would be entirely rewritten allowing radical change to the workflow, only the specific details of the workflow are allowed to change.

The control structure (inversion of control) that is the result of the application of a template pattern is often referred to as the Hollywood Principle: "Don't call us, we'll call you." Using this principle, the template method in a parent class controls the overall process by calling subclass methods as required. This is shown in the following Java example:

## Example

```
/**
 * An abstract class that is
 * common to several games in
 * which players play against
 * the others, but only one is
 * playing at a given time.
 */

abstract class Game {

    protected int playersCount;
    abstract void initializeGame();
    abstract void makePlay(int player);
    abstract boolean endOfGame();
    abstract void printWinner();

    /* A template method : */
    final void playOneGame(int playersCount) {
        this.playersCount = playersCount;
        initializeGame();
        int j = 0;
        while (!endOfGame()) {
            makePlay(j);
        }
    }
}
```

```
        j = (j + 1) % playersCount;
    }
    printWinner();
}
}

//Now we can extend this class in order
//to implement actual games:

class Monopoly extends Game {

    /* Implementation of necessary concrete methods */
    void initializeGame() {
        // Initialize money
    }
    void makePlay(int player) {
        // Process one turn of player
    }
    boolean endOfGame() {
        // Return true if game is over
        // according to Monopoly rules
    }
    void printWinner() {
        // Display who won
    }
    /* Specific declarations for the Monopoly game. */

    // ...
}

class Chess extends Game {

    /* Implementation of necessary concrete methods */
    void initializeGame() {
        // Put the pieces on the board
    }
    void makePlay(int player) {
        // Process a turn for the player
    }
    boolean endOfGame() {
        // Return true if in Checkmate or
        // Stalemate has been reached
    }
    void printWinner() {
        // Display the winning player
    }
    /* Specific declarations for the chess game. */
}
```

```
// ...
}
```

## External links

- Template design pattern in C# and VB.NET <sup>[1]</sup>
- Working with Template Classes in PHP 5 <sup>[2]</sup>
- Template Method pattern in UML and in LePUS3 <sup>[3]</sup> (a formal modelling language)
- Difference between Adapter and Template Method pattern <sup>[4]</sup>
- Template Method Design Pattern <sup>[5]</sup>

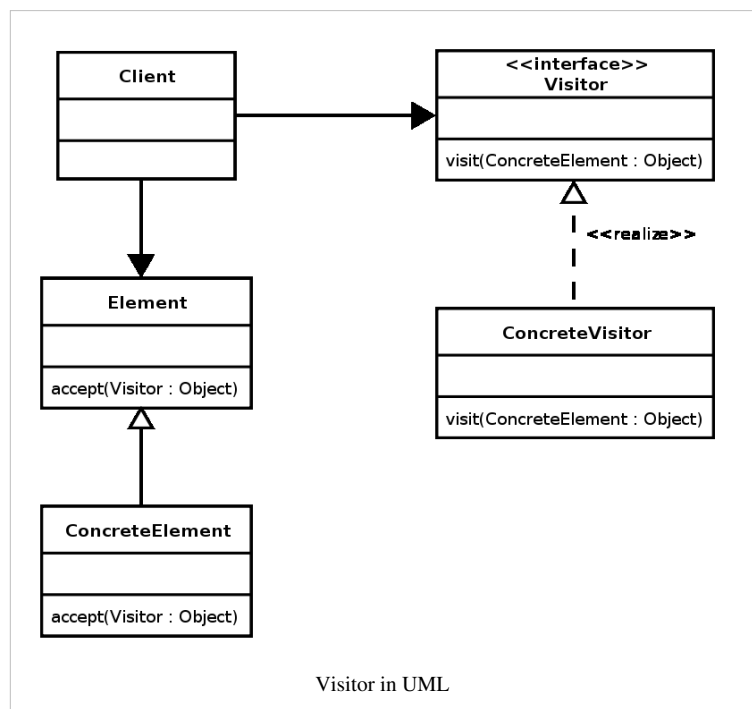
## References

- [1] <http://www.dofactory.com/Patterns/PatternTemplate.aspx>  
 [2] <http://www.devshed.com/c/a/PHP/Working-with-Template-Classes-in-PHP-5/>  
 [3] <http://www.lepus.org.uk/ref/companion/TemplateMethod.xml>  
 [4] <http://programmersnotes.info/2009/03/03/difference-between-adapter-and-template-method-pattern/>  
 [5] [http://sourcemaking.com/design\\_patterns/template\\_method](http://sourcemaking.com/design_patterns/template_method)

# Visitor pattern

In object-oriented programming and software engineering, the **visitor** design pattern is a way of separating an algorithm from an object structure it operates on. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures. It is one way to easily follow the open/closed principle.

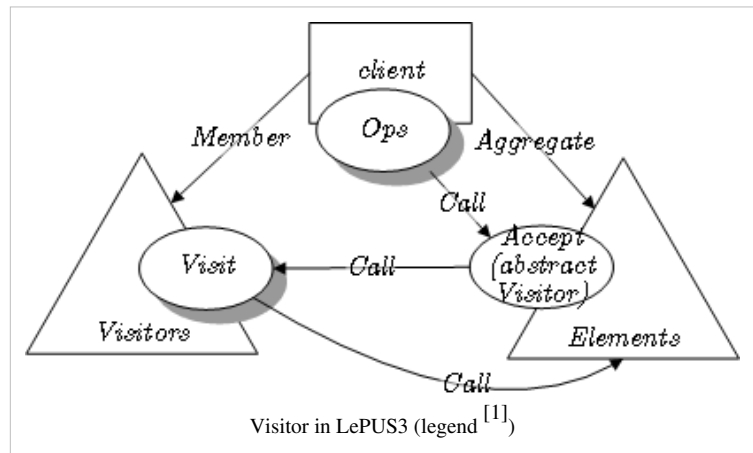
In essence, the visitor allows one to add new virtual functions to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function. The visitor takes the instance reference as input, and implements the goal through double dispatch.



While powerful, the visitor pattern is more limited than conventional virtual functions. It is not possible to create visitors for objects without adding a small callback method inside each class. In naive implementations, the callback method in each of the classes is not inheritable.

## Details

A user object receives a pointer to another object which implements an algorithm. The first is designated 'element class' and the latter 'the visitor class'. The idea is to use a structure of element classes, each of which has an `accept()` method taking a visitor object for an argument. visitor is an interface having a `visit()` method for each element class. The `accept()` method of an element class calls back the `visit()` method for its class. Separate concrete visitor classes can then be written to perform some particular operations, by implementing these operations in their respective `visit()` methods.



One of these `visit()` methods of a concrete visitor can be thought of as a method not of a single class, but rather a method of a pair of classes: the concrete visitor and the particular element class. Thus the visitor pattern simulates double dispatch in a conventional single-dispatch object-oriented language such as Java, Smalltalk, and C++. For an explanation of how double dispatch differs from function overloading, see Double dispatch is more than function overloading in the double dispatch article. In the Java language, two techniques have been documented that use reflection to simplify the mechanics of double dispatch simulation in the visitor pattern: getting rid of `accept()` methods <sup>[1]</sup> (the Walkabout variation), and getting rid of extra `visit()` methods <sup>[2]</sup>.

The visitor pattern also specifies how iteration occurs over the object structure. In the simplest version, where each algorithm needs to iterate in the same way, the `accept()` method of a container element, in addition to calling back the `visit()` method of the visitor, also passes the visitor object to the `accept()` method of all its constituent child elements.

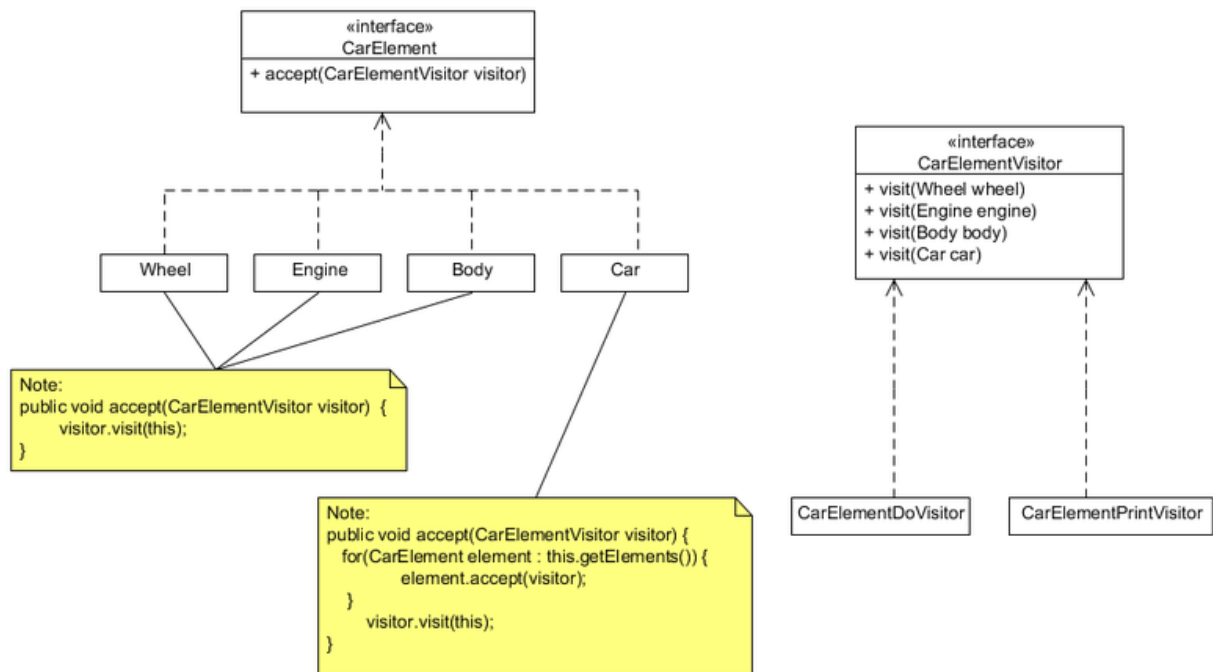
Because the visitor object has one principal function (manifested in a plurality of specialized methods) and that function is called `visit()`, the visitor can be readily identified as a potential function object. Likewise, the `accept()` function can be identified as a function applicator, a mapper, which knows how to traverse a particular type of object and apply a function to its elements. Lisp's object system with its multiple dispatch does not replace the Visitor pattern, but merely provides a more concise implementation of it in which the pattern all but disappears.

## Java example

The following example is in the Java programming language, and shows how the contents of a tree of nodes (in this case describing the components of a car) can be printed. Instead of creating "print" methods for each subclass (Wheel, Engine, Body, and Car), a single class (`CarElementPrintVisitor`) performs the required printing action. Because different subclasses require slightly different actions to print properly, `CarElementDoVisitor` dispatches actions based on the class of the argument passed to it.



## Diagram



## Source

```

interface CarElementVisitor {
    void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Body body);
    void visit(Car car);
}

interface CarElement {
    void accept(CarElementVisitor visitor); // CarElements have to
    provide accept().
}

class Wheel implements CarElement {
    private String name;

    public Wheel(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}
  
```

```
}

class Engine implements CarElement {
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Body implements CarElement {
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Car implements CarElement{
    CarElement[] elements;

    public CarElement[] getElements() {
        return elements.clone(); // Return a copy of the array of
references.
    }

    public Car() {
        this.elements = new CarElement[]
        { new Wheel("front left"), new Wheel("front right"),
          new Wheel("back left") , new Wheel("back right"),
          new Body(), new Engine() };
    }

    public void accept(CarElementVisitor visitor) {
        for(CarElement element : this.getElements()) {
            element.accept(visitor);
        }
        visitor.visit(this);
    }
}

class CarElementPrintVisitor implements CarElementVisitor {
    public void visit(Wheel wheel) {
        System.out.println("Visiting " + wheel.getName()
                           + " wheel");
    }

    public void visit(Engine engine) {
        System.out.println("Visiting engine");
    }
}
```

```
public void visit(Body body) {
    System.out.println("Visiting body");
}

public void visit(Car car) {
    System.out.println("Visiting car");
}
}

class CarElementDoVisitor implements CarElementVisitor {
    public void visit(Wheel wheel) {
        System.out.println("Kicking my " + wheel.getName() + " wheel");
    }

    public void visit(Engine engine) {
        System.out.println("Starting my engine");
    }

    public void visit(Body body) {
        System.out.println("Moving my body");
    }

    public void visit(Car car) {
        System.out.println("Starting my car");
    }
}

public class VisitorDemo {
    static public void main(String[] args){
        Car car = new Car();
        car.accept(new CarElementPrintVisitor());
        car.accept(new CarElementDoVisitor());
    }
}
```

## State

Aside from potentially improving separation of concerns, the visitor pattern has an additional advantage over simply calling a polymorphic method: a visitor object can have state. This is extremely useful in many cases where the action performed on the object depends on previous such actions.

An example of this is a pretty-printer in a programming language implementation (such as a compiler or interpreter). Such a pretty-printer object (implemented as a visitor, in this example), will visit nodes in a data structure that represents a parsed and processed program. The pretty-printer will then generate a textual representation of the program tree. To make the representation human-readable, the pretty-printer should properly indent program statements and expressions. The *current indentation level* can then be tracked by the visitor as its state, correctly applying encapsulation, whereas in a simple polymorphic method invocation, the indentation level would have to be exposed as a parameter and the caller would rely on the method implementation to use and propagate this parameter correctly.

## External links

- The Visitor Family of Design Patterns <sup>[3]</sup> by Robert C. Martin - a rough chapter from *The Principles, Patterns, and Practices of Agile Software Development*, Robert C. Martin, Prentice Hall
- Visitor pattern in UML and in LePUS3 <sup>[4]</sup> (a Design Description Language)
- Article "Componentization: the Visitor Example <sup>[5]</sup>" by Bertrand Meyer and Karine Arnout, *Computer* (IEEE), vol. 39, no. 7, July 2006, pages 23-30.
- Article "Domain Searching Using Visitors <sup>[6]</sup>" by Paul Mukherjee
- Article A Type-theoretic Reconstruction of the Visitor Pattern <sup>[7]</sup>
- Article "The Essence of the Visitor Pattern <sup>[8]</sup>" by Jens Palsberg and C. Barry Jay. 1997 IEEE-CS COMPSAC paper showing that accept() methods are unnecessary when reflection is available; introduces term 'Walkabout' for the technique.
- Article "A Time for Reflection <sup>[9]</sup>" by Bruce Wallace - subtitled "*Java 1.2's reflection capabilities eliminate burdensome accept() methods from your Visitor pattern*"
- Article "Cooperative Visitor: A Template Technique for Visitor Creation <sup>[10]</sup>" by Anand Shankar Krishnamoorthi
- Visitor Patterns <sup>[11]</sup> as a universal model of terminating computation.
- Visitor Pattern <sup>[12]</sup> using reflection(java).
- PerfectJPattern Open Source Project <sup>[13]</sup>, Provides a context-free and type-safe implementation of the Visitor Pattern in Java based on Delegates.
- Visitor Design Pattern <sup>[14]</sup>

## References

- [1] <http://www.cs.ucla.edu/~palsberg/paper/compsac98.pdf>
- [2] <http://www.javaworld.com/javaworld/jvatips/jw-jvatip98.html>
- [3] <http://objectmentor.com/resources/articles/visitor.pdf>
- [4] <http://www.lepus.org.uk/ref/companion/Visitor.xml>
- [5] <http://se.ethz.ch/~meyer/publications/computer/visitor.pdf>
- [6] <http://www.onjava.com/pub/a/onjava/2005/06/01/searchvisitor.html>
- [7] <http://www.cs.bham.ac.uk/~hxt/research/mfps-visitors.pdf>
- [8] <http://citeseer.ist.psu.edu/palsberg97essence.html>
- [9] <http://www.polyglotinc.com/reflection.html>
- [10] [http://www.artima.com/cppsource/cooperative\\_visitor.html](http://www.artima.com/cppsource/cooperative_visitor.html)
- [11] <http://goblin.colourcountry.net/apt1002/Visitor%20patterns>
- [12] [http://www.oodeesign.com/oo\\_design\\_patterns/behavioral\\_patterns/visitor\\_pattern.html](http://www.oodeesign.com/oo_design_patterns/behavioral_patterns/visitor_pattern.html)
- [13] <http://perfectjpattern.sourceforge.net/dp-visitor.html>
- [14] [http://sourcemaking.com/design\\_patterns/visitor](http://sourcemaking.com/design_patterns/visitor)

# Active object

---

The **active object** design pattern decouples method execution from method invocation that reside in their own thread of control.<sup>[1]</sup> The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests.<sup>[2]</sup>

The pattern consists of six elements:<sup>[3]</sup>

- A proxy, which provides an interface towards clients with publicly accessible methods.
- An interface which defines the method request on an active object.
- A list of pending requests from clients
- A scheduler, which decides which request to execute next
- the implementation of the active object method.
- A callback or variable for the client to receive the result.

## References

- [1] Douglas C. Schmidt; Michael Stal, Hans Rohnert, and Frank Buschmann (2000). *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons. ISBN 0-471-60695-2.
- [2] Bass, L., Clements, P., Kazman, R. *Software Architecture in Practice*. Addison Wesley, 2003
- [3] Lavender, R. Greg; Schmidt, Douglas C.. "Active Object" (<http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf>) (PDF). . Retrieved 2007-02-02.

# Balking pattern

---

The **balking pattern** is a software design pattern that only executes an action on an object when the object is in a particular state. For example, if an object reads ZIP files and a calling method invokes a get method on the object when the ZIP file is not open, the object would "balk" at the request. In the Java programming language, for example, an `IllegalStateException` might be thrown under these circumstances.

There are some specialists in this field who think this is more of an anti-pattern, than a design pattern. If an object cannot support its API, it should either limit the API so that the offending call is not available or so that the call can be made without limitation, it should:

- Be created in a sane state
- Not make itself available until it is in a sane state
- Become a facade and answer back an object that is in a sane state

## Usage

Objects that use this pattern are generally only in a state that is prone to balking temporarily but for an unknown amount of time. If objects are to remain in a state which is prone to balking for a known, finite period of time, then the guarded suspension pattern may be preferred.

## Implementation

Below is a general, simple example for an implementation of the balking pattern as originally seen in Grand (2002). As demonstrated by the definition above, notice how the "synchronized" line is utilized. If there are multiple calls to the job method, only one will proceed while the other calls will return with nothing. Another thing to note is the `jobCompleted()` method. The reason it is not synchronized is that there will never be multiple calls to it at the same time, as only one job will execute at a time.

---

```
public class Example{
    private boolean jobInProgress = false;

    public void job(){
        synchronized(this) {
            if(jobInProgress) {
                return;
            }
            jobInProgress = true;
        }
        //Code to execute job goes here
        ...
    }

    void jobCompleted() {
        jobInProgress = false;
    }
}
```

## References

- Grand, Mark (2002), *Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML, Second Edition*, Indianapolis, Ind: John Wiley & Sons.

# Messaging pattern

---

In software architecture, a **messaging pattern** is a network-oriented architectural pattern which describes how two different parts of a message passing system connect and communicate with each other.

In telecommunications, a **message exchange pattern (MEP)** describes the pattern of messages required by a communications protocol to establish or use a communication channel. There are two major message exchange patterns — a *request-response* pattern, and a *one-way* pattern. For example, the TCP is a *request-response* pattern protocol, and the UDP has a *one-way* pattern.

## SOAP

The term "Message Exchange Pattern" has a specific meaning within the SOAP protocol.<sup>[1] [2]</sup> SOAP MEP types include:

1. **In-Only:** This is equivalent to *one-way*. A standard one-way messaging exchange where the consumer sends a message to the provider that provides only a status response.
2. **Robust In-Only:** This pattern is for reliable one-way message exchanges. The consumer initiates with a message to which the provider responds with status. If the response is a status, the exchange is complete, but if the response is a fault, the consumer must respond with a status.
3. **In-Out:** This is equivalent to *request-response*. A standard two-way message exchange where the consumer initiates with a message, the provider responds with a message or fault and the consumer responds with a status.
4. **In Optional-Out:** A standard two-way message exchange where the provider's response is optional.
5. Out-Only
6. Robust Out-Only
7. Out-In
8. Out-Optional-In

## ØMQ

The ØMQ message queueing library provides a so-called *sockets* (a kind of generalization over the traditional IP and Unix sockets) which require to indicate a messaging pattern to be used, and are particularly optimized for that kind of patterns. The basic ØMQ patterns are:<sup>[3]</sup>

- **Request-reply** connects a set of clients to a set of services. This is a remote procedure call and task distribution pattern.
- **Publish-subscribe** connects a set of publishers to a set of subscribers. This is a data distribution pattern.
- **Push-pull** connects nodes in a fan-out / fan-in pattern that can have multiple steps, and loops. This is a parallel task distribution and collection pattern.
- **Exclusive pair** connects two sockets in an exclusive pair. This is a low-level pattern for specific, advanced use cases.

Each pattern defines a particular network topology. Request-reply defines so-called "service bus", publish-subscribe defines "data distribution tree", push-pull defines "parallelised pipeline". All the patterns are deliberately designed in such a way as to be infinitely scalable and thus usable on Internet scale.<sup>[4]</sup>

---

## References

- [1] <http://www.w3.org/TR/soap12-part1/#soapmep> SOAP MEPs in SOAP W3C Recommendation v1.2
- [2] Web Services Description Language (WSDL) Version 2.0: Additional MEPs (<http://www.w3.org/TR/wsdl20-additional-meps/>)
- [3] ØMQ User Guide (<http://www.zeromq.org/docs:user-guide>)
- [4] Scalability Layer Hits the Internet Stack (<http://www.250bpm.com/hits>)

## External links

- Messaging Patterns in Service-Oriented Architecture (<http://msdn.microsoft.com/en-us/library/aa480027.aspx>)

# Double-checked locking

In software engineering, **double-checked locking** (also known as "double-checked locking optimization"<sup>[1]</sup> ".) is a software design pattern used to reduce the overhead of acquiring a lock by first testing the locking criterion (the "lock hint") without actually acquiring the lock. Only if the locking criterion check indicates that locking is required does the actual locking logic proceed.

The pattern, when implemented in some language/hardware combinations, can be unsafe. It can therefore sometimes be considered to be an anti-pattern.

It is typically used to reduce locking overhead when implementing "lazy initialization" in a multi-threaded environment, especially as part of the Singleton pattern. Lazy initialization avoids initializing a value until the first time it is accessed.

## Usage in Java

Consider, for example, this code segment in the Java programming language as given by [2] (as well as all other Java code segments):

```
// Single threaded version
class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null)
            helper = new Helper();
        return helper;
    }

    // other functions and members...
}
```

The problem is that this does not work when using multiple threads. A lock must be obtained in case two threads call `getHelper()` simultaneously. Otherwise, either they may both try to create the object at the same time, or one may wind up getting a reference to an incompletely initialized object.

This can be done by expensive synchronizing, as is shown in the following example.

```
// Correct but possibly expensive multithreaded version
class Foo {
    private Helper helper = null;
    public synchronized Helper getHelper() {
```



```
        if (helper == null)
            helper = new Helper();
        return helper;
    }

    // other functions and members...
}
```

However, the first call to `getHelper()` will create the object and only the few threads trying to access it during that time need to be synchronized; after that all calls just get a reference to the member variable. Since synchronizing a method can decrease performance by a factor of 100 or higher<sup>[3]</sup>, the overhead of acquiring and releasing a lock every time this method is called seems unnecessary: once the initialization has been completed, acquiring and releasing the locks would appear unnecessary. Many programmers have attempted to optimize this situation in the following manner:

1. Check that the variable is initialized (without obtaining the lock). If it is initialized, return it immediately.
2. Obtain the lock.
3. Double-check whether the variable has already been initialized: if another thread acquired the lock first, it may have already done the initialization. If so, return the initialized variable.
4. Otherwise, initialize and return the variable.

```
// Broken multithreaded version
// "Double-Checked Locking" idiom
class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null) {
                    helper = new Helper();
                }
            }
        }
        return helper;
    }

    // other functions and members...
}
```

Intuitively, this algorithm seems like an efficient solution to the problem. However, this technique has many subtle problems and should usually be avoided. For example, consider the following sequence of events:

1. Thread *A* notices that the value is not initialized, so it obtains the lock and begins to initialize the value.
2. Due to the semantics of some programming languages, the code generated by the compiler is allowed to update the shared variable to point to a partially constructed object before *A* has finished performing the initialization.
3. Thread *B* notices that the shared variable has been initialized (or so it appears), and returns its value. Because thread *B* believes the value is already initialized, it does not acquire the lock. If *B* uses the object before all of the initialization done by *A* is seen by *B* (either because *A* has not finished initializing it or because some of the initialized values in the object have not yet percolated to the memory *B* uses (cache coherence)), the program will likely crash.

One of the dangers of using double-checked locking in J2SE 1.4 (and earlier versions) is that it will often appear to work: it is not easy to distinguish between a correct implementation of the technique and one that has subtle problems. Depending on the compiler, the interleaving of threads by the scheduler and the nature of other concurrent system activity, failures resulting from an incorrect implementation of double-checked locking may only occur intermittently. Reproducing the failures can be difficult.

As of J2SE 5.0, this problem has been fixed. The `volatile` keyword now ensures that multiple threads handle the singleton instance correctly. This new idiom is described in [2]:

```
// Works with acquire/release semantics for volatile
// Broken under Java 1.4 and earlier semantics for volatile
class Foo {
    private volatile Helper helper = null;
    public Helper getHelper() {
        Helper result = helper;
        if (result == null) {
            synchronized(this) {
                result = helper;
                if (result == null)
                    helper = result = new Helper();
            }
        }
        return result;
    }

    // other functions and members...
}
```

Note the usage of the local variable `result` which seems unnecessary. For some versions of the Java VM, it will make the code 25% faster and for others, it won't hurt.<sup>[4]</sup>

If the helper object is static (one per class loader), an alternative is the initialization on demand holder idiom<sup>[5]</sup> See Listing 16.6 on<sup>[6]</sup>

```
// Correct lazy initialization in Java
@ThreadSafe
class Foo {
    private static class HelperHolder {
        public static Helper helper = new Helper();
    }
    public static Helper getHelper() {
        return HelperHolder.helper;
    }
}
```

This relies on the fact that inner classes are not loaded until they are referenced.

Semantics of `final` field in Java 5 can be employed to safely publish the helper object without using `volatile`.<sup>[7]</sup>

```
public class FinalWrapper<T>
{
```

```

    public final T value;
    public FinalWrapper(T value) { this.value = value; }
}

public class Foo
{
    private FinalWrapper<Helper> helperWrapper = null;
    public Helper getHelper()
    {
        FinalWrapper<Helper> wrapper = helperWrapper;
        if (wrapper == null)
        {
            synchronized(this)
            {
                if (helperWrapper == null)
                    helperWrapper = new FinalWrapper<Helper>( new Helper() );
                wrapper = helperWrapper;
            }
        }
        return wrapper.value;
    }
}

```

The local variable `wrapper` is required for correctness. Performance of this implementation is not necessarily better than the `volatile` implementation.

## Usage in Microsoft Visual C++

Double-checked locking can be implemented in Visual C++ 5.0 if the pointer to the resource is declared with the C++ keyword **volatile**. Visual C++ 5.0 guarantees that volatile variables will behave as fence instructions, as in J2SE 5.0, preventing both compiler and CPU arrangement of reads and writes with acquire semantics (for reads) and release semantics (for writes)<sup>[8]</sup>. There is no such guarantee in previous versions of Visual C++. However, marking the pointer to the resource as volatile may harm performance elsewhere, if the pointer declaration is visible elsewhere in code, by forcing the compiler to treat it as a fence elsewhere, even when it is not necessary.

## Usage in Microsoft .NET (Visual Basic, C#)

Double-checked locking can be implemented efficiently in .NET with careful use of the `MemoryBarrier` instruction:

```

public class MySingleton
{
    private static object myLock = new object();
    private static MySingleton mySingleton = null;
    private static bool ready = false;

    private MySingleton()
    { }

    public static MySingleton GetInstance()

```

```

{
    if (!ready)    // check
    {
        lock (myLock)
        {
            if (!ready)    // double check
            {
                mySingleton = new MySingleton();
                System.Threading.Thread.MemoryBarrier();    // fence
                ready = true;
            }
        }
    }

    return mySingleton;
}
}

```

In this example, the "lock hint" is the ready flag which can only change after mySingleton is fully constructed and ready for use.

Alternatively, the C# keyword **volatile** can be used to enforce read/write fences around all access of mySingleton, which would negate many of the efficiencies inherent in the double-checked locking strategy.

## References

- [1] Schmidt, D et al. Pattern-Oriented Software Architecture Vol 2, 2000 pp353-363
- [2] <http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>
- [3] Boehm, Hans-J. "Threads Cannot Be Implemented As a Library", ACM 2005, p265
- [4] Joshua Bloch "Effective Java, Second Edition", p. 283
- [5] Brian Goetz et al. Java Concurrency in Practice, 2006 pp348
- [6] (<http://www.javaconcurrencyinpractice.com/listings.html>)
- [7] (<https://mailman.cs.umd.edu/mailman/private/javamemorymodel-discussion/2010-July/000422.html>)Javamemorymodel-discussion mailing list
- [8] [http://msdn.microsoft.com/en-us/library/12a04hfd\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/12a04hfd(VS.100).aspx)

## External links

- Issues with the double checked locking mechanism captured in Jeu George's Blogs (<http://purevirtuals.com/blog/2006/06/16/son-of-a-bug/>) Pure Virtuals (<http://purevirtuals.com/blog/2006/06/16/son-of-a-bug/>)
- Implementation of Various Singleton Patterns including the Double Checked Locking Mechanism (<http://www.tekpool.com/node/2693>) at TEKPOOL (<http://www.tekpool.com/?p=35>)
- "Double Checked Locking" Description from the Portland Pattern Repository
- "Double Checked Locking is Broken" Description from the Portland Pattern Repository
- Paper " C++ and the Perils of Double-Checked Locking ([http://www.aristeia.com/Papers/DDJ\\_Jul\\_Aug\\_2004\\_revised.pdf](http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf))" (475 KB) by Scott Meyers and Andrei Alexandrescu
- Article " Double-checked locking: Clever, but broken (<http://www.javaworld.com/jw-02-2001/jw-0209-double.html>)" by Brian Goetz
- The "Double-Checked Locking is Broken" Declaration (<http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>); David Bacon et al.

- Double-checked locking and the Singleton pattern (<http://www-106.ibm.com/developerworks/java/library/j-dcl.html>)
- Singleton Pattern and Thread Safety (<http://www.oaklib.org/docs/oak/singleton.html>)
- volatile keyword in VC++ 2005 (<http://msdn2.microsoft.com/en-us/library/12a04hfd.aspx>)
- Java Examples and timing of double check locking solutions ([http://blogs.sun.com/cwebster/entry/double\\_check\\_locking](http://blogs.sun.com/cwebster/entry/double_check_locking))

## Asynchronous method invocation

---

In (multithreaded) object-oriented programming, **asynchronous method invocation** (**AMI**), also known as **asynchronous method calls** or **asynchronous pattern** is a design pattern for asynchronous invocation of potentially long-running methods of an object.<sup>[1]</sup> It is equivalent to the **IOU pattern** described in 1996 by Allan Vermeulen.<sup>[2]</sup> <sup>[3]</sup> The **event-based asynchronous pattern** in .NET Framework and the *java.util.concurrent.FutureTask* class in Java use events to solve the same problem. This pattern is a variant of AMI whose implementation carries more overhead, but it is useful for objects representing software components.

In most programming languages a called method is executed synchronously, i.e. in the thread of execution from which it is invoked. If the method needs a long time to completion, e.g. because it is loading data over the internet, the calling thread is blocked until the method has finished. When this is not desired, it is possible to start a "worker thread" and invoke the method from there. In most programming environments this requires many lines of code, especially if care is taken to avoid the overhead that may be caused by creating many threads. AMI solves this problem in that it augments a potentially long-running ("synchronous") object method with an "asynchronous" variant that returns immediately, along with additional methods that make it easy to receive notification of completion, or to wait for completion at a later time.

One common use of AMI is in the active object design pattern. Alternatives are synchronous method invocation and future objects.<sup>[4]</sup> An example for an application that may make use of AMI is a web browser that needs to display a web page even before all images are loaded.

### Example

The following example is loosely based on a standard AMI style used in the .NET Framework.<sup>[5]</sup> Given a method *Accomplish*, one adds two new methods *BeginAccomplish* and *EndAccomplish*:

```
Class Example { Result Accomplish(args ...) IAsyncResult BeginAccomplish(args ...) Result EndAccomplish(IAsyncResult a) ... }
```

Upon calling *BeginAccomplish*, the client immediately receives an object of type *AsyncResult* (which implements the *IAsyncResult* interface), so it can continue the calling thread with unrelated work. In the simplest case, eventually there is no more such work, and the client calls *EndAccomplish* (passing the previously received object), which blocks until the method has completed and the result is available.<sup>[6]</sup> The *AsyncResult* object normally provides at least a method that allows the client to query whether the long-running method has already completed:

```
Interface IAsyncResult { bool HasCompleted() ... }
```

One can also pass a callback method to *BeginAccomplish*, to be invoked when the long-running method completes. It typically calls *EndAccomplish* to obtain the return value of the long-running method. A problem with the callback mechanism is that the callback function is naturally executed in the worker thread (rather than in the original calling thread), which may cause race conditions.<sup>[7] [8]</sup>

In the .NET Framework documentation, the term event-based asynchronous pattern refers to an alternative API style (available since .NET 2.0) using a method named *AccomplishAsync* instead of *BeginAccomplish*.<sup>[9] [10]</sup> A

---

superficial difference is that in this style the return value of the long-running method is passed directly to the callback method. Much more importantly, the API uses a special mechanism to run the callback method (which resides in an event object of type `AccomplishCompleted`) in the same thread in which `BeginAccomplish` was called. This eliminates the danger of race conditions, making the API easier to use and suitable for software components; on the other hand this implementation of the pattern comes with additional object creation and synchronization overhead.<sup>[11]</sup>

## References

- [1] "Asynchronous Method Invocation" (<http://www.zeroc.com/doc/Ice-3.2.1/manual/Async.34.2.html#71139>). *Distributed Programming with Ice*. ZeroC, Inc.. Retrieved 22 November 2008.
- [2] Vermeulen, Allan (June 1996). "An Asynchronous Design Pattern" (<http://www.ddj.com/184409898>). *Dr. Dobbs's Journal*. Retrieved 22 November 2008.
- [3] Nash, Trey (2007). "Threading in C#". *Accelerated C# 2008*. Apress. ISBN 9781590598733.
- [4] Lavender, R. Greg; Douglas C. Schmidt (PDF). *Active Object* (<http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf>). Retrieved 22 November 2008.
- [5] "Asynchronous Programming Design Patterns" (<http://msdn.microsoft.com/en-us/library/ms228969.aspx>). *.NET Framework Developer's Guide*. Microsoft Developer Network. Retrieved 22 November 2008.
- [6] "Asynchronous Programming Overview" (<http://msdn.microsoft.com/en-us/library/ms228963.aspx>). *.NET Framework Developer's Guide*. Microsoft Developer Network. Retrieved 22 November 2008.
- [7] "Using an AsyncCallback Delegate to End an Asynchronous Operation" (<http://msdn.microsoft.com/en-us/library/ms228972.aspx>). *.NET Framework Developer's Guide*. Microsoft Developer Network. Retrieved 22 November 2008.
- [8] "Concurrency Issues" (<http://www.zeroc.com/doc/Ice-3.2.1/manual/Async.34.3.html#76161>). *Distributed Programming with Ice*. ZeroC, Inc.. Retrieved 22 November 2008.
- [9] Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, and Morgan Skinner (2008). "Event-based Asynchronous Pattern". *Professional C# 2008*. Wiley. pp. 570–571. ISBN 0470191376.
- [10] "Multithreaded Programming with the Event-based Asynchronous Pattern" (<http://msdn.microsoft.com/en-us/library/hkasytyf.aspx>). *.NET Framework Developer's Guide*. Microsoft Developer Network. Retrieved 22 November 2008.
- [11] "Deciding When to Implement the Event-based Asynchronous Pattern" (<http://msdn.microsoft.com/en-us/library/ms228966.aspx>). *.NET Framework Developer's Guide*. Microsoft Developer Network. Retrieved 22 November 2008.

## Further reading

- Chris Sells and Ian Griffiths (2007). "Appendix C.3: The Event-Based Asynchronous Pattern". *Programming WPF*. O'Reilly. pp. 747–749. ISBN 0596510373.
- Using asynchronous method calls in C# ([http://articles.techrepublic.com.com/5100-10878\\_11-1044325.html](http://articles.techrepublic.com.com/5100-10878_11-1044325.html))

# Guarded suspension

In concurrent programming, **guarded suspension**<sup>[1]</sup> is a software design pattern for managing operations that require both a lock to be acquired and a precondition to be satisfied before the operation can be executed. The guarded suspension pattern is typically applied to method calls in object-oriented programs, and involves suspending the method call, and the calling thread, until the precondition (acting as a guard) is satisfied.

## Usage

Because it is blocking, the guarded suspension pattern is generally only used when the developer knows that a method call will be suspended for a finite and reasonable period of time. If a method call is suspended for too long, then the overall program will slow down or stop, waiting for the precondition to be satisfied. If the developer knows that the method call suspension will be indefinite or for an unacceptably long period, then the balking pattern may be preferred.

## Implementation

In Java, the `Object` class provides the `wait()` and `notify()` methods to assist with guarded suspension. In the implementation below, originally found in Kuchana (2004), if there is no precondition satisfied for the method call to be successful, then the method will wait until it finally enters a valid state.

```
public class Example {
    synchronized void guardedMethod() {
        while (!preCondition()) {
            try {
                //Continue to wait
                wait();
                //...
            } catch (InterruptedException e) {
                //...
            }
        }
        //Actual task implementation
    }

    synchronized void alterObjectStateMethod() {
        //Change the object state
        //....
        //Inform waiting threads
        notify();
    }
}
```

An example of an actual implementation would be a queue object with a `get` method that has a guard to detect when there are no items in the queue. Once the "put" method notifies the other methods (for example, a `get()` method), then the `get()` method can exit its guarded state and proceed with a call. Once the queue is empty, then the `get()` method will enter a guarded state once again.

## Notes

[1] Lea, Doug (2000). *Concurrent Programming in Java Second Edition*. Reading, MA: Addison-Wesley. ISBN 0-201-31009-0.

## References

- Kuchana, Partha (2004). *Software Architecture Design Patterns in Java*. Boca Raton, Florida: Auerbach Publications.

# Lock (computer science)

---

In computer science, a **lock** is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution. Locks are one way of enforcing concurrency control policies.

## Types

Generally, locks are **advisory locks**, where each thread cooperates by acquiring the lock before accessing the corresponding data. Some systems also implement **mandatory locks**, where attempting unauthorized access to a locked resource will force an exception in the entity attempting to make the access.

A (binary) semaphore is the simplest type of lock. In terms of **access to the data**, no distinction is made between shared (read only) or exclusive (read and write) modes. Other schemes provide for a shared mode, where several threads can acquire a shared lock for read-only access to the data. Other modes such as exclusive, intend-to-exclude and intend-to-upgrade are also widely implemented.

Independent of the type of lock chosen above, locks can be classified by what happens when the lock strategy prevents **progress of a thread**. Most locking designs block the execution of the thread requesting the lock until it is allowed to access the locked resource. A spinlock is a lock where the thread simply waits ("spins") until the lock becomes available. It is very efficient if threads are only likely to be blocked for a short period of time, as it avoids the overhead of operating system process re-scheduling. It is wasteful if the lock is held for a long period of time.

Locks typically require hardware support for efficient implementation. This usually takes the form of one or more atomic instructions such as "test-and-set", "fetch-and-add" or "compare-and-swap". These instructions allow a single process to test if the lock is free, and if free, acquire the lock in a single atomic operation.

Uniprocessor architectures have the option of using uninterruptable sequences of instructions, using special instructions or instruction prefixes to disable interrupts temporarily, but this technique does not work for multiprocessor shared-memory machines. Proper support for locks in a multiprocessor environment can require quite complex hardware and/or software support, with substantial synchronization issues.

The reason an atomic operation is required is because of concurrency, where more than one task executes the same logic. For example, consider the following C code:

```
if (lock == 0) lock = myPID; /* lock free - set it */
```

The above example does not guarantee that the task has the lock, since more than one task can be testing the lock at the same time. Since both tasks will detect that the lock is free, both tasks will attempt to set the lock, not knowing that the other task is also setting the lock. Dekker's or Peterson's algorithm are possible substitutes if atomic locking operations are not available.

Careless use of locks can result in deadlock or livelock. Deadlock occurs when a process holds a lock and then attempts to acquire a second lock. If the second lock is already held by another process, the first process will be blocked. If the second process then attempts to acquire the lock held by the first process, the system has



"deadlocked": no progress will ever be made. A number of strategies can be used to avoid or recover from deadlocks, both at design-time and at run-time. (The most common is to standardize the lock acquisition sequences so that combinations of inter-dependent locks are always acquired and released in a specifically defined "cascade" order.)

## Granularity

Before being introduced to lock granularity, one needs to understand three concepts about locks.

- **lock overhead:** The extra resources for using locks, like the memory space allocated for locks, the CPU time to initialize and destroy locks, and the time for acquiring or releasing locks. The more locks a program uses, the more overhead associated with the usage.
- **lock contention:** This occurs whenever one process or thread attempts to acquire a lock held by another process or thread. The more granular the available locks, the less likely one process/thread will request a lock held by the other. (For example, locking a row rather than the entire table, or locking a cell rather than the entire row.)
- **deadlock:** The situation when each of two tasks is waiting for a lock that the other task holds. Unless something is done, the two tasks will wait forever.

So there is a tradeoff between decreasing lock overhead and decreasing lock contention when choosing the number of locks in synchronization.

An important property of a lock is its **granularity**. The granularity is a measure of the amount of data the lock is protecting. In general, choosing a coarse granularity (a small number of locks, each protecting a large segment of data) results in less **lock overhead** when a single process is accessing the protected data, but worse performance when multiple processes are running concurrently. This is because of increased **lock contention**. The more coarse the lock, the higher the likelihood that the lock will stop an unrelated process from proceeding. Conversely, using a fine granularity (a larger number of locks, each protecting a fairly small amount of data) increases the overhead of the locks themselves but reduces lock contention. More locks also increase the risk of deadlock.

In a database management system, for example, a lock could protect, in order of increasing granularity, part of a field, a field, a record, a data page, or an entire table. Coarse granularity, such as using table locks, tends to give the best performance for a single user, whereas fine granularity, such as record locks, tends to give the best performance for multiple users.

## Database locks

Database locks can be used as a means of ensuring transaction synchronicity. i.e. when making transaction processing concurrent (interleaving transactions), using 2-phased locks ensures that the concurrent execution of the transaction turns out equivalent to some serial ordering of the transaction. However, deadlocks become an unfortunate side-effect of locking in databases. Deadlocks are either prevented by pre-determining the locking order between transactions or are detected using waits-for graphs. An alternate to locking for database synchronicity while avoiding deadlocks involves the use of totally ordered global timestamps.

There are mechanisms employed to manage the actions of multiple concurrent users on a database - the purpose is to prevent lost updates and dirty reads. The two types of locking are Pessimistic and Optimistic Locking.

- **Pessimistic locking:** A user who reads a record, with the intention of updating it, places an exclusive lock on the record to prevent other users from manipulating it. This means no one else can manipulate that record until the user releases the lock. The downside is that users can be locked out for a very long time, thereby slowing the overall system response and causing frustration.
- **Where to use pessimistic locking:** This is mainly used in environments where data-contention (the degree of users request to the database system at any one time) is heavy; where the cost of protecting data through locks is less than the cost of rolling back transactions if concurrency conflicts occur. Pessimistic concurrency is best

implemented when lock times will be short, as in programmatic processing of records. Pessimistic concurrency requires a persistent connection to the database and is not a scalable option when users are interacting with data, because records might be locked for relatively large periods of time. It is not appropriate for use in web application development.

- **Optimistic locking:** this allows multiple concurrent users access to the database whilst the system keeps a copy of the initial-read made by each user. When a user wants to update a record, the application determines whether another user has changed the record since it was last read. The application does this by comparing the initial-read held in memory to the database record to verify any changes made to the record. Any discrepancies between the initial-read and the database record violates concurrency rules and hence causes the system to disregard any update request. An error message is generated and the user is asked to start the update process again. It improves database performance by reducing the amount of locking required, thereby reducing the load on the database server. It works efficiently with tables that require limited updates since no users are locked out. However, some updates may fail. The downside is constant update failures due to high volumes of update requests from multiple concurrent users - it can be frustrating for users.
- **Where to use optimistic locking:** This is appropriate in environments where there is low contention for data, or where read-only access to data is required. Optimistic concurrency is used extensively in .NET to address the needs of mobile and disconnected applications,<sup>[1]</sup> where locking data rows for prolonged periods of time would be infeasible. Also, maintaining record locks requires a persistent connection to the database server, which is not possible in disconnected applications.

## The problems with locks

Lock-based resource protection and thread/process synchronization have many disadvantages:

- They cause blocking, which means some threads/processes have to wait until a lock (or a whole set of locks) is released.
- Lock handling adds overhead for each access to a resource, even when the chances for collision are very rare. (However, any chance for such collisions is a race condition.)
- Locks can be vulnerable to failures and faults that are often very subtle and may be difficult to reproduce reliably. One example is the deadlock. If one thread holding a lock dies, stalls/blocks or goes into any sort of infinite loop, other threads waiting for the lock may wait forever.
- Lock contention limits scalability and adds complexity.
- Balances between lock overhead and contention can be unique to given problem domains (applications) as well as sensitive to design, implementation, and even low-level system architectural changes. These balances may change over the life cycle of any given application/implementation and may entail tremendous changes to update (re-balance).
- Locks are only composable (e.g., managing multiple concurrent locks in order to atomically delete Item X from Table A and insert X into Table B) with relatively elaborate (overhead) software support and perfect adherence by applications programming to rigorous conventions.
- Priority inversion. High priority threads/processes cannot proceed if a low priority thread/process is holding the common lock.
- Convoying. All other threads have to wait if a thread holding a lock is descheduled due to a time-slice interrupt or page fault (*See lock convoy*)
- Hard to debug: Bugs associated with locks are time dependent. They are extremely hard to replicate.
- There must be sufficient resources - exclusively dedicated memory, real or virtual - available for the locking mechanisms to maintain their state information in response to a varying number of contemporaneous invocations, without which the mechanisms will fail, or "crash" bringing down everything depending on them and bringing down the operating region in which they reside. "Failure" is better than crashing, which means a proper locking

mechanism ought to be able to return an "unable to obtain lock for <whatever> reason" status to the critical section in the application, which ought to be able to handle that situation gracefully. The logical design of an application requires these considerations from the very root of conception.

Some people use a concurrency control strategy that doesn't have some or all of these problems. For example, some people use a funnel or serializing tokens, which makes their software immune to the biggest problem -- deadlocks. Other people avoid locks entirely -- using non-blocking synchronization methods, like lock-free programming techniques and transactional memory. However, many of the above disadvantages have analogues with these alternative synchronization methods.

## Language support

Language support for locking depends on the language used:

- There is no API to handle mutexes in the ISO/IEC standards for C or C++. The upcoming revision of the ISO C++ standard, informally known as C++0x, will support threading facilities. The OpenMP standard is supported by some compilers, and this provides critical sections to be specified using pragmas. The POSIX pthread API provides lock support, but its use is not straightforward.<sup>[2]</sup> Visual C++ allows adding the *synchronize* attribute in the code to mark methods that must be synchronized, but this is specific to "COM objects" in the Windows architecture and Visual C++ compiler.<sup>[3]</sup> C and C++ can easily access any native operating system locking features.
- Java provides the keyword *synchronized* to put locks on blocks of code, methods or objects<sup>[4]</sup> and libraries featuring concurrency-safe data structures.
- In the C# programming language, the *lock* keyword can be used to ensure that a thread has exclusive access to a certain resource.
- VB.NET provides a *SyncLock* keyword for the same purpose of C#'s *lock* keyword.
- Python does not provide a lock keyword, but it is possible to use a lower level mutex mechanism to acquire or release a lock.<sup>[5]</sup>
- Ruby also doesn't provide a keyword for synchronization, but it is possible to use an explicit low level mutex object.<sup>[6]</sup>
- In x86 Assembly, the LOCK prefix prevents another processor from doing anything in the middle of certain operations: it guarantees atomicity.
- Objective-C provides the keyword "@synchronized"<sup>[7]</sup> to put locks on blocks of code and also provides the classes NSLock<sup>[8]</sup>, NSRecursiveLock<sup>[9]</sup>, and NSConditionLock<sup>[10]</sup> along with the NSLocking protocol<sup>[11]</sup> for locking as well.
- Ada is probably worth looking at too for a comprehensive overview, with its protected objects<sup>[12]</sup> <sup>[13]</sup> and rendez-vouses.

## References

- [1] "Designing Data Tier Components and Passing Data Through Tiers" (<http://msdn.microsoft.com/en-us/library/ms978496.aspx>). Microsoft. August 2002. . Retrieved 2008-05-30.
- [2] Marshall, Dave (March 1999). "Mutual Exclusion Locks" (<http://www.cs.cf.ac.uk/Dave/C/node31.html#SECTION00311000000000000000>). . Retrieved 2008-05-30.
- [3] "Synchronize" ([http://msdn.microsoft.com/en-us/library/34d2s8k3\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/34d2s8k3(VS.80).aspx)). msdn.microsoft.com. . Retrieved 2008-05-30.
- [4] "Synchronization" (<http://java.sun.com/docs/books/tutorial/essential/concurrency/sync.html>). Sun Microsystems. . Retrieved 2008-05-30.
- [5] Lundh, Fredrik (July 2007). "Thread Synchronization Mechanisms in Python" (<http://effbot.org/zone/thread-synchronization.htm>). . Retrieved 2008-05-30.
- [6] "Programming Ruby: Threads and Processes" ([http://www.ruby-doc.org/docs/ProgrammingRuby/html/tut\\_threads.html](http://www.ruby-doc.org/docs/ProgrammingRuby/html/tut_threads.html)). 2001. . Retrieved 2008-05-30.
- [7] "Apple Threading Reference" (<http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/ObjectiveC/Articles/ocThreading.html>). Apple, inc. . Retrieved 2009-10-17.
- [8] "NSLock Reference" ([http://developer.apple.com/mac/library/documentation/Cocoa/Reference/Foundation/Classes/NSLock\\_Class/Reference/Reference.html](http://developer.apple.com/mac/library/documentation/Cocoa/Reference/Foundation/Classes/NSLock_Class/Reference/Reference.html)). Apple, inc. . Retrieved 2009-10-17.
- [9] "NSRecursiveLock Reference" ([http://developer.apple.com/mac/library/documentation/Cocoa/Reference/Foundation/Classes/NSRecursiveLock\\_Class/Reference/Reference.html](http://developer.apple.com/mac/library/documentation/Cocoa/Reference/Foundation/Classes/NSRecursiveLock_Class/Reference/Reference.html)). Apple, inc. . Retrieved 2009-10-17.
- [10] "NSConditionLock Reference" ([http://developer.apple.com/mac/library/documentation/Cocoa/Reference/Foundation/Classes/NSConditionLock\\_Class/Reference/Reference.html](http://developer.apple.com/mac/library/documentation/Cocoa/Reference/Foundation/Classes/NSConditionLock_Class/Reference/Reference.html)). Apple, inc. . Retrieved 2009-10-17.
- [11] "NSLocking Protocol Reference" ([http://developer.apple.com/mac/library/documentation/Cocoa/Reference/Foundation/Protocols/NSLocking\\_Protocol/Reference/Reference.html](http://developer.apple.com/mac/library/documentation/Cocoa/Reference/Foundation/Protocols/NSLocking_Protocol/Reference/Reference.html)). Apple, inc. . Retrieved 2009-10-17.
- [12] ISO/IEC 8652:2007. "Protected Units and Protected Objects" (<http://www.adaic.com/standards/1zrm/html/RM-9-4.html>). *Ada 2005 Reference Manual*. . Retrieved 2010-02-37. "A protected object provides coordinated access to shared data, through calls on its visible protected operations, which can be protected subprograms or protected entries."
- [13] ISO/IEC 8652:2007. "Example of Tasking and Synchronization" (<http://www.adaic.com/standards/1zrm/html/RM-9-11.html>). *Ada 2005 Reference Manual*. . Retrieved 2010-02-37.

## Monitor (synchronization)

In concurrent programming, a **monitor** is an object or module intended to be used safely by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion. That is, at each point in time, at most one thread may be executing any of its methods. This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared with code that may be executed in parallel.

Monitors also provide a mechanism for threads to temporarily give up exclusive access, in order to wait for some condition to be met, before regaining exclusive access and resuming their task. Monitors also have a mechanism for signaling other threads that such conditions have been met.

Monitors were invented by C. A. R. Hoare <sup>[1]</sup> and Per Brinch Hansen, <sup>[2]</sup> and were first implemented in Brinch Hansen's Concurrent Pascal language.

### Mutual exclusion

As a simple example, consider a monitor for performing transactions on a bank account.

```
monitor class Account {
  private int balance := 0
  invariant balance >= 0

  public method boolean withdraw(int amount)
    precondition amount >= 0
  {
    if balance < amount then return false
```

```

    else { balance := balance - amount ; return true }
}

public method deposit(int amount)
    precondition amount >= 0
{
    balance := balance + amount
}
}

```

While a thread is executing a method of a monitor, it is said to *occupy* the monitor. Monitors are implemented to enforce that *at each point in time, at most one thread may occupy the monitor*. This is the monitor's mutual exclusion property.

Upon calling one of the methods, a thread must wait until no other thread is executing any of the monitor's methods before starting execution of its method. Note that without this mutual exclusion, in the present example, two threads could cause money to be lost or gained for no reason. For example two threads withdrawing 1000 from the account could both return true, while causing the balance to drop by only 1000, as follows: first, both threads fetch the current balance, find it greater than 1000, and subtract 1000 from it; then, both threads store the balance and return.

In a simple implementation, mutual exclusion can be implemented by the compiler equipping each monitor object with a private lock, often in the form of a semaphore. This lock, which is initially unlocked, is locked at the start of each public method, and is unlocked at each return from each public method.

## Waiting and signaling

For many applications, mutual exclusion is not enough. Threads attempting an operation may need to wait until some condition  $P$  holds true. A busy waiting loop

```
while not (  $P$  ) do skip
```

will not work, as mutual exclusion will prevent any other thread from entering the monitor to make the condition true.

The solution is **condition variables**. Conceptually a condition variable is a queue of threads, associated with a monitor, on which a thread may wait for some condition to become true. Thus each condition variable  $c$  is associated with an assertion  $P_c$ . While a thread is waiting on a condition variable, that thread is not considered to occupy the monitor, and so other threads may enter the monitor to change the monitor's state. In most types of monitors, these other threads may signal the condition variable  $c$  to indicate that assertion  $P_c$  is true in the current state.

Thus there are two main operations on condition variables:

- **wait**  $c$  is called by a thread that needs to wait until the assertion  $P_c$  is true before proceeding. While the thread is waiting, it does not occupy the monitor.
- **signal**  $c$  (sometimes written as **notify**  $c$ ) is called by a thread to indicate that the assertion  $P_c$  is true.

As an example, consider a monitor that implements a semaphore. There are methods to increment (V) and to decrement (P) a private integer  $s$ . However, the integer must never be decremented below 0; thus a thread that tries to decrement must wait until the integer is positive. We use a condition variable  $sIsPositive$  with an associated assertion of  $P_{sIsPositive} = (s > 0)$ .

```
monitor class Semaphore
{
    private int s := 0

```

```

invariant s >= 0
private Condition sIsPositive /* associated with s > 0 */

public method P()
{
    if s = 0 then wait sIsPositive
    assert s > 0
    s := s - 1
}

public method V()
{
    s := s + 1
    assert s > 0
    signal sIsPositive
}

```

When a **signal** happens on a condition variable that at least one other thread is waiting on, there are at least two threads that could then occupy the monitor: the thread that signals and any one of the threads that is waiting. In order that at most one thread occupies the monitor at each time, a choice must be made. Two schools of thought exist on how best to resolve this choice. This leads to two kinds of condition variables which will be examined next:

- *Blocking condition variables* or *Signal and Wait* give priority to a signaled thread.
- *Nonblocking condition variables* or *Signal and Continue* give priority to the signaling thread.

### Blocking condition variables

The original proposals by C. A. R. Hoare and Per Brinch Hansen were for *blocking condition variables*. Monitors using blocking condition variables are often called *Hoare style monitors*. With a blocking condition variable, the signaling thread must wait outside the monitor (at least) until the signaled thread relinquishes occupancy of the monitor by either returning or by again waiting on a condition variable.

We assume there are two queues of threads associated with each monitor object

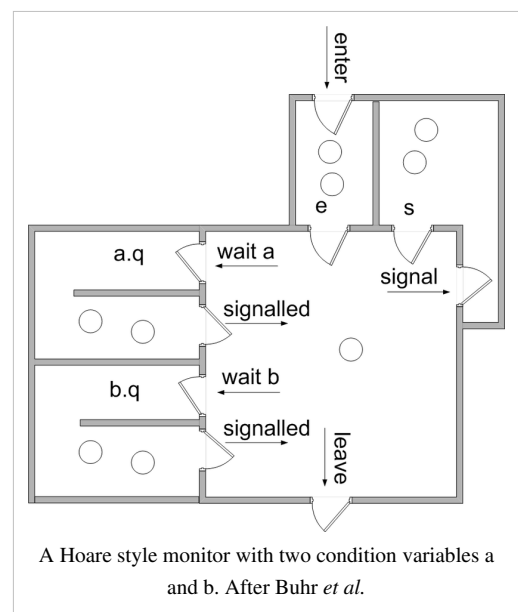
- e is the entrance queue
- s is a queue of threads that have signaled.

In addition we assume that for each condition variable *c*, there is a queue

- *c.q*, which is a queue for threads waiting on condition variable *c*

All queues are typically guaranteed to be fair (i.e. each thread that enters the queue will not be not chosen an infinite number of times citation needed) and, in some implementations, may be guaranteed to be first in first out.

The implementation of each operation is as follows. (We assume that each operation runs in mutual exclusion to the others; thus restarted threads do not begin executing until the operation is complete.)



```

enter the monitor:
    enter the method
    if the monitor is locked
        add this thread to e
        block this thread
    else
        lock the monitor

```

```

leave the monitor:
    schedule
    return from the method

```

```

wait c :
    add this thread to c.q
    schedule
    block this thread

```

```

signal c :
    if there is a thread waiting on c.q
        select and remove one such thread t from c.q
        (t is called "the signaled thread")
        add this thread to s
        restart t
        (so t will occupy the monitor next)
    block this thread

```

```

schedule :
    if there is a thread on s
        select and remove one thread from s and restart it
        (this thread will occupy the monitor next)
    else if there is a thread on e
        select and remove one thread from e and restart it
        (this thread will occupy the monitor next)
    else
        unlock the monitor
        (the monitor will become unoccupied)

```

The schedule routine selects the next thread to occupy the monitor or, in the absence of any candidate threads, unlocks the monitor.

The resulting signaling discipline is known a *"signal and urgent wait,"* as the signaler must wait, but is given priority over threads on the entrance queue. An alternative is *"signal and wait,"* in which there is no s queue and signaler waits on the e queue instead.

Some implementations provide a **signal and return** operation that combines signaling with returning from a procedure.

```

signal c and return :
    if there is a thread waiting on c.q
        select and remove one such thread t from c.q
        (t is called "the signaled thread")

```

```

        restart t
        (so t will occupy the monitor next)
    else
        schedule
    return from the method

```

In either case ("signal and urgent wait" or "signal and wait"), when a condition variable is signaled and there is at least one thread on waiting on the condition variable, the signaling thread hands occupancy over to the signaled thread seamlessly, so that no other thread can gain occupancy in between. If  $P_c$  is true at the start of each **signal**  $c$  operation, it will be true at the end of each **wait**  $c$  operation. This is summarized by the following contracts. In these contracts,  $I$  is the monitor's invariant.

```

enter the monitor:
    postcondition  $I$ 

```

```

leave the monitor:
    precondition  $I$ 

```

```

wait  $c$  :
    precondition  $I$ 
    modifies the state of the monitor
    postcondition  $P_c$  and  $I$ 

```

```

signal  $c$  :
    precondition  $P_c$  and  $I$ 
    modifies the state of the monitor
    postcondition  $I$ 

```

```

signal  $c$  and return :
    precondition  $P_c$  and  $I$ 

```

In these contracts, it is assumed that  $I$  and  $P_c$  do not depend on the contents or lengths of any queues.

(When the condition variable can be queried as to the number of threads waiting on its queue, more sophisticated contracts can be given. For example, a useful pair of contracts, allowing occupancy to be passed without establishing the invariant, is

```

wait  $c$  :
    precondition  $I$ 
    modifies the state of the monitor
    postcondition  $P_c$ 

```

```

signal  $c$ 
    precondition (not empty( $c$ ) and  $P_c$ ) or (empty( $c$ ) and  $I$ )
    modifies the state of the monitor
    postcondition  $I$ 

```

See Howard<sup>[3]</sup> and Buhr *et al.*,<sup>[4]</sup> for more).

It is important to note here that the assertion  $P_c$  is entirely up to the programmer; he or she simply needs to be consistent about what it is.

We conclude this section with an example of a blocking monitor that implements a bounded, thread safe stack.



```
monitor class SharedStack {
  private const capacity := 10
  private int[capacity] A
  private int size := 0
  invariant 0 <= size and size <= capacity
  private BlockingCondition theStackIsNotEmpty /* associated with 0 < size and size <= capacity */
  private BlockingCondition theStackIsNotFull /* associated with 0 <= size and size < capacity */

  public method push(int value)
  {
    if size = capacity then wait theStackIsNotFull
    assert 0 <= size and size < capacity
    A[size] := value ; size := size + 1
    assert 0 < size and size <= capacity
    signal theStackIsNotEmpty and return
  }

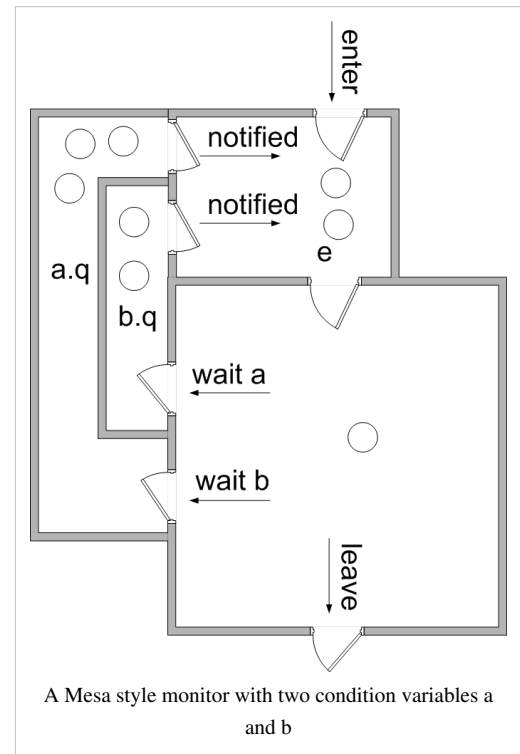
  public method int pop()
  {
    if size = 0 then wait theStackIsNotEmpty
    assert 0 < size and size <= capacity
    size := size - 1 ;
    assert 0 <= size and size < capacity
    signal theStackIsNotFull and return A[size]
  }
}
```

## Nonblocking condition variables

With *nonblocking condition variables* (also called "*Mesa style*" condition variables or "*signal and continue*" condition variables), signaling does not cause the signaling thread to lose occupancy of the monitor. Instead the signaled threads are moved to the e queue. There is no need for the s queue.

With nonblocking condition variables, the **signal** operation is often called **notify** — a terminology we will follow here. It is also common to provide a **notify all** operation that moves all threads waiting on a condition variable to the e queue.

The meaning of various operations are given here. (We assume that each operation runs in mutual exclusion to the others; thus restarted threads do not begin executing until the operation is complete.)



```

enter the monitor:
    enter the method
    if the monitor is locked
        add this thread to e
        block this thread
    else
        lock the monitor

leave the monitor:
    schedule
    return from the method
  
```

```

wait c :
    add this thread to c.q
    schedule
    block this thread
  
```

```

notify c :
    if there is a thread waiting on c.q
        select and remove one thread t from c.q
        (t is called "the notified thread")
        move t to e
  
```

```

notify all c :
    move all threads waiting on c.q to e

schedule :
    if there is a thread on e
        select and remove one thread from e and restart it
    else
        unlock the monitor

```

As a variation on this scheme, the notified thread may be moved to a queue called *w*, which has priority over *e*. See Howard<sup>[5]</sup> and Buhr *et al.*<sup>[6]</sup> for further discussion.

It is possible to associate an assertion  $P_c$  with each condition variable *c* such that  $P_c$  is sure to be true upon return from **wait** *c*. However, one must ensure that  $P_c$  is preserved from the time the **notifying** thread gives up occupancy until the notified thread is selected to re-enter the monitor. Between these times there could be activity by other occupants. Thus it is common for  $P_c$  to simply be *true*.

For this reason, it is usually necessary to enclose each **wait** operation in a loop like this

```

while not ( P ) do wait c

```

where *P* is some condition stronger than  $P_c$ . The operations **notify** *c* and **notify all** *c* are treated as "hints" that *P* may be true for some waiting thread. Every iteration of such a loop past the first represents a lost notification; thus with nonblocking monitors, one must be careful to ensure that too many notifications can not be lost.

As an example of "hinting" consider a bank account in which a withdrawing thread will wait until the account has sufficient funds before proceeding

```

monitor class Account {
    private int balance := 0
    invariant balance >= 0
    private NonblockingCondition balanceMayBeBigEnough

    public method withdraw(int amount)
        precondition amount >= 0
    {
        while balance < amount do wait balanceMayBeBigEnough
        assert balance >= amount
        balance := balance - amount
    }

    public method deposit(int amount)
        precondition amount >= 0
    {
        balance := balance + amount
        notify all balanceMayBeBigEnough
    }
}

```

In this example, the condition being waited for is a function of the amount to be withdrawn, so it is impossible for a depositing thread to *know* that it made such a condition true. It makes sense in this case to allow each waiting thread into the monitor (one at a time) to check if its assertion is true.

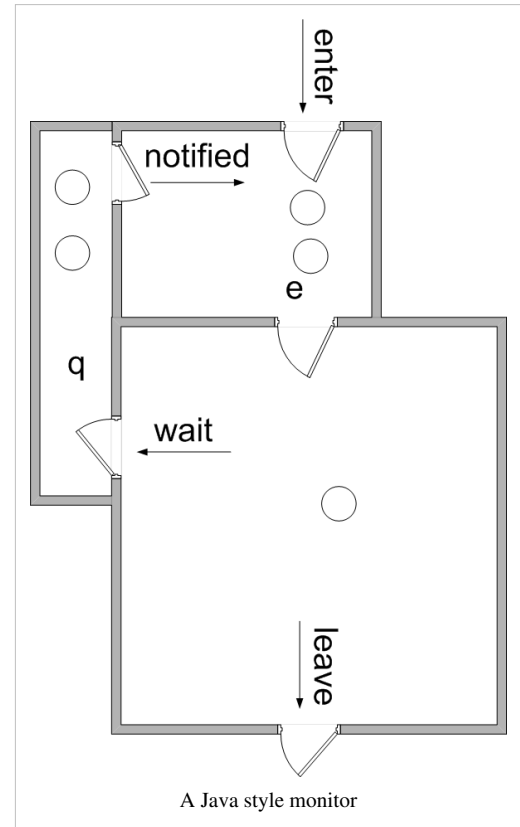
## Implicit condition variable monitors

In the Java language, each object may be used as a monitor. (However, methods that require mutual exclusion must be explicitly marked as **synchronized**.) Rather than having explicit condition variables, each monitor (i.e. object) is equipped with a single wait queue, in addition to its entrance queue. All waiting is done on this single wait queue and all **notify** and **notify all** operations apply to this queue.

This approach has also been adopted in other languages such as C#.

## Implicit signaling

Another approach to signaling is to omit the **signal** operation. Whenever a thread leaves the monitor (by returning or waiting) the assertions of all waiting threads are evaluated until one is found to be true. In such a system, condition variables are not needed, but the assertions must be explicitly coded. The contract for wait is



```
wait P:
    precondition I
    modifies the state of the monitor
    postcondition P and I
```

## History

C. A. R. Hoare and Per Brinch Hansen developed the idea of monitors around 1972, based on earlier ideas of their own and of E. W. Dijkstra.<sup>[7]</sup> Brinch Hansen was the first to implement monitors. Hoare developed the theoretical framework and demonstrated their equivalence to semaphores.

Monitors were soon used to structure inter-process communication in the Solo operating system.

Programming languages that have supported monitors include

- Ada since Ada 95 (as protected objects)
- C# (and other languages that use the .NET Framework)
- Concurrent Euclid
- Concurrent Pascal
- D
- Delphi (Delphi 2009 and above, via TObject.Monitor)
- Java (via the wait and notify keyword)
- Mesa
- Modula-3
- Python (via threading.Condition<sup>[8]</sup> object)
- Ruby

- Squeak Smalltalk
- Turing, Turing+, and Object-Oriented Turing
- $\mu$ C++

A number of libraries have been written that allow monitors to be constructed in languages that do not support them natively. When library calls are used, it is up to the programmer to explicitly mark the start and end of code executed with mutual exclusion. PThreads is one such library.

## Bibliography

- Monitors: an operating system structuring concept, C. A. R. Hoare - Communications of the ACM, v.17 n.10, p. 549-557, Oct. 1974 [9]
- Monitor classification P.A. Buhr, M. Fortier, M.H. Coffin - ACM Computing Surveys, 1995 [10]

## External links

- Java Monitors (lucid explanation) <sup>[11]</sup>
- "Monitors: An Operating System Structuring Concept <sup>[12]</sup>" by C. A. R. Hoare
- "Signalling in Monitors <sup>[13]</sup>" by John H. Howard (computer scientist)
- "Proving Monitors <sup>[14]</sup>" by John H. Howard (computer scientist)
- "Experience with Processes and Monitors in Mesa <sup>[15]</sup>" by Butler W. Lampson and David D. Redell
- pthread\_cond\_wait <sup>[16]</sup> - description from the Open Group Base Specifications Issue 6, IEEE Std 1003.1
- "Block on a Condition Variable <sup>[17]</sup>" by Dave Marshall (computer scientist)
- "Strategies for Implementing POSIX Condition Variables on Win32 <sup>[18]</sup>" by Douglas C. Schmidt and Irfan Pyarali
- Condition Variable Routines <sup>[19]</sup> from the Apache Portable Runtime Library
- wxCondition description <sup>[20]</sup>
- Boost Condition Variables Reference <sup>[21]</sup>
- ZThread Condition Class Reference <sup>[22]</sup>
- Wefts::Condition Class Reference <sup>[23]</sup>
- ACE\_Condition Class Template Reference <sup>[24]</sup>
- QWaitCondition Class Reference <sup>[25]</sup>
- Common C++ Conditional Class Reference <sup>[26]</sup>
- at::ConditionalMutex Class Reference <sup>[27]</sup>
- threads::shared <sup>[28]</sup> - Perl extension for sharing data structures between threads
- Tutorial multiprocessing traps <sup>[29]</sup>
- [http://msdn.microsoft.com/en-us/library/ms682052\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms682052(VS.85).aspx)
- Monitors <sup>[30]</sup> in Visual Prolog.

## Notes

- [1] Hoare, C. A. R. (1974), "Monitors: an operating system structuring concept". *Comm. A.C.M.* **17**(10), 549–57. (<http://doi.acm.org/10.1145/355620.361161>)
- [2] Brinch Hansen, P. (1975). "The programming language Concurrent Pascal". *IEEE Trans. Softw. Eng.* **2** (June), 199–206.
- [3] John Howard (1976), "Signaling in monitors". *Proceedings of the 2nd International Conference on Software Engineering*, 47–52
- [4] Buhr, P.H; Fortier, M., Coffin, M.H. (1995). "Monitor classification". *ACM Computing Surveys (CSUR)* **27**(1). 63–107. (<http://doi.acm.org/10.1145/214037.214100>)
- [5] John Howard (1976), "Signaling in monitors". *Proceedings of the 2nd International Conference on Software Engineering*, 47–52
- [6] Buhr, P.H; Fortier, M., Coffin, M.H. (1995). "Monitor classification". *ACM Computing Surveys (CSUR)* **27**(1). 63–107. (<http://doi.acm.org/10.1145/214037.214100>)
- [7] Brinch Hansen, P. (1993). "Monitors and concurrent Pascal: a personal history", *The second ACM SIGPLAN conference on History of programming languages* 1–35. Also published in *ACM SIGPLAN Notices* **28**(3), March 1993. (<http://doi.acm.org/10.1145/154766.155361>)
- [8] <http://docs.python.org/library/threading.html#condition-objects>
- [9] <http://doi.acm.org/10.1145/355620.361161>
- [10] <http://doi.acm.org/10.1145/214037.214100>
- [11] <http://www.artima.com/insidejvm/ed2/threadsynch.html>
- [12] <http://www.acm.org/classics/feb96/>
- [13] <http://portal.acm.org/citation.cfm?id=807647>
- [14] <http://doi.acm.org/10.1145/360051.360079>
- [15] <http://portal.acm.org/citation.cfm?id=358824>
- [16] [http://www.opengroup.org/onlinepubs/009695399/functions/pthread\\_cond\\_wait.html](http://www.opengroup.org/onlinepubs/009695399/functions/pthread_cond_wait.html)
- [17] <http://gd.tuwien.ac.at/languages/c/programming-dmarshall/node31.html#SECTION00312500000000000000>
- [18] <http://www.cs.wustl.edu/~schmidt/win32-cv-1.html>
- [19] [http://apr.apache.org/docs/apr/group\\_\\_apr\\_\\_thread\\_\\_cond.html](http://apr.apache.org/docs/apr/group__apr__thread__cond.html)
- [20] [http://wxwidgets.org/manuals/2.6.3/wx\\_wxcondition.html](http://wxwidgets.org/manuals/2.6.3/wx_wxcondition.html)
- [21] [http://www.boost.org/doc/html/thread/synchronization.html#thread.synchronization.condvar\\_ref](http://www.boost.org/doc/html/thread/synchronization.html#thread.synchronization.condvar_ref)
- [22] [http://zthread.sourceforge.net/html/classZThread\\_1\\_1Condition.html](http://zthread.sourceforge.net/html/classZThread_1_1Condition.html)
- [23] [http://wefts.sourceforge.net/wefts-apidoc-0.99c/classWefts\\_1\\_1Condition.html](http://wefts.sourceforge.net/wefts-apidoc-0.99c/classWefts_1_1Condition.html)
- [24] [http://www.dre.vanderbilt.edu/Doxygen/Stable/ace/classACE\\_\\_Condition.html](http://www.dre.vanderbilt.edu/Doxygen/Stable/ace/classACE__Condition.html)
- [25] <http://doc.trolltech.com/latest/qwaitcondition.html>
- [26] [http://www.gnu.org/software/commoncpp/docs/refman/html/class\\_conditional.html](http://www.gnu.org/software/commoncpp/docs/refman/html/class_conditional.html)
- [27] [http://austria.sourceforge.net/dox/html/classat\\_1\\_1ConditionalMutex.html](http://austria.sourceforge.net/dox/html/classat_1_1ConditionalMutex.html)
- [28] <http://perldoc.perl.org/threads/shared.html>
- [29] <http://www.asyncop.net/MTnPDirEnum.aspx?treeviewPath=%5bd%5d+Tutorial%5c%5ba%5d+Multiprocessing+Traps+%26+Pitfalls%5c%5bb%5d+Synchronization+API%5c%5bg%5d+Condition+Variables>
- [30] [http://wiki.visual-prolog.com/index.php?title=Language\\_Reference/Monitors](http://wiki.visual-prolog.com/index.php?title=Language_Reference/Monitors)

# Reactor pattern

---

The reactor design pattern is a concurrent programming pattern for handling service requests delivered concurrently to a service handler by one or more inputs. The service handler then demultiplexes the incoming requests and dispatches them synchronously to the associated request handlers.

## Structure

- **Resources:** Any resource that can provide input from or output to the system.
- **Synchronous Event Demultiplexer:** Uses an event loop to block on all resources. When it is possible to start a synchronous operation on a resource without blocking, the demultiplexer sends the resource to the dispatcher.
- **Dispatcher:** Handles registering and unregistering of request handlers. Dispatches resources from the demultiplexer to the associated request handler.
- **Request Handler:** An application defined request handler and its associated resource.

## Properties

All reactor systems are single threaded by definition, but can exist in a multithreaded environment.

## Benefits

The reactor pattern completely separates application specific code from the reactor implementation, which means that application components can be divided into modular, reusable parts. Also, due to the synchronous calling of request handlers, the reactor pattern allows for simple coarse-grain concurrency while not adding the complexity of multiple threads to the system.

## Limitations

The reactor pattern can be more difficult to debug than a procedural pattern due to the inverted flow of control. Also, by only calling request handlers synchronously, the reactor pattern limits maximum concurrency, especially on SMP hardware. The scalability of the reactor pattern is limited not only by calling request handlers synchronously, but also by the demultiplexer. The original Unix select and poll calls for instance have a maximum number of descriptors that may be polled and have performance issues with a high number of descriptors.<sup>[1]</sup> (More recently, more scalable variants of these interfaces have been made available: /dev/poll in Solaris, epoll in Linux and kqueue / kevent in BSD-based systems, allowing the implementation of very high performance systems with large numbers of open descriptors.)

## Implementations

- The ADAPTIVE Communication Environment <sup>[2]</sup> (C++)
  - Apache MINA <sup>[3]</sup> (Java)
  - Apache Cocoon (for XML processing)
  - Boost.Asio <sup>[4]</sup> (C++)
  - JBoss Netty (Java)
  - xSocket <sup>[5]</sup> (Java)
  - AnyEvent <sup>[6]</sup> (Perl)
  - POE (Perl)
  - Twisted (Python)
  - EventMachine <sup>[7]</sup> (Ruby)
  - Node.js (JavaScript)
-

- libowfat<sup>[8]</sup> (C)
- interlace<sup>[9]</sup> (C#)
- after command and Tcl event loop

## References

- [1] Kegel, Dan, *The C10K problem* (<http://www.kegel.com/c10k.html#nb.select>), , retrieved 2007-07-28
- [2] <http://www.cs.wustl.edu/~schmidt/ACE.html>
- [3] <http://mina.apache.org/>
- [4] <http://www.boost.org/>
- [5] <http://xsocket.org/>
- [6] <http://software.schmorp.de/pkg/AnyEvent.html>
- [7] <http://rubyeventmachine.com/>
- [8] <http://www.fefe.de/libowfat/>
- [9] <http://code.google.com/p/interlace/>

## External links

- An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events (<http://www.cs.wustl.edu/~schmidt/PDF/reactor-siemens.pdf>) by Douglas C. Schmidt
- APR Networking & the Reactor Pattern (<http://www.ddj.com/cpp/193101548>)
- Architecture of a Highly Scalable NIO-Based Server (<http://today.java.net/article/2007/02/08/architecture-highly-scalable-nio-based-server>)

# Readers-writer lock

In computer science, a **readers-writer** or **shared-exclusive** lock (also known by the name **multi-reader lock**,<sup>[1]</sup> or by typographical variants such as **readers/writers lock**) is a **synchronization primitive** that solves one of the readers-writers problems. A readers-writer lock is like a mutex, in that it controls access to some shared memory area, but it allows multiple threads to read from the shared area concurrently. Any thread that needs to *write* to the shared memory, of course, needs to acquire an exclusive lock.

One potential problem with a conventional RW lock is that it can lead to write-starvation, meaning that as long as at least one reading thread holds the lock, no writer thread will be able to acquire it. Since multiple reader threads may hold the lock at once, this means that a writer thread may continue waiting for the lock while new reader threads are able to acquire the lock, even to the point where the writer may still be waiting after all of the readers which were holding the lock when it first attempted to acquire it have finished their work in the shared area and released the lock. To avoid writer starvation, a variant on a readers-writer lock can be constructed which prevents any *new* readers from acquiring the lock if there is a writer queued and waiting for the lock, so that the writer will acquire the lock as soon as the readers which were already holding the lock are finished with it. This variation is sometimes known as a "write-preferring" or "write-biased" readers-writer lock.<sup>[2]</sup> <sup>[3]</sup>

Readers-writer locks are usually constructed on top of mutexes and condition variables, or on top of semaphores. They are rarely implemented from scratch.

The read-copy-update (RCU) algorithm is one solution to the readers-writers problem. RCU is wait-free for readers. The Linux-Kernel implements a special solution for few writers called seqlock.

A **read/write lock pattern** or simply **RWL** is a software design pattern that allows concurrent read access to an object but requires exclusive access for write operations.

In this pattern, multiple readers can read the data in parallel but an exclusive lock is needed while writing the data. When a writer is writing the data, readers will be blocked until the writer is finished writing.



Note that operations(either read or write) which you want to allow in parallel should grab the lock in read mode, and operations(either read or write) that you want to be exclusive should grab the lock in write mode.

## Implementations

The current edition of the POSIX standard includes a read-write lock in the form of `pthread_rwlock_t` and the associated operations <sup>[4]</sup>.

Java version 5 or above includes an interface named `java.util.concurrent.locks.ReadWriteLock` that allows the use of this pattern.

A simple Windows API implementation by Glenn Slayden <sup>[5]</sup>.

C# and other .NET languages have `System.Threading.ReaderWriterLockSlim` <sup>[6]</sup>.

Also, the Boost C++ Libraries include a read/write lock in the form of `boost::shared_mutex` <sup>[7]</sup>.

See readers-writers problem for a pseudo-code implementation.

## References

- Scalable Distributed Reader-Writer Mutex <sup>[8]</sup>

[1] "Practical lock-freedom" (<http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-579.pdf>) by Keir Fraser 2004

[2] "ReaderWriterLock Alternative" (<http://www.codeplex.com/ReaderWriterLockAlt>) an open source C# implementation of a write-biased readers-writer lock

[3] `java.util.concurrent.locks.ReentrantReadWriteLock` Java readers-writer lock implementation offers a "fair" mode

[4] [http://www.opengroup.org/onlinepubs/009695399/functions/pthread\\_rwlock\\_init.html](http://www.opengroup.org/onlinepubs/009695399/functions/pthread_rwlock_init.html)

[5] <http://www.glennslayden.com/code/win32/reader-writer-lock>

[6] <http://msdn.microsoft.com/en-us/library/system.threading.readerwriterlockslim.aspx>

[7] [http://www.boost.org/doc/html/thread/synchronization.html#thread.synchronization.mutex\\_types.shared\\_mutex](http://www.boost.org/doc/html/thread/synchronization.html#thread.synchronization.mutex_types.shared_mutex)

[8] <http://www.1024cores.net/home/lock-free-algorithms/reader-writer-problem/distributed-reader-writer-mutex>

---

# Scheduler pattern

In computer programming, the **scheduler pattern** is a software design pattern. It is a concurrency pattern used to explicitly control when threads may execute single-threaded code, like write operation to a file.

The scheduler pattern uses an object that explicitly sequences waiting threads. It provides a mechanism to implement a scheduling policy, but is independent of any specific scheduling policy — the policy is encapsulated in its own class and is reusable.

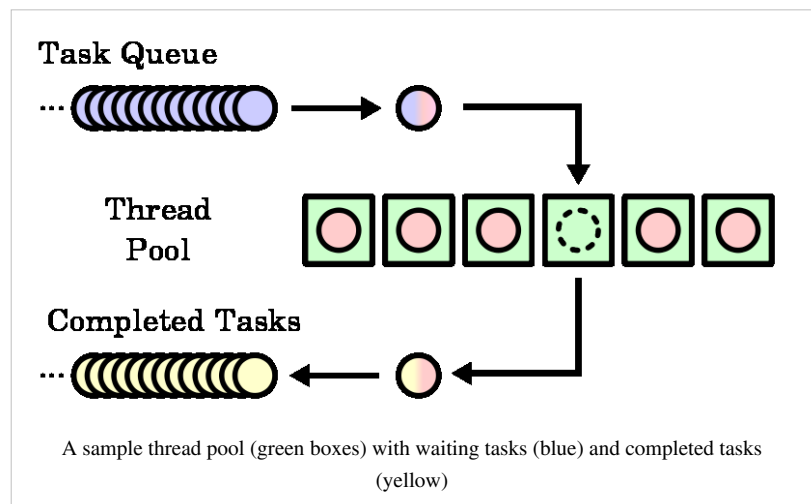
The read/write lock pattern is usually implemented using the scheduler pattern to ensure fairness in scheduling.

Note that the scheduler pattern adds significant overhead beyond that required to call a synchronized method.

The scheduler pattern is not quite the same as the scheduled-task pattern used for real-time systems.

# Thread pool pattern

In computer programming, the **thread pool pattern** is where a number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads. As soon as a thread completes its task, it will request the next task from the queue until all tasks have been completed. The thread can then terminate, or sleep until there are new tasks available.



The number of threads used is a parameter that can be tuned to provide the best performance. Additionally, the number of threads can be dynamic based on the number of waiting tasks. For example, a web server can add threads if numerous web page requests come in and can remove threads when those requests taper down. The cost of having a larger thread pool is increased resource usage. The algorithm used to determine when to create or destroy threads will have an impact on the overall performance:

- create too many threads, and resources are wasted and time also wasted creating any unused threads
- destroy too many threads and more time will be spent later creating them again
- creating threads too slowly might result in poor client performance (long wait times)
- destroying threads too slowly may starve other processes of resources

The algorithm chosen will depend on the problem and the expected usage patterns.

If the number of tasks is very large, then creating a thread for each one may be impractical.

Another advantage of using a thread pool over creating a new thread for each task is thread creation and destruction overhead is negated, which may result in better performance and better system stability. Creating and destroying a thread and its associated resources is an expensive process in terms of time. An excessive number of threads will also waste memory, and context-switching between the runnable threads also damages performance. For example, a socket connection to another machine—which might take thousands (or even millions) of cycles to drop and re-establish—can be avoided by associating it with a thread which lives over the course of more than one transaction.

When implementing this pattern, the programmer should ensure thread-safety of the queue.

Typically, a thread pool executes on a single computer. However, thread pools are conceptually related to server farms in which a master process distributes tasks to worker processes on different computers, in order to increase the overall throughput. Embarrassingly parallel problems are highly amenable to this approach.

## External links

- Article "Query by Slice, Parallel Execute, and Join: A Thread Pool Pattern in Java <sup>[1]</sup>" by Binildas C. A.
- Article "Thread pools and work queues <sup>[2]</sup>" by Brian Goetz
- Article "A Method of Worker Thread Pooling <sup>[3]</sup>" by Pradeep Kumar Sahu
- Article "Work Queue <sup>[4]</sup>" by Uri Twig
- Article "Windows Thread Pooling and Execution Chaining <sup>[5]</sup>"
- Article "Smart Thread Pool <sup>[6]</sup>" by Ami Bar
- Article "Programming the Thread Pool in the .NET Framework <sup>[7]</sup>" by David Carmona
- Article "The Thread Pool and Asynchronous Methods <sup>[8]</sup>" by Jon Skeet
- Article "Creating a Notifying Blocking Thread Pool in Java <sup>[9]</sup>" by Amir Kirsh
- Article "Practical Threaded Programming with Python: Thread Pools and Queues <sup>[10]</sup>" by Noah Gift
- Paper "Optimizing Thread-Pool Strategies for Real-Time CORBA <sup>[11]</sup>" by Irfan Pyarali, Marina Spivak, Douglas C. Schmidt and Ron Cytron
- Conference Paper "Deferred cancellation. A behavioral pattern <sup>[12]</sup>" by Philipp Bachmann

## References

- [1] <http://today.java.net/pub/a/today/2008/01/31/query-by-slice-parallel-execute-join-thread-pool-pattern.html>
- [2] <http://www.ibm.com/developerworks/java/library/j-jtp0730.html>
- [3] [http://www.codeproject.com/threads/thread\\_pooling.asp](http://www.codeproject.com/threads/thread_pooling.asp)
- [4] [http://codeproject.com/threads/work\\_queue.asp](http://codeproject.com/threads/work_queue.asp)
- [5] <http://codeproject.com/threads/Joshthreadpool.asp>
- [6] <http://www.codeproject.com/KB/threads/smartthreadpool.aspx>
- [7] <http://msdn.microsoft.com/en-us/library/ms973903.aspx>
- [8] <http://www.yoda.arachsys.com/csharp/threads/threadpool.shtml>
- [9] <http://java.net/pub/a/today/2008/10/23/creating-a-notifying-blocking-thread-pool-executor.html>
- [10] <http://www.ibm.com/developerworks/aix/library/au-threadingpython/>
- [11] <http://www.cs.wustl.edu/~schmidt/PDF/OM-01.pdf>
- [12] <http://doi.acm.org/10.1145/1753196.1753218>

# Thread-local storage

---

**Thread-local storage** (TLS) is a computer programming method that uses static or global memory local to a thread.

This is sometimes needed because normally all threads in a process share the same address space, which is sometimes undesirable. In other words, data in a static or global variable is normally always located at the same memory location, when referred to by threads from the same process. Variables on the stack however are local to threads, because each thread has its own stack, residing in a different memory location.

Sometimes it is desirable that two threads referring to the same static or global variable are actually referring to different memory locations, thereby making the variable thread local, a canonical example being the C error code variable `errno`.

If it is possible to make at least a memory address sized variable thread local, it is in principle possible to make arbitrarily sized memory blocks thread local, by allocating such a memory block and storing the memory address of that block in a thread local variable.

## Windows implementation

The API function **TlsAlloc** can be used to obtain an unused *TLS slot index*; the *TLS slot index* will then be considered 'used'.

The **TlsGetValue** and **TlsSetValue** functions can then be used to read and write a memory address to a thread local variable identified by the *TLS slot index*. **TlsSetValue** can only affect the variable for the current thread.

The **TlsFree** function can be called to release the *TLS slot index*; the index will then be considered 'unused' and a new call to **TlsAlloc** can return it again.

## Pthreads implementation

TLS with POSIX threads (*Thread-Specific Data* in Pthreads nomenclature) is similar to **TlsAlloc** and related functionality for Windows. **pthread\_key\_create** creates a *key*, with an optional *destructor*, that can later be associated with thread specific data via **pthread\_setspecific**. The data can be retrieved using **pthread\_getspecific**. If the thread specific value is not *NULL*, the *destructor* will be called when the thread exits. Additionally, *key* must be destroyed with **pthread\_key\_delete**.

## Language-specific implementation

Apart from relying on programmers to call the appropriate API functions, it is also possible to extend the programming language to support TLS.

### Object Pascal

In Delphi or Free Pascal you can use the 'threadvar' reserved keyword instead of 'var' to declare variables using the thread-local storage.

```
var
    mydata_process: integer;
threadvar
    mydata_threadlocal: integer;
```

## Java

In Java thread local variables are implemented by the `ThreadLocal` class object. `ThreadLocal` holds variable of type `T`, which is accessible via `get/set` methods. For example `ThreadLocal` variable holding `Integer` value looks like this:

```
private static ThreadLocal<Integer> myThreadLocalInteger = new ThreadLocal<Integer>();
```

## Sun Studio C/C++, IBM XL C/C++, GNU C and Intel C/C++ (Linux systems)

The keyword `__thread` is used like this:

```
__thread int number;
```

- `__thread` defines *number* to be a thread local variable.

## Visual C++, Intel C/C++ (Windows systems)

In Visual C++ the keywords `declspec(thread)` are used like this:

```
__declspec(thread) int number;
```

- `__declspec(thread)` defines *number* to be a thread local variable.
- On operating systems prior to Vista and Server 2008 `__declspec(thread)` works in DLLs only when those DLLs are bound to the executable, and will *not* work for those loaded with `LoadLibrary()` (a protection fault or data corruption may occur).
- There are additional rules: "Rules and Limitations for TLS" <sup>[1]</sup> in MSDN.

## Digital Mars C++

In Digital Mars C++ the keywords `declspec(thread)` are used like this:

```
__declspec(thread) int number;
```

- `__declspec(thread)` defines *number* to be a thread local variable.

## D

In the D programming language version 2, all static and global variables are thread-local by default and are declared with syntax similar to "normal" global and static variables in other languages. Regular global variables must be explicitly requested using the `__gshared` keyword:

```
int threadLocal; // This is a thread local variable.
__gshared int global; // This is a plain old global variable.
```

## Borland C++ Builder

In Borland C++ Builder the keywords `__declspec(thread)` are used like this:

```
__declspec(thread) int number;
```

the same in a more elegant way:

```
int __thread number;
```

- `__declspec(thread)` defines *number* to be a thread local variable. `__thread` is a synonym for `__declspec(thread)`.

## GCC

GCC C/C++ implements `__thread` as above.

The initialiser must be a compile-time constant, even in C++. E.g.

```
__thread int number = 1;
```

but not

```
void f(int number)
{
    static __thread int number_copy = number;
```

or (C++)

```
__thread int number = calculate_number();
```

## C# and other .NET languages

Static fields can be marked with the `ThreadStatic` attribute <sup>[2]</sup>:

```
class FooBar
{
    [ThreadStatic] static int foo;
}
```

In .NET 4.0 the `System.Threading.ThreadLocal<T>` <sup>[3]</sup> class is available for allocating and lazily loading thread local variables.

Also an API <sup>[4]</sup> is available for dynamically allocating thread local variables.

## Python

In Python version 2.4 or later **local** class in **threading** module can be used to create thread-local storage.

```
import threading
mydata = threading.local()
mydata.x = 1
```

## Ruby

In Ruby thread local variables can be created/accessed using `[]=`/`[]` methods.

```
Thread.current[:user_id] = 1
```

## Perl

In Perl threads were added late in the evolution of the language, after a large body of existing code was already present on the Comprehensive Perl Archive Network. As a result, threads in Perl by default take their own local storage for all variables, to minimise the impact of threads on existing non-thread-aware code. In Perl, a thread-shared variable can be created using an attribute:

```
use threads;
use threads::shared;

my $localvar;
my $sharedvar :shared;
```

## Underlying implementation

The above discussion indicates what interface a programmer uses to obtain thread-local storage, but not how this works behind the scenes. The underlying problem is that, since all threads share an address space, no fixed memory location can be used to store the location of the storage. The following discussion applies to Microsoft Windows-based systems, but similar models may be applicable to other systems.

In Windows, the thread-local storage is access via a table. (Actually, two tables, but they appear as one[5]). `TlsAlloc` returns an index to this table, unique per address space, for each call. Each thread has its own copy of the thread-local storage table. Hence, each thread can independently use `TlsSetValue(index)` and obtain the same value via `TlsGetValue(index)`, because these set and look up an entry in the thread's own table. Only a single pointer is stored; any Windows system which offers more than one pointer of storage is either allocating multiple values or, more likely, obtaining storage from heap or stack and storing that in the pointer.

This leaves the question of how a per-thread table is to be found. In fact, each thread must have unique information stored about it somewhere, in a way that the system can easily look it up. In Windows, there is a Thread Information Block for each thread. One of the entries in this block is the thread-local storage table for that thread[6]. In Windows systems, the address of the Thread Information Block is stored in the FS register. Each thread runs on a different processor, and maintains its own separate set of registers.

In this way, access to thread-local storage carries a minimal overhead.

## External links

- ELF Handling For Thread-Local Storage <sup>[7]</sup> — Document about an implementation in C or C++.
- `ACE_TSS< TYPE >` Class Template Reference <sup>[8]</sup>
- `RWTThreadLocal<Type>` Class Template Documentation <sup>[9]</sup>
- Article "Use Thread Local Storage to Pass Thread Specific Data <sup>[10]</sup>" by Doug Doedens
- "Thread-Local Storage <sup>[11]</sup>" by Lawrence Crowl
- "Developer's Reference <sup>[12]</sup>"
- Article "It's Not Always Nice To Share <sup>[13]</sup>" by Walter Bright
- Practical ThreadLocal usage in Java: <http://blogs.captchconsulting.com/blog/balaji-muthuvarathan/persistence-pattern-using-threadlocal-and-ejb-interceptors>

## References

- [1] <http://msdn2.microsoft.com/en-us/library/2s9wt68x.aspx>
- [2] [http://msdn2.microsoft.com/en-us/library/system.threadstaticattribute\(vs.80\).aspx](http://msdn2.microsoft.com/en-us/library/system.threadstaticattribute(vs.80).aspx)
- [3] <http://msdn.microsoft.com/en-us/library/dd642243.aspx>
- [4] [http://msdn2.microsoft.com/en-us/library/system.threading.thread.getnameddataslot\(vs.80\).aspx](http://msdn2.microsoft.com/en-us/library/system.threading.thread.getnameddataslot(vs.80).aspx)
- [5] <http://www.nynaeve.net/?p=181>
- [6] <http://www.microsoft.com/msj/archive/s2ce.aspx>
- [7] <http://www.akkadia.org/drepper/tls.pdf>
- [8] [http://www.dre.vanderbilt.edu/Doxygen/Stable/ace/classACE\\_\\_TSS.html#\\_details](http://www.dre.vanderbilt.edu/Doxygen/Stable/ace/classACE__TSS.html#_details)
- [9] <http://www.roguewave.com/support/docs/hppdocs/thrref/rwtthreadlocal.html#sec4>
- [10] <http://www.c-sharpcorner.com/UploadFile/ddoedens/UseThreadLocals11212005053901AM/UseThreadLocals.aspx>
- [11] <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1966.html>
- [12] <http://www.asyncop.net/Link.aspx?TLS>
- [13] <http://www.ddj.com/cpp/217600495>

# Article Sources and Contributors

**Design pattern (computer science)** *Source:* <http://en.wikipedia.org/w/index.php?oldid=414522385> *Contributors:* 2mcm, 4th-otaku, A. B., ACupOfCoffee, Abelson, AdamBradley, Adaniels.nl, Adjusting, Ahoeresteimer, Ahy1, Airplaneman, Albanaco, AlecMcEachran, Alex.ryzantsev, Alexandernolasco, Allan McInnes, Amazedsaint, Amitch, Anaraug, Andreas Kaufmann, Antonielli, Arch4ngel, Architectureblog, Asphatasawhale, AtheWeatherman, Azbarcea, BD2412, Beland, BenFrantzDale, Benandorsqueaks, BenjaminGittins, Bevo, Bkkrad, BluePlateSpecial, Bogdanmata, Boothy443, Borgx, Bovlb, Bravegag, Breandandaltan, Burschik, Bwoolf, Connett, Cesar Moura, Charliearcuri, Chrislk02, Clubmarx, Coolestdude1, Courcelles, Craig Stuntz, Crdoconnor, Cronian, Cyc, CzarB, DJ Clayworth, Davhdavh, Davhorn, Demonkoryu, Dethomas, Dirk Riehle, Dmyersturnbull, Eaeftremov, Ed Brey, Edenphd, Edward, Edward Z. Yang, Eightcien, El C, EngineerScotty, Enochlau, Enviroboy, Errandir, Evoluzion, FF2010, Fiskah, Forseti, Francois Trazzi, Frau Holle, Fred Bauder, Fred Bradstadt, Fredrik, FreplySpang, Fresheneesz, Fubar Obfusco, Furrykef, Fuzheado, Galoubet, Gary King, Georgenaing, Gimme danger, Gioto, Gnewf, Graue, GregorB, Grubber, Guehene, Gwyddgwyrd, Hairy Dude, Hans Adler, Harborsparrow, Haskellguy, Hede2000, HenkvD, Hirzel, Ibbbi, J.delanoy, JLaTondre, JMSwtik, JWB, Jackhwl, Jafeluv, Jamilsh, Janesconference, Jeff3000, Jerryobject, Jesperborgstrup, Jizzbug, Jleedev, Jonon, Jopincar, Jorend, Josevellezcaldas, Julesd, JustinWick, Katalaveno, Kevin Saff, Khaderv, Khalid, Khalid hassani, Kimmelb, Kirby2010, Kku, Lambyuk, Lancevortex, Landon1980, Lantzilla, LeaveSleaves, Legare, Leibniz, Lelkesa, Liao, Ligulem, Ltruett, Luna Santin, M4gnum0n, Mahanga, Mamouri, Marcoscata, Marek69, Maria C Mosak, MarkSutton, MartinSpamer, Marudubshinki, Mathewdjb, Maximaximax, McSly, Mcavigelli, Mdd, Mdd4696, Melah Hashamaim, Meredyth, Michael Hardy, Michal Jurosz, Mikaea, Mistercupcake, Mohansn, Mormat, Morten Johnsen, Mountain, Msjegan, Msm, Natkeeran, Necklace, Neelix, Neilk, NickShaforstoff, Nixdorf, Nklatt, Oicumayberight, Olff, Orborde, Parseboy, Patrick McDonough, Pcap, Pcb21, Pcwitchid, Perey, Peter S., Pieleric, PierreCaboche, Plasticup, Polonium, Prashanthan, PrologFan, PsyberS, Ptrb, Quagmire, Quuxplusone, RHaworth, RabbiGregory, Realbox, RedWolf, Redeyed Treefrog, Renesis, RickBeton, Rodasmi, Ronz, RossPatterson, Rudyment, Ruidjah, Rwmcfal, S.K., SAE1962, Saiful vonair, Salix alba, Sandman25 again, Seanhan, Sfermigier, Shabbirbhmani, Skorpion87, Skraedingie, SlavMFM, Spdegabrielle, SpeedyGonsales, Stachelfisch, Stephanakib, Stuartroebeck, Sudhanshuss, TakuyaMurata, Tesakoff, Tgmattso, TheJames, Theuser, Thistle172, Tiggeerjay, Tiktuk, TingChong Ma, Tonyfaull, Tonymarston, Torg, Treekids, Tristan ph, TuukkaH, Umbr, Uncle G, Unixer, Vald, VidGa, Vinsci, Vmlaker, WOT, Wapcaplet, Wavelength, Welsh, Wernher, Whiner01, WikiLeon, Wikidrone, Wo.luren, Xypron, Zeno Gantner, Zondor, 619 anonymous edits

**Abstract factory pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=413403267> *Contributors:* A5b, AJCham, Abednigo, Alx359, Anthony Appleyard, Balkrishnatele, Bdean42, BenFrantzDale, Benno, Bigscoop, Bluemoose, Boleslav Bobcik, CanisRufus, Cybercobra, Dan Gluck, Daniel.Cardenas, David.nussio, Doetzee, Dgquintas, Drable, Druckles, Eaeftremov, Eastlaw, Edenphd, Ekamaloff, Enric Naval, Epr123, Evil saltine, Flammifer, Flash200, Fred Bradstadt, Frór, Furrykef, Gilliam, GregorB, Gurjinder.rathore, HalfShadow, Hao2lian, Hephaestos, Hetar, Hu12, Imdonatello, Int19h, Jay, Jay.233, Jonathan Hall, Joriki, JoshHolloway, Jsgoupil, Kallikanzarid, Krzyp, Lukas.gosik, MER-C, Magister Mathematicae, Magnub, Mahanga, Manop, Merutak, Michael Slone, Minimac, Mormat, Msjegan, Msthl, Mtsic, Naasking, Nitinpa2000, Nkochar, Oddity-, Pcb21, Pelister, Peterl, Phresnel, Piano non troppo, Pikiwyn, QuackGuru, Quamaretto, R'n'B, RainbowOLight, RedWolf, Reyjose, Ronabop, Rtushar, Rupendradhillon, Scray, Sdillman, SergejDergatsjev, Shadowjams, Sietse Snel, Sir Nicholas de Mimsy-Porington, Skittleys, Solra Bizna, Stephenb, Superborsuk, Tabletop, TakuyaMurata, The Anome, TheMandarin, Thomas Linder Puls, Tobias Bergemann, Tobiasly, Tomwalden, Topdeck, Treutwein, Tronster, Twxs, Venturor, Vghuston, Vikasgoyal77, Vinhphunguyen, Vinod.pahuja, Vvarkey, WikiJonathanpeter, Wikidudeman, Wikieditor06, Wilfried Loche, Woohookitty, Www.koeberl.priv.at, Yonghuster, YurfDog, Zixyer, 313 anonymous edits

**Builder pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=413403401> *Contributors:* Abednigo, Aesopos, Allan McInnes, AlphaFoobar, Aneeshpu, Anthony Appleyard, Arichnad, Beland, BenFrantzDale, BennyD, Bensaccount, Betacommand, Blankfrack, Brooklyn1, Capt. James T. Kirk, Chiffle, Cmcginnis, Coldboyqn, Cybercobra, Cyrius, Dreadstar, Drmies, Dyker, Edward, Emcmanus, Ezraj, Finlay McHardy, Fred Bradstadt, Geh, Glylvyn, Hairy Dude, Hao2lian, Hephaestos, Hu12, Immyhand77, JForget, JMatthews, Janpolowinski, Jarsyl, JavierMC, Jay, Jesus Escaped, Khalid hassani, Khmerang, Lathspell, Lramos, Lukas.gosik, MER-C, Madoka, Marcusmax, Mikethegreen, Mital d vora, Mormat, Msjegan, Mtsic, Mzedeler, Neoguru12, Nijelprim, Nitinpa2000, Odie5533, Oxymoron83, PeaceNT, Pelister, Per Abrahamson, RedWolf, Rimian, Rossami, Salazard1, Satyr9, Sedziwoj, Shell Kinney, Sligocki, SonOfNothing, Ssimsekler, Supersid01, TakuyaMurata, The Thing That Should Not Be, Thomas Linder Puls, Toteb5, Trashtoy, Umreemala, Uncle G, Vghuston, Vikasgoyal77, Vinod.pahuja, West Brom 4ever, Wik, Yintan, 176 anonymous edits

**Factory method pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=414475584> *Contributors:* 9thbit, Abednigo, Acm, Ain32, Aisaac, Alainr345, Alquantor, Alvin-cs, AmunRa84, Andreas Kaufmann, Andreas Toth, Anthony Appleyard, Apanait, Architectureblog, ArthurDenture, Arturs Licis, Avono, Bearcat, Beland, BenFrantzDale, Bigboojum, Blomskoe, BlueTaRaKa, Bouchecl, Bpssoft, BrokenSegue, Cartoro, Charles Moss, Chaser, Courcelles, Decltype, Dmprantz, Dreflymac, Duplicity, Edenphd, Edward.in.Edmonton, Enselic, Esben, Faradayplank, FellGleaming, Fredrik, Galestar, Gman124, Gnyus, Gwern, Haakon, Hadal, Hairy Dude, Hao2lian, Hemanshu, Hu12, Jamitzky, Jay, Jeffrey Smith, Jkl, Jobriath, Jogle, Joswig, Kefka, KellyCoinGuy, Khalid hassani, Khmer42, Khoango, Khrom, Kkolb, Kshivakumar, Leo Cavalcante, Lexsoft, Liao, Little3lue, LtPowers, MER-C, Mahanga, Marco.panichi, Matthierner, Matthew0028, Mbunyard, Mhtzdt, Michael Slone, Mjworthey, Mormat, Msjegan, Mtsic, NTK, Ncmvocalist, Neshatian, Nforbes, Nielses, Nowforever, OnixWP, OrlinKolev, Paradrop, Pelister, Petur Runolfsson, Philip Trueman, Qwertyus, R'n'B, Rajah9, RedWolf, Redroof, Reyjose, Rjmooney, Rjwilmsi, SergejDergatsjev, Sesse, Slaughter, Ssimsekler, Stannered, Starsareblueandfaraway, TakuyaMurata, Thadgudiry, That Guy, From That Show!, Toddst1, Tommy2010, Tormit, Umreemala, Vald, Vghuston, Vianello, Vikasgoyal77, Vipinhari, Westcoastr13, Will2k, Wlievens, Yamamoto Ichiro, Zch051383471952, Zero4573, 321 anonymous edits

**Lazy initialization** *Source:* <http://en.wikipedia.org/w/index.php?oldid=411785264> *Contributors:* Abednigo, Alexey Luchkovsky, Angilly, Anthony Appleyard, Bilbo1507, Blankfrack, BluePlateSpecial, Britannica, Derek Ross, Doug Bell, Ed Poor, Enochlau, JLaTondre, Jerryobject, JonHarder, Lumning, MASQUERAID, Maximaximax, Michael Hardy, Msjegan, Mtsic, Phresnel, Quagmire, RedWolf, RepeatUntil, Rheun, Rhomboid, Spock lone wolf, Sundar2000, Tackline, TheIncredibleEdibleOompaLoompa, Tormit, VKokielov, YordanGeorgiev, Ztyx, 54 anonymous edits

**Multiton pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=410401992> *Contributors:* .digamma, Abednigo, Beland, Bmord, CliffHall, DanielcWiki, Destynova, Dubai, Ettrig, Galestar, Gang65, Inumedia, Khalid hassani, Mtsic, Pavel Vozenilek, Ranjit.sail, Rentzepopoulos, Ros Original, Spoon!, Steve R Barnes, Viy75, Whiner01, Wookietreiber, 28 anonymous edits

**Object pool pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=390480764> *Contributors:* Andreas Kaufmann, Anthony Appleyard, Arch4ngel, Arunkd13, Audriusa, CesarB, Christoffar, Hairy Dude, Hectorpal, Jkl, Juansempere, Mahanga, Marudubshinki, Moala, Msjegan, Munificent, Noodlez84, PJTrail, Pearle, Prari, Purplefeltangel, Qxz, Ruijoel, Saigyo, Tenbaset, Vinod.pahuja, Vorburger, WikHead, 41 anonymous edits

**Prototype pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=408767965> *Contributors:* Abednigo, Anthony Appleyard, Arturs Licis, Ary29, BenFrantzDale, BluePlateSpecial, Brighterorange, Btx40, Can't sleep, clown will eat me, Carlif, Catch ru, Ceciop, Cheungpat, Cmdjameson, Colonies Chris, Cygon, Der Hakawati, Doug Bell, Eaeftremov, Enric Naval, Errandir, Frecklefoot, Ghettoabaster, Gsp, Hao2lian, Hofoen, Hu12, Io41, J.delanoy, Jengelh, Jianghan, Lathspell, Leszek Jańczuk, Liao, Lookforajt, Loren.wilton, Lukas.gosik, Luis Felipe Braga, M2Ys4U, MER-C, Materialscientist, Michael Devore, Mital d vora, Modster, Mokru, Mormat, Mtsic, PJonDevelopment, Patrickdepinguin, RedWolf, Salazard1, SergejDergatsjev, Stalker134314, TakuyaMurata, TheMandarin, TheParanoidOne, Vghuston, Victorwss, Vikasgoyal77, VladM, XcepticZP, Zarkonnen, Zerokitsune, 149 anonymous edits

**Singleton pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=412903240> *Contributors:* 1qaz-pl, 5 albert square, A Quest For Knowledge, Abednigo, Abhi beckert, Aitias, Aka042, Aks.abhishek, Aleenfl, AndrewVeresov, Anthony Appleyard, Appraiser, Ascänder, Askso, Athlan, Autarch, Beachy, Beland, BenFrantzDale, Bestsss, Bibhatirpathi, Bigbdman, Blankfrack, Bnedelko, BrokenSegue, Bruno.schmidt, Bytebear, C. A. Russell, Caefar, Captchad, Carlopries, Chowbok, Chris the speller, Christian75, Clarkcox3, Craigfis, Crashopper, Cryptophile, Cusp98007, Cyril.wack, DARTH SIDIOUS 2, DaSjieb, DabMachine, Dagostinelli, Derek Ross, DocSigma, Domeika, Doubleplusjeff, Doug Bell, Drmgvry, Dubai, Eduardo.Farias, Educres, Enchanter, Ethaniel, Ethridgela, Ficklephil, Freddy1990, Func, FuzzyBunny, Ghettoabaster, GoCool, Goofyheadepunk, Gorpik, Greg searle, Hao2lian, Hogswild5, Hpsgoj00, Hu12, IanLewis, Innocenceisdeath, Int19h, Inumedia, Isfpoeet, Ixf6d4, Jago84, Jakese, Jamie, Jan Nielsen, Javito554, Jbcc, Jdcook72, Jeugeorge, Jezmck, Jfromcanada, Jleedev, Joswig, Jsgoupil, Jitley, Juanmaiz, Juansempere, Jutiphon, Kate, KeithPinson, Khalid hassani, Khatru2, Kwiki, Lettsurf, LordK, Lordmetroid, Lukas.gosik, Lukebayes, Lulu of the Lotus-Eaters, Lunboks, Luis Felipe Braga, MBisanz, MER-C, MFNickster, Mancini, MarSch, Marty.neal, Marudubshinki, McOsten, Mcease, Mentifisto, Michael Hardy, Michael miceli, Miknight, MisterHand, Mjharrison, Mobile Snail, Mormat, Mountain, Mpradeep, MrWiseGuy, Msjegan, Mtsic, Muxd77, Mydaz, NathanBeach, NeoPhi, Neoguru12, Nitram cero, NormBo, Omicronpersei8, Omnicog, OmveerChaudhary, PI314r, Paralyz3, Paulgear, Pavel Vozenilek, Pcap, Pcb21, Phil Boswell, Piano non troppo, Pnm, Polymerbringer, Pomakis, Ptero, Python eggs, Qc, Quuxplusone, RFST, Rakeshnayak, Ravish9507, Ray Van De Walker, RedWolf, Richard Fusion, Rider kabuto, RossPatterson, Ruaah, Ryan1101, Serkan Kenar, Shadowjams, Shi Hou, Silly rabbit, Simeon H, Sneffel, Spikex, Spoon!, Srittau, Stalker314314, Stwalkerster, Swiftcoder, TakuyaMurata, Tartley, Tarun.Varshney, Taw, Tecle.samsom, Tfischer, TheMandarin, TheRealFennShysa, ThingXYZ, Thomas Kist, Timwi, Tobias Bergemann, Tomerfiliba, Tormit, Trashtoy, Travisowens@hotmail.com, Trimmch, Ultra two, Vazde, Vernicht, Vghuston, Vikasgoyal77, Vina, VladM, Voyaging, Vssun, Wbstory, Weirddemon, Who, Wikidrone, Wingbullet, Winterheat, Xiluwu, Yjixiao, Yukoba, Yurik, Zhen Lin, Ziupo, Zwetan, 608 anonymous edits

**Structural pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=358175340> *Contributors:* Beland, Greedyhalibut, Iridescent, Jeff3000, Jijinhon, Khalid, LazyEditor, Mahanga, Moe Epsilon, Mormat, MrGrigg, Piet Delpont, RedWolf, Rsathish, Ruud Koot, S, Siggimund, StephenWeber, TheParanoidOne, 33 anonymous edits

**Adapter pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=407045339> *Contributors:* Abdull, Abednigo, Alansohn, Alexius08, Andre Engels, Anonymous2420, Anthony Appleyard, Architectureblog, BenFrantzDale, Blytkerchan, Bravegag, BryanG, CannonBallGuy, CatherineMunro, Cyril.wack, Doug Bell, Edenphd, Fred Bradstadt, Giantbicycle, GilHamilton, Hao2lian, Hardywang, Hu12, IanLewis, Intgr, JMatthews, JamesBWatson, Joerg Reiher, JonathanLivingston, Jonon, KevinTeague, Luis Felipe Braga, M4gnum0n, MER-C, MichaelRWolf, Monkeyget, Msjegan, Mtsic, Mxn, OrangeDog, Pcb21, Psg5p, RedWolf, Rimshot, Ronslow, Ronz, SergejDergatsjev, Sietse Snel, Sir Nicholas de Mimsy-Porington, Spoon!, Stalker134314, SteinbDJ, Strib, Synchronism, TakuyaMurata, Tarquin, TheParanoidOne, Tobias Bergemann, Trashtoy, Tuntable, Vghuston, Vikalpasingh, Wapcaplet, Wookietreiber, 108 anonymous edits



**Bridge pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=407286190> *Contributors:* Abednigo, Alain345, Albert cheng, Andriiko, Anthony Appleyard, Areicher, Argel1200, Beland, BenFrantzDale, Billyblind, Burns700, Bytebear, Cduunn2001, Chimply, Chochopk, CIPHERYnx, Cmdrjameson, Cra88y, DCEdwards1966, Daydreamer302000, Dddenton, Deathy, Djmckee1, Drrngryv, Dlugosz, Edenphd, ErikHaugen, Faradayplank, Frecklefoot, Fred Bradstadt, Ftiercel, GiM, Hao2lian, Heron, Honestshrubber, Hu12, IanLewis, ImageObserver, Iridescent, Jimlonj, Joke de Buhr, Kazubon, Kellen`, Khalid hassani, Lathspell, Lexxdark, Lukasz.gosik, Luis Felipe Braga, M4gnum0n, MER-C, Malaki12003, MattGiuca, Mcavigelli, Michael miceli, MichalKwiatkowski, Mikemil, Mintguy, Mjworthey, Mkel85, Mormat, Msjegan, Mtsaic, Mxn, Narcelio, NickelShoe, Odie5533, Omicronpersei8, Patrick Schneider, Pcb21, Phil Boswell, Quuxplusone, RedWolf, Rouven Thimm, Rror, Ryanvolpe, S.K., Seethefellow, Smfaisalabbas, Springnuts, Sugarfish, Supadawg, Supersid01, TakuyaMurata, TheMandarin, Thomas Linder Puls, Tim.simpson, Tobias Bergemann, Topdeck, Torc2, Umreemala, Unyoyega, Usmanmyname, Vghuston, Vladimir Bosnjak, Wapcaplet, Wiki Raja, Wikidicted, Ykh Wong, Zundark, 185 anonymous edits

**Composite pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=411103347> *Contributors:* Aaron Rotenberg, Abednigo, AdrianHesketh, Alain345, Albert cheng, Andrewtkchan, Anonymous2420, Anthony Appleyard, BenFrantzDale, BradBeattie, Braiam, Bravegag, CMosher01, CatharticMoment, Charles Matthews, Chimply, Coffeeflower, DCrazy, Damien Cassou, Darksports, Decltype, Dycedarg, Edenphd, Emufarmers, Ewger, FrankTobia, Fred Bradstadt, Gaurav1146, Gene Thomas, Gkarlic, Hairy Dude, Hao2lian, Harborsparrow, Hephaestos, Hervegirod, Hu12, ImageObserver, J.delanoy, J0no, Jeepday, Jogerh, Joyous!, Judgesurreal777, Kars, Kate, Larry laptop, Lathspell, Lukasz.gosik, Luis Felipe Braga, M4gnum0n, MER-C, Matekm, Matěj Grabovský, Mefyl, Michael miceli, Mik01aj, MikalZiane, Mike926908240680, Mrwojo, Msjegan, Mtsaic, Narcisse, Nick1915, Nikai, Pearle, Pinguin.tk, RedWolf, Rishichauhan, Sebras, Shabbirhimani, Shivendradayal, Skapur, Skarebo, Sneftel, Stwalkerster, TakuyaMurata, Thamaraiselvan, The Thing That Should Not Be, Thomas Linder Puls, Trashtoy, Tubalubalu, Umreemala, Vinhphunguyen, Vishalmamania, Vonkje, Waggess, Wainstead, Whiner01, Will2k, Winnwang, Wkjh, Zarcadia, Zoz, 167 anonymous edits

**Decorator pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=414558462> *Contributors:* ACiD2, Abednigo, Adam Zivner, Ahoerstemeier, Aihdikh, AmunRa84, Anthony Appleyard, Army, Ascänder, Awistaiw, BenFrantzDale, Booler80, Bravegag, BryceHarrington, Codysnider, Connelly, Crucone, Cybercobra, Doradus, Dori, Doug Bell, Drae, EdC, Edenphd, Eric-Western, ErikStewart, FatalError, Francesco Terezani, Gerel, Ggenellina, Grokus, Gslockwood, Hao2lian, Harborsparrow, Hu12, Hunting dog, Itai, J.lonnee, J1o1h1n, JRM, Jeeves, Joriki, Khalid hassani, Khmer42, MER-C, Matt Crypto, Minghong, Mjb, Mjresin, Mostapha, Motine, Msjegan, Mtsaic, Murtasa, Nafeezabrar, Nheirbaut, Odie5533, P0per, Pharaoh of the Wizards, Phresnel, RUL3R, RedWolf, Romanc19s, Ronz, Rrburke, Snooper77, TakuyaMurata, Tgwizard, Tobias Bergemann, Tomerfiliba, Trashtoy, Triskaideka, Urhixidur, Vghuston, Wackimonki, Wik, Wikidrone, Will2k, XcepticZP, 160 anonymous edits

**Facade pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=408713177> *Contributors:* A5b, Abednigo, Aff123a, Andyluciano, Anthony Appleyard, Arch4ngel, Arvindkumar.avinash, Ashlux, Beland, BenFrantzDale, Chestertan, Cybercobra, Darkmane, Dkeithmorris, Dlohcierekim, Dreftymac, Dysprosia, Forseti, Frederikton, Hairy Dude, Hangy, Hao2lian, Hofoen, Hu12, IanLewis, Immure, Jfvalerez, Jogloran, Jorend, Jussist, Kaikko, Khishig.s, Kwamikagami, Lathspell, Lpenz, Lukasz.gosik, Luis Felipe Braga, M4c0, MER-C, Mahanga, MrOllie, Msjegan, Mtsaic, Mxn, Narcissus, Nate Silva, Nibblus, Otterfan, Philip Baird Shearer, RedCrystal, RedWolf, Scullder, Svick, Synook, TakuyaMurata, Thadius856, Tony Sidaway, Tormit, Trashtoy, Vald, Wookie2u, Wwjedsc, Xiaoddkkwe, Xrm0, 에멜무지로, 122 anonymous edits

**Flyweight pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=413403965> *Contributors:* Abednigo, Albert cheng, AlphaWiki, Anirudhvyas010, Anticipation of a New Lover's Arrival, The, Beland, BenFrantzDale, Beryllium, BigDunc, Blastthisinferno, CMosher01, Debajit, Ebswift, Emj, FroZman, Gnp, Hao2lian, Hu12, Justo, Kurt Jansson, Lathspell, Lazerotypx, MER-C, Maxim, Mdjpurdon, Micah headline, Msjegan, Mtsaic, Mxn, Naasking, Northerly Star, Odie5533, Op12, Pcb21, PierreAbbat, Ray Van De Walker, RedWolf, Ryan, Sproul, TakuyaMurata, Teglsbo, The Wild Falcon, Tshak, Twyfrod, Vipinhari, Warren, Wigzob, X00022027, Zozakral, 78 anonymous edits

**Proxy pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=403125095> *Contributors:* 0612, Abednigo, Abelson, Anonymous2420, Anthony Appleyard, Audriusa, Avocado, Azanin, Babomb, Beland, BenFrantzDale, Bravegag, Brownb2, Cholmes75, D3xter, Decltype, Ebswift, Edenphd, Edivorace, Espresso1000, Fred Bradstadt, Fritz freiheit, Ftiercel, Hao2lian, Hu12, Jamelan, Jay, Jd gordon, JeR, Jthiesen, Lathspell, Lukasz.gosik, Lumingz, Lusum, MER-C, Matekm, Msaada, Msjegan, NathanBeach, Nonomy, Nurg, Nzroller, Odie5533, P0per, Pateeludg, Philip Trueman, Prickus, Quuxplusone, RedWolf, Rich Farmbrough, Rich257, Royalguard11, SanthoshRadhakrishnan, Snaxe920, TakuyaMurata, TheParanoidOne, TravisHein, Valodzka, VladM, Will2k, Wlievens, Zoolookas, Zozakral, 100 anonymous edits

**Chain-of-responsibility pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=413404149> *Contributors:* Abednigo, AboutMM, Arch4ngel, Architectureblog, Arjunktul, BenFrantzDale, Bin Sowparnika, Bravegag, Brenont, Brian428, Celephicus, Cmdrjameson, Courcelles, Danny, Eknv, Fmstephe, Hao2lian, Hu12, Ilion2, Itschris, Jhatax, Jdupont, JoaoRicardo, Jokes Free4Me, Khalid hassani, Khym Chanur, King Lopez, Kinguy, Lathspell, Lylez, M4c0, MER-C, Merutak, Mikaey, Nomad7650, Olbat, Pcraven, Porqin, RedWolf, Reyjose, Rhomboid, Rice Yao, Ronz, Soubock, Struggling wang, Tabletop, TakuyaMurata, TheParanoidOne, Thomas Linder Puls, Vghuston, Vikasgoyal77, Wbstory, Wikidrone, Zaletsyy, Zyx, 73 anonymous edits

**Command pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=413402617> *Contributors:* Abdull, Abednigo, Acmeacme, Ascänder, Beland, BenFrantzDale, Bendor, Bipedalpedia, Bluemoose, Bravegag, Crhophkins, Darklilac, Daydreamer302000, Dazhiruoyu, Dharesign, DotnetCarpenter, Doug Bell, Druckles, Dtblullock, Ebainomugisha, Edward.in.Edmonton, Ehn, Exe, Eyewax, Fred Bradstadt, Gene Thomas, Gmoore19, GregorB, Hao2lian, Hooperbloob, Hu12, IanVaughan, JHunterJ, Jevansen, JoaoTrindade, John Lindquist, Jorend, Karuppiiah, Kate, King Lopez, Kku, Lightdarkness, MER-C, Mahanga, Mernen, Michael miceli, Mjworthey, Mloskot, Modify, Nakon, Nathan Ra, Omegatron, Orangeroo, Oubless, Owentaylor25, Pcb21, Pharaoh of the Wizards, Piet Delpot, RedWolf, RexasS, Rijkbenik, SJP, Snottywong, TakuyaMurata, Terheyden, Terry.kfwong, TheParanoidOne, Trevor Johns, Vald, Vghuston, Vikasgoyal77, Will2k, Zixyer, Zozakral, 120 anonymous edits

**Interpreter pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=408588367> *Contributors:* Abednigo, Allan McInnes, Andypayne, Ascilifeform, BenFrantzDale, Borderj, Cybercobra, Davewho2, Fredrik, Furrykef, Gwalla, Hao2lian, Hu12, Komap, L30nc1t0, Lathspell, Lonelybug, Lukasz.gosik, Mahanga, Matekm, Merutak, MithrandirMage, Mrwojo, Odie5533, Pcb21, RJFJR, RedWolf, Rice Yao, Rogervhitey, Rosiestep, SamePaul, TakuyaMurata, TheParanoidOne, Twyfrod, Vghuston, Zozakral, 48 anonymous edits

**Iterator pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=412075300> *Contributors:* Abednigo, Albert cheng, Angilly, Ayvengo21, Beland, BenFrantzDale, Bnedelko, Btx40, Bytebear, D chivaluri, Eaolson, Eknv, Ghthor, Hao2lian, Hu12, Iridescent, J04n, Jogloran, Lababidi, Leibniz, Marj Tiefert, Matekm, Mormat, Odie5533, Ogmios, Radagast83, Ravindra Josyula, RedWolf, Seffer, TakuyaMurata, Tomtheditor, Trashtoy, Vghuston, Winxa, Zerokitsune, Zozakral, 78 anonymous edits

**Mediator pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=413404291> *Contributors:* 16@r, Abednigo, Anonymous2420, Balu213, Bayesianese, BenFrantzDale, C3foncea, Colonies Chris, Cybercobra, Dawynn, Dcflyer, Exe, Flammiifer, Furrykef, Grancalavera, Hao2lian, Hu12, If R-rated movies are porn, it was porn, Jebdm, Jrockley, Kku, Lathspell, Leafyplant, Mariolopes, Mormat, Pietron, RedWolf, Rijnienaber, RobDe68, RoyBoy, Royalguard11, TakuyaMurata, TheParanoidOne, Twyfrod, Vghuston, VladM, Whitepaw, Zozakral, 93 anonymous edits

**Memento pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=402698261> *Contributors:* Aaron Rotenberg, Abdull, Abednigo, Alain345, Alvin-cs, BenFrantzDale, CountRazumovsky, Hao2lian, Heron, Hu12, JLaTondre, Kingturtle, Lathspell, Mild Bill Hiccup, Mormat, Odie5533, Pcb21, Ray Van De Walker, RedWolf, Rich257, Royalguard11, Sardak, Seethefellow, Shadowjams, Spoon!, Stuart Yeates, Sushi Tax, TakuyaMurata, Vghuston, VladM, Zeno Gantner, Zigmarr, Zozakral, 42 anonymous edits

**Null Object pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=412719155> *Contributors:* 16@r, Annrules, Autarch, Cybercobra, Dirk Riehle, Hairy Dude, Ice Keese, Jpp, Mahanga, Neelix, Paulecoyote, Rjwilmsi, Ruijelo, Snaxe920, Spiritia, Svick, Teles, Timwi, Yossi Gil, Zozakral, 45 anonymous edits

**Observer pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=413543545> *Contributors:* Aaron Rotenberg, Abednigo, Albert cheng, Alejandro.isaza, Alexvaughan, Alvin-cs, Amazedsaint, Anna Lincoln, Anonymous Dissident, Ariadie, Arnon Mathias, Arturus, Autarch, AxelBoldt, BBilge, Bdanee, Beland, BenFrantzDale, Bkil, Booyabazooka, Bovlb, Bravegag, Briefstyle, C. lorenz, C3foncea, CBMalloch, Calvin 1998, Can't sleep, clown will eat me, Capricorn42, Ceyockey, Cosmin varlan, Countchoc, Darth Panda, David Levy, Decatur-en, Deineka, Diderot, Dionyziz, Edward, Eknv, Emergeo, Emurphy42, Emw, Eric Le Bigot, Excirial, Flaming Grunt, Frecklefoot, Fred Bradstadt, Furrykef, Gang65, Gregsinclair, Gustavo.mori, Hao2lian, Hervegirod, Hooperbloob, Hu12, HurryPeter, Ipatrol, Jamelan, Jarsyl, Jay, Jebdm, Jems420, Kaquito, Karuppiiah, Kate, KellyCoinGuy, Kelumnpk, Kgeis, Khalid hassani, Kku, Labviewportal, LazyEditor, Leibniz, Lnatans25, Lpsiphi, M.amjad.saeed, Mahanga, MatthieuWiki, Merutak, Michaelschmatz, MirkMeister, Mratzloff, Mrwojo, Mzyxptlk, Nickg, Nickolas.pohiets, OMouse, Oc666, Ohnoitsjamie, OIEnglish, OmveerChaudhary, OrangeDog, Pcb21, Philipp Weis, Pingveno, Pmercier, Porton, Quux, Radagast83, Razafy, RedWolf, Redroof, Retired username, RickBeton, Robbiemuffin, Roy Paveza, Ronz, Rotaerz, RoyBoy, RussBlau, SD6-Agent, Sam Pointon, SamuelHuckins, Shdwfeather, Shirtwaist, Sjmsteffann, SkyWalker, SlubGlub, Stephan Leeds, Sunnychow, SynergyBlades, TakuyaMurata, Terry.kfwong, TheMuuj, TheParanoidOne, Tmkeesey, TobiasHerp, Topdeck, Torc2, Vahrokh, Vghuston, Vipinhari, Voyvf, VvV, Wbstory, WikiSolved, Wikidrone, Willpillp, Xynopsis, YordanGeorgiev, Zozakral, 276 anonymous edits

**Design pattern Servant** *Source:* <http://en.wikipedia.org/w/index.php?oldid=413596002> *Contributors:* AzaToth, Beoros, Invitrovanitas, Novice7, RHaworth, Rudymment, VeselyBrumla, 3 anonymous edits

**Specification pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=404817239> *Contributors:* B4hand, Beland, Esowteric, Ivant, Maralia, RickBeton, Squishymaker, Svallory, Torc2, Wavell2003, Zozakral, 30 anonymous edits

**State pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=413596271> *Contributors:* Abednigo, Allan McInnes, Anonymous2420, Babomb, BenFrantzDale, BennyD, Cedar101, Deltaflux, Edenphd, Eknv, Ertwoc, Frigoris, FunnyMan3595, Garyczek, Hao2lian, Hariharank12, Hu12, IanVaughan, Jamesontai, Jeepday, Jlechem, JoaoTrindade, Julesd, LuckyStarr, Luigi30, M4bwav, Matěj Grabovský, MithrandirMage, Mormat, Naasking, Nitin.nizhawan, Pilotguy, RedWolf, Ruud Koot, Suniltg, TakuyaMurata, Terheyden, TheParanoidOne, Vghuston, 74 anonymous edits

**Strategy pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=412830895> *Contributors:* Abednigo, Anirudhvyas010, Anupk7, Atree, AustinBlues, BenFrantzDale, BlueTaRaKa, Cconnett, Dcadenas, Ddonnell, Decltype, Dougher, Edenphd, Equendil, Finked, Flex, Forseti, Fredrik, Gluegadget, Hao2lian, Helmerh, Hephaestos, Hosamaly, Hu12, Jems420, Jerryobject, Jianwu, Jogloran, Jonasschneider, Joswig, Kingfishr, Landisdesign, Liko81, Loadmaster, Lst27, Luís Felipe Braga, M4gnum0n, MER-C, Mahanga, Marco Krohn, Md2perpe, MichalKwiatkowski, Mik01aj, Minddgo, Mjchonoles, Ospalh, PJonDevelopment, Pcb21, Platypus222, Pzrq, Quuxplusone, Radon210, RedWolf, Redroof, Remuel, Richard.a.paul, Ritchiep, Ronz, Ruud Koot, SlubGluB, Snaxe920, Southen, Spookylukey, SpuriousQ, Sumanthamma, Sungur, Supadawg, TakuyaMurata, The Wild Falcon, TheParanoidOne, TjerkWol, Tombomp, Torc2, Vghuston, Vladimir Bosnjak, Wikidrone, Wmwallace, Woohookitty, XcepticZP, Zhen Lin, Zozakral, 187 anonymous edits

**Template method pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=411940566> *Contributors:* AHMartin, Abednigo, AleaB, Allstarecho, Andreas Kaufmann, Anonymous2420, Antonielly, BenFrantzDale, Coconut99 99, Cp111, Dorftrottel, Dysprosia, Edenphd, Exe, FF2010, Fredrik, GiedrBac, Hao2lian, Hu12, Iridescent, JonatanAndersson, Jsmith030416, KJS77, KJedi, LanceBarber, Liko81, LordTorgamus, MER-C, Mahanga, Mange01, Merutak, Michael Hardy, Mormat, Mrwojo, Normandomac, Oleg Alexandrov, Pbyhistorian, Quuxplusone, RedWolf, Reinis, Remuel, Richard Katz, Rjwilmsi, Roadrunner, Russell Triplehorn, Ruud Koot, Supadawg, T1mmyb, TakuyaMurata, TheParanoidOne, Vghuston, Vladimir Bosnjak, Zarkonnen, Zhen Lin, Zozakral, 81 anonymous edits

**Visitor pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=413404735> *Contributors:* Abednigo, Aeolian145, Afsinbey, Akanksh, AlanUS, Albert cheng, Alfatrion, Alfredo.correa, Alksentrs, Anirudhvyas010, Anonymous2420, Autarch, Beland, BenFrantzDale, Bennor, Brandon, Bravegag, C3oncea, CardinalDan, Computerwguy, D chivaluri, Decltype, Dethomas, Devjava, Edenphd, Eob, Fbahr, Fizzbann1234, Fredrik, Friskeppermint, Gaius Cornelius, Gamma, Gandalfgeek, Gnewf, GregorB, HGoat, Hadal, Hao2lian, Hu12, Iffypop, Int19h, Jcarroll, Jdawson7, Joel falcou, Jokes Free4Me, Joswig, Jrduncans, Karada, Kate, Kraftlos, Kuru, Leibniz, Luiscolorado, Lumingz, M4gnum0n, MER-C, Marau, Martnym, MattGiuca, Matthew0028, Mcsee, Mitchan, Mountain, Nanderman, Nchaimov, Oerjan, Orderud, Pcb21, Pelister, Pewchies, Phil Boswell, Pkirlin, PolyGlot, RedWolf, Reyk, Riki, Salix alba, Sergei49, Skarkkai, Sleske, Szeder, TakuyaMurata, Theone256, Timenkov Fedor, Tobias Bergemann, Versus22, Vghuston, Vinz83, Wiki alf, Wlievens, Woohookitty, Zozakral, Ржавый Хомяк, 214 anonymous edits

**Active object** *Source:* <http://en.wikipedia.org/w/index.php?oldid=406367396> *Contributors:* Beland, Gingerjoos, Hans Adler, Ipsign, JennyRad, Kku, Kobrabones, Nixeagle, RHaworth, ShakespeareFan00, Silesianus, Stimpj, Tim@, Vegarwe, Wigy, Zyx, 7 anonymous edits

**Balking pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=404867780> *Contributors:* Abdull, Amiceli, Beland, Crystallina, DavidBrooks, GregorB, Loopy48, RedWolf, Spliffy, TakuyaMurata, 17 anonymous edits

**Messaging pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=411770062> *Contributors:* 1000Faces, 4th-otaku, Abdull, Anrie Nord, Beland, Djmackenzie, JerryLerman, Munahaf, Prickus, Rwww, Tinucherian, 9 anonymous edits

**Double-checked locking** *Source:* <http://en.wikipedia.org/w/index.php?oldid=412010382> *Contributors:* (, Bdawson, BeeOnRope, Beland, Bultro, Cryptic, Digulla, Doug Bell, Drripp, Dubwai, Flyingped, GhettoBlaster, Gracefool, Haakon, Hairy Dude, Initworks, Javawizard, Jzeuss, Jm307, Julesd, Jwir3, Jyke, Marudubshinki, MikeZar, Msundman, Mithrandir, Neile, Nlu, PJTraill, Quercus basaseachicensis, RedWolf, Rich Farmbrough, ShelfSkewed, SimonTrew, TakuyaMurata, Taw, ToastyKen, 78 anonymous edits

**Asynchronous method invocation** *Source:* <http://en.wikipedia.org/w/index.php?oldid=394569181> *Contributors:* Aervanath, Cander0000, Davhdavh, Hairy Dude, Hans Adler, Ipsign, JulesH, JzG, RHaworth, Ron Ritzman, Sumeet.chhetri, Uncle G

**Guarded suspension** *Source:* <http://en.wikipedia.org/w/index.php?oldid=404870538> *Contributors:* Allan McInnes, Amiceli, Beland, CryptoDerk, Fredrik, HappyDog, Loopy48, MithrandirMage, Pcb21, Reedy, Remuel, TakuyaMurata, TheParanoidOne, TuukkaH, 2 anonymous edits

**Lock (computer science)** *Source:* <http://en.wikipedia.org/w/index.php?oldid=408619965> *Contributors:* (, Allan McInnes, Altenmann, Amorken, Andahazy, Beland, CanisRufus, Chozan, Clausen, Craig Stuntz, CyborgTosser, DavidCary, Dicklyon, Dori, Dlugosz, Edward, Fuhghettaboutit, Furrykef, Gazpacho, Gsln, Hathawayc, Hervegirod, JCLately, Jacobolus, Jay, Jerryobject, JimD, Kaycee srk, KeyStroke, Liao, Mark Renier, Maxal, Michael Slone, Michael Suess, Moonwolf14, Neile, Omikronuk, OrangeDog, Rjwilmsi, Roman12345, Ross bencina, Rsocol, SJP, Shivamohan, StanContributor, Suruena, Svick, TakuyaMurata, Taw, Torc2, Tothwolf, Vald, Wmbolle, Wrp103, Wykypydy, YB3, Zeroflag, Zigger, Zoicon5, Zundark, <sup>212</sup>, 84 anonymous edits

**Monitor (synchronization)** *Source:* <http://en.wikipedia.org/w/index.php?oldid=407142371> *Contributors:* AKEB, Abdull, Anna Lincoln, Asafshelly, Assimil8or, AxelBoldt, Charlesb, Chruck, Cleared as filed, DARTH SIDIOUS 2, Dcoetzee, Dellidot, Flex, Franl, Gazpacho, Gennaro Prota, Gonnet, Herrturtur, Ianb1469, Intgr, Ipatrol, JLaTondre, Jacosi, Jeroen74, Jerryobject, John Vandenberg, Joy, Kislay kishore2003, Marudubshinki, Miym, Mormegil, Moshewe, Moulaali, Musiphil, Niuye, Pacman128, Parklandspanaway, Pburka, Pdcok, Raul654, Rich Farmbrough, RickBeton, Rv74, Tcnvc, Theodore.norvell, Thiagomacieira, Thomas Linder Puls, Torc2, Voskanyan, Waqas1987, WikHead, Yukoba, 99 anonymous edits

**Reactor pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=413229718> *Contributors:* Alejop, Apoc2400, Benjamin.booth, Bigcheese, Chocolateboy, Danio, Distalzo, Ehames, Ipsign, JCarlos, JLaTondre, Krauss, Obankston, Praxis, Tomerfiliba, UlrichAAB, 19 anonymous edits

**Readers-writer lock** *Source:* <http://en.wikipedia.org/w/index.php?oldid=413126227> *Contributors:* Andreas Kaufmann, Babobibo, Ber, Darth Panda, Dublet, Dvyukov, Fuzzbox, Greensburger, Jakarr, Jeenuv, Loopy48, Peepeedia, Quuxplusone, Ronklein, Spoonboy42, Vald, ZeroOne, 16 anonymous edits

**Scheduler pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=347950331> *Contributors:* CarlHewitt, DanMS, David Gerard, Fram, M2Ys4U, RedWolf, Shoeofdeath, TSFS, TakuyaMurata, Vansong

**Thread pool pattern** *Source:* <http://en.wikipedia.org/w/index.php?oldid=405076004> *Contributors:* Andreas Kaufmann, Arkanosis, Asimahme tx1, Bezenek, Caitlinhalstead, CanisRufus, Cburnett, Charles Matthews, Check, Cybercobra, Denny-cn, Doug Bell, GhettoBlaster, IceMan42, Jdavidw13, JonHarder, Leuk he, Psm, Ptrb, Red Thrush, RedWolf, RickBeton, Soumyasch, Stokestack, Swguy3, Ysangkok, 26 anonymous edits

**Thread-local storage** *Source:* <http://en.wikipedia.org/w/index.php?oldid=414445545> *Contributors:* AKGhetto, Andareed, Andreas Kaufmann, AndrewHowse, Antithesisw, Asafshelly, B4hand, BevinBrett, Brianski, Christian75, D.scain.farenzena, Doug Bell, ENeville, Frap, Gerbrant, Heron, Hmains, Hutteman, Jengelh, Jorend, Kamasutra, LeoNerd, Mais oui!, Marudubshinki, NathanBeach, Nealcardwell, Orderud, Parklandspanaway, Pavel Vozenilek, Pjvpjv, Qutezuze, Raul654, Raysonho, Rvalls, SJFriedl, Scottielad, Suruena, Tackline, Thomerow, TimBentley, 65 anonymous edits

# Image Sources, Licenses and Contributors

**Image:Abstract factory.svg** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Abstract\\_factory.svg](http://en.wikipedia.org/w/index.php?title=File:Abstract_factory.svg) *License:* unknown *Contributors:* User:DoktorMandrake

**Image:Abstract Factory in LePUS3.png** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Abstract\\_Factory\\_in\\_LePUS3.png](http://en.wikipedia.org/w/index.php?title=File:Abstract_Factory_in_LePUS3.png) *License:* Public Domain *Contributors:* Amnon Eden (User:Edenphd)

**Image:builder2.png** *Source:* <http://en.wikipedia.org/w/index.php?title=File:Builder2.png> *License:* GNU Free Documentation License *Contributors:* BennyD, Ftiercel, Joey-das-WBF

**Image:FactoryMethod.svg** *Source:* <http://en.wikipedia.org/w/index.php?title=File:FactoryMethod.svg> *License:* Public Domain *Contributors:* User:Stannered

**Image:Factory Method pattern in LePUS3.png** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Factory\\_Method\\_pattern\\_in\\_LePUS3.png](http://en.wikipedia.org/w/index.php?title=File:Factory_Method_pattern_in_LePUS3.png) *License:* Public Domain *Contributors:* Amnon Eden (User:Edenphd)

**Image:Singleton UML class diagram.svg** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Singleton\\_UML\\_class\\_diagram.svg](http://en.wikipedia.org/w/index.php?title=File:Singleton_UML_class_diagram.svg) *License:* Public Domain *Contributors:* User:Trashtoy

**Image:ObjectAdapter.png** *Source:* <http://en.wikipedia.org/w/index.php?title=File:ObjectAdapter.png> *License:* Public Domain *Contributors:* Original uploader was GilHamilton at en.wikipedia

**Image:Adapter(Object) pattern in LePUS3.png** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Adapter\(Object\)\\_pattern\\_in\\_LePUS3.png](http://en.wikipedia.org/w/index.php?title=File:Adapter(Object)_pattern_in_LePUS3.png) *License:* Public Domain *Contributors:* Amnon Eden (User:Edenphd)

**Image:ClassAdapter.png** *Source:* <http://en.wikipedia.org/w/index.php?title=File:ClassAdapter.png> *License:* Public Domain *Contributors:* Original uploader was GilHamilton at en.wikipedia

**Image:Adapter(Class) pattern in LePUS3.png** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Adapter\(Class\)\\_pattern\\_in\\_LePUS3.png](http://en.wikipedia.org/w/index.php?title=File:Adapter(Class)_pattern_in_LePUS3.png) *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Edenphd

**Image:Bridge pattern in LePUS3.1.gif** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Bridge\\_pattern\\_in\\_LePUS3.1.gif](http://en.wikipedia.org/w/index.php?title=File:Bridge_pattern_in_LePUS3.1.gif) *License:* Public Domain *Contributors:* Ian Atkin (User:Sugarfish) [correction of image originally created by Amnon Eden (User:Edenphd)]

**Image:Bridge UML class diagram.svg** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Bridge\\_UML\\_class\\_diagram.svg](http://en.wikipedia.org/w/index.php?title=File:Bridge_UML_class_diagram.svg) *License:* Public Domain *Contributors:* User:Trashtoy

**Image:Composite UML class diagram (fixed).svg** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Composite\\_UML\\_class\\_diagram\\_\(fixed\).svg](http://en.wikipedia.org/w/index.php?title=File:Composite_UML_class_diagram_(fixed).svg) *License:* Public Domain *Contributors:* User:Aaron Rotenberg, User:Trashtoy

**Image:Composite pattern in LePUS3.png** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Composite\\_pattern\\_in\\_LePUS3.png](http://en.wikipedia.org/w/index.php?title=File:Composite_pattern_in_LePUS3.png) *License:* Public Domain *Contributors:* Amnon Eden (User:Edenphd)

**File:Decorator UML class diagram.svg** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Decorator\\_UML\\_class\\_diagram.svg](http://en.wikipedia.org/w/index.php?title=File:Decorator_UML_class_diagram.svg) *License:* Public Domain *Contributors:* User:Trashtoy

**Image:Decorator pattern in LePUS3.png** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Decorator\\_pattern\\_in\\_LePUS3.png](http://en.wikipedia.org/w/index.php?title=File:Decorator_pattern_in_LePUS3.png) *License:* Public Domain *Contributors:* Amnon Eden (User:Edenphd)

**Image:UML2 Decorator Pattern.png** *Source:* [http://en.wikipedia.org/w/index.php?title=File:UML2\\_Decorator\\_Pattern.png](http://en.wikipedia.org/w/index.php?title=File:UML2_Decorator_Pattern.png) *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Nheirbaut

**Image:FacadeDesignPattern.png** *Source:* <http://en.wikipedia.org/w/index.php?title=File:FacadeDesignPattern.png> *License:* Creative Commons Attribution-Sharealike 2.5 *Contributors:* Original uploader was 에델무지로 at en.wikipedia

**Image:proxy\_pattern\_diagram.svg** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Proxy\\_pattern\\_diagram.svg](http://en.wikipedia.org/w/index.php?title=File:Proxy_pattern_diagram.svg) *License:* GNU Free Documentation License *Contributors:* User:Stannered

**Image:Proxy pattern in LePUS3.gif** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Proxy\\_pattern\\_in\\_LePUS3.gif](http://en.wikipedia.org/w/index.php?title=File:Proxy_pattern_in_LePUS3.gif) *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Amnon Eden (talk)

**Image:Command Design Pattern Class Diagram.png** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Command\\_Design\\_Pattern\\_Class\\_Diagram.png](http://en.wikipedia.org/w/index.php?title=File:Command_Design_Pattern_Class_Diagram.png) *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Original uploader was JoaoTrindade at en.wikipedia Later version(s) were uploaded by Trevor Johns at en.wikipedia.

**Image:Interpreter\_UML\_class\_diagram.jpg** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Interpreter\\_UML\\_class\\_diagram.jpg](http://en.wikipedia.org/w/index.php?title=File:Interpreter_UML_class_diagram.jpg) *License:* GNU Free Documentation License *Contributors:* Rice Yao

**File:Observer.svg** *Source:* <http://en.wikipedia.org/w/index.php?title=File:Observer.svg> *License:* Public Domain *Contributors:* User:WikiSolved

**File:DesignPatternServantFigure1.png** *Source:* <http://en.wikipedia.org/w/index.php?title=File:DesignPatternServantFigure1.png> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Creator:VeselyBrumlaVeselyBrumla

**File:DesignPatternServantFigure2.png** *Source:* <http://en.wikipedia.org/w/index.php?title=File:DesignPatternServantFigure2.png> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Creator:VeselyBrumlaVeselyBrumla

**Image:Specification UML.png** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Specification\\_UML.png](http://en.wikipedia.org/w/index.php?title=File:Specification_UML.png) *License:* Free Art License *Contributors:* User:IToshkov

**Image:State Design Pattern UML Class Diagram.svg** *Source:* [http://en.wikipedia.org/w/index.php?title=File:State\\_Design\\_Pattern\\_UML\\_Class\\_Diagram.svg](http://en.wikipedia.org/w/index.php?title=File:State_Design_Pattern_UML_Class_Diagram.svg) *License:* GNU Free Documentation License *Contributors:* User:Ertwroc

**Image:State pattern in LePUS3.gif** *Source:* [http://en.wikipedia.org/w/index.php?title=File:State\\_pattern\\_in\\_LePUS3.gif](http://en.wikipedia.org/w/index.php?title=File:State_pattern_in_LePUS3.gif) *License:* Public Domain *Contributors:* Amnon Eden (User:Edenphd)

**Image:Strategy\_Pattern\_Diagram\_ZP.svg** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Strategy\\_Pattern\\_Diagram\\_ZP.svg](http://en.wikipedia.org/w/index.php?title=File:Strategy_Pattern_Diagram_ZP.svg) *License:* Public Domain *Contributors:* Original uploader was XcepticZP at en.wikipedia

**Image:Strategy pattern in LePUS3.gif** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Strategy\\_pattern\\_in\\_LePUS3.gif](http://en.wikipedia.org/w/index.php?title=File:Strategy_pattern_in_LePUS3.gif) *License:* Public Domain *Contributors:* Amnon Eden (User:Edenphd)

**Image:Strategy.JPG** *Source:* <http://en.wikipedia.org/w/index.php?title=File:Strategy.JPG> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Original uploader was Sungur at en.wikipedia (Original text : Guray Sungur)

**Image:Template Method UML.svg** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Template\\_Method\\_UML.svg](http://en.wikipedia.org/w/index.php?title=File:Template_Method_UML.svg) *License:* GNU Free Documentation License *Contributors:* Giacomo Ritucci

**Image:Template Method pattern in LePUS3.gif** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Template\\_Method\\_pattern\\_in\\_LePUS3.gif](http://en.wikipedia.org/w/index.php?title=File:Template_Method_pattern_in_LePUS3.gif) *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Amnon Eden (User:Edenphd)

**Image:VisitorClassDiagram.svg** *Source:* <http://en.wikipedia.org/w/index.php?title=File:VisitorClassDiagram.svg> *License:* Public Domain *Contributors:* User:EatMyShortz

**Image:Visitor pattern in LePUS3.gif** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Visitor\\_pattern\\_in\\_LePUS3.gif](http://en.wikipedia.org/w/index.php?title=File:Visitor_pattern_in_LePUS3.gif) *License:* Public Domain *Contributors:* Amnon Eden (User:Edenphd)

**Image:VisitorPatternUML.png** *Source:* <http://en.wikipedia.org/w/index.php?title=File:VisitorPatternUML.png> *License:* Public Domain *Contributors:* {{subst:usernamepxand}}

**Image:Monitor (synchronization)-SU.png** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Monitor\\_\(synchronization\)-SU.png](http://en.wikipedia.org/w/index.php?title=File:Monitor_(synchronization)-SU.png) *License:* Creative Commons Attribution 3.0 *Contributors:* Theodore.norvell

**Image:Monitor (synchronization)-Mesa.png** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Monitor\\_\(synchronization\)-Mesa.png](http://en.wikipedia.org/w/index.php?title=File:Monitor_(synchronization)-Mesa.png) *License:* Creative Commons Attribution 3.0 *Contributors:* User:Theodore.norvell

**Image:Monitor (synchronization)-Java.png** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Monitor\\_\(synchronization\)-Java.png](http://en.wikipedia.org/w/index.php?title=File:Monitor_(synchronization)-Java.png) *License:* Creative Commons Attribution 3.0 *Contributors:* User:Theodore.norvell

**Image:Thread pool.svg** *Source:* [http://en.wikipedia.org/w/index.php?title=File:Thread\\_pool.svg](http://en.wikipedia.org/w/index.php?title=File:Thread_pool.svg) *License:* GNU Free Documentation License *Contributors:* User:Cburnett

# License

---

Creative Commons Attribution-Share Alike 3.0 Unported  
<http://creativecommons.org/licenses/by-sa/3.0/>

---