Design Patterns

PDF generated at: Wed, 29 Aug 2012 18:35:21 UTC

Contents

Articles

Overview **1**

Design pattern 1

Creational Patterns **10**

Creational pattern 10

Abstract factory pattern 12

Builder pattern 23

Factory method pattern 31

Lazy initialization 38

Multiton pattern 50

Object pool pattern 57

Prototype pattern 59

Resource Acquisition Is Initialization 65

Singleton pattern 74

Structural patterns **80**

Structural pattern 80

Adapter pattern 81

Bridge pattern 87

Composite pattern 90

Decorator pattern 97

Facade pattern 105

Front Controller pattern 108

Flyweight pattern 109

Proxy pattern 112

Behavioral patterns **116**

Behavioral pattern 116

Chain-of-responsibility pattern 117

Command pattern 126

Interpreter pattern 135

Iterator pattern 138

Mediator pattern 140

Memento pattern 144

Null Object pattern 147
Observer pattern 153
Publish/subscribe 160
Design pattern Servant 162
Specification pattern 166
State pattern 171
Strategy pattern 175
Template method pattern 179
Visitor pattern 183

Concurrency patterns **192**

Concurrency pattern 192
Active object 193
Balking pattern 193
Messaging pattern 195
Double-checked locking 196
Asynchronous method invocation 202
Guarded suspension 204
Lock 205
Monitor 211
Reactor pattern 221
Readers-writer lock 223
Scheduler pattern 224
Thread pool pattern 225
Thread-local storage 227

References

Article Sources and Contributors 231
Image Sources, Licenses and Contributors 235

Article Licenses

License 236

Overview

Design pattern

In software engineering, a **design pattern** is a general reusable solution to a commonly occurring problem within a given context 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. Patterns are formalized best practices that the programmer must implement themselves in the application.[1] 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.[2]

There are many types of design patterns, like

- **Algorithm strategy patterns** addressing concerns related to high-level strategies describing how to exploit application characteristics on a computing platform.
- **Computational design patterns** addressing concerns related to key computation identification.
- **Execution patterns** that address concerns related to supporting application execution, including strategies in executing streams of tasks and building blocks to support task synchronization.
- **Implementation strategy patterns** addressing concerns related to implementing source code to support
  1. program organization, and
  2. the common data structures specific to parallel programming.
- **Structural design patterns** addressing concerns related to high-level structures of applications being developed.

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.[3][4] 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.), which is frequently abbreviated as "GOF". 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 978-0-321-12742-6.
- 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.[5]

In 2009 over 30 contributors collaborated with Thomas Erl on his book, *SOA Design Patterns*.[6] The goal of this book was to establish a de facto catalog of design patterns for SOA and service-orientation.[7] (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 [8]

Practice

Design patterns can speed up the development process by providing tested, proven development paradigms.[9] 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.[10]

Software design 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,[11] information visualization,[12] secure design,[13] "secure usability",[14] Web design [15] and business model design.[16]

The annual Pattern Languages of Programming Conference proceedings [17] 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.

Creational patterns				
Name	**Description**	**In *Design Patterns***	**In *Code Complete*[18]**	**Other**
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 (dependency injection[19]).	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*[20]
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

Structural patterns				
Name	**Description**	**In *Design Patterns***	**In *Code Complete*[18]**	**Other**
Adapter or Wrapper or Translator [21].	Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces. The enterprise integration pattern equivalent is the Translator [21].	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
Flyweight	Use sharing to support large numbers of similar objects efficiently.	Yes	No	N/A
Front Controller	The pattern relates to the design of Web applications. It provides a centralized entry point for handling requests.	No	Yes	N/A
Module	Group several related elements, such as classes, singletons, methods, globally used, into a single conceptual entity.	No	No	N/A
Proxy	Provide a surrogate or placeholder for another object to control access to it.	Yes	No	N/A

Behavioral patterns				
Name	**Description**	**In *Design Patterns***	**In *Code Complete*[18]**	**Other**
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 in 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

Concurrency patterns			
Name	**Description**	**In *POSA2*[22]**	**Other**
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.[23]	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.[24]	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.[25]	No	*PoEAA*[20]
Messaging design pattern (MDP)	Allows the interchange of information (i.e. messages) between components and applications.	No	N/A
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.[26] 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.[27] 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.

Criticism

The concept of design patterns has been criticized in several ways.

The design patterns may just be a sign of some missing features of a given programming language (Java or C++ for instance). Peter Norvig demonstrates that 16 out of the 23 patterns in the Design Patterns book (that is primarily focused on C++) are simplified or eliminated (via direct language support) in Lisp or Dylan.[28] See also Paul Graham's essay "Revenge of the Nerds".[29]

Moreover, inappropriate use of patterns may unnecessarily increase complexity.[30]

References

[1] 1. Introduction to Spring Framework (http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/overview.html)

[2] Martin, Robert C.. "Design Principles and Design Patterns" (http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf). . Retrieved 2000.

[3] 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."*

[4] 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.

[5] 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 2012-08-11.
[6] Erl, Thomas (2009). *SOA Design Patterns*. New York: Prentice Hall/PearsonPTR. p. 864. ISBN 0-13-613516-1.
[7] Prentice Hall Announces Publication of SOA Design Patterns and SOAPatterns.org Community Site | SOA World Magazine (http://soa.sys-con.com/node/809800)
[8] http://www.soapatterns.org/
[9] Judith Bishop. "C# 3.0 Design Patterns: Use the Power of C# 3.0 to Solve Real-World Problems" (http://msdn.microsoft.com/en-us/vstudio/ff729657). C# Books from O'Reilly Media. . Retrieved 2012-05-15. "If you want to speed up the development of your .NET applications, you're ready for C# design patterns -- elegant, accepted and proven ways to tackle common programming problems."
[10] 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. .
[11] 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.
[12] Heer, J.; M. Agrawala (2006). "Software Design Patterns for Information Visualization" (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. .
[13] Chad Dougherty et al (2009). *Secure Design Patterns* (http://www.cert.org/archive/pdf/09tr010.pdf). .
[14] Simson L. Garfinkel (2005). *Design Principles and Patterns for Computer Systems That Are Simultaneously Secure and Usable* (http://www.simson.net/thesis/). .
[15] "Yahoo! Design Pattern Library" (http://developer.yahoo.com/ypatterns/). . Retrieved 2008-01-31.
[16] "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.
[17] Pattern Languages of Programming, Conference proceedings (annual, 1994—) (http://hillside.net/plop/pastconferences.html)
[18] McConnell, Steve (June 2004). "Design in Construction". *Code Complete* (2nd ed.). Microsoft Press. p. 104. ISBN 978-0-7356-1967-8. "Table 5.1 Popular Design Patterns"
[19] "Design Patterns: Dependency injection" (http://msdn.microsoft.com/de-de/magazine/cc163739(en-us).aspx). . Retrieved 2011-04-13. "The use of a factory class is one common way to implement DI."
[20] Fowler, Martin (2002). *Patterns of Enterprise Application Architecture* (http://martinfowler.com/books.html#eaa). Addison-Wesley. ISBN 978-0-321-12742-6. .
[21] http://www.eaipatterns.com/MessageTranslator.html
[22] 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.
[23] Binding Properties (http://c2.com/cgi/wiki?BindingProperties)
[24] Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, and Morgan Skinner (2008). "Event-based Asynchronous Pattern". *Professional C# 2008*. Wiley. pp. 570–571. ISBN 0-470-19137-6.
[25] Lock Pattern (http://c2.com/cgi/wiki?LockPattern)
[26] 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.
[27] Fowler, Martin (2006-08-01). "Writing Software Patterns" (http://www.martinfowler.com/articles/writingPatterns.html). . Retrieved 2007-03-06.
[28] Norvig, Peter (1998). "Design Patterns in Dynamic Languages" (http://www.norvig.com/design-patterns/). .
[29] Graham, Paul (2002). "Revenge of the Nerds" (http://www.paulgraham.com/icad.html). . Retrieved 2012-08-11.
[30] McConnell, Steve (2004). *Code Complete: A Practical Handbook of Software Construction, 2nd Edition*. p. 105.

Further reading

- 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 978-0-19-501919-3.
- Alur, Deepak; John Crupi, Dan Malks (May 2003). *Core J2EE Patterns: Best Practices and Design Strategies (2nd Edition)*. Prentice Hall. ISBN 0-13-142246-4.
- Beck, Kent (October 2007). *Implementation Patterns*. Addison-Wesley. ISBN 978-0-321-41309-3.
- 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 978-0-321-12742-6.
- 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. p. 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 0-201-59607-5.

External links

- 101 Design Patterns & Tips for Developers (http://sourcemaking.com/design-patterns-and-tips)
- Design Patterns Reference (http://www.oodesign.com/) at oodesign.com
- Directory of websites that provide pattern catalogs (http://hillside.net/patterns/onlinepatterncatalog.htm) at hillside.net.
- Jt (http://jt.dev.java.net/) J2EE Pattern Oriented Framework
- 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
- 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).
- On Patterns and Pattern Languages (http://media.wiley.com/product_data/excerpt/28/04700590/0470059028.pdf) by Buschmann, Henney, and Schmidt
- Patterns and Anti-Patterns (http://www.dmoz.org/Computers/Programming/Methodologies/Patterns_and_Anti-Patterns//) at the Open Directory Project
- Patterns for Scripted Applications (http://www.doc.ic.ac.uk/~np2/patterns/scripting/)
- 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.
- JPattern (http://jpatterns.org/) JPatterns is a collection of annotations for Design Patterns.
- Printable Design Patterns Quick Reference Cards (http://www.mcdonaldland.info/2007/11/28/40/)
- Are Design Patterns Missing Language Features? at the Portland Pattern Repository
- History of Patterns at the Portland Pattern Repository
- Show Trial of the Gang of Four at the Portland Pattern Repository
- Category: Pattern at the Portland Pattern Repository
- "Design Patterns in Modern Day Software Factories (WCSF)" (http://www.xosoftware.co.uk/Articles/WCSFDesignPatterns/). XO Software, Ltd.
- "Core J2EE Patterns: Patterns index page" (http://java.sun.com/blueprints/corej2eepatterns/Patterns/). Sun Microsystems, Inc.. Retrieved 2012-08-11.

Creational Patterns

Creational pattern

In software engineering, **creational design patterns** are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

Creational design patterns are composed of two dominant ideas. One is encapsulating knowledge about which concrete classes the system use. Another is hiding how instances of these concrete classes are created and combined.[1]

Creational design patterns are further categorized into Object-creational patterns and Class-creational patterns, where Object-creational patterns deal with Object creation and Class-creational patterns deal with Class-instantiation. In greater details, Object-creational patterns defer part of its object creation to another object, while Class-creational patterns defer its objection creation to subclasses.[2]

Five well-known design patterns that are parts of creational patterns are the

- Abstract factory pattern, which provides an interface for creating related or dependent objects without specifying the objects' concrete classes.[3]
- Builder pattern, which separates the construction of a complex object from its representation so that the same construction process can create different representation.
- Factory method pattern, which allows a class to defer instantiation to subclasses.[4]
- Prototype pattern, which specifies the kind of object to create using a prototypical instance, and creates new objects by cloning this prototype.
- Singleton pattern, which ensures that a class only has one instance, and provides a global point of access to it.[5]

Definition

The creational patterns aim to separate a system from how its objects are created, composed, and represented. They increase the system's flexibility in terms of the what, who, how, and when of object creation.[6]

Usage

As modern software engineering depends more on object composition than class inheritance, emphasis shifts away from hard-coding behaviors toward defining a smaller set of basic behaviors that can be composed into more complex ones.[7] Hard-coding behaviors are inflexible because they require overriding or re-implementing the whole thing in order to change parts of the design. Additionally, hard-coding does not promote reuse and is hard to keep track of errors. For these reasons, creational patterns are more useful than hard-coding behaviors. Creational patterns make design become more flexible. They provide different ways (patterns) to remove explicit references in the concrete classes from the code that needs to instantiate them.[8] In other words, they create independency for objects and classes.

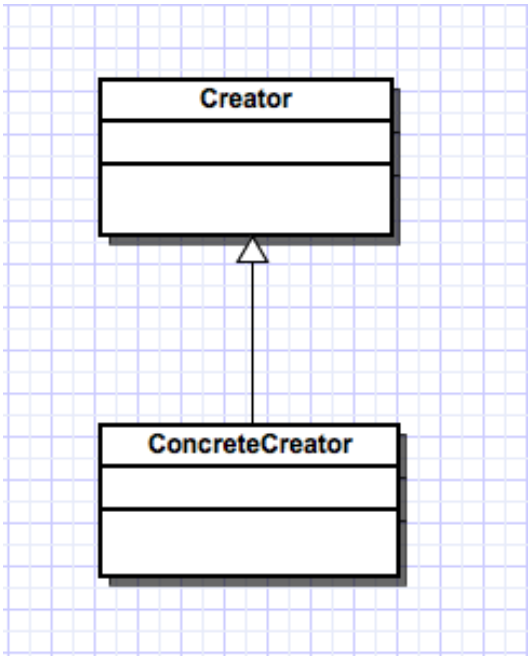
Consider applying creational patterns when:

- A system should be independent of how its objects and products are created.
- A set of related objects is designed to be used together.
- Hiding the implementations of a class library of product, revealing only their interfaces.

- Constructing different representation of independent complex objects.
- A class want its subclass to implement the object it creates.
- The class instantiations are specified at run-time.
- There must be a single instance and client can access this instance at all times.
- Instance should be extensible without being modified.

Structure

Below is a simple class diagram that most creational patterns have in common. Note that different creational patterns require additional and different participated classes.



Creational Pattern class diagram.

Participants:

- **Creator**: Declares object interface. Returns object.
- **ConcreateCreator**: Implements object's interface.

Examples

Some examples of creational design patterns include:

- Abstract factory pattern: centralize decision of what factory to instantiate
- Factory method pattern: centralize creation of an object of a specific type choosing one of several implementations
- Builder pattern: separate the construction of a complex object from its representation so that the same construction process can create different representations
- Lazy initialization pattern: 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
- Object pool pattern: avoid expensive acquisition and release of resources by recycling objects that are no longer in use
- Prototype pattern: used when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects
- Singleton pattern: restrict instantiation of a class to one object

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns*. Massachusetts: Addison-Wesley. p. 81. ISBN 0-201-63361-2.

[2] Gang Of Four

[3] Freeman, Eric; Freeman, Elisabeth; Kathy, Sierra; Bert, Bates (2004). Hendrickson, Mike. ed (in English) (paperback). *Head First Design Patterns* (http://it-ebooks.info/book/252/). **1**. O'REILLY. pp. 156. ISBN 978-0-596-00712-6. . Retrieved 2012-07-23.

[4] Freeman, Eric; Freeman, Elisabeth; Kathy, Sierra; Bert, Bates (2004). Hendrickson, Mike. ed (in English) (paperback). *Head First Design Patterns* (http://it-ebooks.info/book/252/). **1**. O'REILLY. pp. 134. ISBN 978-0-596-00712-6. . Retrieved 2012-07-23.

[5] Freeman, Eric; Freeman, Elisabeth; Kathy, Sierra; Bert, Bates (2004). Hendrickson, Mike. ed (in English) (paperback). *Head First Design Patterns* (http://it-ebooks.info/book/252/). **1**. O'REILLY. pp. 177. ISBN 978-0-596-00712-6. . Retrieved 2012-07-23.

[6] Judith, Bishop (December 2007). *C# 3.0 Design Patterns* (http://shop.oreilly.com/product/9780596527730.do?cmp=af-orm-msdn-book-9780596527730). O'Reilly Media. pp. 336. ISBN 978-0-596-52773-0. .

[7] *Design Patterns*. Massachusetts: Addison Wesley. 1995. p. 84. ISBN 0-201-63361-2.

[8] *Design Patterns*. Massachusetts: Addision Wesley. 1995. p. 85. ISBN 0-201-63361.

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.

A **factory** is the location or a concrete class 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 implementations 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 implementation of the factory.

Definition

The essence of the Abstract Factory method Pattern is to "Provide an interface for creating families of related or dependent objects without specifying their concrete classes".[1]

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.[2]

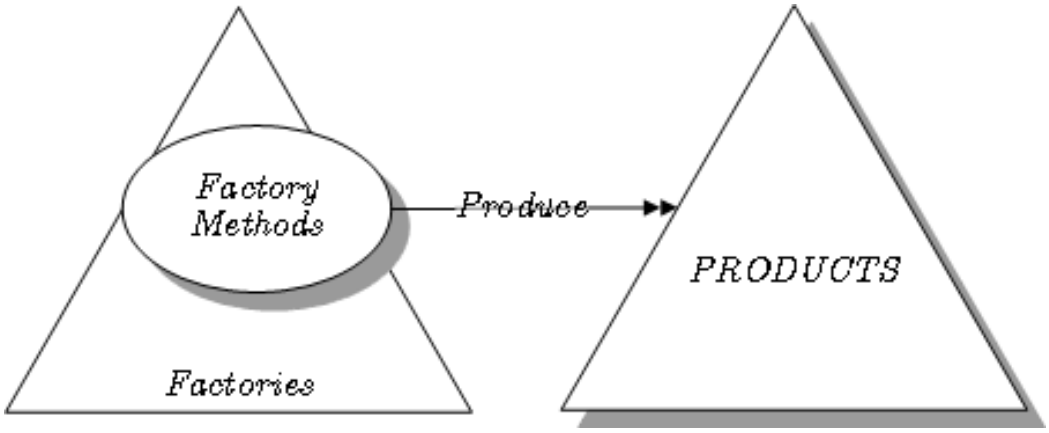
As the factory only returns an abstract pointer, the client code (that requested the object from the factory) does not know – and is not burdened by – the actual concrete type of the object that was just created. However, the type of a concrete object (and hence a concrete factory) is known by the abstract 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 related to it. 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.[3]

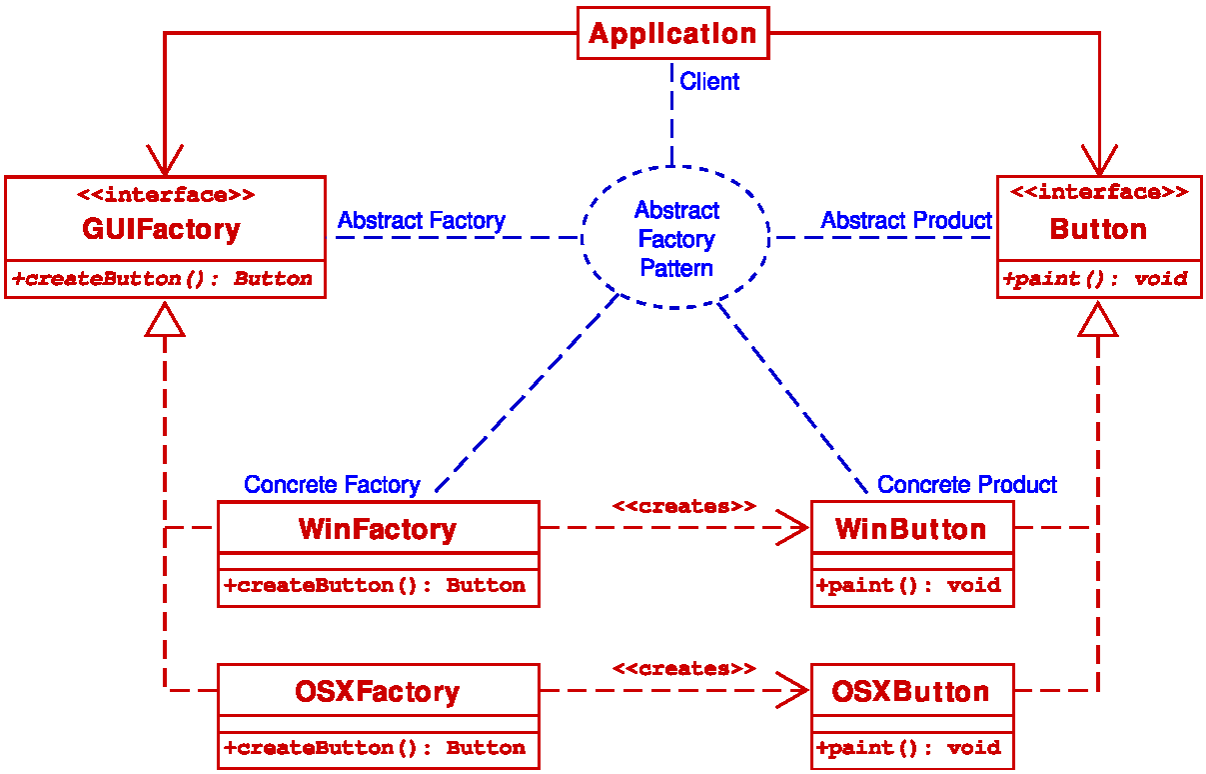
- Adding new concrete types is done by modifying the client code to use a different factory, a modification that is typically one line in one file. (The different factory then creates objects of a *different* concrete type, but still returns a pointer of the *same* abstract type as before – thus insulating 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.[3]

Structure

Lepus3 chart (legend [4])



Class diagram



The method `createButton` on the `GuiFactory` interface returns objects of type `Button`. What implementation of `Button` is returned depends on which implementation of `GuiFactory` is handling the method call.

Note that, for conciseness, this class diagram only shows the class relations for creating one type of object.

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.

C#

```
/* GUIFactory example -- */

using System;
using System.Configuration;

namespace AbstractFactory
{
    public interface IButton
    {
        void Paint();
    }

    public interface IGUIFactory
    {
        IButton CreateButton();
    }

    public class OSXButton : IButton // Executes fourth if OS:OSX
    {
        public void Paint()
        {
            System.Console.WriteLine("I'm an OSXButton");
        }
    }

    public class WinButton : IButton // Executes fourth if OS:WIN
    {
        public void Paint()
        {
            System.Console.WriteLine("I'm a WinButton");
        }
    }

    public class OSXFactory : IGUIFactory // Executes third if OS:OSX
    {
        IButton IGUIFactory.CreateButton()
        {
            return new OSXButton();
```

```
        }
    }

    public class WinFactory : IGUIFactory // Executes third if OS:WIN
    {
        IButton IGUIFactory.CreateButton()
        {
            return new WinButton();
        }
    }

    public class Application
    {
        public Application(IGUIFactory factory)
        {
            IButton button = factory.CreateButton();
            button.Paint();
        }
    }

    public class ApplicationRunner
    {
        static IGUIFactory CreateOsSpecificFactory()  // Executes
second
        {
            // Contents of App{{Not a typo|.}}Config associated with
this C# project
            //<?xml version="1.0" encoding="utf-8" ?>
            //<configuration>
            //  <appSettings>
            //    <!-- Uncomment either Win or OSX OS_TYPE to test -->
            //    <add key="OS_TYPE" value="Win" />
            //    <!-- <add key="OS_TYPE" value="OSX" /> -->
            //  </appSettings>
            //</configuration>
            string sysType =
ConfigurationSettings.AppSettings["OS_TYPE"];
            if (sysType == "Win")
            {
                return new WinFactory();
            }
            else
            {
                return new OSXFactory();
            }
        }
```

```
        static void Main(string[] args) // Executes first
        {
            new Application(CreateOsSpecificFactory());
            Console.ReadLine();
        }
    }
}
```

C++

```
/* GUIFactory example -- */

#include<iostream>
using namespace std;

class Button
{
public:
      virtual void paint() = 0;
    virtual ~Button() { }
};

class WinButton: public Button
{
public:
      void paint()
      {
            cout << "I'm a WinButton";
      }
};

class OSXButton: public Button
{
public:
      void paint()
      {
            cout << "I'm an OSXButton";
      }
};

class GUIFactory
{
public:
      virtual Button* createButton() = 0;
    virtual ~GUIFactory() { }
};

class WinFactory: public GUIFactory
```

```
{
public:
      Button* createButton()
      {
            return new WinButton();
      }

    ~WinFactory() { }
};

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

    ~OSXFactory() { }
};

class Application
{
public:
      Application(GUIFactory* factory)
      {
            Button* button = factory->createButton();
            button->paint();
        delete button;
        delete factory;
      }
};

GUIFactory* createOsSpecificFactory() {
      int sys;
      cout << endl << "Enter OS Type(0 - Win, 1 - OSX): ";
      cin >> sys;

      if (sys == 0) return new WinFactory();
      else return new OSXFactory();
}

int main(int argc, char** argv)
 {
      Application* newApplication = new
Application(createOsSpecificFactory());
      delete newApplication;
      return 0;
}
```

Java

```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();
    }
}
```

Objective-C

```
/* GUIFactory example -- */

#import <Foundation/Foundation.h>

@protocol Button <NSObject>
- (void)paint;
@end

@interface WinButton : NSObject <Button>
@end

@interface OSXButton : NSObject <Button>
@end

@protocol GUIFactory
- (id<Button>)createButton;
@end

@interface WinFactory : NSObject <GUIFactory>
@end

@interface OSXFactory : NSObject <GUIFactory>
@end

@interface Application : NSObject
- (id)initWithGUIFactory:(id)factory;
+ (id)createOsSpecificFactory:(int)type;
@end

@implementation WinButton
- (void)paint {
    NSLog(@"I am a WinButton.");
}
@end

@implementation OSXButton
- (void)paint {
    NSLog(@"I am a OSXButton.");
}
@end
```

```
@implementation WinFactory
- (id<Button>)createButton {
    return [[[WinButton alloc] init] autorelease];
}
@end

@implementation OSXFactory
- (id<Button>)createButton {
    return [[[OSXButton alloc] init] autorelease];
}
@end

@implementation Application
- (id)initWithGUIFactory:(id)factory {
    if (self = [super init]) {
        id button = [factory createButton];
        [button paint];
    }
    return self;
}
+ (id)createOsSpecificFactory:(int)type {
    if (type == 0) {
        return [[[WinFactory alloc] init] autorelease];
    } else {
        return [[[OSXFactory alloc] init] autorelease];
    }
}
@end

int main(int argc, char* argv[]) {
    @autoreleasepool {
        [[Application alloc] initWithGUIFactory:[Application
createOsSpecificFactory:0]];// 0 is WinButton
    }
    return 0;
}
```

Lua

```
--[[
      Because Lua is a highly dynamic Language, an OOP scheme is
implemented by the programmer.
      The OOP scheme implemented here implements interfaces using
documentation.


      A Factory Supports:
       - factory:CreateButton()
```

```
      A Button Supports:
       - button:Paint()
]]


-- Create the OSXButton Class
do
      OSXButton = {}
      local mt = { __index = OSXButton }

      function OSXButton:new()
            local inst = {}
            setmetatable(inst, mt)
            return inst
      end

      function OSXButton:Paint()
            print("I'm a fancy OSX button!")
      end
end

-- Create the WinButton Class
do
      WinButton = {}
      local mt = { __index = WinButton }

      function WinButton:new()
            local inst = {}
            setmetatable(inst, mt)
            return inst
      end

      function WinButton:Paint()
            print("I'm a fancy Windows button!")
      end
end

-- Create the OSXGuiFactory Class
do
      OSXGuiFactory = {}
      local mt = { __index = OSXGuiFactory }

      function OSXGuiFactory:new()
            local inst = {}
            setmetatable(inst, mt)
            return inst
```

```
	end

	function OSXGuiFactory:CreateButton()
		return OSXButton:new()
	end
end

-- Create the WinGuiFactory Class
do
	WinGuiFactory = {}
	local mt = { __index = WinGuiFactory }

	function WinGuiFactory:new()
		local inst = {}
		setmetatable(inst, mt)
		return inst
	end

	function WinGuiFactory:CreateButton()
		return WinButton:new()
	end
end

-- Table to keep track of what GuiFactories are available
GuiFactories = {
	["Win"] = WinGuiFactory,
	["OSX"] = OSXGuiFactory,
}

--[[ Inside an OS config script ]]
OS_VERSION = "Win"

--[[ Using the Abstract Factory in some the application script ]]

-- Selecting the factory based on OS version
MyGuiFactory = GuiFactories[OS_VERSION]:new()

-- Using the factory
osButton = MyGuiFactory:CreateButton()
osButton:Paint()
```

References

[1] [|Gamma, Erich (http://www.informit.com/authors/bio.aspx?a=725735c6-e618-488a-9f9b-a3b8344570dc)]; Richard Helm, Ralph Johnson, John M. Vlissides (2009-10-23). "Design Patterns: Abstract Factory" (http://www.informit.com/articles/article.aspx?p=1398599) (in English) (HTML). informIT. Archived from the original (http://www.informit.com/) on 2009-10-23. . Retrieved 2012-05-16. "Object Creational: Abstract Factory: Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes."

[2] [|Veeneman, David (http://www.codeproject.com/script/Membership/View.aspx?mid=319264)] (2009-10-23). "Object Design for the Perplexed" (http://www.codeproject.com/Articles/4079/Object-Design-for-the-Perplexed) (in English) (HTML). The Code Project. Archived from the original (http://www.codeproject.com/) on 2011-09-18. . Retrieved 2012-05-16. "The factory insulates the client from changes to the product or how it is created, and it can provide this insulation across objects derived from very different abstract interfaces."

[3] "Abstract Factory: Implementation" (http://www.oodesign.com/abstract-factory-pattern.html) (in English) (HTML). OODesign.com. . Retrieved 2012-05-16.

[4] http://www.lepus.org.uk/ref/legend/legend.xml

External links

- Abstract Factory (http://www.lepus.org.uk/ref/companion/AbstractFactory.xml) UML diagram + formal specification in LePUS3 and Class-Z (a Design Description Language)

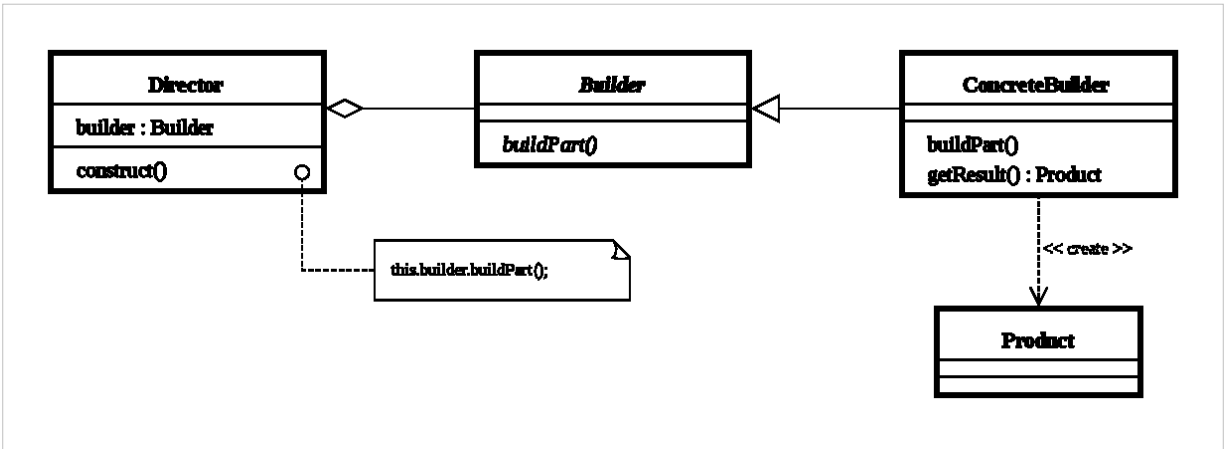
Builder pattern

The **builder pattern** is an object creation 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 with the composite pattern.

Definition

The intent of the Builder design pattern is to separate the construction of a complex object from its representation. By doing so, the same construction process can create different representations. [1]

Structure



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.

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 one of the other patterns to implement which components are built. Abstract Factory, Builder, and Prototype can use Singleton in their implementations.
- Builders are good candidates for a fluent interface.

Examples

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 Waiter {
   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 customer ordering a pizza. */
class BuilderExample {
   public static void main(String[] args) {
       Waiter waiter = new Waiter();
       PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
       PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();

       waiter.setPizzaBuilder( hawaiianPizzaBuilder );
       waiter.constructPizza();

       Pizza pizza = waiter.getPizza();
   }
}
```

C#

```
using System;

namespace BuilderPattern
{
  // Builder - abstract interface for creating objects (the product, in
 this case)
  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();
  }
  // Concrete Builder - provides implementation for Builder; an object
able to construct other objects.
  // Constructs and assembles parts to build the objects
  class HawaiianPizzaBuilder : PizzaBuilder
  {
    public override void BuildDough()
    {
      pizza.Dough = "cross";
    }

    public override void BuildSauce()
    {
      pizza.Sauce = "mild";
    }

    public override void BuildTopping()
    {
      pizza.Topping = "ham+pineapple";
    }
  }
  // Concrete Builder - provides implementation for Builder; an object
able to construct other objects.
  // Constructs and assembles parts to build the objects
  class SpicyPizzaBuilder : PizzaBuilder
  {
    public override void BuildDough()
    {
      pizza.Dough = "pan baked";
    }
```

```
    public override void BuildSauce()
    {
      pizza.Sauce = "hot";
    }

    public override void BuildTopping()
    {
      pizza.Topping = "pepperoni + salami";
    }
  }

  // Director - responsible for managing the correct sequence of object
 creation.
  // Receives a Concrete Builder as a parameter and executes the
necessary operations on it.
  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();
    }
  }

  // Product - The final object that will be created by the Director
using Builder
  public class Pizza
  {
    public string Dough = string.Empty;
    public string Sauce = string.Empty;
    public string Topping = string.Empty;
  }
```

```
  class Program
  {
    static void Main(string[] args)
    {
      PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
      Cook cook = new Cook();
      cook.SetPizzaBuilder(hawaiianPizzaBuilder);
      cook.ConstructPizza();
      // create the product
      Pizza hawaiian = cook.GetPizza();

      PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();
      cook.SetPizzaBuilder(spicyPizzaBuilder);
      cook.ConstructPizza();
      // create another product
      Pizza spicy = cook.GetPizza();
    }
  }

}
```

C++

```
#include <string>
#include <iostream>
using namespace std;

// "Product"
class Pizza {
public:
      void dough(const string& dough) {
            dough_ = dough;
      }

      void sauce(const string& sauce) {
            sauce_ = sauce;
      }

      void topping(const string& topping) {
            topping_ = topping;
      }

      void open() const {
            cout << "Pizza with " << dough_ << " dough, " << sauce_ << " sauce and "
                  << topping_ << " topping. Mmm." << endl;
      }

private:
```

```
      string dough_;
      string sauce_;
      string topping_;
};

// "Abstract Builder"
class PizzaBuilder {
public:
      const Pizza& pizza() {
            return pizza_;
      }

      virtual void buildDough() = 0;
      virtual void buildSauce() = 0;
      virtual void buildTopping() = 0;

protected:
      Pizza pizza_;
};

//-----------------------------------------------------------------

class HawaiianPizzaBuilder : public PizzaBuilder {
public:
      void buildDough() {
            pizza_.dough("cross");
      }

      void buildSauce() {
            pizza_.sauce("mild");
      }

      void buildTopping() {
            pizza_.topping("ham+pineapple");
      }
};

class SpicyPizzaBuilder : public PizzaBuilder {
public:
      void buildDough() {
            pizza_.dough("pan baked");
      }

      void buildSauce() {
            pizza_.sauce("hot");
      }
```

```
	void buildTopping() {
		pizza_.topping("pepperoni+salami");
	}
};

//------------------------------------------------------------------

class Cook {
public:
	Cook()
		: pizzaBuilder_(nullptr)
	{	 }

	~Cook() {
		if (pizzaBuilder_)
			delete pizzaBuilder_;
	}

	void pizzaBuilder(PizzaBuilder* pizzaBuilder) {
		if (pizzaBuilder_)
			delete pizzaBuilder_;

		pizzaBuilder_ = pizzaBuilder;
	}

	const Pizza& getPizza() {
		return pizzaBuilder_->pizza();
	}

	void constructPizza() {
		pizzaBuilder_->buildDough();
		pizzaBuilder_->buildSauce();
		pizzaBuilder_->buildTopping();
	}

private:
	PizzaBuilder* pizzaBuilder_;
};

int main() {
	Cook cook;
	cook.pizzaBuilder(new HawaiianPizzaBuilder);
	cook.constructPizza();

	Pizza hawaiian = cook.getPizza();
	hawaiian.open();
```

```
        cook.pizzaBuilder(new SpicyPizzaBuilder);
        cook.constructPizza();

        Pizza spicy = cook.getPizza();
        spicy.open();
}
```

References

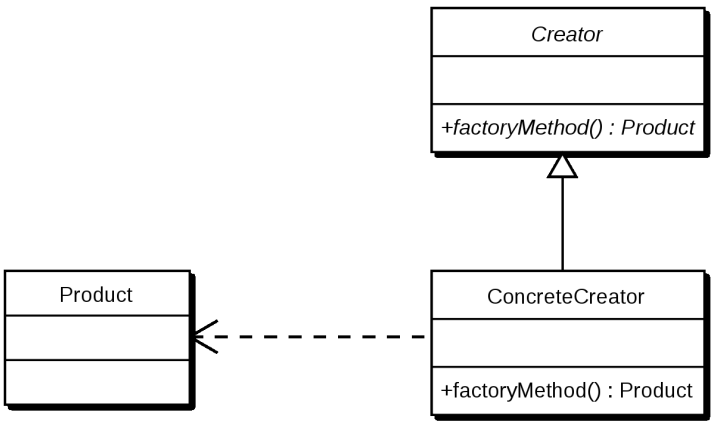
[1] Gang Of Four

External links

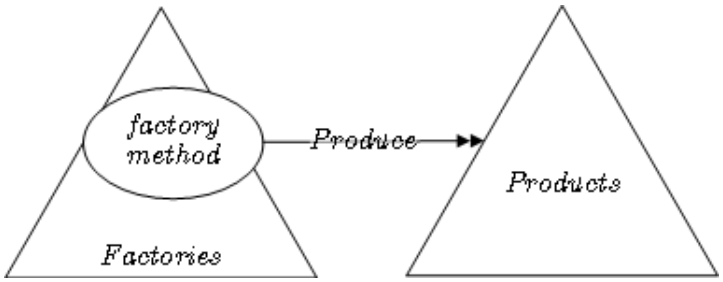
- The JavaWorld article Build user interfaces without getters and setters (http://www.javaworld.com/javaworld/jw-01-2004/jw-0102-toolbox.html) (Allen Holub) shows the complete Java source code for a Builder.
- Item 2: Consider a builder (http://www.ddj.com/java/208403883?pgno=2) by Joshua Bloch

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



Factory method in UML



Factory Method in LePUS3

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.

Applicability

The factory pattern can be used when:

- The creation of an object precludes its reuse without significant duplication of code.
- The creation of an object requires access to information or resources that should not be contained within the composing class.
- The lifetime management of the generated objects must be centralized to ensure a consistent behavior within the application.

Factory methods are common in toolkits and frameworks, where library code needs to create objects of types that 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.[2] 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.

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.

Java

The following example shows the implementation of complex numbers in Java:

```
class Complex {
     public static Complex fromCartesianFactory(double real, double
imaginary) {
          return new Complex(real, imaginary);
     }
     public static Complex fromPolarFactory(double modulus, double
angle) {
          return new Complex(modulus * cos(angle), modulus *
sin(angle));
```

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

Complex product = Complex.fromPolarFactory(1, pi);
```

Strictly speaking this is not an example of the Factory Method pattern (we can not override static), but rather an example of static factory method, which has no direct equivalent in Design Patterns. See for example Effective Java 2nd edition by Joshua Bloch, page 5.

VB.NET

The same example from above follows in VB.NET:

```
Public Class Complex
    Public Shared Function fromCartesianFactory(real As Double,
imaginary As Double) As Complex
        Return (New Complex(real, imaginary))
    End Function

    Public Shared Function fromPolarFactory(modulus As Double, angle As
 Double) As Complex
        Return (New Complex(modulus * Math.Cos(angle), modulus *
Math.Sin(angle)))
    End Function

    Private Sub New(a As Double, b As Double)
        '...
    End Sub
End Class

Complex product = Complex.fromPolarFactory(1, pi);
```

C#

```
   public class Complex
    {
        public double real;
        public double imaginary;

        public static Complex FromCartesianFactory(double real, double
imaginary )
        {
            return new Complex(real, imaginary);
        }

        public static Complex FromPolarFactory(double modulus , double
angle )
```

```
        {
            return new Complex(modulus * Math.Cos(angle), modulus *
Math.Sin(angle));
        }


        private Complex (double real, double imaginary)
        {
            this.real = real;
            this.imaginary = imaginary;
        }
    }

Complex product = Complex.FromPolarFactory(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 that reads image files. The program supports different image formats, represented by a reader class for each format.
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 imageReaderFactoryMethod(InputStream is) {
        ImageReader product = null;

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

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, the `MazeGame` constructor is a template method that makes some common logic. It refers to the `makeRoom` factory method that 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 {
  @Override
  protected Room makeRoom() {
      return new MagicRoom();
  }
}
```

PHP

Another example in PHP follows:

```
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. The reflection features of some languages can obviate this issue.

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

Uses

- In ADO.NET, IDbCommand.CreateParameter [4] is an example of the use of factory method to connect parallel class hierarchies.
- In Qt, QMainWindow::createPopupMenu [5] is a factory method declared in a framework that can be overridden in application code.
- In Java, several factories are used in the javax.xml.parsers [6] 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 978-0-13-117705-5

[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://download.oracle.com/javase/1.5.0/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 978-0-201-10393-9.
- 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 in UML and in LePUS3 (http://www.lepus.org.uk/ref/companion/FactoryMethod.xml) (a Design Description Language)
- Consider static factory methods (http://drdobbs.com/java/208403883) by Joshua Bloch

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

Actionscript 3

The following is an example of a class with Lazy initialization implemented in Actionscript:

```
package examples.lazyinstantiation
{
      public class Fruit
      {
            private var _typeName:String;
            private static var instancesByTypeName:Dictionary = new
Dictionary();

            public function Fruit(typeName:String):void
            {
                  this._typeName = typeName;
            }

            public function get typeName():String
            {
                  return _typeName;
            }

            public static function
getFruitByTypeName(typeName:String):Fruit
            {
                  return instancesByTypeName[typeName] ||= new
Fruit(typeName);
            }
```

```
            public static function printCurrentTypes():void
            {
                  for each (var fruit:Fruit in instancesByTypeName)
                  {
                        // iterates through each value
                        trace(fruit.typeName);
                  }
            }
      }
}
```

Basic Usage:

```
package
{
      import examples.lazyinstantiation;

      public class Main
      {
            public function Main():void
            {
                  Fruit.getFruitByTypeName("Banana");
                  Fruit.printCurrentTypes();

                  Fruit.getFruitByTypeName("Apple");
                  Fruit.printCurrentTypes();

                  Fruit.getFruitByTypeName("Banana");
                  Fruit.printCurrentTypes();
            }
      }
}
```

C#

In .NET 4.0 Microsoft has included a `Lazy` class that can be used to do lazy loading. Below is some dummy code that does lazy loading of Class `Fruit`

```
Lazy<Fruit> lazyFruit = new Lazy<Fruit>();
Fruit fruit = lazyFruit.Value;
```

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{{Not a typo|.}}Count > 0)
        {
            Console.WriteLine("Number of instances made = {0}",
_typesDictionary.Count);

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

            Console.WriteLine();
        }
    }

    public Fruit()
    {
        // required so the sample compiles
    }
}

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();
    }
}
```

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) {
    map<string,Fruit*>::iterator it = types.find(type);  // try to find an existing
instance; if not found std::map will return types.end()

    Fruit *f;
    if (it == types.end()) { // 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
    } else { //if already had an instance
        f = it->second; //The return value will be the found fruit
    }
    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
*/
```

Java

Here is an example in Java.

```
import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;

public enum FRUIT_TYPE {
	NONE,
	APPLE,
	BANANA,
}

class Program {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		Fruit.getFruitByTypeName(FRUIT_TYPE.BANANA);
		Fruit.showAll();
		Fruit.getFruitByTypeName(FRUIT_TYPE.APPLE);
		Fruit.showAll();
		Fruit.getFruitByTypeName(FRUIT_TYPE.BANANA);
		Fruit.showAll();
	}
}

public class Fruit {
	private static Map<FRUIT_TYPE, Fruit> types = new HashMap<FRUIT_TYPE, Fruit>();

	/**
	 * Using a private constructor to force the use of the factory
```

```
method.
       * @param type
       */
      private Fruit(FRUIT_TYPE type) {
      }

      /**
       * Lazy Factory method, gets the Fruit instance associated with a
 certain
       * type. Instantiates new ones as needed.
       * @param type Any allowed fruit type, e.g. APPLE
       * @return The Fruit instance associated with that type.
       */
      public static Fruit getFruitByTypeName(FRUIT_TYPE type) {
            Fruit fruit;

            if (!types.containsKey(type)) {
                  // Lazy initialisation
                  fruit = new Fruit(type);
                  types.put(type, fruit);
            } else {
                  // OK, it's available currently
                  fruit = types.get(type);
            }

            return fruit;
      }

      /**
       * Lazy Factory method, gets the Fruit instance associated with a
 certain
        * type. Instantiates new ones as needed. Uses double-checked
locking
       * pattern for using in highly concurrent environments.
       * @param type Any allowed fruit type, e.g. APPLE
       * @return The Fruit instance associated with that type.
       */
      public static Fruit
getFruitByTypeNameHighConcurrentVersion(FRUIT_TYPE type) {
            if (!types.containsKey(type)) {
                  synchronized (types) {
                        // Check again, after having acquired the lock
to make sure
                        // the instance was not created meanwhile by
another thread
                        if (!types.containsKey(type)) {
                              // Lazy initialisation
```

```
                              types.put(type, new Fruit(type));
                        }
                  }
            }

            return types.get(type);
      }

      /**
       * Displays all entered fruits.
       */
      public static void showAll() {
            if (types.size() > 0) {
                  System.out.println("Number of instances made = " +
types.size());

                  for (Entry<FRUIT_TYPE, Fruit> entry : types.entrySet()) {
                        System.out.println(

Constants.firstLetterToUpper(entry.getKey().toString()));
                  }

                  System.out.println();
            }
      }
}
```

Output

```
Number of instances made = 1
Banana

Number of instances made = 2
Banana
Apple

Number of instances made = 2
Banana
Apple
```

JavaScript

Here is an example in JavaScript.

```
var Fruit = (function() {
  var types = {};
  function Fruit() {};

  // count 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 (typeof 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
```

PHP

Here is an example of lazy initialization in PHP 5:

```php
<?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::$types)) {
            return self::$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
*/

?>
```

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')
```

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
```

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].
```

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.

References

External links

- Article " Java Tip 67: Lazy instantiation (http://www.javaworld.com/javaworld/javatips/jw-javatip67.html) - Balancing performance and resource usage" by Philip Bishop and Nigel Warren
- Java code examples (http://javapractices.com/Topic34.cjp)
- Use Lazy Initialization to Conserve Resources (http://devx.com/tips/Tip/18007)
- Description from the Portland Pattern Repository (http://c2.com/cgi/wiki?LazyInitialization)
- Lazy Initialization of Application Server Services (http://weblogs.java.net/blog/binod/archive/2005/09/lazy_initializa.html)
- Lazy Inheritance in JavaScript (http://sourceforge.net/projects/jsiner)
- Lazy Inheritance in C# (http://labs.kaliko.com/2011/04/lazy-loading-in-c-net.html)

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:

This code is wrong, it could result in concurrent get() and put() which is not allowed for a HashMap.

```
    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 ...

    }
```

Action Script 3.0/ Flex

```
    import flash.utils.Dictionary;

    public class InternalModelLocator {
            private static var instances:Dictionary = new Dictionary();

            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;
            }
      }
```

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

```
                return instance;
            }
        }
    }
}
```

C++

Implementation from StackOverflow [1]

```
#ifndef MULTITON_H
#define MULTITON_H

#include <map>
#include <string>

template <typename T, typename Key = std::string>
class Multiton
{
public:
    static void destroy()
    {
        for (typename std::map<Key, T*>::const_iterator it = instances.begin();
it != instances.end(); ++it)
            delete (*it).second;
        instances.clear();
    }

    static T* getPtr(const Key& key) {
        typename std::map<Key, T*>::const_iterator it = instances.find(key);

        if (it != instances.end()) {
            return (T*)(it->second);
        }

        T* instance = new T();
        instances[key] = instance;
        return instance;
    }

    static T& getRef(const Key& key) {
        return *getPtr(key);
    }

protected:
    Multiton() {}
    ~Multiton() {}

private:
```

```
    Multiton(const Multiton&) {}
    Multiton& operator= (const Multiton&) { return *this; }

    static std::map<Key, T*> instances;
};

template <typename Key, typename T>
std::map<Key, T*> Multiton<Key, T>::instances;

#endif

// example usage
class Foo : public Multiton<Foo> {};
Foo& foo1 = Foo::getRef("foobar");
Foo* foo2 = Foo::getPtr("foobar");
Foo::destroy();
```

PHP

```
<?php
//orochi
// 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
            $key = get_called_class() . serialize(func_get_args());
            if (!isset(self::$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
```

Python

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

multiton = {}
a0 = multiton.setdefault('a0', A()) # get object by key, or create new
and return it
a1 = multiton.setdefault('a1', A())
print multiton.get('a0')
print multiton.get('a1')
```

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
```

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[2], as it introduces global state into an application.

With garbage collected languages it may become a source of memory leaks as it introduces global strong references to the objects.

References

[1] http://stackoverflow.com/questions/2346091/c-templated-class-implementation-of-the-multiton-pattern
[2] 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)
- 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)

The **object pool pattern** is a software creational design pattern that uses a set of initialised objects kept ready to use, rather than allocating and destroying them 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, it returns the object, which is a specific type of factory object, to the pool rather than destroying it.

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 that 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 (that 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.

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 ten 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.

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

[1] Goetz, Brian (2005-09-27). "Java theory and practice: Urban performance legends, revisited" (http://www.ibm.com/developerworks/java/library/j-jtp09275/index.html). IBM developerWorks. Archived from the original (http://www.ibm.com/) on 2005-09-27. . Retrieved 2012-08-28.

[2] Goetz, Brian (2005-09-27). "Java theory and practice: Garbage collection in the HotSpot JVM" (http://www.ibm.com/developerworks/java/library/j-jtp11253/). IBM developerWorks. Archived from the original (http://www.ibm.com/) on 2003-11-25. . Retrieved 2012-08-28.

- Kircher, Michael; Prashant Jain; (2002-07-04). "Pooling Pattern" (http://www.kircher-schwanninger.de/michael/publications/Pooling.pdf). *EuroPLoP 2002*. Germany. Retrieved 2007-06-09.

External links

- OODesign article (http://www.oodesign.com/object-pool-pattern.html)
- Improving Performance with Object Pooling (Microsoft Developer Network) (http://msdn2.microsoft.com/en-us/library/ms682822.aspx)
- Developer.com article (http://www.developer.com/tech/article.php/626171/Pattern-Summaries-Object-Pool.htm)
- Portland Pattern Repository entry (http://c2.com/cgi-bin/wiki?ObjectPoolPattern)
- Apache Commons Pool: A mini-framework to correctly implement object pooling in Java (http://commons.apache.org/pool/)
- Game Programming Patterns: Object Pool (http://gameprogrammingpatterns.com/object-pool.html)

Prototype pattern

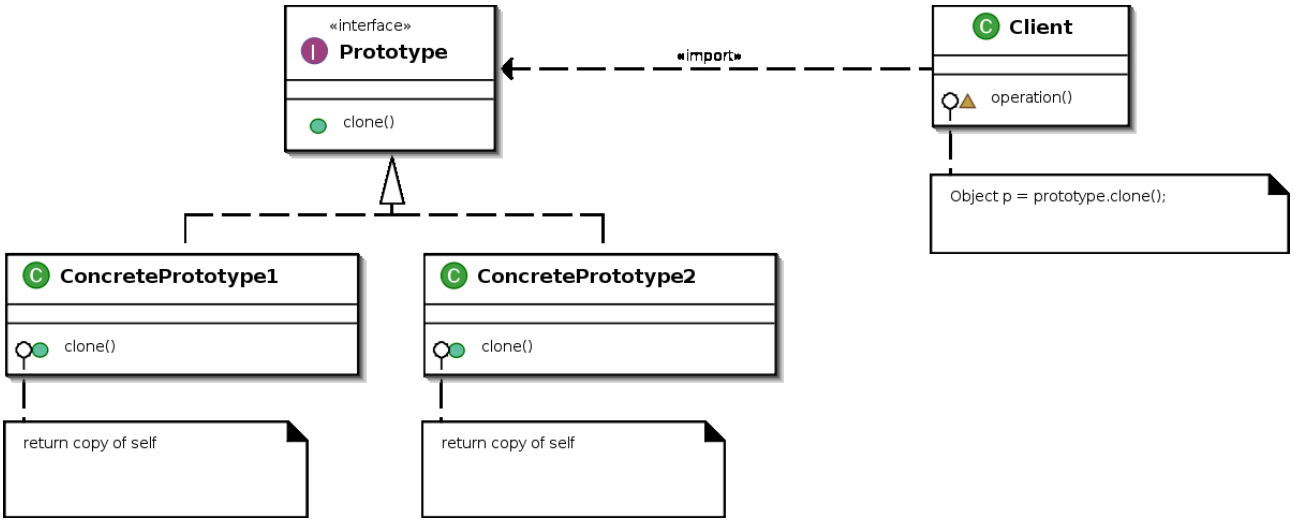
The **prototype pattern** is a creational design pattern used in software development when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects. This pattern is used to:

- 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.

Structure



UML class diagram describing the Prototype 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.[1]

Java

```
/**
 * Prototype class
 */
abstract class Prototype implements Cloneable {
      @Override
```

```
	public Prototype clone() throws CloneNotSupportedException {
		return (Prototype)super.clone();
	}

	public abstract void setX(int x);

	public abstract void printX();

	public abstract int getX();
}

/**
 * Implementation of prototype class
 */
class PrototypeImpl extends Prototype {
	int x;

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

	public void setX(int x) {
		this.x = x;
	}

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

	public int getX() {
		return x;
	}
}

/**
 * Client code
 */
public class PrototypeTest {
	public static void main(String args[]) throws
CloneNotSupportedException {
		Prototype prototype = new PrototypeImpl(1000);

		for (int i = 1; i < 10; i++) {
			Prototype tempotype =  prototype.clone();

			// Usage of values in prototype to derive a new
value.
```

```
                    tempotype.setX( tempotype.getX() * i);
                    tempotype.printX();
                }
        }
}

/*
**Code output**

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

C#

```
//Note: In this example ICloneable interface (defined in .Net
Framework) acts as Prototype

class ConcretePrototype : ICloneable
{
    public int X { get; set; }

    public ConcretePrototype(int x)
    {
        this.X = x;
    }

    public void PrintX()
    {
        Console.WriteLine("Value :" + X);
    }

    public object Clone()
    {
        return this.MemberwiseClone();
    }
}

/**
* Client code
*/
```

```
public class PrototypeTest
{
    public static void Main()
    {
        var prototype = new ConcretePrototype(1000);

        for (int i = 1; i < 10; i++)
        {
            ConcretePrototype tempotype = prototype.Clone() as
ConcretePrototype;

            // Usage of values in prototype to derive a new value.
            tempotype.X *= i;
            tempotype.PrintX();
        }
        Console.ReadKey();
    }
}

/*
**Code output**

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

PHP

```
<?php
class ConcretePrototype
{
    private $x;

    public function __construct($p_valueX)
    {
        $this->x = (int) $p_valueX;
    }

    public function printX()
    {
        print sprintf('Value: %5d' . PHP_EOL, $this->x);
```

```
    }

    public function setX($p_valueX)
    {
        $this->x *= (int) $p_valueX;
    }

    public function __clone()
    {
        /*
        * This method is not required for cloning, although when
implemented,
        * PHP will trigger it after the process in order to permit you
some
        * change in the cloned object.
        *
        * Reference: http://php.net/manual/en/language.oop5.cloning.php
        */
        // $this->x = 1;
    }
}

/**
* Client code
*/
$prototype = new ConcretePrototype(1000);

foreach (range(1, 10) as $i) {
    $tempotype = clone $prototype;
    $tempotype->setX($i);
    $tempotype->printX();
}

/*
**Code output**
Value:  1000
Value:  2000
Value:  3000
Value:  4000
Value:  5000
Value:  6000
Value:  7000
Value:  8000
Value:  9000
Value: 10000
*/
```

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.

Resource Acquisition Is Initialization

Resource Acquisition Is Initialization (**RAII**, sometimes RIIA) is a programming idiom used in several object-oriented languages like C++, D and Ada. The technique was invented by Bjarne Stroustrup[1] to deal with resource allocation and deallocation in C++. In this language, the only code that can be guaranteed to be executed after an exception is thrown are the destructors of objects residing on the stack. Resource management therefore needs to be tied to the lifespan of suitable objects in order to gain automatic allocation and reclamation. They are acquired during initialization, when there is no chance of them being used before they are available, and released with the destruction of the same objects, which is guaranteed to take place even in case of errors.

RAII is vital in writing exception-safe C++ code: to release resources before permitting exceptions to propagate (in order to avoid resource leaks) one can write appropriate destructors once rather than dispersing and duplicating cleanup logic between exception handling blocks that may or may not be executed.

Language support

C++ and D allow objects to be allocated on the stack and their scoping rules ensure that destructors are called when a local object's scope ends. By putting the resource release logic in the destructor, C++'s and D's scoping provide direct support for RAII.

The C language does not directly support RAII, though there are some ad-hoc mechanisms available to emulate it. However, some compilers provide non-standard extensions that implement RAII. For example, the "cleanup" variable attribute extension of GCC is one of them.

Typical uses

The RAII technique is often used for controlling mutex locks in multi-threaded applications. In that use, the object releases the lock when destroyed. Without RAII in this scenario the potential for deadlock would be high and the logic to lock the mutex would be far from the logic to unlock it. With RAII, the code that locks the mutex essentially includes the logic that the lock will be released when the RAII object goes out of scope. Another typical example is interacting with files: We could have an object that represents a file that is open for writing, wherein the file is opened in the constructor and closed when the object goes out of scope. In both cases, RAII only ensures that the resource in question is released appropriately; care must still be taken to maintain exception safety. If the code modifying the data structure or file is not exception-safe, the mutex could be unlocked or the file closed with the data structure or file corrupted.

The ownership of dynamically allocated memory (such as memory allocated with `new` in C++ code) can be controlled with RAII, such that the memory is released when the RAII object is destroyed. For this purpose, the C++ Standard Library defines the smart pointer class `std::auto_ptr`, deprecated in C++11 in favor of `std::unique_ptr`. Furthermore, smart pointer with shared-ownership semantics such as `tr1::shared_ptr` (defined in C++ by TR1 and marked for inclusion in the current C++11 standard), or policy based smart pointers such as `Loki::SmartPtr` (from Loki), can also be used to manage the lifetime of shared objects.

C++ example

The following RAII class is a lightweight wrapper of the C standard library file system calls.

```
#include <cstdio>
#include <stdexcept> // std::runtime_error
class file {
public:
    file(const char* filename)
        : file_(std::fopen(filename, "w+")) {
        if (!file_) {
            throw std::runtime_error("file open failure");
        }
    }

    ~file() {
        if (std::fclose(file_)) {
           // failed to flush latest changes.
           // handle it
        }
    }

    void write(const char* str) {
        if (EOF == std::fputs(str, file_)) {
            throw std::runtime_error("file write failure");
        }
    }

private:
    std::FILE* file_;

    // prevent copying and assignment; not implemented
    file(const file &);
    file& operator=(const file &);
};
```

The class `file` can then be used as follows:

```
void example_usage() {
    file logfile("logfile.txt"); // open file (acquire resource)
    logfile.write("hello logfile!");
    // continue using logfile ...
    // throw exceptions or return without
    //  worrying about closing the log;
    // it is closed automatically when
    // logfile goes out of scope
}
```

This works because the class `file` encapsulates the management of the `FILE*` file handle. When `file` objects are local to a function, C++ guarantees that they are destroyed at the end of the enclosing scope (the function in the

example), and the `file` destructor releases the file by calling `std::fclose(file_)`. Furthermore, `file` instances guarantee that a file is available by throwing an exception if the file could not be opened when creating the object.

Local variables easily manage multiple resources within a single function: They are destroyed in the reverse order of their construction, and an object is only destroyed if fully constructed. That is, if no exception propagates from its constructor.

Using RAII-enabled resources simplifies and reduces overall code size and helps ensure program correctness. Notice that the C++ standard library contains RAII-enabled file I/O in the fstream header and the above file class is only provided as an example of the concept.

C example using GCC extensions

The GNU Compiler Collection implements a non-standard extension to the C language that allows it to support RAII: the "cleanup" variable attribute [2]. For example, the following macro defines a variable with a given type and attaches a function to it that will be called when the variable goes out of scope:

```
#define RAII_VARIABLE(vartype,varname,initval,dtor) \
    void _dtor_ ## varname (vartype * v) { dtor(*v); } \
    vartype varname __attribute__((cleanup(_dtor_ ## varname))) =
(initval)
```

This macro can then be used as follows:

```
void example_usage() {
  RAII_VARIABLE(FILE*, logfile, fopen("logfile.txt", "w+"), fclose);
  fputs("hello logfile!", logfile);
}
```

In this example, the compiler arranges for the *fclose()* function to be called before *example_usage()* returns.

Resource management without RAII

Finalizers

In Java, objects are not allocated on the stack and must be accessed through references; hence, one cannot have automatic variables of objects that "go out of scope". Instead, all objects are dynamically allocated. In principle, dynamic allocation does not make RAII unfeasible per se; it could still be feasible if there were a guarantee that a "destructor" ("finalize") method would be called as soon as an object were pointed to by no references (i.e., if the object lifetime management were performed according to reference counting).

However, Java objects have indefinite lifetimes which cannot be controlled by the programmer, because, according to the Java Virtual Machine specification, it is unpredictable when the garbage collector will act. Indeed, the garbage collector may never act at all to collect objects pointed to by no references. Hence the "finalize" method of an unreferenced object might never be called or be called long after the object became unreferenced. Resources must thus be closed manually by the programmer, using something like the dispose pattern.

In Java versions prior to Java 7, the preceding example would be written like this:

```
class JavaExample {

    void exampleMethod() {
        // open file (acquire resource)
```

```
        final LogFile logfile = new LogFile("logfile.txt");

        try {
            logfile.write("hello logfile!");

            // continue using logfile ...
            // throw exceptions or return without worrying about
closing the log;
            // it is closed automatically when exiting this block
        } finally {
            // explicitly release the resource
            logfile.close();
        }
    }

}
```

In Java 7 and later, using the new try-with-resources construct, this pattern can be declared a bit more concisely (but is still essentially equivalent to the above code, assuming no exceptions are thrown by the close() method):

```
class JavaExample {

    void exampleMethod() {
        // open file (acquire resource)
        try (LogFile logfile = new LogFile("logfile.txt")) {
            logfile.write("hello logfile!");

            // continue using logfile ...
            // throw exceptions or return without worrying about
closing the log;
            // it is closed automatically when exiting this block
        }
    }

}
```

(The class used in the statement must implement `java.lang.AutoCloseable`.) In both cases, the burden of releasing resources falls on the programmer each time a resource is used: The programmer must either explicitly write `finally` blocks, or must remember to construct the object in the "try-with-resources" construct in order to ensure that the resource is released.

Note, however, that at least one of the benefits of RAII-style coding is still applicable to Java: Construction and initialization can be tightly coupled. For instance, in the above examples, note that it is impossible for code to refer to the variable `logfile` before it points to a valid object, since the variable `logfile` is declared explicitly `final` in the first example (and is implicitly final in the second due to the semantics of the try-with-resources construct) and that in both cases it is initialized on the same line as it is declared, with a `new` expression that both creates and initializes the object it will reference.

Closure blocks

Ruby and Smalltalk do not support RAII, but have a pattern that makes use of methods that pass resources to closure blocks. Here is an example in Ruby:

```
File.open("logfile.txt", "w+") do |logfile|
   logfile.write("hello logfile!")
end
```

The `open` method ensures that the file handle is closed without special precautions by the code writing to the file. This is similar to Common Lisp's `unwind-protect` [3]-based macros:

```
(with-open-file (logfile "logfile.txt" :direction :output :if-exists
:append)
  (format logfile "hello logfile!"))
```

Python's `with` statement [4] and *context manager* system is also similar and provides deterministic resource management within a block, doing away with the requirement for explicit `finally`-based cleanup and release.

```
with open("log.txt", 'r') as logfile:
    logfile.write("hello logfile!")
```

Before entering the block, the context manager (in this case, a `file` object) has its `__enter__()` method called. Before exiting the block (including when this is due to an exception being raised), the context manager's `__exit__()` method is called.

In C#, the same goal is accomplished by wrapping any object that implements the `IDisposable` interface in a `using` statement. When execution leaves the scope of the `using` statement body, the `Dispose` method on the wrapped object is executed giving it a deterministic way to clean up any resources.

```
public class CSharpExample {
    public static void Main() {
        using (FileStream fs = new FileStream("log.txt",
FileMode.OpenOrCreate)) {
            using (StreamWriter log = new StreamWriter(fs)) {
                log.WriteLine("hello logfile!");
            }
        }
    }
}
```

Visual Basic 2005 also supports the `using` statement.

Disadvantages of scope bound resource management alternatives

Where both finalizers and closure blocks work as a good alternative to RAII for "shallow" resources, it is important to note however that the compositional properties of RAII differ greatly from these scope bound forms of resource management. Where RAII allows for full encapsulation of resources behind an abstraction, with scope bound resource management this isn't the case. In an environment that purely depends on scope bound resource management, "being a resource" becomes a property that is transitive to composition. That is, using only scope bound resource management, any object that is composed using a resource that requires resource management effectively itself becomes a resource that requires resource management. RAII effectively breaks the transitivity of this property allowing for the existence of "deep" resources to be effectively abstracted away.

Let's for example say we have an object of type A that by composition holds an object of type B that by composition holds an object of type C. Now lets see what happens when we create a new implementation of C that by composition holds a resource R. R will have some `close` or `release` method that must be invoked prior to C going out of scope. We could make C into a RAII object for R that invokes `release` on destruction.

Basically now we have the situation where from the point of view of R and C (shallow resources), scope bound resource management alternatives functionally are fully equivalent to RAII. From a point of view of A and B (deep resources) however, we see a difference in the transitivity of "being a resource" emerging. With C as a RAII object, the interface and implementation of A, B and any code using a scoped A or B will remain unchanged and unaware of the newly introduced existence of "deep" resources. Without RAII however, the fact that C holds R means that C will need its own `release` method for proxying the `release` of R. B will now need its own `release` method for proxying the release of C. A will need its own `release` method for proxying the release of B, and the scope where either A or B is used will also require of the alternative resource management techniques, where with RAII C provides the abstraction barrier that hides the implementation detail of "implemented using a resource" from the view of A, B and any users of an A or B. This difference shows that RAII effectively breaks the compositional transitivity of the "being a resource" property.

Additionally, the scope-based alternatives do not operate in the same manner as RAII for objects with expression lifetimes. This causes the same lack of transparency mentioned above for object containment with RAII alternatives – that objects must know the resource status of their contained objects – to also be a problem for expressions that return anonymous or temporary objects. Expressions must know if any of their subexpressions return or evaluate to a resource. In languages with RAII, many idioms take advantage of automatic destruction routines, such as expression templates, stream formatters, and function call wrappers. The lack of resource transparency at the expression level means that one cannot write generic or polymorphic code except through promoting all expressions to resources and implementing disposal for expressions that do not themselves have resource semantics. These idioms then lose the benefit of the information hiding and syntactic ease of use that they are often created to provide.

Reference counting

Perl and CPython[5] manage object lifetime by reference counting, making it possible to use RAII in a limited form. Objects that are no longer referenced are immediately released, so a destructor can release the resource at that time. However, object lifetime isn't necessarily bound to any lexical scope. One can store a reference to an object in a global variable, for example, thus keeping the object (and resource) alive indeterminately long. This makes it possible to accidentally leak resources that should have been released at the end of some scope. Circular references can also cause an object's reference count to never drop to zero; another potential memory leak under reference counting. Also, in the case of Python, the actual garbage collection strategy is an implementation detail, and running with an alternative interpreter (such as IronPython or Jython) could result in the RAII implementation not working.

Ad-hoc mechanisms

Languages with no direct support for defining constructors and destructors sometimes still have the ability to support ad-hoc RAII methods using programmer idioms. C, for example, lacks direct support for these features, yet programming languages with such support are often implemented in C, and fragile support can be achieved in C itself, although at the cost of additional cognitive overhead on the part of the programmer.

The lack of any standardized idiom for dynamic object allocation beyond malloc and free can be addressed numerous ways, such as through a programming idiom of having TNew, TDelete, and T*method* functions for any given type T. The ability of **throw** to do non-local jumps is analogous to the C-standard setjmp/longjmp library functions, and the syntactic sugar needed to make the result readable can be injected using the C preprocessor. Using variadic macros and in-expression variable declarations makes code like the following possible.

```c
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

jmp_buf       _jmp_bufs[100]; /* avoid recursion with funcs using
try/RAII */
int           _jmp_i = 0;       /* _jmp_bufs[_jmp_i] is the next to use
 */
const char   *_jmp_text = 0;

/* a variant of Catch without "break;" might be useful for partial
cleanups */
#define Try        do{ switch( setjmp(_jmp_bufs[_jmp_i++]) ){ case 0:
while(1)
#define Throw(x)        (longjmp(_jmp_bufs[--_jmp_i], x))
#define ThrowText(x,s) (_jmp_text = s, longjmp(_jmp_bufs[--_jmp_i], x))
#define CatchText()     (_jmp_text )
#define Catch(x)       break; case x:
#define Finally        break; default:
#define Done           } }while(0)

enum { ExceptionOpen = 1 /* must be =1 */, ExceptionWrite,
ExceptionClose };

/* In GCC, use option "-std=c99"
 * Avoid using 'return', 'break', 'goto' to escape RAII blocks early!
 * Without the exception handling, this macro would be simpler:
 *   #define RAII(t, v, a...) for(t *v = t ## New(a, __VA_ARGS__) ; v ;
 t ## Delete(&v))
 */
#define RAII(t, v, a,...) \
    for(int _outer_once = 1, _inner_once = 1, _jumped = 0 ;
_outer_once-- ; \
        _jumped && (Throw(_jumped), 1), 1)
                                \
        for(_jumped = setjmp(_jmp_bufs[_jmp_i++]) ; _inner_once-- ; )
   \
            for( t *v = t ## New(a, __VA_ARGS__) ; v ; t ## Delete(&v))

typedef struct _File { FILE *fp; } File;

void FileDelete(File **faddr) {
    if((*faddr)->fp)
        if(EOF == fclose((*faddr)->fp))
            Throw(ExceptionClose);
    free(*faddr);
    *faddr = 0;
```

```
}

File *FileNew(const char *pathname, char *mode) {
    File *f;
    if(f = malloc(1,sizeof(FILE))) {
        if( ! (f->fp = fopen(pathname, mode))) {
            FileDelete(&f);
            ThrowText(ExceptionOpen, pathname);
        }
    }

    return f;
}

void FilePutc(File *f, int c) {
    if(EOF == fputc(c, f->fp)) Throw(ExceptionWrite);
}

int FileGetc(File *f) { return fgetc(f->fp); }

int main(int ac, char **av) {
    Try {
        RAII(File, from, "file-from", "r+") { /* r+ to ban opening dirs
 */
            RAII(File, to, "file-to", "w") {
                int c;
                while(EOF != (c = FileGetc(from)))
                    FilePutc(to, c);
                puts("copy complete");
            }
        }
    }
    Catch(ExceptionOpen)  { printf(">>> open error on \"%s\"\n",
CatchText()); }
    Catch(ExceptionClose) { puts(">>> close error!"); }
    Catch(ExceptionWrite) { puts(">>> write error!"); }
    Finally { puts("finally :-)"); } Done;
    return 0;
}
```

The use of goto,[6] is a common C idiom, used notably in the Linux kernel to do RAII. Each label acts as an entry-point into the chain of deinitialization that takes place at the end of the function, preventing not-yet-allocated resources from being released. At the cost of verbosity and potential repetition, it avoids exceptions, dynamic memory, and ad-hoc object systems.

```
#include <stdio.h>

int main(int ac, char **av) {
```

```
    int ret = EXIT_FAILURE;
    FILE *from, *to;

    if((from = fopen("file-from", "r+"))==NULL)
        goto from_failed;
    if((to = fopen("file-to", "w"))==NULL)
        goto to_failed;

    int c;
    while(EOF != (c = fgetc(from))) {
        if(EOF == fputc(c, to))
            goto putc_failed;
    }
    puts("copy complete");

success:
    ret = EXIT_SUCCESS;
putc_failed:
    fclose(to);
to_failed:
    fclose(from);
from_failed:

    return ret;
}
```

There are many different, often partial, solutions to the problem, such as handling object cleanup with goto, code that creates objects for the exceptions,[7] or addresses multithreading issues.[8] The real weakness of not having direct language support for RAII features is in situations where code from multiple sources is being combined, and all the ad-hoc methods being used fail to interact properly, or worse yet, directly conflict with each other.

Even where mixing with other ad-hoc approaches can be obviated, there is still an issue of questionable code resilience. Ad-hocs are typically specific to individual projects and usually haven't been subjected to the same rigorous testing and code review that RAII and exception mechanisms are when supported as a primary language feature. A project relying heavily on these types of patterns might reasonably use the exception handling of an internationally supported C++ compiler, rather than an ad-hoc C mechanism such as the example provided above, or even mechanisms constructed in house by a single team.

References

[1] Stroustrup, Bjarne (1994). *The Design and Evolution of C++*. Addison-Wesley. ISBN 0-201-54330-3.

[2] http://gcc.gnu.org/onlinedocs/gcc/Variable-Attributes.html

[3] http://www.lispworks.com/documentation/HyperSpec/Body/s_unwind.htm

[4] http://www.python.org/dev/peps/pep-0343/

[5] Python 2.5.2 manual: 1.10 Reference Counts (http://www.python.org/doc/2.5.2/ext/refcounts.html) Accessed on 2009-05-15.

[6] CERT MEM12-C. Consider using a Goto-Chain when leaving a function on error when using and releasing resources (https://www.securecoding.cert.org/confluence/display/seccode/MEM12-C.+Consider+using+a+Goto-Chain+when+leaving+a+function+on+error+when+using+and+releasing+resources?rootCommentId=29033072#comments)

[7] Exceptions in C (http://adomas.org/excc/)

[8] Exception Handling in C without C++ (http://www.on-time.com/ddj0011.htm)

External links

- Sample Chapter " Gotcha #67: Failure to Employ Resource Acquisition Is Initialization (http://www.informit.com/articles/article.aspx?p=30642&seqNum=8)" by Stephen Dewhurst
- Interview " A Conversation with Bjarne Stroustrup (http://artima.com/intv/modern3.html)" by Bill Venners
- Article " The Law of The Big Two (http://artima.com/cppsource/bigtwo3.html)" by Bjorn Karlsson and Matthew Wilson
- Article " Implementing the 'Resource Acquisition is Initialization' Idiom (http://web.archive.org/web/20080129235233/http://gethelp.devx.com/techtips/cpp_pro/10min/2001/november/10min1101.asp)" by Danny Kalev
- Article " RAII, Dynamic Objects, and Factories in C++ (http://www.codeproject.com/KB/cpp/RAIIFactory.aspx)" by Roland Pibinger
- Article " Managing Dynamic Objects in C++ (http://www.ddj.com/184409895)" by Tom Cargill
- Blog Entry " RAII in C++ (http://jrdodds.blogs.com/blog/2004/08/raii_in_c.html)" by Jonathan Dodds
- RAII in Delphi " One-liner RAII in Delphi (http://blog.barrkel.com/2010/01/one-liner-raii-in-delphi.html)" by Barry Kelly

Singleton pattern

In software engineering, the **singleton pattern** is a design pattern that restricts 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. The term comes from the mathematical concept of a singleton.

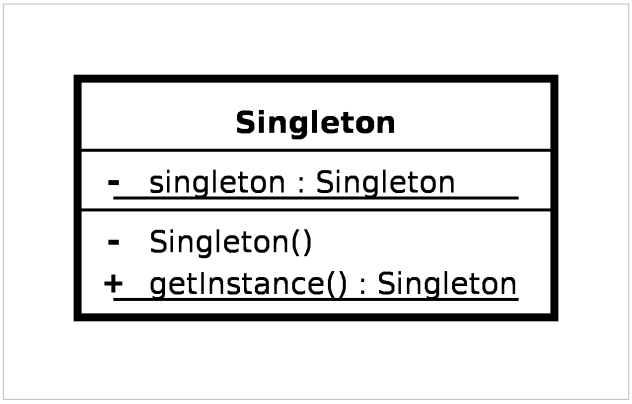
There is criticism of the use of the singleton pattern, as some consider it an anti-pattern, judging that it is overused, introduces unnecessary restrictions in situations where a sole instance of a class is not actually required, and introduces global state into an application.[1][2][3][4][5][6]

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 do not pollute the global name space (or, in languages with namespaces, their containing namespace) with unnecessary variables.[7]
  - They permit lazy allocation and initialization, whereas global variables in many languages will always consume resources.

Structure



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 private. 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. Since Java 5.0, the easiest way to create a Singleton is the enum type approach, given at the end of this section.

Lazy initialization

This method uses double-checked locking, which should not be used prior to J2SE 5.0, as it is vulnerable to subtle bugs. The problem is that an out-of-order write may allow the `instance` reference to be returned before the `Singleton` constructor is executed.[8]

```
public class Singleton {
      private static volatile Singleton instance = null;

      private Singleton() {      }

      public static Singleton getInstance() {
```

```
            if (instance == null) {
                        synchronized (Singleton.class){
                          if (instance == null) {
                                        instance = new Singleton();
                                  }
                          }
            }
            return instance;
      }
}
```

Eager initialization

If the program will always need an instance, or if the cost of creating the instance is not too large in terms of time/resources, the programmer can switch to eager initialization, which always creates an instance:

```
public class Singleton {
    private static final Singleton instance = new Singleton();

    private Singleton() {}

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

This method has a number of advantages:

- The instance is not constructed until the class is used.
- There is no need to `synchronize` the `getInstance()` method, meaning all threads will see the same instance and no (expensive) locking is required.
- The `final` keyword means that the instance cannot be redefined, ensuring that one (and only one) instance ever exists.

This method also has some drawbacks:

- If the program uses the class, but perhaps not the singleton instance itself, then you may want to switch to lazy initialization.

Static block initialization

Some authors[9] refer to a similar solution allowing some pre-processing (e.g. for error-checking). In this sense, the traditional approach could be seen as a particular case of this one, as the class loader would do exactly the same processing.

```
public class Singleton {
  private static final Singleton instance;

  static {
    try {
      instance = new Singleton();
    } catch (IOException e) {
      throw new RuntimeException("Darn, an error occurred!", e);
```

```
    }
  }

  public static Singleton getInstance() {
    return instance;
  }

  private Singleton() {
    // ...
  }
}
```

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.[10] 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;
      }
}
```

Alternatively, the inner class SingletonHolder can be also substituted by implementing a Property which provides also access to the static final/read-only class members. Just like the lazy object in C#, whenever the Singleton.INSTANCE property is called, this singleton is instantiated for the very first time.

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"[11] 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;
        public void execute (String arg) {
                //... perform operation here ...
        }
}
```

The public method can be written to take any desired types of arguments; a single String argument is used here as an example.

This approach implements the singleton by taking advantage of Java's guarantee that any enum value is instantiated only once in a Java program. Since Java enum values are globally accessible, so is the singleton. The drawback is that the enum type is somewhat inflexible; for example, it does not allow lazy initialization.

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,[6] 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[12][13] or PHP.[14]

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] Haggar, Peter (1 May 2002). "Double-checked locking and the Singleton pattern" (http://www.ibm.com/developerworks/java/library/j-dcl/index.html). IBM. .
[9] Coffey, Neil (November 16, 2008). "JavaMex tutorials" (http://www.javamex.com/tutorials/double_checked_locking_fixing.shtml). . Retrieved April 10, 2012.
[10] Pugh, Bill (November 16, 2008). "The Java Memory Model" (http://www.cs.umd.edu/~pugh/java/memoryModel/). . Retrieved April 27, 2009.
[11] Joshua Bloch: *Effective Java* 2nd edition, ISBN 978-0-321-35668-0, 2008, p. 18
[12] Breaking the Singleton (http://blog.yohanliyanage.com/2009/09/breaking-the-singleton/), September 21, 2009
[13] Testing a Singleton (http://dp4j.com), March 7, 2011
[14] 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") 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

- Book extract: Implementing the Singleton Pattern in C# (http://csharpindepth.com/Articles/General/Singleton.aspx) by Jon Skeet
- Singleton at Microsoft patterns & practices Developer Center (http://msdn.microsoft.com/en-us/library/ms998426.aspx)
- Ruby standard library documentation for Singleton (http://ruby-doc.org/stdlib/libdoc/singleton/rdoc/index.html)
- IBM article " Double-checked locking and the Singleton pattern (http://www-128.ibm.com/developerworks/java/library/j-dcl.html?loc=j)" by Peter Haggar
- IBM article " Use your singletons wisely (http://www-106.ibm.com/developerworks/library/co-single.html)" by J. B. Rainsberger
- Javaworld article " Simply Singleton (http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns.html)" by David Geary
- Google article " Why Singletons Are Controversial (http://code.google.com/p/google-singleton-detector/wiki/WhySingletonsAreControversial)"
- Google Singleton Detector (http://code.google.com/p/google-singleton-detector/) (analyzes Java bytecode to detect singletons)

Structural patterns

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
  - Adapter pipeline: Use multiple adapters for debugging purposes.[1]
  - Retrofit Interface Pattern:[2][3] An adapter used as a new interface for multiple classes at the same time.
- 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.[4]
- 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
- 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
- Proxy pattern: a class functioning as an interface to another thing

References

[1] "Adapter Pipeline" (http://c2.com/cgi/wiki?AdapterPipeline). Cunningham & Cunningham, Inc.. 2010-12-31. Archived from the original (http://c2.com/) on 2010-12-31. . Retrieved 2012-07-20.

[2] [|BobbyWoolf (http://c2.com/cgi/wiki?BobbyWoolf)] (2002-06-19). "Retrofit Interface Pattern" (http://c2.com/cgi/wiki?RetrofitInterfacePattern). Cunningham & Cunningham, Inc.. Archived from the original (http://c2.com/) on 2002-06-19. . Retrieved 2012-07-20.

[3] [|MartinZarate (http://c2.com/cgi/wiki?MartinZarate)] (2010-12-31). "External Polymorphism" (http://c2.com/cgi/wiki?ExternalPolymorphism). Cunningham & Cunningham, Inc.. Archived from the original (http://c2.com/) on 2010-12-31. . Retrieved 2012-07-20.

[4] "Tomb Stone" (http://c2.com/cgi/wiki?AdapterPipeline). Cunningham & Cunningham, Inc.. 2007-06-17. Archived from the original (http://c2.com/) on 2007-06-17. . Retrieved 2012-07-20.

Adapter pattern

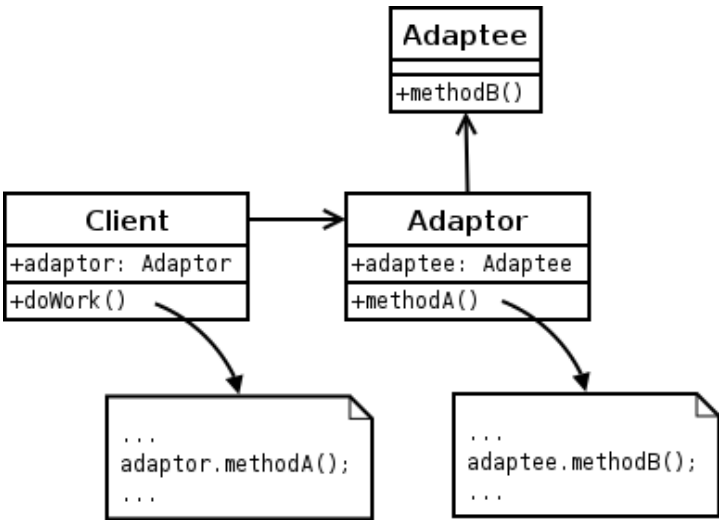
In computer programming, the **adapter pattern** (often referred to as the **wrapper pattern** or simply a **wrapper**) is a design pattern that translates one interface for a class into a compatible interface.[1] An *adapter* allows classes to work together that normally could not because of incompatible 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 client 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

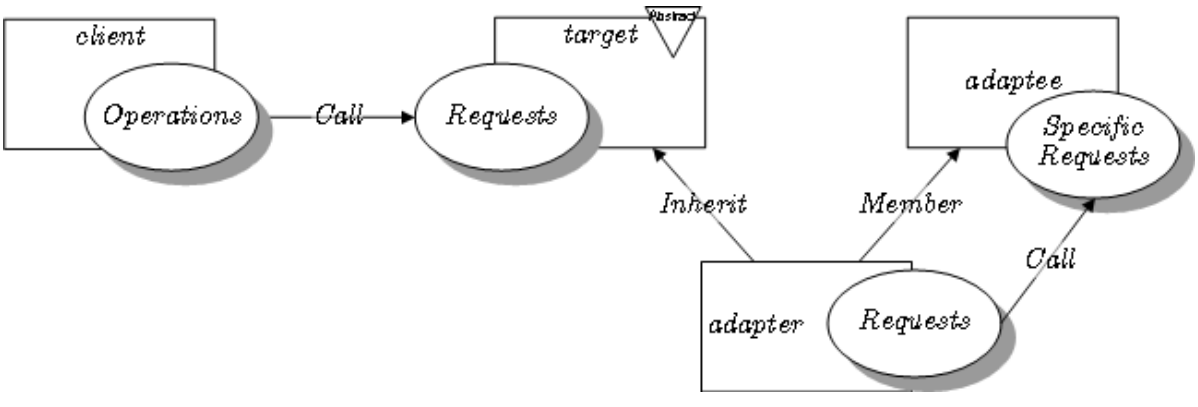
There are two types of adapter patterns:[1]

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.



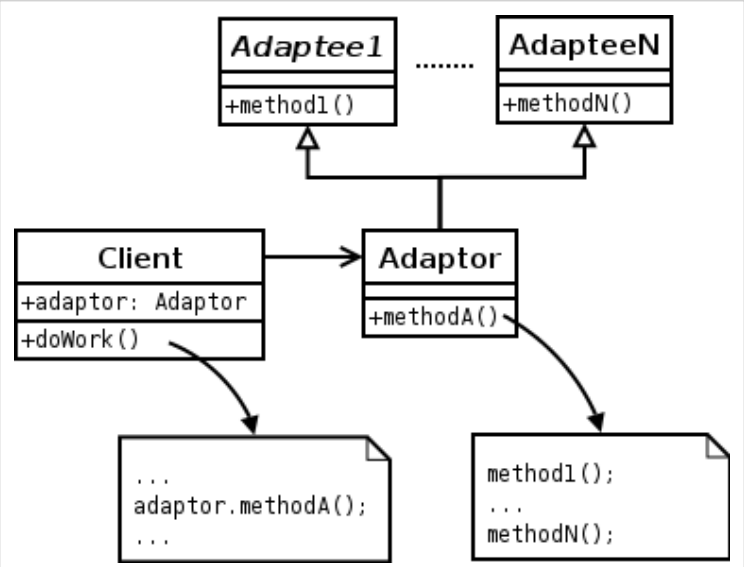
The object adapter pattern expressed in UML. The adapter *hides* the adaptee's interface from the client.



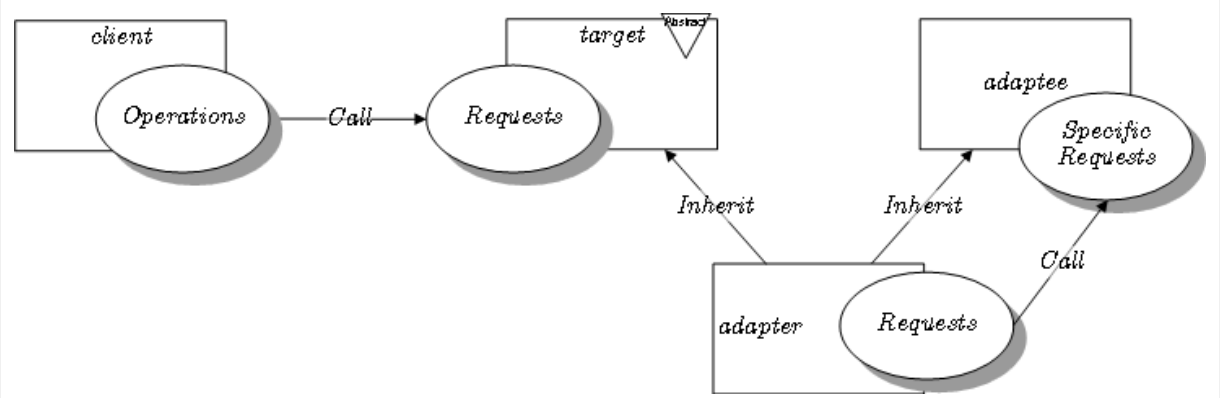
The object adapter pattern expressed in LePUS3.

Class Adapter pattern

This type of adapter uses multiple polymorphic interfaces to achieve its goal. The adapter is created by implementing or inheriting 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.[1]



The class adapter pattern expressed in UML.



The class adapter pattern expressed in LePUS3

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 that 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 {
    private ClassA classA = null;

    public ClassAFormat1(final ClassA A) {
        classA = A;
    }

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

```
}
```

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

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

(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 that can be retrofitted to an existing object hierarchy.

Implementation of Adapter pattern

When implementing the adapter pattern, for clarity use the class name

```
[AdapteeClassName]To[Interface]Adapter
```

, for example

```
DAOToProviderAdapter
```

. It should have a constructor method with adaptee class variable as parameter. This parameter will be passed to the instance member of

```
[AdapteeClassName]To[Interface]Adapter
```

.

```
Class SampleAdapter implements ClientClass
{
    private AdapteeClass mInstance;
```

```
    public SampleAdapter(final AdapteeClass INSTANCE)
    {
         mInstance = INSTANCE;
    }
    @Override
    public void ClientClassMethod()
    {
       // call AdapteeClass's method to implement ClientClassMethod
    }

}
```

References

[1] Freeman, Eric; Freeman, Elisabeth; Kathy, Sierra; Bert, Bates (2004). Hendrickson, Mike. ed (in English) (paperback). *Head First Design Patterns* (http://it-ebooks.info/book/252/). **1**. O'REILLY. pp. 244. ISBN 978-0-596-00712-6. . Retrieved 2012-07-02.

External links

- Adapter in UML and in LePUS3 (a formal modelling language) (http://www.lepus.org.uk/ref/companion/Adapter(Class).xml)
- Description in Portland Pattern Repository's Wiki (http://www.c2.com/cgi/wiki?AdapterPattern)
- Adapter Design Pattern (http://www.codeproject.com/Articles/186003/Adapter-Design-Pattern.aspx) in CodeProject
- PerfectJPattern Open Source Project (http://perfectjpattern.sourceforge.net/dp-adapter.html), Provides componentized implementation of the Adapter Pattern in Java
- The Adapter Pattern in Python (http://ginstrom.com/scribbles/2009/03/27/the-adapter-pattern-in-python/) (detailed tutorial)
- The Adapter Pattern in Eclipse RCP (http://www.eclipse.org/articles/article.php?file=Article-Adapters/index.html)
- Adapter Design Pattern (http://sourcemaking.com/design_patterns/adapter)

Bridge pattern

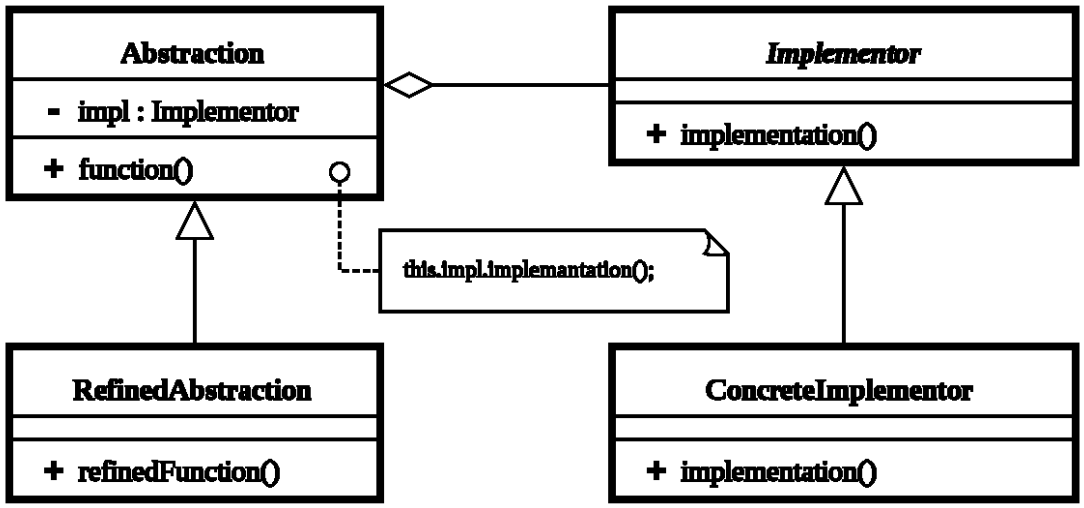
The **bridge pattern** is a design pattern used in software engineering which is meant to *"decouple an abstraction from its implementation so that the two can vary independently"*.[1] The *bridge* uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes.

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

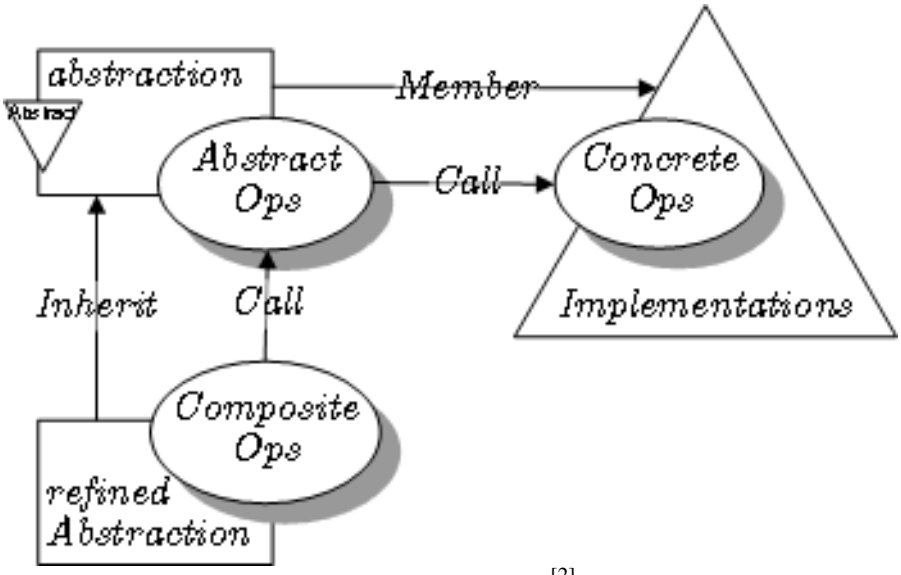
extends the interface defined by Abstraction

Implementor

defines the interface for implementation classes

ConcreteImplementor

implements the Implementor interface



Bridge in LePUS3 (legend [2])

Example

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

```
API1.circle at 1.000000:2.000000 radius 7.5000000
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" */
abstract class Shape {
   protected DrawingAPI drawingAPI;
```

```
   protected Shape(DrawingAPI drawingAPI){
      this.drawingAPI = drawingAPI;
   }

   public abstract void draw();                                  //
low-level
   public abstract void resizeByPercentage(double pct);      //
high-level
}

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

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

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=0-201-84453-2). Addison-Wesley. ISBN 0-201-84453-2.

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.[1]

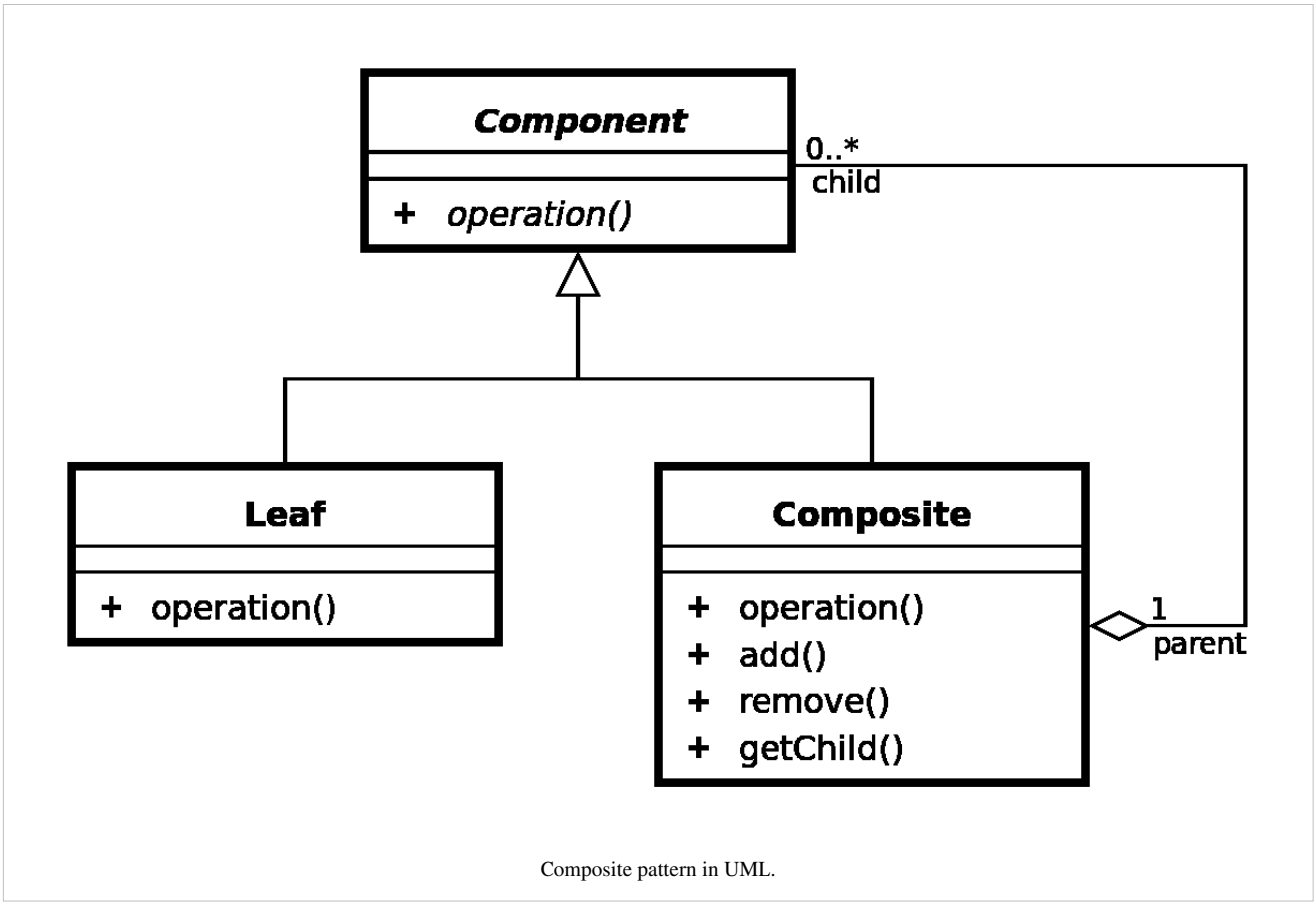
Motivation

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.[2] 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.[1] 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



Composite pattern in UML.

Component

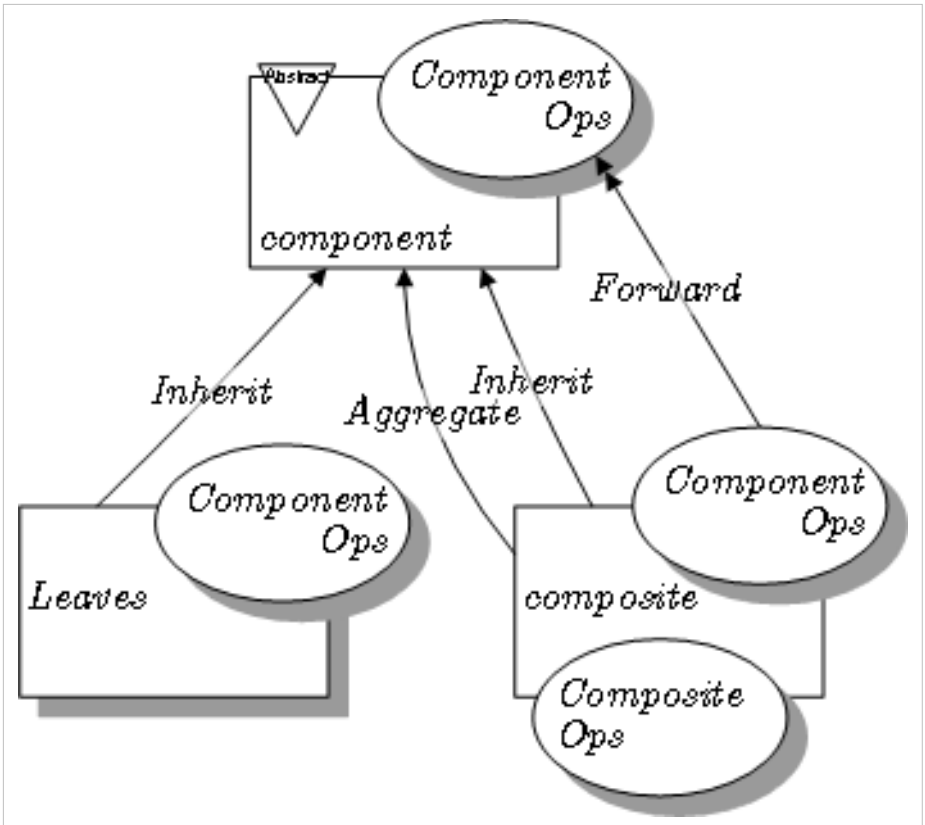
- is the abstraction for all components, including composite ones
- declares the interface for objects in the composition
- (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate

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



Composite pattern in LePUS3.

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.[3]

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> childGraphics = new ArrayList<Graphic>();

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

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

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

The following example, written in C#.

```
namespace CompositePattern
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    //Client
    class Program
    {
        static void Main(string[] args)
        {
            // initialize variables
            var compositeGraphic = new CompositeGraphic();
            var compositeGraphic1 = new CompositeGraphic();
            var compositeGraphic2 = new CompositeGraphic();

            //Add 1 Graphic to compositeGraphic1
            compositeGraphic1.Add(new Ellipse());

            //Add 2 Graphic to compositeGraphic2
            compositeGraphic2.AddRange(new Ellipse(),
                new Ellipse());

            /*Add 1 Graphic, compositeGraphic1, and
              compositeGraphic2 to compositeGraphic */
            compositeGraphic.AddRange(new Ellipse(),
                compositeGraphic1,
                compositeGraphic2);

            /*Prints the complete graphic
            (four times the string "Ellipse").*/
            compositeGraphic.Print();
```

```
            Console.ReadLine();
        }
    }
    //Component
    public interface IGraphic
    {
        void Print();
    }
    //Leaf
    public class Ellipse : IGraphic
    {
        //Prints the graphic
          public void Print()
        {
              Console.WriteLine("Ellipse");
        }
    }
    //Composite
    public class CompositeGraphic : IGraphic
    {
        //Collection of Graphics.
        private readonly List<IGraphic> _Graphics;

        //Constructor
        public CompositeGraphic()
        {
            //initialize generic Colleciton(Composition)
            _Graphics = new List<IGraphic>();
        }
        //Adds the graphic to the composition
        public void Add(IGraphic graphic)
        {
            _Graphics.Add(graphic);
        }
        //Adds multiple graphics to the composition
        public void AddRange(params IGraphic[] graphic)
        {
            _Graphics.AddRange(graphic);
        }
        //Removes the graphic from the composition
        public void Delete(IGraphic graphic)
        {
            _Graphics.Remove(graphic);
        }
        //Prints the graphic.
        public void Print()
        {
```

```
            foreach (var childGraphic in _Graphics)
            {
                childGraphic.Print();
            }
        }
    }
}
```

External links

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

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 0-201-63361-2.

[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

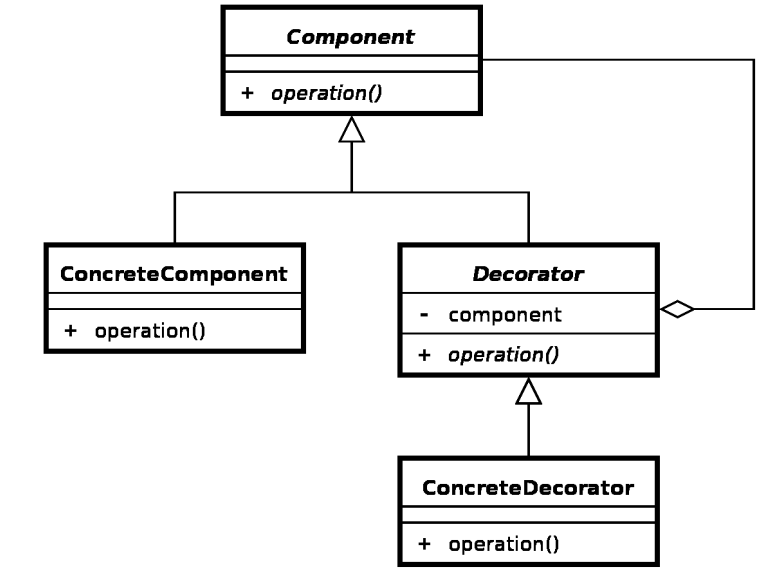
[10] http://sourcemaking.com/design_patterns/composite

Decorator pattern

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

Introduction

The decorator pattern can be used to extend (decorate) the functionality of a certain object at run-time, 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:



Decorator UML class diagram

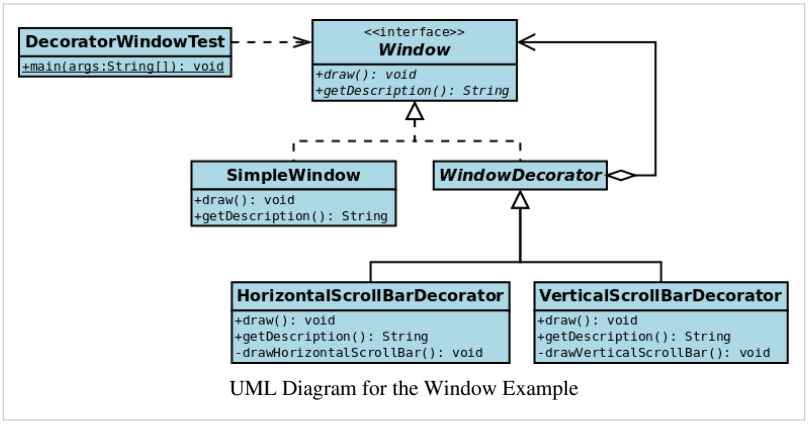
1. Subclass the original "Decorator" class into a "Component" class (see UML diagram);
2. In the Decorator class, add a Component pointer as a field;
3. Pass a Component to the Decorator constructor to initialize the Component pointer;
4. In the Decorator class, redirect all "Component" methods to the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



UML Diagram for the Window Example

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 the coffee
    public String getIngredients(); // returns the ingredients of the
coffee
}

// implementation of a simple coffee without any extra ingredients
public class SimpleCoffee implements Coffee {
    public double getCost() {
```

```
        return 1;
    }

    public String getIngredients() {
        return "Coffee";
    }
}
```

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 final Coffee decoratedCoffee;
    protected String ingredientSeparator = ", ";

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

    public double getCost() { // implementing methods of the interface
        return decoratedCoffee.getCost();
    }

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

// Decorator Milk that mixes milk with coffee
// note it extends CoffeeDecorator
public class Milk extends CoffeeDecorator {
    public Milk(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    public double getCost() { // overriding methods defined in the
abstract superclass
        return super.getCost() + 0.5;
    }

    public String getIngredients() {
        return super.getIngredients() + ingredientSeparator + "Milk";
    }
}

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

```
    public Whip(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    public double getCost() {
        return super.getCost() + 0.7;
    }

    public String getIngredients() {
        return super.getIngredients() + ingredientSeparator + "Whip";
    }
}

// Decorator Sprinkles that mixes sprinkles with coffee
// note it extends CoffeeDecorator
public class Sprinkles extends CoffeeDecorator {
    public Sprinkles(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    public double getCost() {
        return super.getCost() + 0.2;
    }

    public String getIngredients() {
        return super.getIngredients() + ingredientSeparator +
"Sprinkles";
    }
}
```

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 c = new SimpleCoffee();
        System.out.println("Cost: " + c.getCost() + "; Ingredients: " +
 c.getIngredients());

        c = new Milk(c);
        System.out.println("Cost: " + c.getCost() + "; Ingredients: " +
 c.getIngredients());

        c = new Sprinkles(c);
        System.out.println("Cost: " + c.getCost() + "; Ingredients: " +
 c.getIngredients());
```

```
        c = new Whip(c);
        System.out.println("Cost: " + c.getCost() + "; Ingredients: " +
 c.getIngredients());

        // Note that you can also stack more than one decorator of the
same type
        c = new Sprinkles(c);
        System.out.println("Cost: " + c.getCost() + "; Ingredients: " +
 c.getIngredients());
    }
}
```

The output of this program is given below:

```
Cost: 1.0; Ingredients: Coffee
Cost: 1.5; Ingredients: Coffee, Milk
Cost: 1.7; Ingredients: Coffee, Milk, Sprinkles
Cost: 2.4; Ingredients: Coffee, Milk, Sprinkles, Whip
Cost: 2.6; Ingredients: Coffee, Milk, Sprinkles, Whip, Sprinkles
```

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() {

}

Coffee.prototype.cost = function() {
      return 1;
};

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

      return coffee;
}

// Decorator B
function Whip(coffee) {
      var currentCost = coffee.cost();
```

```
      coffee.cost = function() {
            return currentCost + 0.7;
      };

      return coffee;
}

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

      return coffee;
}

// 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 [1]

References

[1] http://c2.com/cgi/wiki?DecoratorPattern

Facade pattern

The **facade pattern** (or **façade pattern**) is a software 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:

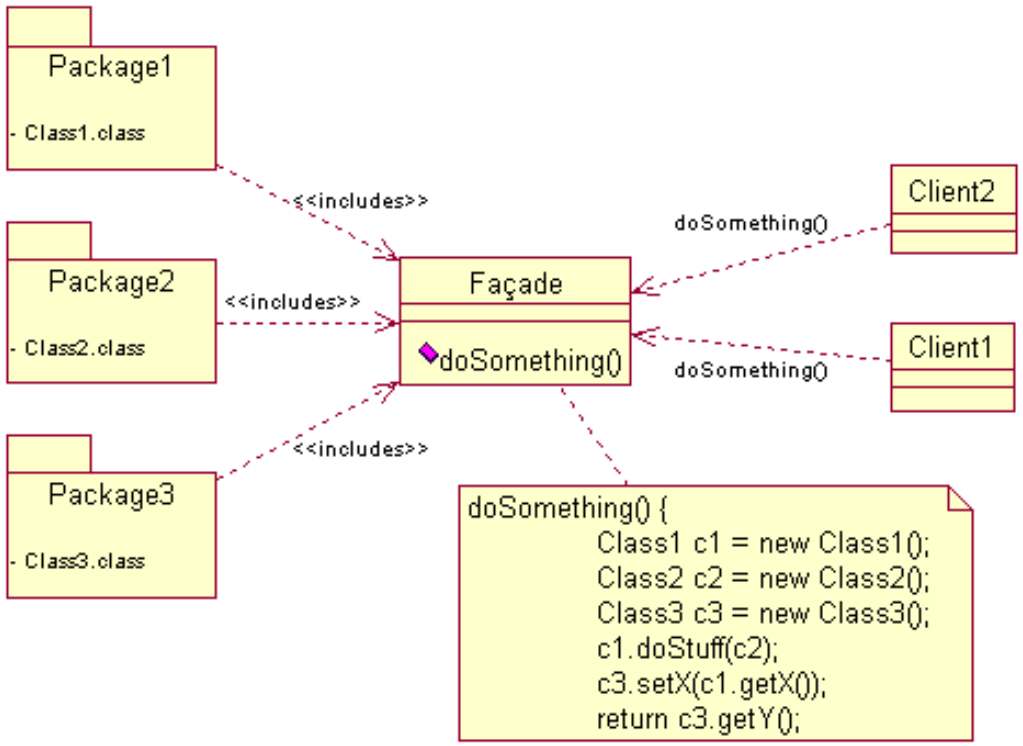
- make a software library easier to use, understand and test, since the facade has convenient methods for common tasks;
- make the library more readable, for the same reason;
- reduce dependencies of outside code on the inner workings of a library, since most code uses the facade, thus allowing more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

Usage

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. Together with the similar decorator pattern, they have slightly different usage:[1]

Pattern	Intent
Adapter	Converts one interface to another so that it matches what the client is expecting
Decorator	Adds responsibility to the interface without altering it
Facade	Provides a simplified interface

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();
    }
}
```

References

[1] Freeman, Eric; Freeman, Elisabeth; Kathy, Sierra; Bert, Bates (2004). Hendrickson, Mike. ed (in English) (paperback). *Head First Design Patterns* (http://it-ebooks.info/book/252/). **1**. O'REILLY. pp. 243, 252, 258, 260. ISBN 978-0-596-00712-6. . Retrieved 2012-07-02.

External links

- Description from the Portland Pattern Repository (http://c2.com/cgi/wiki?FacadePattern)

Front Controller pattern

The **Front Controller Pattern** is a software design pattern listed in several pattern catalogs. The pattern relates to the design of web applications. It "provides a centralized entry point for handling requests."[1]

Front controllers are often used in web applications to implement workflows. While not strictly required, it is much easier to control navigation across a set of related pages (for instance, multiple pages might be used in an online purchase) from a front controller than it is to make the individual pages responsible for navigation.

The front controller may be implemented as a Java object, or as a script in a script language like PHP, ASP, CFML or JSP that is called on every request of a web session. This script, for example an *index.php*, would handle all tasks that are common to the application or the framework, such as session handling, caching, and input filtering. Based on the specific request it would then instantiate further objects and call methods to handle the particular task(s) required.

The alternative to a front controller would be individual scripts like *login.php* and *order.php* that would each then satisfy the type of request. Each script would have to duplicate code or objects that are common to all tasks. But each script might also have more flexibility to implement the particular task required.

Examples

Several web-tier application frameworks implement the Front Controller pattern, among them:

- Ruby on Rails
- ColdBox, a ColdFusion MVC framework.
- Spring MVC, a Java MVC framework
- Yii, Cake, Symfony, Kohana, CodeIgniter and Zend Framework, MVC frameworks written with PHP
- Cairngorm framework in Adobe Flex.
- Microsoft's ASP.NET MVC Framework.
- Yesod web application framework [2] written in Haskell.

Notes

[1] Alur et al., p. 166.

[2] http://www.yesodweb.com/

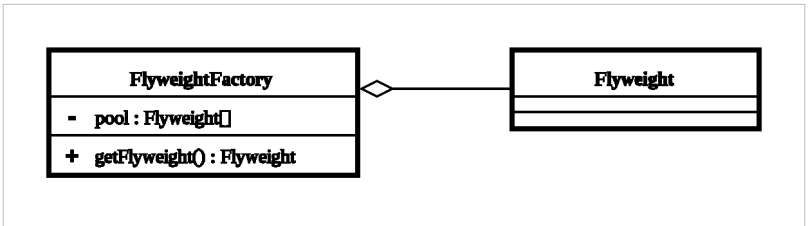
External links

- Bear Bibeault's Front Man™ (http://www.bibeault.org/frontman/), A lightweight Java implementation.

References

- Alur, Deepak; John Crup, Dan Malks (2003). *Core J2EE Patterns, Best Practices and Design Strategies, 2nd Ed.*. Sun Microsystems Press. pp. 650pp. ISBN 0-13-142246-4.
- Fowler, Martin. *Patterns of Enterprise Application Architecture* (http://www.martinfowler.com/books.html#eaa). pp. 560pp. ISBN 978-0-321-12742-6.
- Fowler, Martin. "Front Controller" (http://www.martinfowler.com/eaaCatalog/frontController.html). Retrieved February 2, 2008.

Flyweight pattern



In computer programming, **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. Often some parts of the object state can be shared, and it is common practice to hold 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.

History

According to a textbook Design Patterns: Elements of Reusable Object-Oriented Software,[1] the flyweight pattern was first coined and extensively explored by Paul Calder and Mark Linton in 1990[2] to efficiently handle glyph information in a WYSIWYG document editor, although similar techniques were already used in other systems, e.g., an application framework by Weinand et al. (1988).[3]

Example in Java

```
// Flyweight object interface
public interface CoffeeOrder {
    public void serveCoffee(CoffeeOrderContext context);
}

// ConcreteFlyweight object that creates ConcreteFlyweight
public class CoffeeFlavor implements CoffeeOrder {
    private String flavor;

    public CoffeeFlavor(String newFlavor) {
        this.flavor = newFlavor;
    }

    public String getFlavor() {
        return this.flavor;
    }

    public void serveCoffee(CoffeeOrderContext context) {
```

```
        System.out.println("Serving Coffee flavor " + flavor + " to
table number " + context.getTable());
    }
}

public class CoffeeOrderContext {
   private int tableNumber;

   public CoffeeOrderContext(int tableNumber) {
       this.tableNumber = tableNumber;
   }

   public int getTable() {
       return this.tableNumber;
   }
}

import java.util.HashMap;
import java.util.Map;

//FlyweightFactory object
public class CoffeeFlavorFactory {
    private Map<String, CoffeeFlavor> flavors = new HashMap<String, CoffeeFlavor>();

    public CoffeeFlavor getCoffeeFlavor(String flavorName) {
        CoffeeFlavor flavor = flavors.get(flavorName);
        if (flavor == null) {
            flavor = new CoffeeFlavor(flavorName);
            flavors.put(flavorName, flavor);
        }
        return flavor;
    }

    public int getTotalCoffeeFlavorsMade() {
        return flavors.size();
    }
}

public class TestFlyweight {
   /** The flavors ordered. */
   private static CoffeeFlavor[] flavors = new CoffeeFlavor[100];
   /** The tables for the orders. */
   private static CoffeeOrderContext[] tables = new
CoffeeOrderContext[100];
   private static int ordersMade = 0;
   private static CoffeeFlavorFactory flavorFactory;
```

```
    public static void takeOrders(String flavorIn, int table) {
        flavors[ordersMade] = flavorFactory.getCoffeeFlavor(flavorIn);
        tables[ordersMade++] = new CoffeeOrderContext(table);
    }

    public static void main(String[] args) {
        flavorFactory = new CoffeeFlavorFactory();

        takeOrders("Cappuccino", 2);
        takeOrders("Cappuccino", 2);
        takeOrders("Frappe", 1);
        takeOrders("Frappe", 1);
        takeOrders("Xpresso", 1);
        takeOrders("Frappe", 897);
        takeOrders("Cappuccino", 97);
        takeOrders("Cappuccino", 97);
        takeOrders("Frappe", 3);
        takeOrders("Xpresso", 3);
        takeOrders("Cappuccino", 3);
        takeOrders("Xpresso", 96);
        takeOrders("Frappe", 552);
        takeOrders("Cappuccino", 121);
        takeOrders("Xpresso", 121);

        for (int i = 0; i < ordersMade; ++i) {
            flavors[i].serveCoffee(tables[i]);
        }
        System.out.println(" ");
        System.out.println("total CoffeeFlavor objects made: " +
flavorFactory.getTotalCoffeeFlavorsMade());
    }
}
```

References

[1] *Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. pp. 205–206. ISBN 0-201-63361-2.

[2] Calder, Paul R.; Linton, Mark A. (October 1990). "Glyphs: Flyweight Objects for User Interfaces". The 3rd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology. Snowbird, Utah, United States. pp. 92-101. doi:10.1145/97924.97935. ISBN 0-89791-410-4.

[3] Weinand, Andre; Gamma, Erich; Marty, Rudolf (1988). "ET++—an object oriented application framework in C++". OOPSLA (Object-Oriented Programming Systems, Languages and Applications). San Diego, California, United States. pp. 46-57. doi:10.1145/62083.62089. ISBN 0-89791-284-5.

External links

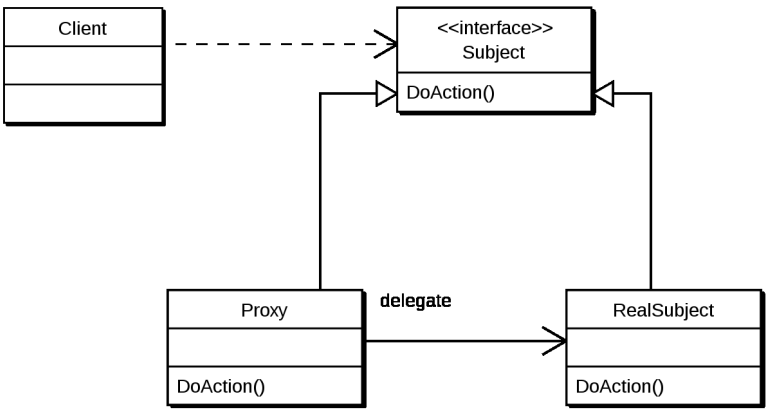
- Flyweight in UML and in LePUS3 (a formal modelling language) (http://www.lepus.org.uk/ref/companion/Flyweight.xml)
- Article " Enhancing Web Application Performance with Caching (http://theserverside.com/articles/article.tss?l=Caching)" by Neal Ford
- Section " Flyweight Text Entry Fields (archive.org) (http://web.archive.org/web/20070404160614/http://btl.usc.edu/rides/documentn/refMan/rf21_d.html)" from the RIDES Reference Manual by Allen Munro and Quentin A. Pizzini
- Description (http://c2.com/cgi/wiki?FlyweightPattern) from Portland's Pattern Repository
- Sourdough Design (http://sourdough.phpee.com/index.php?node=18)
- Class::Flyweight - implement the flyweight pattern in OO perl (http://www.perlmonks.org/?node_id=94783)
- Boost.Flyweight - A generic C++ implementation (http://boost.org/libs/flyweight/index.html)

Proxy pattern

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

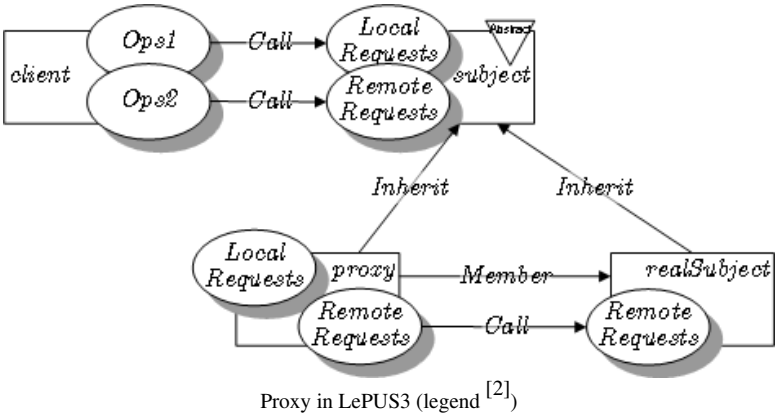
A proxy, in its most general form, is a class functioning as an interface to something else. The proxy could interface to anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate.

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



Proxy in UML

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 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.



Proxy in LePUS3 (legend [2])

Example

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

The example creates first an interface against which the pattern creates the classes. This interface contains only one method to display the image, called `displayImage()`, that hast to be coded by all classes implementing it.

The proxy class `ProxyImage` is running on another system than the real image class itself and can represent the real image `RealImage` over there. The image information is accessed from the disk. Using the proxy pattern, the code of the `ProxyImage` avoids multiple loading of the image, accessing it from the other system in a memory-saving manner.

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

//on System A
class RealImage implements Image {

    private String filename = null;
    /**
     * Constructor
     * @param FILENAME
     */
    public RealImage(final String FILENAME) {
        filename = FILENAME;
        loadImageFromDisk();
    }

    /**
     * Loads the image from the disk
     */
    private void loadImageFromDisk() {
        System.out.println("Loading   " + filename);
    }

    /**
     * Displays the image
     */
    public void displayImage() {
        System.out.println("Displaying " + filename);
    }

}

//on System B
class ProxyImage implements Image {

    private RealImage image = null;
    private String filename = null;
```

```
    /**
     * Constructor
     * @param FILENAME
     */
    public ProxyImage(final String FILENAME) {
        filename = FILENAME;
    }

    /**
     * Displays the image
     */
    public void displayImage() {
        if (image == null) {
           image = new RealImage(filename);
        }
        image.displayImage();
    }

}

class ProxyExample {

   /**
    * Test method
    */
   public static void main(String[] args) {
        final Image IMAGE1 = new ProxyImage("HiRes_10MB_Photo1");
        final Image IMAGE2 = new ProxyImage("HiRes_10MB_Photo2");

        IMAGE1.displayImage(); // loading necessary
        IMAGE1.displayImage(); // loading unnecessary
        IMAGE2.displayImage(); // loading necessary
        IMAGE2.displayImage(); // loading unnecessary
        IMAGE1.displayImage(); // loading unnecessary
    }

}
```

The program's output is:

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

References

External links

- Proxy pattern in Java (http://wiki.java.net/bin/view/Javapedia/ProxyPattern)
- Proxy pattern in UML and in LePUS3 (a formal modelling language) (http://www.lepus.org.uk/ref/companion/Proxy.xml)
- Take control with the Proxy design pattern (http://www.javaworld.com/javaworld/jw-02-2002/jw-0222-designpatterns.html) by David Geary, JavaWorld.com
- PerfectJPattern Open Source Project (http://perfectjpattern.sourceforge.net/dp-proxy.html), Provides componentized implementation of the Proxy Pattern in Java
- Adapter vs. Proxy vs. Facade Pattern Comparison (http://www.netobjectives.com/PatternRepository/index.php?title=AdapterVersusProxyVersusFacadePatternComparison)
- Proxy Design Pattern (http://sourcemaking.com/design_patterns/proxy)
- Proxy pattern description from the Portland Pattern Repository (http://c2.com/cgi/wiki?ProxyPattern)

Behavioral patterns

Behavioral pattern

In software engineering, **behavioral design patterns** are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

Examples of this type of design pattern include:

- Chain of responsibility pattern: Command objects are handled or passed on to other objects by logic-containing processing objects
- Command pattern: Command objects encapsulate an action and its parameters
- "Externalize the Stack": Turn a recursive function into an iterative one that uses a stack.[1]
- Hierarchical visitor pattern: Provide a way to visit every node in a hierarchical data structure such as a tree.
- Interpreter pattern: Implement a specialized computer language to rapidly solve a specific set of problems
- Iterator pattern: Iterators are used to access the elements of an aggregate object sequentially without exposing its underlying representation
- Mediator pattern: Provides a unified interface to a set of interfaces in a subsystem
- Memento pattern: Provides the ability to restore an object to its previous state (rollback)
- Null Object pattern: designed to act as a default value of an object
- Observer pattern: aka Publish/Subscribe or Event Listener. Objects register to observe an event that may be raised by another object
  - Weak reference pattern: De-couple an observer from an observable.[2]
- Protocol stack: Communications are handled by multiple layers, which form an encapsulation hierarchy.[3]
- Scheduled-task pattern: A task is scheduled to be performed at a particular interval or clock time (used in real-time computing)
- Single-serving visitor pattern: Optimise the implementation of a visitor that is allocated, used only once, and then deleted
- Specification pattern: Recombinable Business logic in a boolean fashion
- State pattern: A clean way for an object to partially change its type at runtime
- Strategy pattern: Algorithms can be selected on the fly
- Template method pattern: Describes the program skeleton of a program
- Visitor pattern: A way to separate an algorithm from an object

References

[1] "Externalize The Stack" (http://c2.com/cgi/wiki?ExternalizeTheStack). c2.com. 2010-01-19. Archived from the original (http://c2.com/) on 2010-01-19. . Retrieved 2012-05-21.

[2] [|Ashod Nakashian (http://c2.com/cgi/wiki?AshodNakashian)] (2004-04-11). "Weak Reference Pattern" (http://c2.com/cgi/wiki?WeakReferencePattern). c2.com. Archived from the original (http://c2.com/) on 2004-04-11. . Retrieved 2012-05-21.

[3] "Protocol Stack" (http://c2.com/cgi/wiki?ProtocolStack). c2.com. 2006-09-05. Archived from the original (http://c2.com/) on 2006-09-05. . Retrieved 2012-05-21.

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 logic that defines the types of command objects that it can handle; the rest are passed 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 might work in this manner.

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

Example

The following 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. 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.

Java

```
package chainofresp;

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 log) {
        next = log;
        return log;
    }

    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 logger, logger1,logger2;
        logger = new StdoutLogger(Logger.DEBUG);
        logger1 = logger.setNext(new EmailLogger(Logger.NOTICE));
        logger2 = logger1.setNext(new StderrLogger(Logger.ERR));

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

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

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

C#

```
using System;
using System.IO;

namespace ChainOfResponsibility
{
    public enum LogLevel
    {
        Info=1,
        Debug=2,
        Warning=4,
        Error=8,
        FunctionalMessage=16,
        FunctionalError=32,
        All = 63
    }

    /// <summary>
    /// Abstract Handler in chain of responsibility Pattern
    /// </summary>
    public abstract class Logger
    {
        protected LogLevel logMask;

        // The next Handler in the chain
        protected Logger next;

        public Logger(LogLevel mask)
        {
            this.logMask = mask;
```

```
        }

        /// <summary>
        /// Sets the Next logger to make a list/chain of Handlers
        /// </summary>
        public Logger SetNext(Logger nextlogger)
        {
            next = nextlogger;
            return nextlogger;
        }

        public void Message(string msg, LogLevel severity)
        {
            if ((severity & logMask) != 0)
            {
                WriteMessage(msg);
            }
            if (next != null)
            {
                next.Message(msg, severity);
            }
        }

        abstract protected void WriteMessage(string msg);
    }

    public class ConsoleLogger : Logger
    {
        public ConsoleLogger(LogLevel mask)
            : base(mask)
        { }

        protected override void WriteMessage(string msg)
        {
            Console.WriteLine("Writing to console: " + msg);
        }
    }

    public class EmailLogger : Logger
    {
        public EmailLogger(LogLevel mask)
            : base(mask)
        { }

        protected override void WriteMessage(string msg)
        {
            //Placeholder for mail send logic, usually the email
```

```
configurations are saved in config file.
            Console.WriteLine("Sending via email: " + msg);
        }
    }

    class FileLogger : Logger
    {
        public FileLogger(LogLevel mask)
            : base(mask)
        { }

        protected override void WriteMessage(string msg)
        {
            //Placeholder for File writing logic
            Console.WriteLine("Writing to Log File: " + msg);
        }
    }

    public class Program
    {
        public static void Main(string[] args)
        {
            // Build the chain of responsibility
            Logger logger, logger1, logger2;
            logger1 = logger = new ConsoleLogger(LogLevel.All);
            logger1 = logger1.SetNext(new
EmailLogger(LogLevel.FunctionalMessage | LogLevel.FunctionalError));
            logger2 = logger1.SetNext(new FileLogger(LogLevel{{Not a
typo|.}}Warning | LogLevel.Error));

            // Handled by ConsoleLogger
            logger.Message("Entering function ProcessOrder().",
LogLevel.Debug);
            logger.Message("Order record retrieved.", LogLevel.Info);

            // Handled by ConsoleLogger and FileLogger
            logger.Message("Customer Address details missing in Branch
DataBase.", LogLevel.Warning);
            logger.Message("Customer Address details missing in
Organization DataBase.", LogLevel.Error);

            // Handled by ConsoleLogger and EmailLogger
            logger.Message("Unable to Process Order ORD1 Dated D1 For
Customer C1.", LogLevel.FunctionalError);

            // Handled by ConsoleLogger and EmailLogger
            logger.Message("Order Dispatched.",
```

```
LogLevel.FunctionalMessage);
        }
    }
}

/* Output
Writing to console: Entering function ProcessOrder().
Writing to console: Order record retrieved.
Writing to console: Customer Address details missing in Branch
DataBase.
Writing to Log File: Customer Address details missing in Branch
DataBase.
Writing to console: Customer Address details missing in Organization
DataBase.
Writing to Log File: Customer Address details missing in Organization
DataBase.
Writing to console: Unable to Process Order ORD1 Dated D1 For Customer
C1.
Sending via email: Unable to Process Order ORD1 Dated D1 For Customer
C1.
Writing to console: Order Dispatched.
Sending via email: Order Dispatched.
*/
```

Another Example

Below is another example of this pattern in Java. In this example we have different roles, each having a fixed purchasing limit and a successor. Every time a user in a role receives a purchase request that exceeds his or her limit, the request is passed to his or her 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 following code defines the PurchaseRequest class that 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;
    }
}
```

In the following usage example, the successors are set as follows: 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);

        // Press Ctrl+C to end.
```

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

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://allapplabs.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

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.

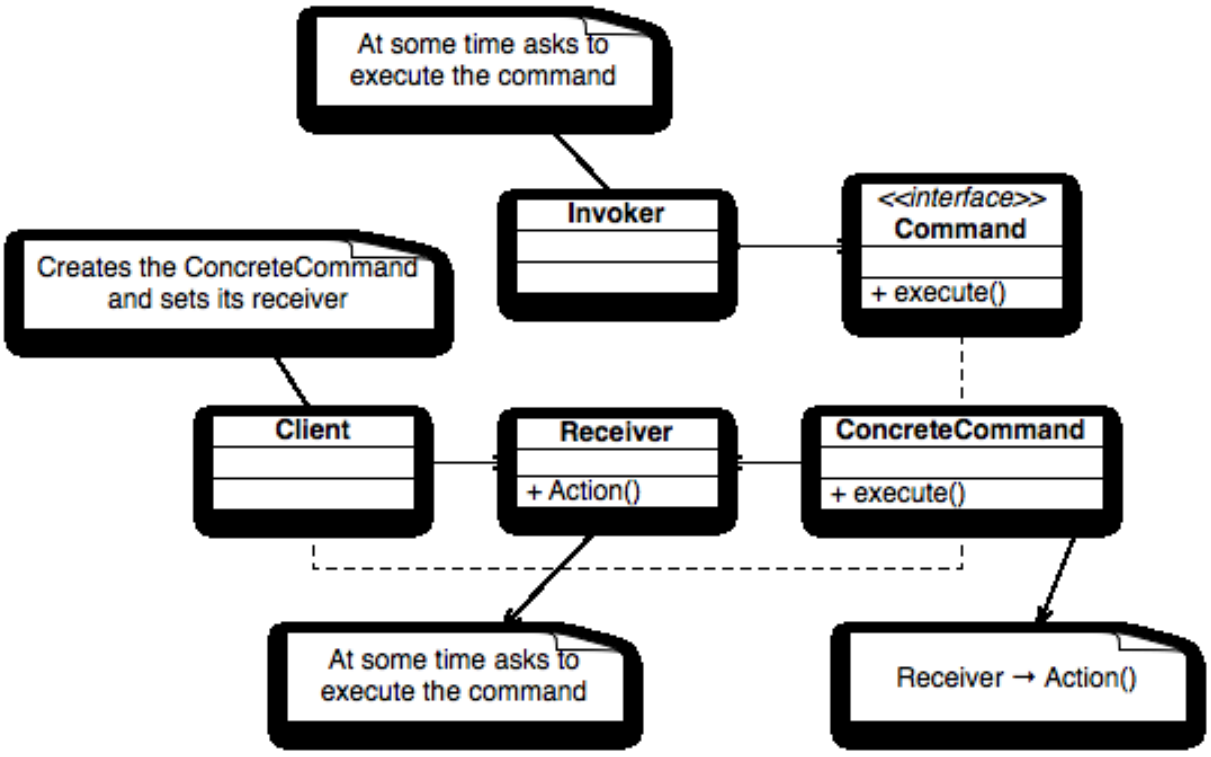
Parallel Processing

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/invoker object calls the Command/Action object's execute/performAction method. The Command/Action object notifies the appropriate source/invoker 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 [1] (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.

Example

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 (i.e. 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.

Java

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



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

/*the Invoker class*/
public class Switch {
   private List<Command> history = new ArrayList<Command>();

   public Switch() {
   }

   public void storeAndExecute(Command cmd) {
      this.history.add(cmd); // optional
      cmd.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 for turning on the light - ConcreteCommand #1*/
public class FlipUpCommand implements Command {
    private Light theLight;

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

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


/*the Command for turning off the light - ConcreteCommand #2*/
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);

      Switch s = new Switch();

      try {
         if (args[0].equalsIgnoreCase("ON")) {
            s.storeAndExecute(switchUp);
            System.exit(0);
         }
         if (args[0].equalsIgnoreCase("OFF")) {
            s.storeAndExecute(switchDown);
            System.exit(0);
         }
         System.out.println("Argument \"ON\" or \"OFF\" is required.");
      } catch (Exception e) {
         System.out.println("Arguments required.");
      }
   }
}
```

Python

The following code is an implementation of Command pattern in Python.

```
class Switch(object):
    """The INVOKER class"""
    def __init__(self, flip_up_cmd, flip_down_cmd):
        self.flip_up = flip_up_cmd
        self.flip_down = flip_down_cmd

class Light(object):
    """The RECEIVER class"""
    def turn_on(self):
        print "The light is on"
    def turn_off(self):
        print "The light is off"

class LightSwitch(object):
    """The CLIENT class"""
    def __init__(self):
        lamp = Light()
        self._switch = Switch(lamp.turn_on, lamp.turn_off)

    def switch(self, cmd):
        cmd = cmd.strip().upper()
```

```
        if cmd == "ON":
            self._switch.flip_up()
        elif cmd == "OFF":
            self._switch.flip_down()
        else:
            print 'Argument "ON" or "OFF" is required.'

# Execute if this file is run as a script and not imported as a module
if __name__ == "__main__":
    light_switch = LightSwitch()
    print "Switch ON test."
    light_switch.switch("ON")
    print "Switch OFF test."
    light_switch.switch("OFF")
    print "Invalid Command test."
    light_switch.switch("****")
```

C#

The following code is an implementation of Command pattern in C#.

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

namespace CommandPattern
{
    public interface ICommand
    {
        void Execute();
    }

    public class Switch
    {
        private List<ICommand> _commands = new List<ICommand>();

        public void StoreAndExecute(ICommand command)
        {
            _commands.Add(command);
            command.Execute();
        }
    }

    public class Light
    {
        public void TurnOn()
        {
            Console.WriteLine("The light is on");
        }
```

```
        public void TurnOff()
        {
            Console.WriteLine("The light is off");
        }
    }

    public class FlipUpCommand : ICommand
    {
        private Light _light;

        public FlipUpCommand(Light light)
        {
            _light = light;
        }

        public void Execute()
        {
            _light.TurnOn();
        }
    }

    public class FlipDownCommand : ICommand
    {
        private Light _light;

        public FlipDownCommand(Light light)
        {
            _light = light;
        }

        public void Execute()
        {
            _light.TurnOff();
        }
    }

    internal class Program
    {
        private static void Main(string[] args)
        {
            Light lamp = new Light();
            ICommand switchUp = new FlipUpCommand(lamp);
            ICommand switchDown = new FlipDownCommand(lamp);

            Switch s = new Switch();

            try
```

```
            {
                if (args[0].ToUpper().Equals("ON"))
                {
                    s.StoreAndExecute(switchUp);
                    return;
                }
                if (args[0].ToUpper().Equals("OFF"))
                {
                    s.StoreAndExecute(switchDown);
                    return;
                }
                Console.WriteLine("Argument \"ON\" or \"OFF\" is
required.");
            }
            catch (Exception e)
            {
                Console.WriteLine("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/javatips/jw-javatip68.html
- PerfectJPattern Open Source Project [2], Provides a componentized i.e. context-free and type-safe implementation of the Command Pattern in Java
- Command Pattern Video Tutorial [3], 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

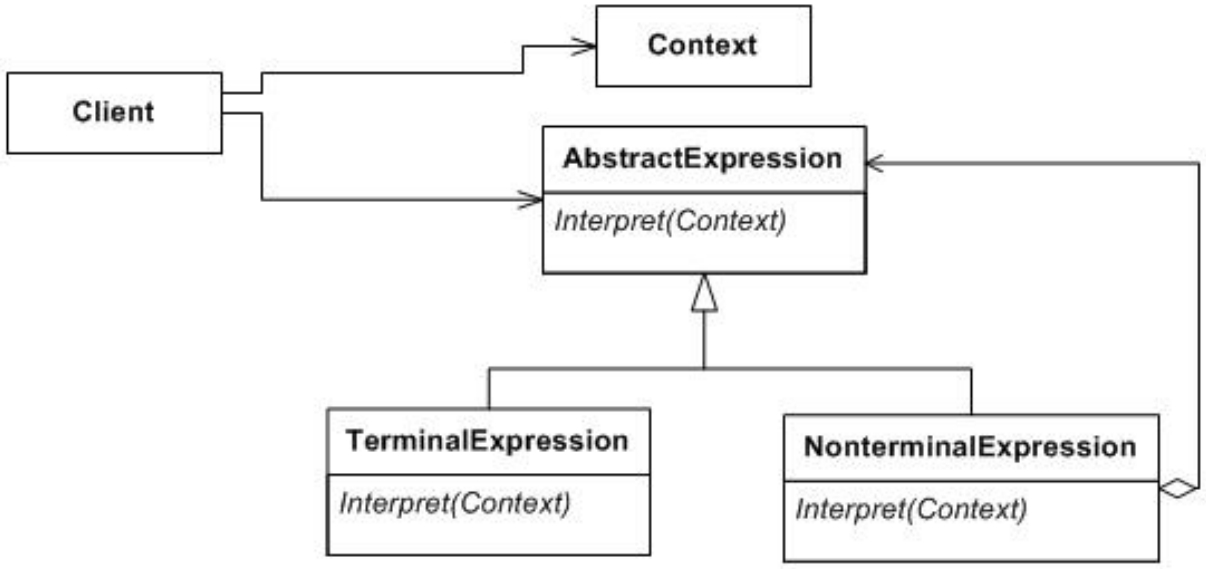
Interpreter pattern

In computer programming, the **interpreter pattern** is a design pattern that 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.[1]

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.Map;

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

class Number implements Expression {
    private int number;
    public Number(int number)       { this.number = number; }
    public int interpret(Map<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(Map<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(Map<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(Map<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[2] a parser is provided for completeness.

```
import java.util.Map;
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(Map<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.Map;
import java.util.HashMap;

public class InterpreterExample {
    public static void main(String[] args) {
        String expression = "w x z - +";
        Evaluator sentence = new Evaluator(expression);
        Map<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://lukaszwrobel.pl/blog/interpreter-design-pattern) in Ruby
- SourceMaking tutorial (http://sourcemaking.com/design_patterns/interpreter)
- Interpreter pattern description from the Portland Pattern Repository (http://c2.com/cgi/wiki?InterpreterPattern)

Iterator pattern

In object-oriented programming, the **iterator pattern** is a design pattern in which an iterator is used to traverse a container and access the container's elements. The iterator pattern decouples algorithms from containers; in some cases, algorithms are necessarily container-specific and thus cannot be decoupled.

For example, the hypothetical algorithm *SearchForElement* can be implemented generally using a specified type of iterator rather than implementing it as a container-specific algorithm. This allows *SearchForElement* to be used on any container that supports the required type of iterator.

Definition

The essence of the Iterator Factory method Pattern is to "Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.".[1]

Language-specific implementation

Some languages standardize syntax. C++ and Python are notable examples.

C++

C++ implements iterators with the semantics of pointers in that language. 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.

Java

Java has an Iterator interface that the Collections should implement in order to traverse the elements of the collection. Classic implementations were using the `next()` method, which is the same for the Java interface. However, there are no `currentItem()`, `first()`, and `isDone()` methods defined. Instead, the Java interface adds the `hasNext()` and `remove()` methods.

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__().)`[2]

References

[1] Gang Of Four

[2] "Python v2.7.1 documentation: The Python Standard Library: 5. Built-in Types" (http://docs.python.org/library/stdtypes.html). . Retrieved 2 May 2011.

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)
- Iterator Pattern (http://c2.com/cgi/wiki?IteratorPattern)

Mediator pattern

The **mediator pattern** defines an object that encapsulates how a set of objects interact. 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.

Definition

The essence of the Mediator Pattern is to "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. ".[1]

Example

In the following example a mediator object controls the status of three collaborating buttons: for this it contains three methods (`book(),view()` and `search()`) that set the status of the buttons. The methods are called by each button upon activation (via the `execute()` method in each of them).

Hence here the collaboration pattern is that each participant (here the buttons) communicates to the mediator its activity and the mediator dispatches the expected behavior to the other participants.

```
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();
}

//Abstract Mediator
interface IMediator {
    void book();
    void view();
    void search();
    void registerView(BtnView v);
    void registerSearch(BtnSearch s);
```

```
    void registerBook(BtnBook b);
    void registerDisplay(LblDisplay d);
}

//Concrete mediator
class Mediator implements IMediator {

    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 {

    IMediator med;

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

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

}

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

    IMediator med;

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

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

}

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

    IMediator med;

    BtnBook(ActionListener al, IMediator m) {
```

```
        super("Book");
        addActionListener(al);
        med = m;
        med.registerBook(this);
    }

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

}

class LblDisplay extends JLabel {

    IMediator med;

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

}

class MediatorDemo extends JFrame implements ActionListener {

    IMediator 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 comd = (Command) ae.getSource();
        comd.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*

References

[1] Gang Of Four

External links

- Mediator Design Pattern (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 implemented with 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;

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

        public 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
```

This example uses a String as the state, which by default is an immutable type in java. In real life scenarios the state will almost always be an object. In which case the state has to be cloned before putting in the memento.

```
private Memento(State state)
{
    //state has to be cloned before returning the
    //memento, or successive calls to get Memento
    //return a reference to the same object
    this.mementoState = state.clone();
}
```

It must be said that this latter implementation has a drawback: it declares an internal class. Better would be that the memento strategy could apply on more that one object.

There are mainly 3 other ways to achieve Memento: 1- Serialization. 2- A class declared in the same package. 3- The object can also be accessed via a proxy, which can achieve any save/restore operation on the object.

External links

- Description of Memento Pattern [1] in Ada
- Memento UML Class Diagram [2] with C# and .NET code samples
- SourceMaking Tutorial [3]
- Memento Design Pattern using C# [4] by Nilesh Gule

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
[4] http://www.nileshgule.com/2012/08/memento-design-pattern.html

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.[1]

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 methods typically cannot be invoked on null references.

The Objective-C language takes another approach to this problem and does not invoke methods on nil but instead returns nil for all such invocations.

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.

Example

Given a binary tree, with this node structure:

```
class node {
  node left
  node right
}
```

One may implement a tree size procedure recursively:

```
function tree_size(node) {
  return 1 + tree_size(node.left) + tree_size(node.right)
}
```

Since the child nodes may not exist, one must modify the procedure by adding non-existence or null checks:

```
function tree_size(node) {
  set sum = 1
  if node.left exists {
    sum = sum + tree_size(node.left)
```

```
  }
  if node.right exists {
    sum = sum + tree_size(node.right)
  }
  return sum
}
```

This however makes the procedure more complicated by mixing boundary checks with normal logic, and it becomes harder to read. Using the null object pattern, one can create a special version of the procedure but only for null nodes:

```
function tree_size(node) {
  return 1 + tree_size(node.left) + tree_size(node.right)
}
```

```
function tree_size(null_node) {
  return 0
}
```

This separates normal logic from special case handling, and makes the code easier to understand.

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*[2] and Joshua Kerievsky's[3] 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*[4]

In various languages

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#

C# is a language in which the Null Object pattern can be properly implemented. This example shows animal objects that display sounds and a NullAnimal instance used in place of the C# null keyword. The Null Object provides consistent behaviour and prevents a runtime Null Reference Exception that would occur if the C# null keyword were used instead.

```
/* Null Object Pattern implementation:
 */
using System;

// Animal interface is the key to compatibility for Animal
implementations below.
interface IAnimal
{
    void MakeSound();
}

// Dog is a real animal.
class Dog : IAnimal
{
    public void MakeSound()
    {
        Console.WriteLine("Woof!");
    }
}

// The Null Case: this NullAnimal class should be instantiated and used
 in place of C# null keyword.
class NullAnimal : IAnimal
{
    public void MakeSound()
    {
        // Purposefully provides no behaviour.
    }
}

/* ========================
 * Simplistic usage example in a Main entry point.
 */
```

```
static class Program
{
    static void Main()
    {
        IAnimal dog = new Dog();
        dog.MakeSound(); // outputs "Woof!"

        /* Instead of using C# null, use a NullAnimal instance.
         * This example is simplistic but conveys the idea that if a
NullAnimal instance is used then the program
         * will never experience a .NET System.NullReferenceException
at runtime, unlike if C# null was used.
         */
        IAnimal unknown = new NullAnimal();  //<< replaces: IAnimal unknown = null;
        unknown.MakeSound(); // outputs nothing, but does not throw a
runtime exception
    }
}
```

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`.

Ruby

In the Ruby language

```
the null object is a fully featured object of the NilClass.
```

It takes a special role in boolean comparisons as being defined as false.

Ruby allows to enhance the NilClass So if you want to behave the NilClass like it does in ObjC way, just add this (your mileage may vary)

```
 class NilClass
   def method_missing(*)
     return nil
   end
```

`end`

Java

```
public interface Animal {
      public void makeSound();
}

public class Dog implements Animal {
      public void makeSound() {
            System.out.println("woof!");
      }
}

public class NullAnimal implements Animal {
      public void makeSound() {
      }
}
```

This code illustrates a variation of the C++ example, above, using the Java language. As with C++, a null class can be instantiated in situations where a reference to an `Animal` object is required, but there is no appropriate object available. A null `Animal` object is possible (`Animal myAnimal = null;`) and could be useful as a place-holder, but may not be used for calling a method. In this example, `myAnimal.makeSound();` will throw a NullPointerException. Therefore, additional code may be necessary to test for null objects.

The null object pattern solves this problem by providing a special `NullAnimal` class which can be instantiated as an object of type `Animal`. As with C++ and related languages, that special null class must be created for each class hierarchy that needs a null object, since a `NullAnimal` is of no use when what is needed is a null object that does not implement the `Animal` interface.

Criticism

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

External links

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

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://martinfowler.com/eaaCatalog/specialCase.html

[8] http://www.owlnet.rice.edu/~comp212/00-spring/handouts/week06/null_object_revisited.htm

[9] http://refactoring.com/catalog/introduceNullObject.html

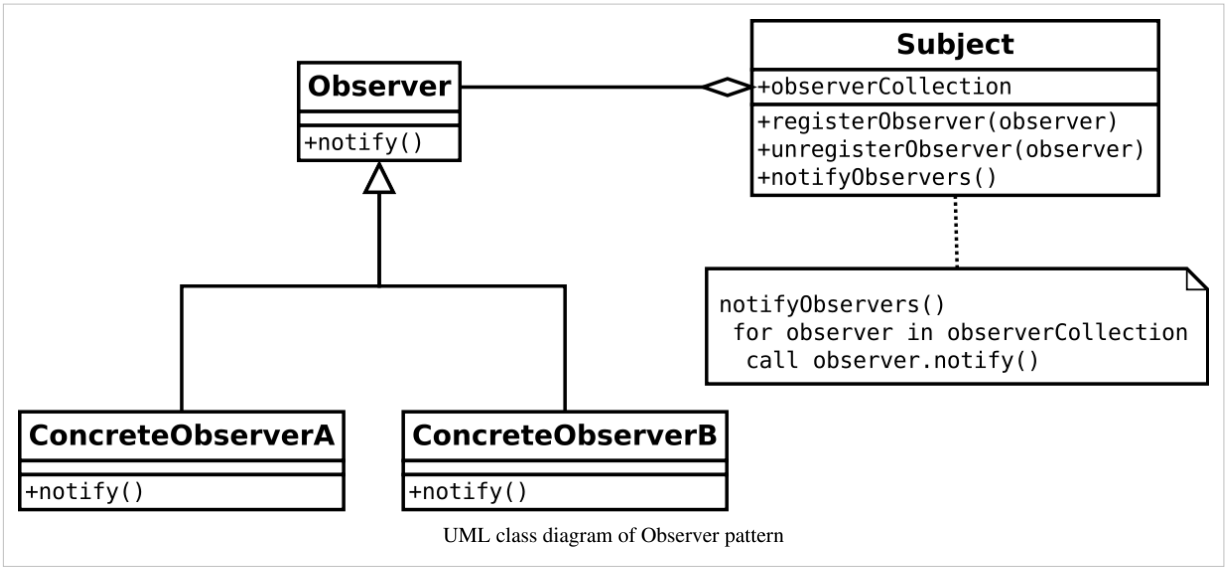
[10] http://sourcemaking.com/design_patterns/null_object

Observer pattern

The **observer 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. Observer is also a key part in the familiar MVC architectural pattern. In fact the observer pattern was first implemented in Smalltalk's MVC based user interface framework.[1]

Related patterns: Publish–subscribe pattern, mediator, singleton.

Structure



UML class diagram of Observer pattern

Definition

The essence of the Observer Pattern is to "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. ".[1]

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` [2] and `java.util.Observable` [3]. 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 org.wikipedia.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 org.wikipedia.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 org.wikipedia.obs;

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

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

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

        // subscribe the observer to the event source
        eventSource.addObserver(responseHandler);

        // starts the event thread
        Thread thread = new Thread(eventSource);
        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 [4], 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++ [5] - the C++ signalling template library.
- sigslot [6] - C++ Signal/Slot Library
- Cpp::Events [7] - Template-based C++ implementation that introduces separation of connection management interface of the event object from the invocation interface.
- XLObject [8] - Template-based C++ signal/slot model patterned after Qt.
- Signals [9] - A lightweight and non-intrusive C++ signal/slot model implementation.
- libevent [10] - Multi-threaded Crossplatform Signal/Slot C++ Library
- Boost.Signals [11], an implementation of signal/slot model
- MFC's CDocument-CView-framework
- The Qt C++ framework's signal/slot model
- The Advanced Component Framework's component-based Model/Observer pattern implementation.
- The MRPT robotics C++ framework's observer/observable [12] model.

Objective-C

- NSNotificationCenter [13] - The Objective-C NSNotificationCenter class.
- NSKeyValueObserving [14] - The Objective-C NSKeyValueObserving protocol.

C#

- The IObserver<T> Interface [15] - The .NET Framework 4+ supported way of implementing the observer pattern.
- Exploring the Observer Design Pattern [16] - 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/
- http://coldfusioneventmanager.riaforge.org/

D

- Phobos - std.signal [17]

Delphi

- Delphi Observer Pattern [18], a Delphi implementation

Java

- The class `java.util.Observer`[19] provides a simple observer implementation.
- Events are typically implemented in Java through the callback pattern: one piece of code provides a common interface with as many methods as many events are required,[20] another piece of code provides an implementation of that interface, and another one receives that implementation, and raises events as necessary.[21]

JavaScript

- EventDispatcher singleton [22], a JavaScript core API based Signals and slots implementation - an observer concept different from Publish/subscribe - pretty lightweighted but still type-safety enforcing.
- DevShop [23] DevShop Js is a pocket-sized minimalist framework of common design patterns for JavaScript. Includes Observer pattern.
- The observer pattern is implemented in jQuery using '.on() [24]', '.trigger() [25]' and 'off() [26]' to create, fire and remove event handlers. jQuery uses the actual in-memory DOM Elements by attaching event handlers to them, and automatically removing the events when an element disappears from the DOM. See this jQuery Event Model [27] slideshow for more info.

Lisp

- Cells [28], a dataflow extension to Common Lisp that uses meta-programming to hide some of the details of Observer pattern implementation.

Perl

- Class::Observable [29] Basic observer pattern implementation
- Class::Publisher [30] a slightly more advanced implementation
- POE::Component::Syndicator [31] Observer pattern for POE [32]

PHP

- Symfony2 Event Dispatcher [33], a standalone library component by the Symfony team
- Event_Dispatcher [34], a PEAR implementation
- SPLSubject [35] & SPLObserver [36], part of the Standard PHP Library
- prggmr [37], an event processing engine for PHP 5.4+ designed to be lightweight, fast and very simple to use.

Python

- Louie [38], an implementation by Patrick K. O'Brien.
- PyDispatcher [39], the implementation on which the Django [40] web framework's signals are based.
- Py-notify [41], a Python (plus a little C) implementation
- Observer Pattern using Weak References [42] implementation by Michael Kent
- PyPubSub [43] an in-application Pub/Sub library for Observer behavior
- NotificationFramework [44] classes directly implementing Observer patterns
- Blinker [45], an implementation which can be used with decorators.
- Jpblib [46], an implementation that minimises boilerplate.

Ruby

- Observer [47], from the Ruby Standard Library.

Other/Misc

- CSP [48] - *Observer Pattern* using *CSP-like Rendezvous* (each actor is a process, communication is via rendezvous).
- YUI Event utility [49] implements custom events through the observer pattern
- Publish/Subscribe with LabVIEW [50], Implementation example of Observer or Publish/Subscribe using G.

Critics

The Observer pattern is criticized[51] for being too verbose, introducing too many bugs and violating software engineering principles, such as not promoting side-effects, encapsulation, composability, separation of concerns, scalability, uniformity, abstraction, resource management, semantic distance . The recommended approach is to gradually deprecate observers in favor of reactive programming abstractions .

References

[1] Gang Of Four
[2] http://docs.oracle.com/javase/7/docs/api/java/util/Observer.html
[3] http://docs.oracle.com/javase/7/docs/api/java/util/Observable.html
[4] http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/events/package-detail.html
[5] http://libsigc.sourceforge.net
[6] http://sigslot.sourceforge.net/
[7] http://code.google.com/p/cpp-events
[8] http://xlobject.sourceforge.net/
[9] http://github.com/pbhogan/Signals
[10] http://www.monkey.org/~provos/libevent/
[11] http://www.boost.org/doc/html/signals.html
[12] http://reference.mrpt.org/svn/classmrpt_1_1utils_1_1_c_observable.html
[13] http://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSNotificationCenter_Class/Reference/Reference.html
[14] http://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Protocols/NSKeyValueObserving_Protocol/Reference/Reference.html
[15] http://msdn.microsoft.com/en-us/library/dd783449.aspx
[16] http://msdn.microsoft.com/en-us/library/ee817669.aspx
[17] http://dlang.org/phobos/std_signals.html
[18] http://blogs.teamb.com/joannacarter/2004/06/30/690
[19] java.util.Observer (http://download.oracle.com/javase/6/docs/api/java/util/Observer.html)
[20] Which will typically implement interface java.util.EventListener (http://download.oracle.com/javase/6/docs/api/java/util/EventListener.html)
[21] http://download.oracle.com/javase/tutorial/uiswing/events/generalrules.html
[22] http://www.refactory.org/s/eventdispatcher_singleton_a_core_api_based_signals_and_slots_implementation/view/latest
[23] https://github.com/rgr-myrg/DevShop-JS
[24] http://api.jquery.com/on/
[25] http://api.jquery.com/trigger/
[26] http://api.jquery.com/off/
[27] http://www.slideshare.net/wildan.m/jquery-events-are-where-it-happens
[28] http://common-lisp.net/project/cells/
[29] http://search.cpan.org/~cwinters/Class-Observable-1.04/lib/Class/Observable.pm
[30] http://search.cpan.org/~simonflk/Class-Publisher-0.2/lib/Class/Publisher.pm
[31] http://search.cpan.org/perldoc?POE::Component::Syndicator
[32] http://poe.perl.org
[33] http://symfony.com/doc/master/components/event_dispatcher/index.html
[34] http://pear.php.net/package/Event_Dispatcher

[35] http://php.net/splsubject
[36] http://php.net/splobserver
[37] http://github.com/nwhitingx/prggmr
[38] http://github.com/11craft/louie
[39] http://pydispatcher.sourceforge.net/
[40] http://www.djangoproject.com/
[41] http://home.gna.org/py-notify/
[42] http://radio-weblogs.com/0124960/2004/06/15.html
[43] http://sourceforge.net/projects/pubsub/
[44] http://pypi.python.org/pypi/NotificationFramework/
[45] http://pypi.python.org/pypi/blinker/1.1
[46] http://www.sinusoid.es/jpblib
[47] http://ruby-doc.org/stdlib/libdoc/observer/rdoc/index.html
[48] http://ptolemy.eecs.berkeley.edu/presentations/06/FutureOfEmbeddedSoftware_Lee_Graz.ppt
[49] http://developer.yahoo.com/yui/event/
[50] http://www.labviewportal.eu/viewtopic.php?f=19&t=9
[51] Deprecating the Observer Pattern by Ingo Maier, Tiark Rompf, Martin Odersky (2010) PDF (http://lamp.epfl.ch/~imaier/pub/DeprecatingObserversTR2010.pdf)

- http://www.research.ibm.com/designpatterns/example.htm
- http://msdn.microsoft.com/en-us/library/ms954621.aspx
- "Speaking on the Observer pattern" (http://www.javaworld.com/javaworld/javaqa/2001-05/04-qa-0525-observer.html) - JavaWorld

External links

- Observer Pattern implementation in JDK 7 (http://docs.oracle.com/javase/7/docs/api/java/util/Observable.html)
- Observer Pattern in Java (http://www.javaworld.com/javaworld/javaqa/2001-05/04-qa-0525-observer.html)
- Definition, C# example & UML diagram (http://www.dofactory.com/Patterns/PatternObserver.aspx)
- Subject Observer example in C++ (http://rtmatheson.com/2010/03/working-on-the-subject-observer-pattern/)
- Observer Pattern recipe in Python (http://code.activestate.com/recipes/131499-observer-pattern/)
- SourceMaking Tutorial (http://sourcemaking.com/design_patterns/observer)
- Observer Pattern in Objective-C (http://www.a-coding.com/2010/10/observer-pattern-in-objective-c.html)
- Observer Pattern in Java (Portuguese) (http://www.patternizando.com.br/2011/03/design-pattern-observer-com-aplicacao-swing-jse/)

Publish/subscribe

In software architecture, **publish–subscribe** is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers. Instead, published messages are characterized into classes, without knowledge of what, if any, subscribers there may be. Similarly, subscribers express interest in one or more classes, and only receive messages that are of interest, without knowledge of what, if any, publishers there are.

Pub/sub is a sibling of the message queue paradigm, and is typically one part of a larger message-oriented middleware system. Most messaging systems support both the pub/sub and message queue models in their API, e.g. Java Message Service (JMS).

This pattern provides greater network scalability and a more dynamic network topology.

Message filtering

In the published/sub model, subscribers typically receive only a subset of the total messages published. The process of selecting messages for reception and processing is called *filtering*. There are two common forms of filtering: topic-based and content-based.

In a **topic-based** system, messages are published to "topics" or named logical channels. Subscribers in a topic-based system will receive all messages published to the topics to which they subscribe, and all subscribers to a topic will receive the same messages. The publisher is responsible for defining the classes of messages to which subscribers can subscribe.

In a **content-based** system, messages are only delivered to a subscriber if the attributes or content of those messages match constraints defined by the subscriber. The subscriber is responsible for classifying the messages.

Some systems support a **hybrid** of the two; publishers post messages to a topic while subscribers register content-based subscriptions to one or more topics.

Topologies

In many pub/sub systems, publishers post messages to an intermediary message broker and subscribers register subscriptions with that broker, letting the broker perform the filtering. The broker normally performs a store and forward function to route messages from publishers to subscribers.

History

One of the earliest publicly described pub/sub systems was the "news" subsystem of the Isis Toolkit, described at the 1987 Association for Computing Machinery (ACM) Symposium on Operating Systems Principles conference (SOSP '87), in a paper "Exploiting Virtual Synchrony in Distributed Systems. 123-138".[1] The pub/sub technology described therein was invented by Frank Schmuck.

Advantages

Loose coupling

> Publishers are loosely coupled to subscribers, and need not even know of their existence. With the topic being the focus, publishers and subscribers are allowed to remain ignorant of system topology. Each can continue to operate normally regardless of the other. In the traditional tightly coupled client–server paradigm, the client cannot post messages to the server while the server process is not running, nor can the server receive messages unless the client is running. Many pub/sub systems decouple not only the locations of the publishers and

subscribers, but also decouple them temporally. A common strategy used by middleware analysts with such pub/sub systems is to take down a publisher to allow the subscriber to work through the backlog (a form of bandwidth throttling).

Scalability

Pub/sub provides the opportunity for better scalability than traditional client–server, through parallel operation, message caching, tree-based or network-based routing, etc. However, in certain types of tightly coupled, high-volume enterprise environments, as systems scale up to become data centers with thousands of servers sharing the pub/sub infrastructure, current vendor systems often lose this benefit; scalability for pub/sub products under high load in these contexts is a research challenge.

Outside of the enterprise environment, on the other hand, the pub/sub paradigm has proven its scalability to volumes far beyond those of a single data centre, providing Internet-wide distributed messaging through web syndication protocols such as RSS and Atom (standard). These syndication protocols accept higher latency and lack of delivery guarantees in exchange for the ability for even a low-end web server to syndicate messages to (potentially) millions of separate subscriber nodes.

Disadvantages

The most serious problems with pub/sub systems are a side-effect of their main advantage: the decoupling of publisher from subscriber. The problem is that it can be hard to specify stronger properties that the application might need on an end-to-end basis:

- As a first example, many pub/sub systems will try to deliver messages for a little while, but then give up. If an application actually needs a stronger guarantee (such as: messages will always be delivered or, if delivery cannot be confirmed, the publisher will be informed), the pub/sub system probably won't have a way to provide that property.
- Another example arises when a publisher "assumes" that a subscriber is listening. Suppose that we use a pub/sub system to log problems in a factory: any application that senses an error publishes an appropriate message, and the messages are displayed on a console by the logger daemon, which subscribes to the "errors" topic. If the logger happens to crash, publishers won't have any way to see this, and all the error messages will vanish. (It should be noted that in a client/server system, you still have the question of how the error generating clients would deal with the un-logged errors if an error logging server crashes. Adding redundant error logging servers to a client/server system adds code complexity and contact info maintenance tasks for each of the error generating applications. On the other hand, in a pub/sub system, adding a number of redundant logging subscribers requires no changes to any of the error publishers.)

As noted above, while pub/sub scales very well with small installations, a major difficulty is that the technology often scales poorly in larger ones. These manifest themselves as instabilities in throughput (load surges followed by long silence periods), slowdowns as more and more applications use the system (even if they are communicating on disjoint topics), and so-called IP broadcast storms, which can shut down a local area network by saturating it with overhead messages that choke out all normal traffic, even traffic unrelated to pub/sub.

For pub/sub systems that use brokers (servers), the agreement for a broker to send messages to a subscriber is in-band, and can be subject to security problems. Brokers might be fooled into sending notifications to the wrong client, amplifying denial of service requests against the client. Brokers themselves could be overloaded as they allocate resources to track created subscriptions.

Even with systems that do not rely on brokers, a subscriber might be able to receive data that it is not authorized to receive. An unauthorized publisher may be able to introduce incorrect or damaging messages into the pub/sub system. This is especially true with systems that broadcast or multicast their messages. Encryption (e.g. Transport Layer Security (SSL/TLS)) can be the only strong defense against unauthorized access.

Semantic coupling

Pub/sub systems have loose coupling within space, time and synchronization, providing a scalable infrastructure for information exchange and distributed workflows. However, pub/sub are tightly coupled, via event subscriptions and patterns, to the semantics of the underlying event schema and values. The high degree of semantic heterogeneity of events in large and open deployments such as smart cities and the sensor web makes it difficult to develop and maintain pub/sub systems. In order to address semantic coupling within pub/subsystems the use of approximate semantic matching of events is an active area of research [2].

References

[1] Birman, K. and Joseph, T. " Exploiting virtual synchrony in distributed systems (http://portal.acm.org/citation.cfm?id=41457.37515&coll=portal&dl=ACM&CFID=97240647&CFTOKEN=78805594)" *in Proceedings of the eleventh ACM Symposium on Operating systems principles (SOSP '87)*, 1987. pp. 123-138.

[2] Hasan, Souleiman, Sean O'Riain, and Edward Curry. 2012. "Approximate Semantic Matching of Heterogeneous Events." (http://www.edwardcurry.org/publications/Hasan_DEBS_2012.pdf) In 6th ACM International Conference on Distributed Event-Based Systems (DEBS 2012), 252–263. Berlin, Germany: ACM. "DOI" (http://dx.doi.org/10.1145/2335484.2335512).

External links

- XMPP XEP-0060: Publish-Subscribe (http://xmpp.org/extensions/xep-0060.html)
- For an open source example which is in production on MSN.com and Microsoft.com, see Distributed Publish/Subscribe Event System (http://www.codeplex.com/pubsub)
- Python PubSub (http://pubsub.sf.net) a Python Publish-Subscribe broker for messages *within* an application (NOT network)

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 methods that take care of a desired service, while objects for which (or with whom) the servant does something, are taken as parameters.

Description and simple example

Servant is used for providing some behavior to a group of classes. Instead of defining that behavior in each class - or when we cannot factor out this behavior in the common parent class - it is defined once in the 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 "Movable" specifying that every class implementing this interface needs to implement method "getPosition" and "setPosition". The first method gets the position of an object on a canvas and second one sets the position of an object and draws it on a canvas. Then we define a servant class "MoveServant", which has two methods "moveTo(Movable movedObject, Position where)" and moveBy(Movable movedObject, int dx, int dy). The Servant class can now be used to move every object which implements the Movable. Thus the "moving" code appears in only one class which respects the "Separation of Concerns" rule.

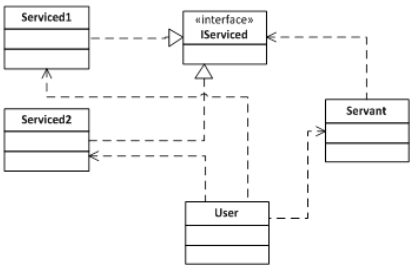
Two ways of implementation

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.

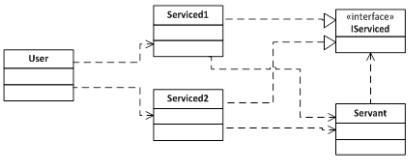
1. On figure 2 is shown opposite situation, where user don't know about servant class and calls directly serviced classes. Serviced classes then asks servant themselves to achieve desired functionality.



User uses servant to achieve some functionality and passes the serviced objects as parameters.

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.
2. Analyze what abilities serviced classes must have, so they can be properly serviced.



User requests operations from serviced instances, which then asks servant to do it for them.

3. We define an interface, which will enforce implementation of declared methods.
4. Define an interface specifying requested behavior of serviced objects. If some instance wants to be served by servant, it must implement this interface.
5. Define (or acquire somehow) specified servant (his class).
6. 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
// Movable Interface
public class MoveServant {
      // Method, which will move Movable implementing class to position
 where
      public void moveTo(Movable 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 Movable implementing class by dx and
dy
      public void moveBy(Movable 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 Movable {
      public void setPosition(Position p);

      public Position getPosition();
}

// One of geometric classes
public class Triangle implements Movable {
      // Position of the geometric object on some canvas
      private Position p;

        // Method, which sets position of geometric object
      public void setPosition(Position p) {
            this.p = p;
      }

      // Method, which returns position of geometric object
      public Position getPosition() {
            return this.p;
      }
}

// One of geometric classes
public class Ellipse implements Movable {
      // Position of the geometric object on some canvas
      private Position p;

      // Method, which sets position of geometric object
      public void setPosition(Position p) {
```

```
            this.p = p;
      }

      // Method, which returns position of geometric object
      public Position getPosition() {
            return this.p;
      }
}

// One of geometric classes
public class Rectangle implements Movable {
      // Position of the geometric object on some canvas
      private Position p;

      // Method, which sets position of geometric object
      public void setPosition(Position p) {
            this.p = p;
      }

      // Method, which returns 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

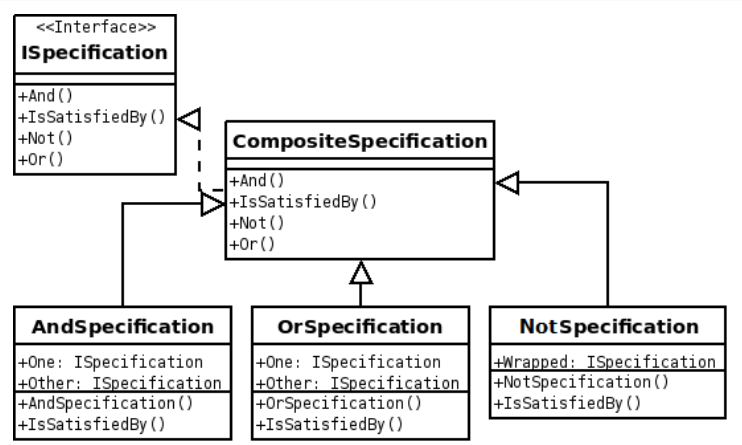
Pecinovský, Rudolf; Jarmila Pavlíčková, Luboš Pavlíček (6 2006). "Let's Modify the Objects First Approach into Design Patterns First" [1]. Eleventh Annual Conference on Innovation and Technology in Computer Science Education, University of Bologna [2].

References

[1] 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 rules can be recombined by chaining the business rules together using boolean logic.



Specification Pattern in UML

A specification pattern outlines a business rule that is combinable with other business rules. 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 [1] by Eric Evans and Martin Fowler
- The Specification Pattern: A Primer [2] by Matt Berther
- The Specification Pattern: A Four Part Introduction using VB.Net [3] by Richard Dalton
- specification pattern in flash actionscript 3 [4] by Rolf Vreijdenberger

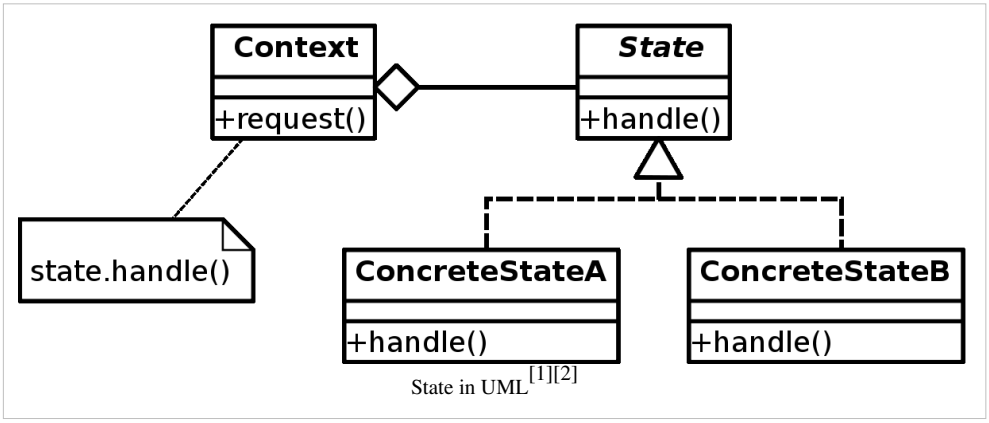
References

[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.dpdk.nl/opensource/specification-pattern-for-selection-on-lists

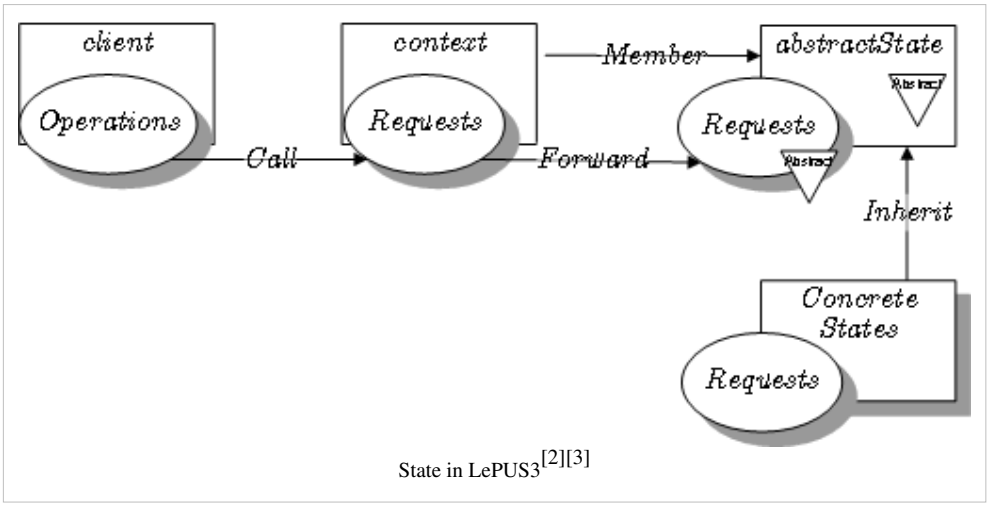
State pattern

The **state pattern**, which closely resembles Strategy 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[1]:395.

Structure



State in UML[1][2]



State in LePUS3[2][3]

Example

Pseudocode

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.

Java

The state interface and two implementations. The state's method has a reference to the context object and is able to change its state.

```
interface State {
      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
twice
            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 [4]

References

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

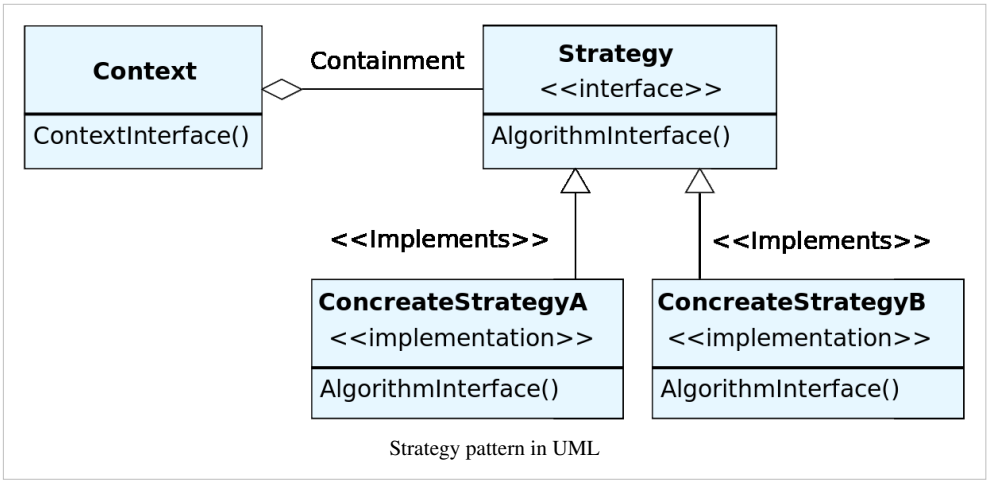
Strategy pattern

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. Formally speaking, 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.[1]

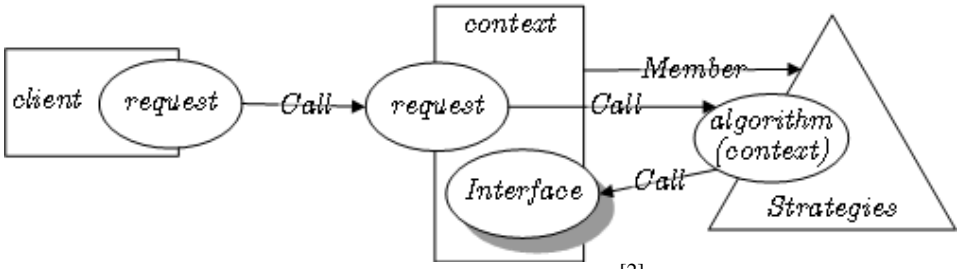
For instance, a class that performs validation on incoming data may use a strategy pattern to select a validation algorithm based on the type of data, the source of the data, user choice, and/or other discriminating factors. These factors are not known for each case until run-time, and may require radically 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.

Structure



Strategy pattern in UML



Strategy pattern in LePUS3 (legend [2])

Example

The following example is in Java.

```
// The classes that implement a concrete strategy should implement
this.
// The Context class uses this to call the concrete strategy.
interface IStrategy {
    int execute(int a, int b);
}

// Implements the algorithm using the strategy interface
class ConcreteStrategyAdd implements IStrategy {

    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 IStrategy {

    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 IStrategy {

    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategyMultiply's
execute()");
        return a * b;   // Do a multiplication with a and b
    }
}
```

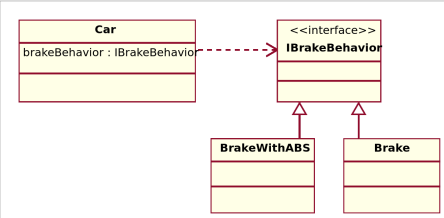
```
// Configured with a ConcreteStrategy object and maintains a reference
to a Strategy object
class Context {

    private IStrategy strategy;

    // Constructor
    public Context(IStrategy strategy) {
        this.strategy = strategy;
    }

    public int executeStrategy(int a, int b) {
        return strategy.execute(a, b);
    }
}

// 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);
    }
}
```

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



Accelerate and brake behaviors must be declared in each new car model

According to the 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 aggregation 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 that 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 for CSharp article (http://www.webbiscuit.co.uk/articles/the-strategy-pattern/)
- Strategy pattern in UML and in LePUS3 (http://www.lepus.org.uk/ref/companion/Strategy.xml) (a formal modelling notation)
- Refactoring: Replace Type Code with State/Strategy (http://martinfowler.com/refactoring/catalog/replaceTypeCodeWithStateStrategy.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.

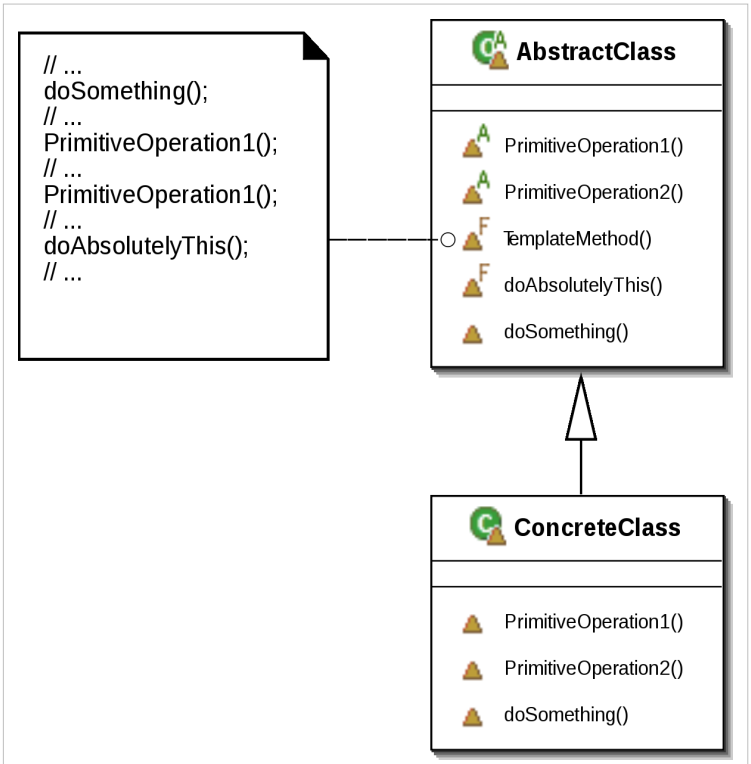
The template method thus manages the larger picture of task semantics, and more refined implementation details of selection and sequence of methods. This larger picture calls abstract and non-abstract methods for the task at hand. The non-abstract methods are completely controlled by the template method 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.

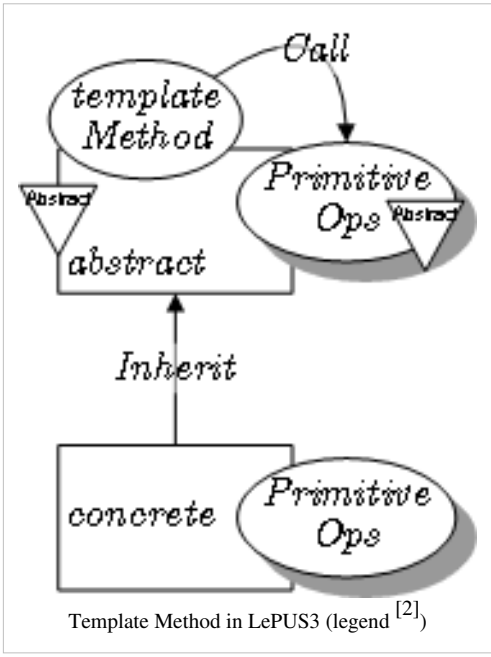
The Template method pattern is strongly related to the Non-Virtual Interface (NVI) pattern. The NVI pattern recognizes the benefits of a non-abstract method invoking the subordinate abstract methods. This level of indirection allows for pre and post operations relative to the abstract operations both 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.

Structure



Template method: UML class diagram.



Template Method in LePUS3 (legend [2])

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 : */
    public 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 players
        // 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() {
        // Initialize players
        // 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 [1]
- Working with Template Classes in PHP 5 [2]
- Template Method pattern in UML and in LePUS3 [3] (a formal modelling language)
- Difference between Adapter and Template Method pattern [4]
- Template Method Design Pattern [5]

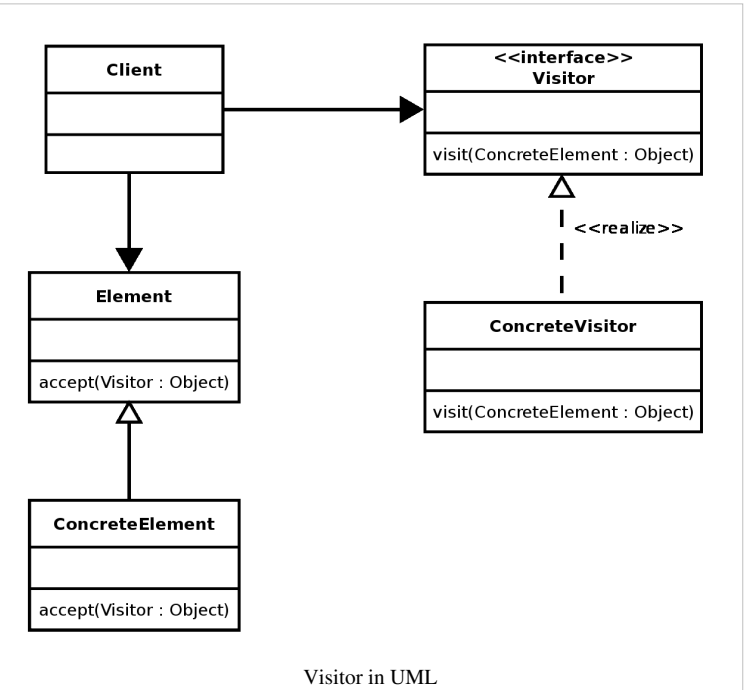
References

[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

Visitor pattern

In object-oriented programming and software engineering, the **visitor** design pattern is a way of separating an algorithm from an object structure on which it operates. 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.

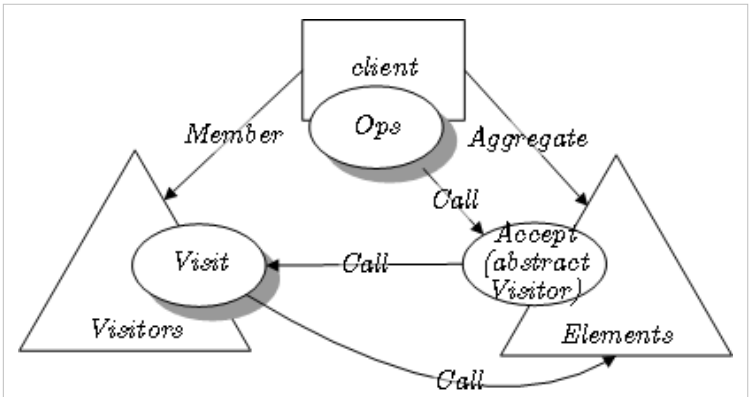
In essence, the visitor allows one to add new virtual functions to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function. The visitor takes the instance reference as input, and implements the goal through double dispatch.



Visitor in UML

Motivation

Consider the design of a 2D CAD system. At its core there are several types to represent basic geometric shapes like circles, lines and arcs. The entities are ordered into layers, and at the top of the type hierarchy is the drawing, which is simply a list of layers, plus some additional properties.



Visitor in LePUS3 (legend [2])

A fundamental operation on this type hierarchy is saving the drawing to the system's native file format. It seems relatively fine to add local save methods to all types in the hierarchy. But then we also want to be able to save drawings to other file formats, and adding more and more methods for saving into lots of different file formats soon clutters the relatively pure geometric data structure we started out with.

A naive way to solve this would be to maintain separate functions for each file format. Such a save function would take a drawing as input, traverse it and encode into that specific file format. But if you do this for several different formats, you soon begin to see lots of duplication between the functions, e.g. lots of type-of if statements and traversal loops. Another problem with this approach is how easy it is to miss a certain shape in some saver.

Instead, you could apply the Visitor pattern. The Visitor pattern encodes a logical operation on the whole hierarchy into a single class containing one method per type. In our CAD example, each save function would be implemented as a separate Visitor subclass. This would remove all duplication of type checks and traversal steps. It would also make the compiler complain if you leave out a shape.

Details

The visitor pattern requires a programming language that supports single dispatch and method overloading. Under these conditions, consider two objects, each of some class type; one is called the "element", and the other is called the "visitor". An element has an `accept()` method that can take the visitor as an argument. The `accept()` method calls a `visit()` method of the visitor; the element passes itself as an argument to the `visit()` method. Thus:

- When the `accept()` method is called in the program, its implementation is chosen based on both:
  - The dynamic type of the element.
  - The static type of the visitor.
- When the associated `visit()` method is called, its implementation is chosen based on both:
  - The dynamic type of the visitor.
  - The static type of the element as known from within the implementation of the `accept()` method, which is the same as the dynamic type of the element. (As a bonus, if the visitor can't handle an argument of the given element's type, then the compiler will catch the error.)
- Consequently, the implementation of the `visit()` method is chosen based on both:
  - The dynamic type of the element.
  - The dynamic type of the visitor.

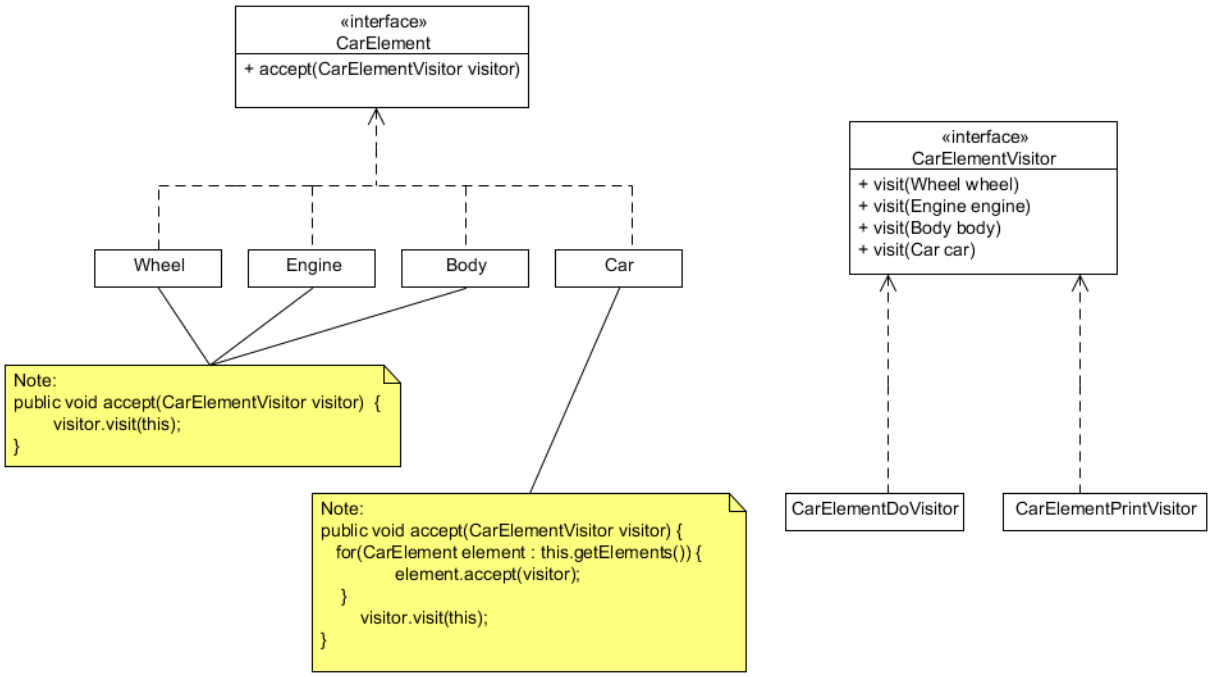
    This effectively implements double dispatch; indeed, because the Lisp language's object system supports multiple dispatch (not just single dispatch), implementing the visitor pattern in Lisp is trivial.

In this way, a single algorithm can be written for traversing a graph of elements, and many different kinds of operations can be performed during that traversal by supplying different kinds of visitors to interact with the elements based on the dynamic types of both the elements and the visitors.

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) {
        /*
         * accept(CarElementVisitor) in Wheel implements
         * accept(CarElementVisitor) in CarElement, so the call
         * to accept is bound at run time. This can be considered
         * the first dispatch. However, the decision to call
         * visit(Wheel) (as opposed to visit(Engine) etc.) can be
         * made during compile time since 'this' is known at compile
         * time to be a Wheel. Moreover, each implementation of
         * CarElementVisitor implements the visit(Wheel), which is
         * another decision that is made at run time. This can be
         * considered the second dispatch.
         */
        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 Car() {
        //create new Array of elements
        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 elem : elements) {
            elem.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());
    }
```

```
}
```

Note: A more flexible approach to this pattern is to create a wrapper class implementing the interface defining the accept method. The wrapper contains a reference pointing to the CarElement which could be initialized through the constructor. This approach avoids having to implement an interface on each element. [see article Java Tip 98 article below]

Output

```
Visiting front left wheel
Visiting front right wheel
Visiting back left wheel
Visiting back right wheel
Visiting body
Visiting engine
Visiting car
Kicking my front left wheel
Kicking my front right wheel
Kicking my back left wheel
Kicking my back right wheel
Moving my body
Starting my engine
Starting my car
```

Lisp Example

Source

```
(defclass auto ()
  ((elements :initarg :elements)))

(defclass auto-part ()
  ((name :initarg :name :initform "<unnamed-car-part>")))

(defmethod print-object ((p auto-part) stream)
  (print-object (slot-value p 'name) stream))

(defclass wheel (auto-part) ())

(defclass body (auto-part) ())

(defclass engine (auto-part) ())

(defgeneric traverse (function object other-object))

(defmethod traverse (function (a auto) other-object)
  (with-slots (elements) a
    (dolist (e elements)
      (funcall function e other-object))))
```

```
;; do-something visitations

;; catch all
(defmethod do-something (object other-object)
  (format t "don't know how ~s and ~s should interact~%" object
other-object))

;; visitation involving wheel and integer
(defmethod do-something ((object wheel) (other-object integer))
  (format t "kicking wheel ~s ~s times~%" object other-object))

;; visitation involving wheel and symbol
(defmethod do-something ((object wheel) (other-object symbol))
  (format t "kicking wheel ~s symbolically using symbol ~s~%" object
other-object))

(defmethod do-something ((object engine) (other-object integer))
  (format t "starting engine ~s ~s times~%" object other-object))

(defmethod do-something ((object engine) (other-object symbol))
  (format t "starting engine ~s symbolically using symbol ~s~%" object
other-object))

(let ((a (make-instance 'auto
                        :elements `(,(make-instance 'wheel :name
"front-left-wheel")
                                    ,(make-instance 'wheel :name
"front-right-wheel")
                                    ,(make-instance 'wheel :name
"rear-right-wheel")
                                    ,(make-instance 'wheel :name
"rear-right-wheel")
                                    ,(make-instance 'body :name "body")
                                    ,(make-instance 'engine :name
"engine")))))
  ;; traverse to print elements
  ;; stream *standard-output* plays the role of other-object here
  (traverse #'print a *standard-output*)

  (terpri) ;; print newline

  ;; traverse with arbitrary context from other object
  (traverse #'do-something a 42)

  ;; traverse with arbitrary context from other object
  (traverse #'do-something a 'abc))
```

Output

```
"front-left-wheel"
"front-right-wheel"
"rear-right-wheel"
"rear-right-wheel"
"body"
"engine"
kicking wheel "front-left-wheel" 42 times
kicking wheel "front-right-wheel" 42 times
kicking wheel "rear-right-wheel" 42 times
kicking wheel "rear-right-wheel" 42 times
don't know how "body" and 42 should interact
starting engine "engine" 42 times
kicking wheel "front-left-wheel" symbolically using symbol ABC
kicking wheel "front-right-wheel" symbolically using symbol ABC
kicking wheel "rear-right-wheel" symbolically using symbol ABC
kicking wheel "rear-right-wheel" symbolically using symbol ABC
don't know how "body" and ABC should interact
starting engine "engine" symbolically using symbol ABC
```

Notes

The `other-object` parameter is superfluous in `traverse`. The reason is that it is possible to use an anonymous function which calls the desired target method with a lexically captured object:

```
(defmethod traverse (function (a auto)) ;; other-object removed
  (with-slots (elements) a
    (dolist (e elements)
      (funcall function e)))) ;; from here too

;; ...

  ;; alternative way to print-traverse
  (traverse (lambda (o) (print o *standard-output*)) a)

  ;; alternative way to do-something with
  ;; elements of a and integer 42
  (traverse (lambda (o) (do-something o 42)) a)
```

Now, the multiple dispatch occurs in the call issued from the body of the anonymous function, and so `traverse` is just a mapping function which distributes a function application over the elements of an object. Thus all traces of the Visitor Pattern disappear, except for the mapping function, in which there is no evidence of two objects being involved. All knowledge of there being two objects and a dispatch on their types is in the lambda function.

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 [1] 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 [2] (a Design Description Language)
- Article "Componentization: the Visitor Example [3] by Bertrand Meyer and Karine Arnout, *Computer* (IEEE), vol. 39, no. 7, July 2006, pages 23-30.
- Article A Type-theoretic Reconstruction of the Visitor Pattern [4]
- Article "The Essence of the Visitor Pattern [5]" 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 [6]" by Bruce Wallace - subtitled *"Java 1.2's reflection capabilities eliminate burdensome accept() methods from your Visitor pattern"*
- Visitor Patterns [7] as a universal model of terminating computation.
- Visitor Pattern [8] using reflection(java).
- PerfectJPattern Open Source Project [9], Provides a context-free and type-safe implementation of the Visitor Pattern in Java based on Delegates.
- Visitor Design Pattern [10]
- Article Java Tip 98: Reflect on the Visitor design pattern [11]

References

[1] http://objectmentor.com/resources/articles/visitor.pdf
[2] http://www.lepus.org.uk/ref/companion/Visitor.xml
[3] http://se.ethz.ch/~meyer/publications/computer/visitor.pdf
[4] http://www.cs.bham.ac.uk/~hxt/research/mfps-visitors.pdf
[5] http://citeseer.ist.psu.edu/palsberg97essence.html
[6] http://www.polyglotinc.com/reflection.html
[7] http://goblin.colourcountry.net/apt1002/Visitor%20patterns
[8] http://www.oodesign.com/oo_design_patterns/behavioral_patterns/visitor_pattern.html
[9] http://perfectjpattern.sourceforge.net/dp-visitor.html
[10] http://sourcemaking.com/design_patterns/visitor
[11] http://www.javaworld.com/javaworld/javatips/jw-javatip98.html

Concurrency patterns

Concurrency pattern

In software engineering, **concurrency patterns** are those types of design patterns that deal with the multi-threaded programming paradigm. Examples of this class of patterns include:

- Active Object[1][2]
- Balking pattern
- Double checked locking pattern
- Guarded suspension
- Leaders/followers pattern
- Monitor Object
- Reactor pattern
- Read write lock pattern
- Scheduler pattern
- Thread pool pattern
- Thread-Specific Storage

References

[1] Douglas C. Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann "Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects", Wiley, 2000

[2] R. Greg Lavender, Doublas C Scmidt (1995). "Active Object" (http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf). . Retrieved 2010-06-17.

External links

Recordings about concurrency patterns from Software Engineering Radio:

- Episode 12: Concurrency Pt. 1 (http://www.se-radio.net/2006/04/episode-12-concurrency-pt-1/)
- Episode 19: Concurrency Pt. 2 (http://www.se-radio.net/2006/06/episode-19-concurrency-pt-2/)
- Episode 29: Concurrency Pt. 3 (http://www.se-radio.net/2006/09/episode-29-concurrency-pt-3/)

Active object

The **active object** design pattern decouples method execution from method invocation that reside in their own thread of control.[1] The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests.[2]

The pattern consists of six elements:[3]

- 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 synchronized is because the only way to guarantee another thread will see a change to a field is to synchronize access all access to it or declare it as volatile.

```
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() {
        synchronized(this) {
            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 HTTP is a *request-response* pattern protocol, and the UDP has a *one-way* pattern.[1]

SOAP

The term "Message Exchange Pattern" has a specific meaning within the SOAP protocol.[2][3] 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: The reverse of In-Only. It primarily supports event notification. It cannot trigger a fault message.
6. Robust Out-Only: similar to the out-only pattern, except it can trigger a fault message. The outbound message initiates the transmission.
7. Out-In: The reverse of In-Out. The provider transmits the request and initiates the exchange.
8. Out-Optional-In: The reverse of In-Optional-Out. The service produces an outbound message. The incoming message is optional ("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:[4]

- **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. [5]

References

[1] Erl, Thomas (2005). *Service Oriented Architecture: Concepts, Technology, and Design*. Indiana: Pearson Education. pp. 171. ISBN 0-13-185858-0.
[2] http://www.w3.org/TR/soap12-part1/#soapmep SOAP MEPs in SOAP W3C Recommendation v1.2
[3] Web Services Description Language (WSDL) Version 2.0: Additional MEPs (http://www.w3.org/TR/wsdl20-additional-meps/)
[4] ØMQ User Guide (http://www.zeromq.org/docs:user-guide)
[5] 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)
- Enterprise Integration Patterns - Pattern Catalog (http://www.eaipatterns.com/toc.html)

Double-checked locking

In software engineering, **double-checked locking** (also known as "double-checked locking optimization"[1]) 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. At times, it can be considered an anti-pattern.[2]

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 [3] (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.

The lock is obtained 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,[4] 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.

For example, in Java if a call to a constructor has been inlined then the shared variable may immediately be updated once the storage has been allocated but before the inlined constructor initializes the object.[5]

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 [3]:

```
// 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.[6]

If the helper object is static (one per class loader), an alternative is the initialization on demand holder idiom [7] See Listing 16.6 on [8]

```
// 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`:[9]

```
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++ 2005 and above if the pointer to the resource is declared with the C++ keyword **volatile**. Visual C++ 2005 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).[10] 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) { // 1st check
            lock (myLock) {
                if (!ready) { // 2nd (double) check
                    mySingleton = new MySingleton();
                    // Fence for write-release semantics
                    System.Threading.Thread.MemoryBarrier();
                    ready = true;
                }
            }
        }
        // Fence for read-acquire semantics
        // This fence is needed to avoid reordering of reads that might
        // cause stale values of mySingleton to be seen.
        System.Threading.Thread.MemoryBarrier();
        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.

```
public class MySingleton {
    private static object myLock = new object();
    private static volatile MySingleton mySingleton = null;

    private MySingleton() {
    }

    public static MySingleton GetInstance() {
        if (mySingleton == null) { // check
            lock (myLock) {
                if (mySingleton == null) { // double check, volatile
ensures that the value is re-read
                    mySingleton = new MySingleton();
```

```
                }
            }
        }
        return mySingleton;
    }
}
```

In .NET Framework 4.0, the `Lazy<T>` class was introduced, which internally uses double-checked locking by default:[11]

```
public class MySingleton
{
    private static readonly Lazy<MySingleton> mySingleton =
        new Lazy<MySingleton>(() => new MySingleton());

    private MySingleton()
    { }

    public static MySingleton GetInstance()
    {
        return mySingleton.Value;
    }
}
```

References

[1] Schmidt, D et al. Pattern-Oriented Software Architecture Vol 2, 2000 pp353-363

[2] David Bacon et al. The "Double-Checked Locking is Broken" Declaration (http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html).

[3] http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html

[4] Boehm, Hans-J. "Threads Cannot Be Implemented As a Library", ACM 2005, p265

[5] Haggar, Peter (1 May 2002). "Double-checked locking and the Singleton pattern" (http://www.ibm.com/developerworks/java/library/j-dcl/index.html). IBM. .

[6] Joshua Bloch "Effective Java, Second Edition", p. 283

[7] Brian Goetz et al. Java Concurrency in Practice, 2006 pp348

[8] (http://www.javaconcurrencyinpractice.com/listings.html)

[9] (https://mailman.cs.umd.edu/mailman/private/javamemorymodel-discussion/2010-July/000422.html) Javamemorymodel-discussion mailing list

[10] http://msdn.microsoft.com/en-us/library/12a04hfd(VS.100).aspx

[11] Albahari, Joseph (2010). "Threading in C#: Using Threads" (http://www.albahari.com/threading/part3.aspx#_LazyT). *C# 4.0 in a Nutshell*. O'Reilly Media. ISBN 0596800959. . "`Lazy<T>` actually implements […] double-checked locking. Double-checked locking performs an additional volatile read to avoid the cost of obtaining a lock if the object is already initialized."

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)" (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
- Article " Warning! Threading in a multiprocessor world (http://www.javaworld.com/javaworld/jw-02-2001/jw-0209-toolbox.html)" by Allen Holub
- 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)

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.[1] It is equivalent to the **IOU pattern** described in 1996 by Allan Vermeulen.[2][3] 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.[4] 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.[5] 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.[6] 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.[7][8]

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`.[9][10] 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.[11]

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. Dobb's Journal*. . Retrieved 22 November 2008.

[3] Nash, Trey (2007). "Threading in C#". *Accelerated C# 2008*. Apress. ISBN 978-1-59059-873-3.

[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. Archived (http://web.archive.org/web/20081122091746/http://msdn.microsoft.com/en-us/library/ms228969.aspx) from the original on 22 November 2008. . 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. Archived (http://web.archive.org/web/20081207092841/http://msdn.microsoft.com/en-us/library/ms228963.aspx) from the original on 7 December 2008. . 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. Archived (http://web.archive.org/web/20081223205326/http://msdn.microsoft.com/en-us/library/ms228972.aspx) from the original on 23 December 2008. . 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 0-470-19137-6.

[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. Archived (http://web.archive.org/web/20081225175311/http://msdn.microsoft.com/en-us/library/hkasytyf.aspx) from the original on 25 December 2008. . 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. Archived (http://web.archive.org/web/20081122092048/http://msdn.microsoft.com/en-us/library/ms228966.aspx) from the original on 22 November 2008. . 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 0-596-51037-3.
- Using asynchronous method calls in C# (http://articles.techrepublic.com.com/5100-10878_11-1044325.html)

Guarded suspension

In concurrent programming, **guarded suspension**[1] 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

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 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 free - set it */
    lock = myPID;
}
```

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. A number of strategies can be used to avoid or recover from deadlocks or livelocks, 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.)

Some languages do support locks syntactically. An example in C# follows:

```
class Account {      // this is a monitor of an account
  long val = 0;
  object thisLock = new object();
  public void Deposit(const long x) {
    lock (thisLock) {    // only 1 thread at a time may execute this
statement
      val += x;
    }
  }

  public void Withdraw(const long x) {
    lock (thisLock) {
      val -= x;
    }
  }
}
```

lock (this) is a problem if the instance can be accessed publicly.[1]

Similar to Java, C# can also synchronize entire methods, by using the MethodImplOptions.Synchronized attribute.[2][3]

```
 [MethodImpl(MethodImplOptions.Synchronized)]
 public void SomeMethod () {
     // do stuff
  }
```

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.

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. Granular locking where each process must hold multiple locks from a common set of locks can create subtle lock dependencies. This subtlety can increase the chance that a programmer will unknowingly introduce a **deadlock** .

In a database management system, for example, a lock could protect, in order of decreasing 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,[4] 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, where (at least) two threads all wait for a lock that another thread holds and will not give up until it has acquired the other lock.
- 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.

Any such "concurrency control strategy" would require actual lock mechanisms implemented at a more fundamental level of the operating software (the analogues mentioned above), which may only relieve the application level from the details of implementation. The "problems" remain, but are dealt with beneath the application. In fact, proper locking ultimately depends upon the CPU hardware itself providing a method of atomic instruction stream synchronization. For example, the addition or deletion of an item into a pipeline requires that all contemporaneous operations needing to add or delete other items in the pipe be suspended during the manipulation of the memory content required to add or delete the specific item. The design of an application is better when it recognizes the burdens it places upon an operating system and is capable of graciously recognizing the reporting of impossible demands.

Language support

Language support for locking depends on the language used:

- There is no API to handle mutexes in the ISO/IEC standard for C. The current ISO C++ standard, C++11, supports 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.[5] 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.[6] 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[7] 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.[8]
- Ruby also doesn't provide a keyword for synchronization, but it is possible to use an explicit low level mutex object.[9]
- 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"[10] to put locks on blocks of code and also provides the classes NSLock,[11] NSRecursiveLock,[12] and NSConditionLock[13] along with the NSLocking protocol[14] for locking as well.
- Ada is probably worth looking at too for a comprehensive overview, with its protected objects[15][16] and rendezvous.

References

[1] "lock Statement (C# Reference)" (http://msdn.microsoft.com/en-us/library/c5kehkcz(v=vs.100).aspx). .

[2] "ThreadPoolPriority, and MethodImplAttribute" (http://msdn.microsoft.com/en-us/magazine/cc163896.aspx). : MSDN. p. ??. . Retrieved 2011-11-22.

[3] "C# From a Java Developer's Perspective" (http://www.25hoursaday.com/CsharpVsJava.html#attributes). . . Retrieved 2011-11-22.

[4] "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.

[5] Marshall, Dave (March 1999). "Mutual Exclusion Locks" (http://www.cs.cf.ac.uk/Dave/C/node31.html#SECTION00311000000000000000). . Retrieved 2008-05-30.

[6] "Synchronize" (http://msdn.microsoft.com/en-us/library/34d2s8k3(VS.80).aspx). msdn.microsoft.com. . Retrieved 2008-05-30.

[7] "Synchronization" (http://java.sun.com/docs/books/tutorial/essential/concurrency/sync.html). Sun Microsystems. . Retrieved 2008-05-30.

[8] Lundh, Fredrik (July 2007). "Thread Synchronization Mechanisms in Python" (http://effbot.org/zone/thread-synchronization.htm). . Retrieved 2008-05-30.

[9] "Programming Ruby: Threads and Processes" (http://www.ruby-doc.org/docs/ProgrammingRuby/html/tut_threads.html). 2001. . Retrieved 2008-05-30.

[10] "Apple Threading Reference" (http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjectiveC/Chapters/ocThreading.html). Apple, inc. . Retrieved 2009-10-17.

[11] "NSLock Reference" (http://developer.apple.com/mac/library/documentation/Cocoa/Reference/Foundation/Classes/NSLock_Class/Reference/Reference.html). Apple, inc. . Retrieved 2009-10-17.

[12] "NSRecursiveLock Reference" (http://developer.apple.com/mac/library/documentation/Cocoa/Reference/Foundation/Classes/NSRecursiveLock_Class/Reference/Reference.html). Apple, inc. . Retrieved 2009-10-17.

[13] "NSConditionLock Reference" (http://developer.apple.com/mac/library/documentation/Cocoa/Reference/Foundation/Classes/NSConditionLock_Class/Reference/Reference.html). Apple, inc. . Retrieved 2009-10-17.

[14] "NSLocking Protocol Reference" (http://developer.apple.com/mac/library/documentation/Cocoa/Reference/Foundation/Protocols/NSLocking_Protocol/Reference/Reference.html). Apple, inc. . Retrieved 2009-10-17.

[15] 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."

[16] 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.

External links

- Tutorial on Locks and Critical Sections http://www.futurechips.org/tips-for-power-coders/parallel-programming-understanding-impact-critical-sections.html

Monitor

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 to reasoning about parallel code that updates a data structure.

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 [1] and Per Brinch Hansen, [2] 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* give priority to a signaled thread.
- *Nonblocking condition variables* 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*. 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. Monitors using blocking condition variables are often called *Hoare-style* monitors or *signal-and-urgent-wait* monitors.

We assume there are two queues of threads associated with each monitor object

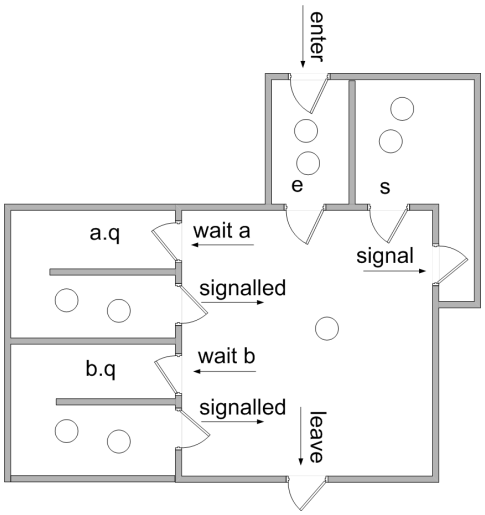
- `e` is the entrance queue
- `s` is a queue of threads that have signaled.

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 (in all futures, each thread that enters the queue will be chosen infinitely often) 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.)



A Hoare style monitor with two condition variables `a` and `b`. After Buhr *et al.*

```
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[3] and Buhr *et al.*,[4] 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 ;
```

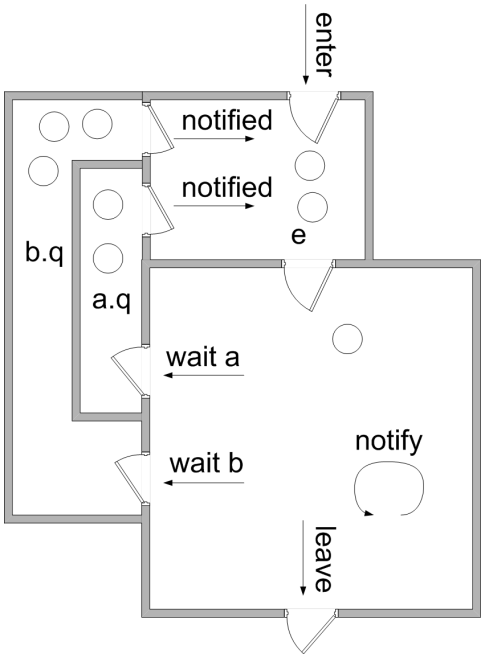
```
    assert 0 <= size and size < capacity
    signal theStackIsNotFull  and return A[size]
  }
}
```

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.)



A Mesa style monitor with two condition variables `a` and `b`

```
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[5] and Buhr *et al.*[6] 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 **notify**ing 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
```

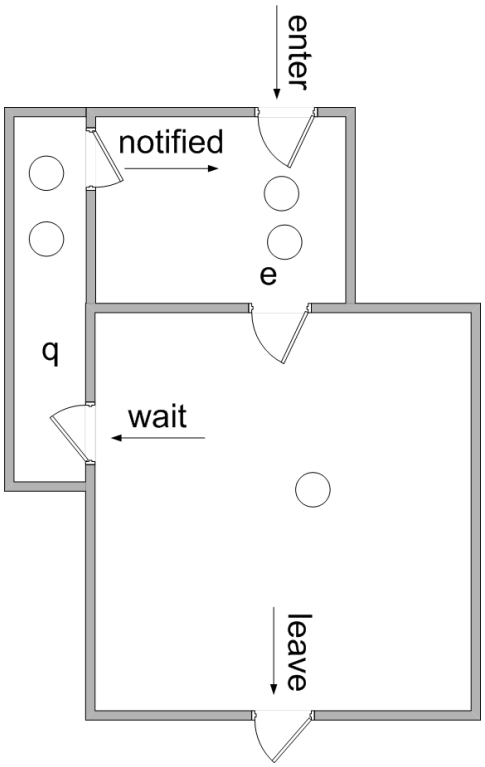
```
    }
}
```

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. Methods requiring mutual exclusion must be explicitly marked with the **synchronized** keyword. Blocks of code may also be marked by **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 been adopted in other languages, for example C#.



A Java style monitor

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. [7] 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 methods)
- Mesa
- Modula-3
- Python (via threading.Condition [8] object)
- Ruby
- Squeak Smalltalk
- Turing, Turing+, and Object-Oriented Turing
- μC++

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) [11]
- "Monitors: An Operating System Structuring Concept [12]" by C. A. R. Hoare
- "Signalling in Monitors [13]" by John H. Howard (computer scientist)
- "Proving Monitors [14]" by John H. Howard (computer scientist)
- "Experience with Processes and Monitors in Mesa [15]" by Butler W. Lampson and David D. Redell
- pthread_cond_wait [16] – description from the Open Group Base Specifications Issue 6, IEEE Std 1003.1
- "Block on a Condition Variable [17]" by Dave Marshall (computer scientist)
- "Strategies for Implementing POSIX Condition Variables on Win32 [18]" by Douglas C. Schmidt and Irfan Pyarali
- Condition Variable Routines [19] from the Apache Portable Runtime Library
- wxCondition description [20]
- Boost Condition Variables Reference [21]
- ZThread Condition Class Reference [22]
- Wefts::Condition Class Reference [23]
- ACE_Condition Class Template Reference [24]
- QWaitCondition Class Reference [25]
- Common C++ Conditional Class Reference [26]
- at::ConditionalMutex Class Reference [27]
- threads::shared [28] – Perl extension for sharing data structures between threads
- Tutorial multiprocessing traps [29]
- http://msdn.microsoft.com/en-us/library/ms682052(VS.85).aspx
- Monitors [30] 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

[17] http://gd.tuwien.ac.at/languages/c/programming-dmarshall/node31.html#SECTION003125000000000000000

[18] http://www.cs.wustl.edu/~schmidt/win32-cv-1.html

[19] http://apr.apache.org/docs/apr/group__apr__thread__cond.html

[20] http://wxwidgets.org/manuals/2.6.3/wx_wxcondition.html

[21] http://www.boost.org/doc/html/thread/synchronization.html#thread.synchronization.condvar_ref

[22] http://zthread.sourceforge.net/html/classZThread_1_1Condition.html

[23] http://wefts.sourceforge.net/wefts-apidoc-0.99c/classWefts_1_1Condition.html

[24] 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

[27] 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

Reactor pattern

The reactor design pattern is an event handling 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.[1] (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

C

- libevent

C++

- POCO C++ Libraries

Java

- Apache MINA
- Apache Cocoon (for XML processing)
- JBoss Netty

JavaScript

- Node.js

Perl

- POE

Python

- Twisted

Ruby

- EventMachine

References

[1] Kegel, Dan, *The C10K problem* (http://www.kegel.com/c10k.html#nb.select), , retrieved 2007-07-28

External links

- An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events (http://www.dre.vanderbilt.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 as the **multiple readers / single-writer lock**[1] or the **multi-reader lock**,[2] 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 a shared resource, allowing concurrent access to multiple threads for reading but restricting access to a single thread for writes (or other changes) to the resource. A common use might be to control access to a data structure in memory that can't be updated atomically but isn't valid (and shouldn't be read by another thread) until the update is complete.

One potential problem with a conventional RW lock is that it can lead to write-starvation if contention is high enough, 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.[3] The downside is that it's less performant because each operation, taking or releasing the lock for either read or write, is more complex, internally requiring taking and releasing two mutexes instead of one.[3][4] This variation is sometimes known as a "write-preferring" or "write-biased" readers-writer lock.[5][6]

Readers–writer locks are usually constructed on top of mutexes and condition variables, or on top of semaphores.

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 POSIX standard `pthread_rwlock_t` and associated operations.[7]
- The C language Win32 multiple-reader/single-writer lock used in Hamilton C shell.[1][4] The Hamilton lock presumes contention is low enough that writers are unlikely to be starved,[8] prompting Jordan Zimmerman to suggest a modified version to avoid starvation.[3]
- The ReadWriteLock[9] interface and the ReentrantReadWriteLock[10] locks in Java version 5 or above.
- A simple Windows API implementation by Glenn Slayden.[11]
- The Microsoft `System.Threading.ReaderWriterLockSlim` lock for C# and other .NET languages.[12]
- The `boost::shared_mutex` in read/write lock the Boost C++ Libraries.[13]
- A pseudo-code implementation in the Readers-writers problem article.

References

[1] Hamilton, Doug (21 April 1995). "[news:hamilton.798430053@BIX.com Suggestions for multiple-reader/single-writer lock?]". [news:comp.os.ms-windows.nt.misc comp.os.ms-windows.nt.misc]. (Web link) (http://groups.google.com/group/comp.os.ms-windows.programmer.win32/msg/77533bcc6197c627?hl=en). Retrieved 8 October 2010.

[2] "Practical lock-freedom" (http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-579.pdf) by Keir Fraser 2004

[3] Jordan Zimmerman (21 October 1999). "Single-writer Multi-Reader lock for Win98". [news:comp.programming.threads comp.programming.threads]. (Web link) (http://groups.google.com/group/comp.programming.threads/msg/b831174c04245657?hl=en). Retrieved 17 May 2011.

[4] Nicole Hamilton (19 October 1999). "Single-writer Multi-Reader lock for Win98". [news:comp.programming.threads comp.programming.threads]. (Web link) (http://groups.google.com/group/comp.programming.threads/msg/ea070d971d7663e2?hl=en). Retrieved 17 May 2011.

[5] "ReaderWriterLock Alternative" (http://www.codeplex.com/ReaderWriterLockAlt) an open source C# implementation of a write-biased readers-writer lock

[6] `java.util.concurrent.locks.ReentrantReadWriteLock` Java readers-writer lock implementation offers a "fair" mode

[7] "The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition: pthread_rwlock_destroy" (http://www.opengroup.org/onlinepubs/009695399/functions/pthread_rwlock_init.html). The IEEE and The Open Group. . Retrieved 14 May 2011.

[8] Ziv Caspi (20 October 1999). "Re: Single-writer Multi-Reader lock for Win98". [news:comp.programming.threads comp.programming.threads]. (Web link) (http://groups.google.com/group/comp.programming.threads/msg/52224268c2952cda?hl=en). "Forgive me for saying so, but this implementation favors readers instead of the writer. If there are many readers, the writer will never have a chance to write". Retrieved 7 October 2011.

[9] `java.util.concurrent.locks.ReadWriteLock`

[10] `java.util.concurrent.locks.ReentrantReadWriteLock`

[11] Glenn Slayden. "Multiple-Reader, Single-Writer Synchronization Lock Class" (http://www.glennslayden.com/code/win32/reader-writer-lock). . Retrieved 14 May 2011.

[12] "ReaderWriteLockSlim Class (System.Threading)" (http://msdn.microsoft.com/en-us/library/system.threading.readerwriterlockslim.aspx). Microsoft Corporation. . Retrieved 14 May 2011.

[13] Anthony Williams. "Synchronization – Boost 1.46.1" (http://www.boost.org/doc/html/thread/synchronization.html#thread.synchronization.mutex_types.shared_mutex). . Retrieved 14 May 2011.

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.

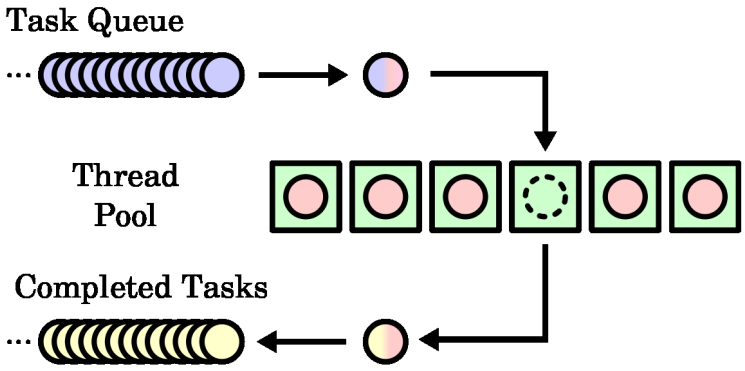
The scheduler pattern uses an object that explicitly sequences waiting threads. It provides a mechanism to implement a scheduling policy, but is independent of any specific scheduling policy — the policy is encapsulated in its own class and is reusable.

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



A sample thread pool (green boxes) with waiting tasks (blue) and completed tasks (yellow)

In computer programming, the **thread pool pattern** (also **replicated workers**) is where a number of threads are created to perform a number of tasks, which are usually organized in a queue. The results from the tasks being executed might also be placed in a queue, or if the tasks return no result (for example, if the task is for animation).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. In Java, you can synchronize the relevant method using the `synchronized` keyword. This will bind the block modified with `synchronized` into one atomic structure, therefore forcing any threads using the associated resource to wait until there are no threads using the resource. As a drawback to this method, synchronization is rather expensive. You can also create an object that holds a list of all the jobs in a queue, which could be a Singleton.

Typically, a thread pool executes on a single computer. However, thread pools are conceptually related to server farms in which a master process, which might be a thread pool itself, 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 [1]" by Binildas C. A.
- Article "Thread pools and work queues [2]" by Brian Goetz
- Article "A Method of Worker Thread Pooling [3]" by Pradeep Kumar Sahu
- Article "Work Queue [4]" by Uri Twig
- Article "Windows Thread Pooling and Execution Chaining [5]"
- Article "Smart Thread Pool [6]" by Ami Bar
- Article "Programming the Thread Pool in the .NET Framework [7]" by David Carmona
- Article "The Thread Pool and Asynchronous Methods [8]" by Jon Skeet
- Article "Creating a Notifying Blocking Thread Pool in Java [9]" by Amir Kirsh
- Article "Practical Threaded Programming with Python: Thread Pools and Queues [10]" by Noah Gift
- Paper "Optimizing Thread-Pool Strategies for Real-Time CORBA [11]" by Irfan Pyarali, Marina Spivak, Douglas C. Schmidt and Ron Cytron
- Conference Paper "Deferred cancellation. A behavioral pattern [12]" 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
[4] 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://today.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 call 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 application programming interface (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.

C++

C++11 introduces the `thread_local`[1] keyword which can be used in the following cases

- Namespace level (global) variables
- File static variables
- Function static variables
- Static member variables

Aside from that, various C++ compiler implementations provide specific ways to declare thread-local variables:

- Solaris Studio C/C++, IBM XL C/C++, GNU C and Intel C/C++ (Linux systems) use the syntax:

```
__thread int number;
```

- Visual C++[2], Intel C/C++ (Windows systems), C++Builder, and Digital Mars C++ use the syntax:

```
__declspec(thread) int number;
```

- C++Builder also supports the syntax:

```
int __thread number;
```

On Windows versions before 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).[3]

Common Lisp (and maybe other dialects)

Common Lisp provides a feature called dynamically scoped variables.

Dynamic variables have a binding which is private to the invocation of a function and all of the children called by that function.

This abstraction naturally maps to thread-specific storage, and Lisp implementations that provide threads do this. Common Lisp has numerous standard dynamic variables, and so threads cannot be sensibly added to an implementation of the language without these variables having thread-local semantics in dynamic binding.

For instance the standard variable `*print-base*` determines the default radix in which integers are printed. If this variable is overridden, then all enclosing code will print integers in an alternate radix:

```
;;; function foo and its children will print
;; in hexadecimal:
(let ((*print-base* 16)) (foo))
```

If functions can execute concurrently on different threads, this binding has to be properly thread local, otherwise each thread will fight over who controls a global printing radix.

D

In D 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.
```

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>();
```

.NET languages: C# and others

In .NET Framework languages such as C#, static fields can be marked with the ThreadStatic attribute [4]:

```
class FooBar {
   [ThreadStatic] static int foo;
}
```

In .NET 4.0 the System.Threading.ThreadLocal<T> [5] class is available for allocating and lazily loading thread local variables.

```
class FooBar {
   private static System.Threading.ThreadLocal<int> foo;
}
```

Also an API [6] is available for dynamically allocating thread local variables.

Object Pascal

In Object Pascal (Delphi) or Free Pascal the *threadvar* reserved keyword can be used instead of 'var' to declare variables using the thread-local storage.

```
var
   mydata_process: integer;
threadvar
   mydata_threadlocal: integer;
```

Perl

In Perl threads were added late in the evolution of the language, after a large body of extant code was already present on the Comprehensive Perl Archive Network (CPAN). Thus, threads in Perl by default take their own local storage for all variables, to minimise the impact of threads on extant 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;
```

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
```

Underlying implementation in Microsoft Windows

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 accessed via a table. (Actually, two tables, but they appear as one.[7]) 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 Windows, there is a Win32 Thread Information Block for each thread. One of the entries in this block is the thread-local storage table for that thread.[8] On x86 systems, the address of the Thread Information Block is stored in the FS register. In this way, access to thread-local storage carries a minimal overhead.

References

[1] Section 3.7.2 in C++11 standard

[2] The thread extended storage-class modifier (http://msdn.microsoft.com/en-us/library/9w1sdazb.aspx)

[3] "Rules and Limitations for TLS" (http://msdn2.microsoft.com/en-us/library/2s9wt68x.aspx)

[4] http://msdn2.microsoft.com/en-us/library/system.threadstaticattribute(vs.80).aspx

[5] http://msdn.microsoft.com/en-us/library/dd642243.aspx

[6] http://msdn2.microsoft.com/en-us/library/system.threading.thread.getnameddataslot(vs.80).aspx

[7] Johnson, Ken (23 October 2007). "Thread Local storage, part 2: Explicit TLS" (http://www.nynaeve.net/?p=181). . Retrieved 6 April 2011.

[8] Pietrek, Matt (May 2006). "Under the Hood" (http://www.microsoft.com/msj/archive/s2ce.aspx). *MSDN*. . Retrieved 6 April 2010.

External links

- ELF Handling For Thread-Local Storage (http://www.akkadia.org/drepper/tls.pdf) — Document about an implementation in C or C++.
- ACE_TSS< TYPE > Class Template Reference (http://www.dre.vanderbilt.edu/Doxygen/Stable/ace/classACE__TSS.html#_details)
- RWTThreadLocal<Type> Class Template Documentation (http://www.roguewave.com/support/docs/hppdocs/thrref/rwtthreadlocal.html#sec4)
- Article " Use Thread Local Storage to Pass Thread Specific Data (http://www.c-sharpcorner.com/UploadFile/ddoedens/UseThreadLocals11212005053901AM/UseThreadLocals.aspx)" by Doug Doedens
- " Thread-Local Storage (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1966.html)" by Lawrence Crowl
- " Developer's Reference (http://www.asyncop.net/Link.aspx?TLS)"
- Article " It's Not Always Nice To Share (http://www.ddj.com/cpp/217600495)" by Walter Bright
- Practical ThreadLocal usage in Java: http://blogs.captechconsulting.com/blog/balaji-muthuvarathan/persistence-pattern-using-threadlocal-and-ejb-interceptors
- GCC " (http://gcc.gnu.org/onlinedocs/gcc-3.3.1/gcc/Thread-Local.html)"

Article Sources and Contributors

Design pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=509776193 *Contributors*: 2mcm, 4th-otaku, A. B., A3 nm, A876, ACupOfCoffee, Abelson, AdamBradley, Adaniels.nl, Adjusting, Ahoerstemeier, Ahy1, Airplaneman, Albanaco, AlecMcEachran, Alex.ryazantsev, Alexandernolasco, Alexmadon, Allan McInnes, Amazedsaint, Amitch, Amitdipsite, Anaraug, Andreas Kaufmann, Andy Dingley, Antonielly, Arch4ngel, Architectureblog, Asphatasawhale, AtheWeatherman, Azbarcea, BBL5660, BD2412, Beland, BenFrantzDale, Benandorsqueaks, BenjaminGittins, Bevo, Bkkbrad, BluePlateSpecial, Bogdanmata, Bogdanmionescu, Boothy443, Borgx, Bovlb, BozzieBear, Bravegag, Breandandalton, Burschik, Bwoolf, CaptainPinko, Cconnett, Cesar Moura, Charliearcuri, Chrislk02, Clubmarx, Coolestdude1, Courcelles, Craig Stuntz, Crdoconnor, Cronian, Cybercobra, Cyc, CzarB, DJ Clayworth, Danger, Davhdavh, Davhorn, Demonkoryu, Dethomas, Devintseo, Dirk Riehle, Dmyersturnbull, Docsman, Eaefremov, Ed Brey, Edenphd, Edward, Edward Z. Yang, Egg, 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, Gioto, Gnewf, Graue, GregorB, Grubber, Guehene, Gwyddgwyrdd, Hairy Dude, Halfdan, Hans Adler, Harborsparrow, Haskellguy, Hede2000, HenkvD, Hirzel, Hu12, Ibbbi, J.delanoy, JLaTondre, JMSwtlk, JWB, Jackhwl, Jafeluv, Jamilsh, Janesconference, Jeff3000, Jerryobject, Jesperborgstrup, Jim1138, Jizzbug, Jleedev, JohnCD, 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, Lost on belmont, Ltruett, Luna Santin, M4gnum0n, Mahanga, Mamouri, Marcoscata, Marek69, Maria C Mosak, Mark Renier, MarkSutton, MartinSpamer, Marudubshinki, Matthewdjb, Maximaximax, McSly, Mcavigelli, Mdd, Mdd4696, Melah Hashamaim, Meredyth, Michael Hardy, Michal Jurosz, Mikaey, Mistercupcake, Mohansn, Mormat, Morten Johnsen, Mountain, Msjegan, Msm, Natkeeran, Nbarth, Necklace, Neelix, Neilc, NickShaforostoff, Nixdorf, Nklatt, Oicumayberight, Olff, Opticyclic, Orborde, Parsecboy, Patrick McDonough, Pcap, Pcb21, Pcwitchdr, Perey, Peter S., Pieleric, PierreCaboche, Plasticup, Pnm, Polonium, Prashanthns, PrologFan, PsyberS, Ptrb, Quagmire, QuiteUnusual, Quuxplusone, RHaworth, RabbiGregory, Rcsprinter123, Realbox, RedWolf, Redeyed Treefrog, Renesis, RickBeton, Rodasmith, Ronz, RossPatterson, Rudyment, Ruudjah, Rwmcfa1, S.K., SAE1962, Sae1962, Saiful vonair, Salix alba, Sandman25 again, Seanhan, Sfermigier, Shabbirbhimani, Skorpion87, Skraedingie, SlavMFM, Snarpel, Sonicsuns, Spdegabrielle, SpeedyGonsales, Stachelfisch, Stephanakib, SteveLoughran, Stokito, Stryn, Stuartroebuck, SudhakarNimmala, Sudhanshuss, Sxilderik, TakuyaMurata, Tassedethe, Tesakoff, Tgmattso, TheJames, Theuser, Thistle172, Thornbirda, Tiggerjay, Tiktuk, TingChong Ma, Tobias Bergemann, Tonyfaull, Tonymarston, Torrg, Treekids, Tristan ph, TuukkaH, Umbr, Umlcat, Uncle G, Unixer, Vald, VidGa, Vinsci, Vmlaker, WOT, Wapcaplet, Wavelength, Welsh, Wernher, WikiLeon, Wikidrone, Wo.luren, Xtian, Xypron, Zeno Gantner, Zondor, 665 anonymous edits

Creational pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=509503577 *Contributors*: Acflorea, Cdiggins, Fredrik, Jacob Robertson, Kellen`, Khalid, Khalid hassani, Kusunose, Lancevortex, Marraco, Mrwojo, Ninad Nagwekar, RedWolf, Sae1962, Saigyo, Steinsky, TakuyaMurata, Yungtrang, 40 anonymous edits

Abstract factory pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=509445544 *Contributors*: A5b, AJCham, Abednigo, Alx359, Anthony Appleyard, Balkrishnatalele, Bdean42, BenFrantzDale, Benno, Bigscoop, Bluemoose, BluesLf, Boleslav Bobcik, CanisRufus, Cybercobra, Dan Gluck, Daniel.Cardenas, David.nussio, Dcoetzee, Demonkoryu, Dgquintas, Drable, Druckles, Eaefremov, Eastlaw, Edbecnel, Edenphd, Ekamaloff, Enric Naval, Epbr123, Evil saltine, Firetinder, Flammifer, Fluffernutter, Fred Bradstadt, Frór, Furrykef, Fylbecatulous, Gilliam, GregorB, Gurjinder.rathore, HalfShadow, Hao2lian, Hephaestos, Hetar, Hu12, Imdonatello, Int19h, Jay, Jay.233, Jonathan Hall, Joriki, JoshHolloway, Jsgoupil, Kallikanzarid, Krzyp, Linuraj, Lukasz.gosik, MER-C, Magister Mathematicae, Magnub, Mahanga, Manop, Matt.forestpath, Mecanismo, Merutak, Michael Slone, Minimac, Mormat, Msjegan, Msthil, Mtasic, Mumeriqbal786, Naasking, Niceguyedc, Nitinpai2000, Nkocharh, Oddity-, Pcb21, Pelister, Peterl, Phresnel, Piano non troppo, Pikiwyn, Pilode, Pointillist, QuackGuru, Quamaretto, R'n'B, RainbowOfLight, RedWolf, Reyjose, Rnapier, Ronabop, Rory O'Kane, Rtushar, Rupendradhillon, Sae1962, Scray, Sdillman, SergejDergatsjev, Shadowjams, Shakethehill, Sietse Snel, Sir Nicholas de Mimsy-Porpington, Skittleys, Smithrondive, Solra Bizna, Stephenb, Sun Creator, Superborsuk, Tabletop, TakuyaMurata, The Anome, TheMandarin, Thomas Linder Puls, Tobias Bergemann, Tobiasly, Tomwalden, Topdeck, Treutwein, Tronster, Twsx, Venturor, Vghuston, Vikasgoyal77, Vinhphunguyen, Vinod.pahuja, Vvarkey, WikiJonathanpeter, Wikidudeman, Wikieditor06, Wilfried Loche, Woohookitty, Www.koeberl.priv.at, Yonghuster, YurfDog, Zixyer, 351 anonymous edits

Builder pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=508489852 *Contributors*: Abba-best, Abednigo, Aesopos, Allan McInnes, AlphaFoobar, Amniarix, Aneeshpu, Anthony Appleyard, Arichnad, Beland, BenFrantzDale, BennyD, Bensaccount, Betacommand, Blankfrack, Brooklyn1, Capt. James T. Kirk, Cbiffle, Cmcginnis, Coldboyqn, Craig Pemberton, Cybercobra, Cyrius, Daniel.Cardenas, Demonkoryu, Dreadstar, Drmies, Dyker, Ecabiac, Edward, Emcmanus, Ezraj, Finlay McWalter, Fred Bradstadt, Ftiercel, Gallas.aladdin, Geh, Ghuckabone, Glylvyn, Hairy Dude, Hao2lian, Hephaestos, Hu12, Inmyhand77, JForget, JMatthews, Janpolowinski, Jarsyl, JavierMC, Jay, Jesus Escaped, Ketorin, Khalid hassani, Khmerang, Lathspell, Lramos, Lukasz.gosik, MER-C, Madoka, Marcusmax, Mikethegreen, Mital d vora, Mormat, MrOllie, Msjegan, Mtasic, Mzedeler, Naaram, Nafeezabrar, Neoguru12, Nijelprim, Nitinpai2000, Oddbodz, Odie5533, Oxymoron83, PeaceNT, Pelister, Per Abrahamsen, Pmussler, Pointillist, RedWolf, Rh.vieira, Rimian, Rossami, Salazard1, Satyr9, Sedziwoj, Shakethehill, Shell Kinney, Shyal-b, Sligocki, SonOfNothing, Ssimsekler, Strike Eagle, Sun Creator, Supersid01, TakuyaMurata, The Thing That Should Not Be, Thomas Linder Puls, Toteb5, Trashtoy, Umreemala, Uncle G, Vdevigere, Vghuston, Vikasgoyal77, Vinod.pahuja, West Brom 4ever, Wik, WinerFresh, Yintan, 210 anonymous edits

Factory method pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=506582969 *Contributors*: 1typesetter, 9thbit, Abednigo, Acm, Ain32, Aisaac, Alainr345, AlanUS, Allens, Alquantor, Alvin-cs, AmunRa84, Andreas Kaufmann, Andreas Toth, Anthony Appleyard, Apanait, Aprock, Architectureblog, ArthurDenture, Arturs Licis, Avono, Bearcat, Beland, BenFrantzDale, Bigboojum, Binksternet, Blomskoe, BlueTaRaKa, Bouchecl, Bpssoft, BrokenSegue, CWenger, Cartoro, Charles Moss, Chaser, Courcelles, Decltype, Demonkoryu, Dmprantz, Dreftymac, Duplicity, Ebraminio, Edenphd, Edward.in.Edmonton, Enselic, Esben, Faradayplank, FellGleaming, Fraggle81, Fredrik, Galestar, Gary King, Gman124, Gnyus, Guentherwagner, Gwern, Haakon, Hadal, Hairy Dude, Hao2lian, Hemanshu, Hu12, JIP, JMatthews, Jakew, Jamitzky, Jay, Jeffrey Smith, Jkl, Jobriath, Jogla, Joswig, Jowa fan, Juansempere, Kefka, KellyCoinGuy, Khalid hassani, Khmer42, Khoango, Khrom, Kjkolb, Kshivakumar, Leo Cavalcante, Lexxsoft, Liao, Little3lue, LtPowers, Lucascullen, MER-C, Mahanga, Marco.panichi, MartinRinehart, Mattbierner, Matthew0028, MauiBoyInMaltby, Mbunyard, Mephistophelian, Mhtzdt, Michael Slone, MichielDMN, MikeEagling, Mjworthey, Mormat, Msjegan, Mtasic, NTK, Ncmvocalist, Neshatian, Nforbes, Nielses, Nomad311, Nowforever, OnixWP, OrlinKolev, Paradrop, Pelister, Petur Runolfsson, Philip Trueman, Pointillist, Qwertyus, R'n'B, RA0808, Rajah9, RedWolf, RedWordSmith, Redroof, Reyjose, Rjmooney, Rjwilmsi, Sae1962, SergejDergatsjev, Sesse, Slaunger, Soufiane2011, Ssimsekler, Stannered, Starsareblueandfaraway, Sumone10154, Svick, TakuyaMurata, Thadguidry, That Guy, From That Show!, Toddst1, Tommy2010, Tormit, Umreemala, Vald, Vghuston, Vianello, Vikasgoyal77, Vipinhari, Westcoastr13, Will2k, Wlievens, Yamamoto Ichiro, Zch051383471952, Zero4573, 376 anonymous edits

Lazy initialization *Source*: http://en.wikipedia.org/w/index.php?oldid=506450348 *Contributors*: Abednigo, Alexey Luchkovsky, Angilly, Anthony Appleyard, Bilbo1507, Blankfrack, BluePlateSpecial, Britannica, Demonkoryu, Derek Ross, Dexp, Doug Bell, Ed Poor, Enochlau, Gustavo.mori, JLaTondre, Jerryobject, JonHarder, Lumingz, MASQUERAID, Mark.knol, Maximaximax, Michael Hardy, Msjegan, Mtasic, Phresnel, Psychogears, Quagmire, RedWolf, RepeatUntil, Rheun, Rhomboid, Sae1962, Spock lone wolf, Sun Creator, Sundar2000, Tackline, TheIncredibleEdibleOompaLoompa, Tormit, VKokielov, YordanGeorgiev, Ztyx, 65 anonymous edits

Multiton pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=505376650 *Contributors*: .digamma, A876, Abednigo, Beland, Bmord, CliffHall, DanielcWiki, Demonkoryu, Destynova, Drealecs, Dubwai, Ettrig, Galestar, Gang65, Heavenlyhash, Inumedia, Khalid hassani, Mtasic, Pavel Vozenilek, Ranjit.sail, Rentzepopoulos, Ros Original, Sae1962, Spoon!, Steve R Barnes, Viy75, Wavelength, Wookietreiber, 38 anonymous edits

Object pool pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=509605850 *Contributors*: Aaron Schulz, Andreas Kaufmann, Anthony Appleyard, Arch4ngel, Arunkd13, Audriusa, CesarB, Christoofar, Hairy Dude, Hectorpal, Jkl, Juansempere, M4gnum0n, Mahanga, Marudubshinki, Moala, Msjegan, Munificent, Noodlez84, PJTraill, Pearle, Prari, Purplefeltangel, Qxz, Ruijoel, Sae1962, Saigyo, Tenbaset, Vinod.pahuja, Vorburger, WikHead, 42 anonymous edits

Prototype pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=502322478 *Contributors*: Abednigo, Anthony Appleyard, Arglbr, Arturs Licis, Ary29, BenFrantzDale, BluePlateSpecial, Brighterorange, Btx40, Can't sleep, clown will eat me, Carlif, Catch ru, Ceciop, Cheungpat, Cmdrjameson, Colonies Chris, Cygon, Der Hakawati, Doug Bell, Eaefremov, Enric Naval, Errandir, Frecklefoot, Ghettoblaster, Gotovikas, Gsp, Hao2lian, Hofoen, Hu12, Igitur, Io41, J.delanoy, Jarble, Jcollo, Jengelh, Jianghan, Lathspell, Leirith, Leszek Jańczuk, Liao, Lookforajit, Loren.wilton, Lukasz.gosik, Luís Felipe Braga, M2Ys4U, MER-C, Materialscientist, Megazoid9000, Michael Devore, Mital d vora, Modster, Mokru, Mormat, Msjegan, Mtasic, PJonDevelopment, Patrickdepinguin, Pmussler, Pointillist, RedWolf, Salazard1, ScottyBerg, SergejDergatsjev, Stalker314314, TakuyaMurata, TheMandarin, TheParanoidOne, Vghuston, Victorwss, Vikasgoyal77, VladM, XcepticZP, Zarkonnen, Zerokitsune, 158 anonymous edits

Resource Acquisition Is Initialization *Source*: http://en.wikipedia.org/w/index.php?oldid=501005290 *Contributors*: Abednigo, Alca Isilon, Alexvaughan, Anphanax, Antonielly, Arto B, BenFrantzDale, Bporopat, CRGreathouse, CanisRufus, CesarB, Clements, Clitte bbt, Cybercobra, Damian Yerrick, DanielKO, DavePeixotto, DavidDecotigny, Decltype, Demonkoryu, Dougher, Download, E=MC^2, Echuck215, Eelis, Enerjazzer, Eriol Ancalagon, EvanED, Ex0du5 5utu7e, F0rbidik, Falcon8765, FatalError, Fplay, Fresheneesz, Frosty976, Furrykef, Gadfium, Giles Bathgate, Guenther Brunthaler, Honeyman, Intgr, Jgrahn, Jjcolotti, Jonathan.Dodds, Karmastan, Kinu, Kjwhitefoot, Kku, Lectonar, Ligulem, Lkarayan, Maghnus, Mcanismo, Medinoc, Mirokado, Mortense, Neilc, Nutate, Orderud, Oysta, Phonetagger, Pibara, Poldi, Ptrb, Quietbritishjim, RJFJR, Rbonvall, Regenspaziergang, RoySmith, Runtime, ShaneKing, Siodhe, Smurfix, Spoon!, Tagishsimon, Talliesin, That Guy, From That Show!, The Nameless, The deprecator, Tidigare upphovsman, TimBentley, Torc2, Trevor MacInnis, Tronic2, Tulcod, Wikante, Wikidrone, Wlievens, Xerxesnine, 160 anonymous edits

Singleton pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=509327484 *Contributors*: 1qaz-pl, 5 albert square, A Quest For Knowledge, APerson241, Abednigo, Abhi beckert, Aitias, Aka042, Aks.abhishek, Aleenf1, AndrewVeresov, Andy Dingley, Anthony Appleyard, Appraiser, Ascánder, Asksol, Athlan, Atreys, Autarch, B3t, Beachy, Beland, BenFrantzDale, Bestsss, Bibhatripathi, Bigbadman, Blankfrack, Bnedelko, Bogdanmionescu, BrokenSegue, Bruno.schmidt, Bytebear, C. A. Russell, Caefer, Captchad, Carlopires, Chowbok, Chris the speller, Christian75, Clarkcox3, Craigfis, Crasshopper, Cryptophile, Cusp98007, Cyril.wack, DARTH SIDIOUS 2, DaSjieb, DabMachine, Dagostinelli, Debashish, Demonkoryu, Derek Ross, DocSigma,

Domeika, Doubleplusjeff, Doug Bell, Drrngrvy, Dubwai, Eduardo.Farias, Educres, Ekamara, Enchanter, Ethaniel, Ethridgela, Ficklephil, Freddy1990, Func, FuzzyBunny, Gbouzon, Ghettoblaster, GoCooL, Goofyheadedpunk, Gorpik, Greg searle, Hao2lian, Heidirose12345, Hoggwild5, Hpesoj00, Hu12, IanLewis, Innocenceisdeath, Int19h, Inumedia, Isfppoet, Ixfd64, Jago84, Jahudgins, Jakesee, Jamie, Jan Nielsen, Javito554, Jbcc, Jdcook72, Jeugeorge, Jezmck, Jfromcanada, Jleedev, Jobin Bennett, Joswig, Jsgoupil, Jtitley, Juanmaiz, Juansempere, Jutiphan, Kate, KeithPinson, Khalid hassani, Khatru2, Kwiki, Letssurf, LordK, Lordmetroid, Lukasz.gosik, Lukebayes, Lulu of the Lotus-Eaters, Lunboks, Luís Felipe Braga, MBisanz, MER-C, MFNickster, Mancini, MarSch, Marty.neal, Marudubshinki, McOsten, Mcsee, Mentifisto, Michael Hardy, Michael miceli, Miknight, MisterHand, Mjharrison, Mormat, Mountain, Mpradeep, MrWiseGuy, Msjegan, Mtasic, Mudx77, Mydaz, NathanBeach, NeoPhi, Neoguru12, Nitram cero, Norm mit, NormBo, Omicronpersei8, Omnicog, OmveerChaudhary, PI314r, Paralyz3, PatheticCopyEditor, Paulgear, Pavel Vozenilek, Pcap, Pcb21, Phil Boswell, Piano non troppo, Pluma, Pmussler, Pnm, Pointillist, Polymerbringer, Pomakis, Ptero, Python eggs, Qc, Quuxplusone, RFST, Rakeshnayak, Ravish9507, Ray Van De Walker, RedWolf, Richard Fusion, Rider kabuto, Robert Illes, Robo Cop, Rogério Brito, RossPatterson, Ruakh, Ryan1101, Serkan Kenar, Serketan, Shadowjams, Shi Hou, Silly rabbit, Simeon H, Simpatico qa, Slinkp, Sneftel, SouthLake, Spikex, Spoon!, Srittau, Stalker314314, Stwalkerster, Sun Creator, Swiftcoder, TakuyaMurata, Tartley, Tarun.Varshney, Taw, Tecle.samson, Tfischer, TheMandarin, TheRealFennShysa, ThingXYZ, Thinking Stone, Thomas Kist, Thong10, Timp21337, Timwi, Tobias Bergemann, Tomerfiliba, Tormit, Trashtoy, Travisowens@hotmail.com, Trimmch, Ultra two, Vazde, Vernicht, Vghuston, Vikasgoyal77, Vina, VladM, Voyaging, Vrenator, Vssun, Wbstory, Weirddemon, Who, WikiLaurent, Wikidrone, Wingbullet, Winterheat, Xiluwu, Xuranium, Yjxiao, Yohanliyanage, Yukoba, Yurik, Zhen Lin, Zuiopo, Zwetan, Ігор Дзюба, 663 anonymous edits

Structural pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=503273242 *Contributors*: Beland, Greedyhalibut, Iridescent, Jeff3000, Jijinjohn, Khalid, LazyEditor, Mahanga, Moe Epsilon, Mormat, MrGrigg, Phoenix720, Piet Delport, RedWolf, Rsathish, Ruud Koot, S, Sae1962, Siggimund, StephenWeber, TheParanoidOne, 36 anonymous edits

Adapter pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=505101215 *Contributors*: Abdull, Abednigo, Alansohn, Alexius08, Andre Engels, Anonymous2420, Anthony Appleyard, Architectureblog, BenFrantzDale, Blytkerchan, Bravegag, BryanG, CannonBallGuy, CatherineMunro, Conditioner1, Conditioner123, Cyril.wack, Demonkoryu, Doug Bell, Edenphd, Eudoxie, Fred Bradstadt, Giantbicycle, GilHamilton, Hao2lian, Happyser, Hardywang, Hu12, IanLewis, Intgr, JMatthews, JamesBWatson, Jk2q3jrklse, Joerg Reiher, JonathanLivingston, Jonon, Jordanarseno, KevinTeague, Luís Felipe Braga, M4gnum0n, MER-C, Materialscientist, MichaelRWolf, Monkeyget, Mshravan wiki, Msjegan, Mtasic, Mxn, OrangeDog, Pcb21, Psg5p, RA0808, RedWolf, Rimshot, Roger.wernersson, Ronslow, Ronz, Sae1962, SergejDergatsjev, Sietse Snel, Sir Nicholas de Mimsy-Porpington, Spoon!, Stalker314314, SteinbDJ, Strib, Synchronism, TakuyaMurata, Tarquin, TheParanoidOne, Tobias Bergemann, Trashtoy, Tuntable, Vanmaple, Vghuston, Vikalpsingh, Wapcaplet, Wikipelli, Wookietreiber, 126 anonymous edits

Bridge pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=496090135 *Contributors*: Abednigo, Alainr345, Albert cheng, Andriiko, Anthony Appleyard, Areicher, Argel1200, Beland, BenFrantzDale, Billyblind, Burns700, Bytebear, Cdunn2001, Chimply, Chochopk, Cipherynx, Cmdrjameson, Cra88y, DCEdwards1966, Daydreamer302000, Ddenton, Deathy, Djmckee1, Drrngrvy, Długosz, 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, Luís Felipe Braga, M4gnum0n, MER-C, Malaki12003, MattGiuca, Mcavigelli, Michael miceli, MichalKwiatkowski, Mikemill, Mintguy, Mjworthey, Mkell85, Mormat, Msjegan, Mtasic, Mxn, Narcelio, NickelShoe, Odie5533, Omicronpersei8, Patrick Schneider, Pcb21, Phil Boswell, Pmussler, Profaisal, Quuxplusone, RedWolf, Rouven Thimm, Rror, Ryanvolpe, S.K., Seethefellow, 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, Ykhwong, Zundark, 196 anonymous edits

Composite pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=509795043 *Contributors*: A876, Aaron Rotenberg, Abednigo, AdrianHesketh, Alainr345, Albert cheng, Andrewtkchan, Anonymous2420, Anthony Appleyard, BenFrantzDale, BradBeattie, Braiam, Bravegag, CMosher01, CatharticMoment, Charles Matthews, Chimply, Coffeeflower, DCrazy, Damien Cassou, Darkspots, 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, Luís Felipe Braga, M4gnum0n, MER-C, Matekm, Matěj Grabovský, Mefyl, Michael miceli, Mik01aj, MikalZiane, Mike926908240680, Mrwojo, Msjegan, Mtasic, Nagnatron, Narcisse, Nick1915, Nikai, Pearle, Pinguin.tk, Pmussler, RedWolf, Rishichauhan, Sebras, Shabbirbhimani, Shivendradayal, Skapur, Skarebo, Sneftel, Stwalkerster, TakuyaMurata, Thamaraiselvan, The Thing That Should Not Be, Thomas Linder Puls, Trashtoy, Tubalubalu, Umreemala, Vinhphunguyen, Vishalmamania, Vonkje, Waggers, Wainstead, Whirlium, Will2k, Winnwang, Wkjyh, Zarcadia, Zoz, 173 anonymous edits

Decorator pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=509728154 *Contributors*: ACiD2, Abednigo, Adam Zivner, Ahoerstemeier, Aihtdikh, AmunRa84, Anakin101, Anthony Appleyard, Arny, Ascánder, Awistaiw, BenFrantzDale, Booler80, Bravegag, BryceHarrington, Codysnider, Conditioner1, Connelly, Cruccone, Cybercobra, Doradus, Dori, Doug Bell, Drae, EdC, Edenphd, Eric-Wester, ErikStewart, FatalError, Francesco Terenzani, Gasparovicm, Gerel, Ggenellina, Grokus, Gslockwood, Hao2lian, Harborsparrow, Hu12, Hunting dog, Itai, J.lonnee, J1o1h1n, JRM, Jeeves, Jim1138, Joriki, Khalid hassani, Khmer42, MER-C, Matt Crypto, Minghong, Mjb, Mjresin, Mostapha, Motine, Msjegan, Mtasic, Murtasa, Nafeezabrar, Nheirbaut, Odie5533, P0per, Pharaoh of the Wizards, Phresnel, Pmussler, RUL3R, RedWolf, Romanc19s, Ronz, Rrburke, Servel333, Snooper77, TakuyaMurata, Tgwizard, Tobias Bergemann, Tomerfiliba, Trashtoy, Triskaideka, Urhixidur, Vghuston, Wackimonki, Wik, Wikidrone, Will2k, XcepticZP, Yuqingalex, 187 anonymous edits

Facade pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=508482033 *Contributors*: A5b, Abdull, Abednigo, Aff123a, Andyluciano, Anthony Appleyard, Arch4ngel, Arvindkumar.avinash, Ashlux, B3t, Beland, BenFrantzDale, Chestertan, Cybercobra, Darkmane, Deflective, Demonkoryu, Dkeithmorris, Dlohcierekim, Dreftymac, Dysprosia, Forseti, Frederikton, Hairy Dude, Hangy, Hao2lian, Hofoen, Hu12, IanLewis, Immure, Jfalvarez, Jogloran, Jorend, Jussist, Kaikko, Khishig.s, Kwamikagami, Lathspell, Leonard G., Lpenz, Lukasz.gosik, Luís Felipe Braga, M4c0, MER-C, Mahanga, Materialscientist, MrOllie, Msjegan, Mtasic, Mxn, Narcissus, Nate Silva, Nibblus, Otterfan, Owen214, PBS, Puchiko, RedCrystal, RedWolf, Sae1962, Scullder, Svick, Synook, TakuyaMurata, Thadius856, Tony Sidaway, Tormit, Trashtoy, Vald, Wookie2u, Wwjdcsk, Xiaoddkkwe, Xrm0, 에멜무지로, 130 anonymous edits

Front Controller pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=504136712 *Contributors*: Andreas Kaufmann, Angrytoast, BBL5660, Bcosca, Chadmyers, Dmforcier, Ekerazha, Jase21, Jpp, Kentoshi, Ludde23, Matekm, MrOllie, Nperriault, Userform, Wavelength, 43 anonymous edits

Flyweight pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=506450320 *Contributors*: Abednigo, Accelerometer, Albert cheng, AlphaWiki, Anirudhvyas010, Anticipation of a New Lover's Arrival, The, Beland, BenFrantzDale, Beryllium, BigDunc, Blastthisinferno, CMosher01, Cgodley, DanielBrauer, Debajit, Demonkoryu, Diego Moya, Ebswift, Emj, FroZman, Gnp, Hao2lian, Hu12, Jk2q3jrklse, Justo, Kurt Jansson, Lathspell, Lazeroptyx, LyricCoder, MER-C, Maxim, Mdjpurdon, Micah hainline, Msjegan, Mtasic, Mxn, Naasking, Northerly Star, Odie5533, Op12, Pcb21, PierreAbbat, Ray Van De Walker, RedWolf, Ryanl, Senu, Sproul, Sun Creator, TakuyaMurata, Teglsbo, The Wild Falcon, Tshak, Twyford, Vipinhari, Warren, WestwoodMatt, Wizgob, X00022027, ZacBowling, Zozakral, 103 anonymous edits

Proxy pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=509436112 *Contributors*: 0612, Abdull, Abednigo, Abelson, Ad1976, Anonymous2420, Anthony Appleyard, Audriusa, Avocado, Azanin, Babomb, Beland, BenFrantzDale, Bravegag, Brownb2, Cholmes75, D3xter, Decltype, Demonkoryu, Ebswift, Edenphd, Edivorce, Elector101, Espresso1000, Fred Bradstadt, Fritz freiheit, Ftiercel, Hao2lian, Hu12, Jamelan, Jay, Jd gordon, JeR, Jthiesen, Lathspell, Lukasz.gosik, Lumingz, Lusum, MER-C, Marianobianchi, Matekm, Msaada, Msjegan, NathanBeach, Nonomy, Nurg, Nzroller, Odie5533, P0per, Pateeludg, Philip Trueman, Prickus, Quuxplusone, RedWolf, Rich Farmbrough, Rich257, Royalguard11, Sae1962, SanthoshRadhakrishnan, Snaxe920, TakuyaMurata, TheParanoidOne, TravisHein, Valodzka, VladM, Will2k, Wlievens, Zolookas, Zozakral, 109 anonymous edits

Behavioral pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=503275299 *Contributors*: Beland, Fredrik, Khalid, Mahanga, Michael Slone, Mormat, Petrb, RedWolf, Sae1962, Stalker314314, Starfoxy, TheParanoidOne, Wavell2003, 13 anonymous edits

Chain-of-responsibility pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=506450250 *Contributors*: Abednigo, AboutMM, AlanUS, Arch4ngel, Architectureblog, Arjunkul, B3t, BenFrantzDale, Bin Sowparnika, Bravegag, Brenont, Brian428, Celephicus, Cmdrjameson, Courcelles, Danny, Demonkoryu, Eknv, Ermunishsodhi, Estevoaei, Fmstephe, FoCuSandLeArN, Gotovikas, Hao2lian, Hu12, Ilion2, Itschris, Jhatax, Jldupont, 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, Seaphoto, Shades97, Soubok, Struggling wang, Sun Creator, Tabletop, TakuyaMurata, TheParanoidOne, Thomas Linder Puls, Vghuston, Vikasgoyal77, Vyuschuk, Wbstory, Wikidrone, Zaletskyy, Zyx, 79 anonymous edits

Command pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=496770527 *Contributors*: Abdull, Abednigo, Acmeacme, Ascánder, B3t, Beland, BenFrantzDale, Bennor, Bipedalpedia, Bluemoose, Bravegag, Chester Markel, Crhopkins, Darklilac, DavidBourguignon, Daydreamer302000, Dazhiruoyu, Demonkoryu, Dharesign, DotnetCarpenter, Doug Bell, Druckles, Dtbullock, Ebainomugisha, Edward.in.Edmonton, Ehn, Exe, Eyewax, Fred Bradstadt, Gene Thomas, Gmoore19, GregorB, Hao2lian, Hooperbloob, Hu12, IanVaughan, JHunterJ, Jensck93, Jevansen, JoaoTrindade, John Lindquist, Jorend, Karuppiah, Kate, King Lopez, Kku, Lightdarkness, LilHelpa, MER-C, Mahanga, Mernen, Michael miceli, Mjworthey, Mloskot, Modify, Nakon, Nathan Ra, Omegatron, Orangeroof, Oster54, Oubless, Owentaylor25, Pcb21, Pharaoh of the Wizards, Piet Delport, Rahul verma india, RedWolf, RexxS, Rijkbenik, SJP, Scottywong, Seaphoto, Svick, TakuyaMurata, Terheyden, Terry.kfwong, TheParanoidOne, Trevor Johns, Vald, Vghuston, Vikasgoyal77, Will2k, Zixyer, Zozakral, 137 anonymous edits

Interpreter pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=489508686 *Contributors*: Abednigo, Allan McInnes, Andypayne, Asciilifeform, BenFrantzDale, Borderj, Cmccormick8, Cybercobra, Davewho2, Fredrik, Furrykef, Gwalla, Hao2lian, Hu12, Komap, L30nc1t0, Lathspell, Lonelybug, Lukasz.gosik, Mahanga, Matekm, Mecanismo, Merutak, MithrandirMage, Mrwojo, Odie5533, Pcb21, RJFJR, RedWolf, Rice Yao, Rogerwhitney, Rosiestep, SamePaul, Seaphoto, TakuyaMurata, TheParanoidOne, Vghuston, Yuqingalex, Zozakral, 53 anonymous edits

Iterator pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=506982968 *Contributors*: Abednigo, Albert cheng, Angilly, Ayvengo21, Beland, BenFrantzDale, Bnedelko, Btx40, Bytebear, Cybercobra, D chivaluri, Dhulme, Eaolson, Eknv, Ghthor, Hao2lian, Hu12, Iridescence, J04n, Jerryobject, Jogloran, KAMEDA, Akihiro, Lababidi, Leibniz, Marj Tiefert, Matekm, Mormat, Odie5533, Ogmios, Radagast83, Ravindra Josyula, RedWolf, Sae1962, Seffer, TakuyaMurata, TechTony, Tomtheeditor, Trashtoy, Vghuston, Winxa, Yuqingalex, Zerokitsune, Zozakral, 83 anonymous edits

Mediator pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=497692688 *Contributors*: 16@r, Abednigo, Anonymous2420, Balu213, Bayesianese, BenFrantzDale, C3foncea, Colonies Chris, Cybercobra, Dawynn, Dcflyer, Exe, Flammifer, Furrykef, Grancalavera, Grizzly33, Hao2lian, Hu12, If R-rated movies are porn, it was porn, Jebdm, Jrockley, Kku, Lathspell, Leafyplant, Mariolopes, Mormat, Pietrodn, RedWolf, Rjnienaber, RobDe68, RoyBoy, Royalguard11, TakuyaMurata, TheParanoidOne, Twyford, Vghuston, Vijayekm, VladM, Whitepaw, Yuqingalex, Zozakral, 100 anonymous edits

Memento pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=508136065 *Contributors*: Aaron Rotenberg, Abdull, Abednigo, Alainr345, Alvin-cs, BenFrantzDale, CountRazumovsky, Galyathee, Hao2lian, Heron, Hu12, JLaTondre, Kingturtle, Lathspell, MaheshZ, 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, Zigmar, Zozakral, 48 anonymous edits

Null Object pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=509244012 *Contributors*: 16@r, Alexander Gorbylev, Annrules, Autarch, Bunnyhop11, C xong, Cybercobra, Demonkoryu, Dirk Riehle, Ferdinand Pienaar, Hairy Dude, Ice Keese, Johndkane, Jpp, Kwikkode, Mahanga, Michael Anon, Neelix, Paulecoyote, Rjwilmsi, Ruijoel, Snaxe920, Spiritia, Svick, Teles, Teukros, Timwi, Tom Morris, Vaclav.Makes, Yossi Gil, Zozakral, 62 anonymous edits

Observer pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=509392845 *Contributors*: Aaron Rotenberg, Abednigo, Albert cheng, Alejandro.isaza, Alexvaughan, Alvin-cs, Amazedsaint, Anna Lincoln, Anonymous Dissident, Antonschastny, Ariadie, Arnon Mathias, Arturus, Autarch, AxelBoldt, BBilge, Bdanee, Beland, BenFrantzDale, Bkil, Bmusician, Booyabazooka, Bovlb, Bravegag, Briefstyle, C. lorenz, C3foncea, CBMalloch, Calvin 1998, Can't sleep, clown will eat me, Capricorn42, Ceyockey, Cosmicnet, Cosmin varlan, Countchoc, Darth Panda, David Levy, DdEe4Aai, Decatur-en, Deineka, Devshop.me, Diderot, Dionyziz, Ebraminio, Ed Poor, Edward, Eknv, Emergeo, Emurphy42, Emw, Eric Le Bigot, Excirial, Frecklefoot, Fred Bradstadt, Furrykef, Gang65, Gregsinclair, Gustavo.mori, Hao2lian, Hervegirod, Hooperbloob, Hu12, HurryPeter, Ipatrol, Jamelan, Jarble, Jarsyl, Jay, Jebdm, Jems420, Jphollanti, Kaquito, Karuppiah, Kate, KellyCoinGuy, Kelumnpk, Kgeis, Khalid hassani, Kku, Komap, Labviewportal, LazyEditor, Leibniz, Lnatan25, Lpsiphi, M.amjad.saeed, Mahanga, Marie Poise, Martnym, MatthieuWiki, Mattjball, Mecanismo, Merutak, Michaelschmatz, MirkMeister, Mratzloff, Mrwojo, Mzyxptlk, Niceguyedc, Nickg, Nickolas.pohilets, OMouse, Oc666, Ohnoitsjamie, OlEnglish, OmveerChaudhary, OrangeDog, Pcb21, PersonWhoLivesInHollyhurst, Philipp Weis, Pingveno, Pmercier, Pmussler, Porton, Quux, Radagast83, Razafy, Reaper555, RedWolf, Redroof, Retired username, RickBeton, Robbiemuffin, Robert Paveza, Ronz, Rotaercz, RoyBoy, RussBlau, SD6-Agent, Sam Pointon, SamuelHuckins, Sdesalas, Shdwfeather, Shirtwaist, Sjmsteffann, SkyWalker, SlubGlub, Stephan Leeds, Sunnychow, SynergyBlades, TakuyaMurata, Terry.kfwong, TheMuuj, TheParanoidOne, Tmkeesey, TobiasHerp, Topdeck, Torc2, Uzume, Vahrokh, Vanished user giiw8u4ikmfw823foinc2, Vghuston, Vipinhari, Voyvf, VvV, Wbstory, WikiSolved, Wikidrone, Willpwillp, Xynopsis, YordanGeorgiev, Yuqingalex, Zozakral, 322 anonymous edits

Publish/subscribe *Source*: http://en.wikipedia.org/w/index.php?oldid=460488206 *Contributors*: 4th-otaku, Abdull, Altenmann, Anaholic, Beland, Berndfo, Bjfletcher, Bob Wyman, Bunnyhop11, CanisRufus, Clarevoyent, Damiens.rf, Dougher, Dpm64, DuncanCragg, Edward Vielmetti, Enochlau, Fordsfords, Francesco-lelli, FuzzyBSc, Gaius Cornelius, Group29, Harburg, Hidbaty223, ImaginationDave, JHunterJ, Jacobolus, Jamelan, Jerryobject, Jolest, Josemariasaldana, Juergen.Boehm, KagamiNoMiko, Ken Birman, Kernel.package, Kgardner2007, Kmorozov, Kstevenham, Kyle Hardgrave, L200817s, Lancelotlinc, Lwihwr, Malin Tokyo, Mandarax, Marc Esnouf, MeltBanana, Mfrisch, Michael.jaeger, Mxn, Nanodance, Neustradamus, Nickg, Obcg, Peter Hitchmough, Pnm, Puffin, Ray Van De Walker, Reflex Reaction, RichardBanks, Robina Fox, Ruud Koot, Schollii, Secretkeeper81, Spike Wilbury, Stephan Leeds, Subversive.sound, Supriyadevidutta, Suruena, SymlynX, T.J.A. Uhm, Tedernst, Tervela, Themfromspace, Tzoght, Vegaswikian, Wavelength, WikHead, Willpwillp, Yworo, □, 用心阁, 104 anonymous edits

Design pattern Servant *Source*: http://en.wikipedia.org/w/index.php?oldid=466664862 *Contributors*: Aerovistae, AzaToth, Beoros, Combovercool, Dexp, Dthomsen8, Faisalbegins, Ibic, Invitrovanitas, Mathwiz777, RHaworth, Rudyment, Ryoga Godai, Stfg, VeselyBrumla, 7 anonymous edits

Specification pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=509529790 *Contributors*: B4hand, Beland, Demonkoryu, Esowteric, Ivant, Juancadavid, Maralia, RickBeton, Sotonika, Squishymaker, Svallory, Torc2, Wavell2003, Zozakral, 31 anonymous edits

State pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=493707707 *Contributors*: Abednigo, Allan McInnes, Anonymous2420, Babomb, BenFrantzDale, BennyD, Cedar101, Deltaflux, Edenphd, Eknv, Ertwroc, Estevoaei, Frigoris, FunnyMan3595, Garyczek, Hao2lian, Hariharank12, Hu12, IanVaughan, Jamesontai, Jeepday, Jlechem, JoaoTrindade, Julesd, LuckyStarr, Luigi30, M4bwav, Matěj Grabovský, MithrandirMage, Mormat, Naasking, Nitin.nizhawan, Pdhooper, Pilotguy, Pmussler, RedWolf, Ruud Koot, Suniltg, TakuyaMurata, Tassedethe, Terheyden, TheParanoidOne, ThomasO1989, Userask, Vghuston, 80 anonymous edits

Strategy pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=507445472 *Contributors*: Abdull, Abednigo, Anirudhvyas010, Anupk7, Atree, AustinBlues, BenFrantzDale, BlueTaRaKa, Bocsika, Cconnett, Cst17, Cybercobra, Daemonicky, Danny childs19, Dcadenas, Ddonnell, Decltype, Dougher, Edenphd, Equendil, Finked, Flex, Forseti, Fredrik, Gluegadget, Hao2lian, Helmerh, Hephaestos, Hosamaly, Hu12, Jems420, Jerryobject, Jianwu, Jncraton, Jogloran, Jonasschneider, Joswig, Kingfishr, Landisdesign, Liko81, Loadmaster, Lotje, Lst27, Luís Felipe Braga, M4gnum0n, MER-C, Mahanga, Marco Krohn, Md2perpe, MichalKwiatkowski, Mik01aj, Minddog, Mjchonoles, Mlibby, Omer Hassan, Ospalh, PJonDevelopment, Pcb21, Pgan002, Pinethicket, Platypus222, Pmussler, Ptoonen, Pzrq, Quuxplusone, Radon210, RedWolf, Redroof, Remuel, Richard.a.paul, Rikesh kisnah, Riki, Ritchiep, Ronz, Ruud Koot, SlubGlub, Snaxe920, Southen, Spookylukey, SpuriousQ, Sumanthtamma, Sungur, Supadawg, TakuyaMurata, The Wild Falcon, TheParanoidOne, TjerkWol, Tombomp, Torc2, Vghuston, Vladimir Bosnjak, Wikidrone, Wmwallace, Woohookitty, XcepticZP, Zhen Lin, Zozakral, 210 anonymous edits

Template method pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=500451589 *Contributors*: AHMartin, Abednigo, AleaB, Allstarecho, Andreas Kaufmann, Anjsimmo, 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, Pmussler, Quuxplusone, RedWolf, Reinis, Remuel, Richard Katz, Rjwilmsi, Roadrunner, Russell Triplehorn, Ruud Koot, Supadawg, T1mmyb, TJ09, TakuyaMurata, Teles, TheParanoidOne, Vghuston, Vladimir Bosnjak, Wavelength, Zarkonnen, Zhen Lin, Zozakral, 92 anonymous edits

Visitor pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=507678539 *Contributors*: Abednigo, Aeolian145, Afsinbey, Akanksh, AlanUS, Albert cheng, Alfatrion, Alfredo.correa, Alksentrs, Andy.m.jost, Anirudhvyas010, Anonymous2420, Autarch, B3t, Beland, BenFrantzDale, Bennor, Brandon, Bravegag, C3foncea, CardinalDan, Computerwguy, D chivaluri, Decltype, Demonkoryu, Dethomas, Devjava, Edenphd, Eob, Fbahr, Fizzbann1234, Fredrik, Friskpeppermint, Gaius Cornelius, Gamma, Gandalfgeek, Gnewf, Goyalsachin22, GregorB, Gyrobo, HGoat, Hadal, Hao2lian, Hazard-SJ, Hu12, Ibic, Iffypop, Int19h, JamesPoulson, Jcarroll, Jjdawson7, Joel falcou, Jokes Free4Me, Joswig, Jrduncans, Karada, Kate, Kraftlos, Kuru, Leibniz, Luiscolorado, Lumingz, M4gnum0n, MER-C, Marau, Martnym, Materialscientist, MattGiuca, Matthew0028, Mcsee, Mfwitten, Mitchan, Mountain, My Name is Christopher, Nanderman, Nchaimov, Ntalamai, Objarni, Oerjan, Ohnoitsjamie, Orderud, Pcb21, Pelister, Pewchies, Phil Boswell, Pkirlin, PolyGlot, RedWolf, Reyk, Riki, Salix alba, Sergei49, Skarkkai, Sleske, Svick, Szeder, TakuyaMurata, Theone256, Thinking Stone, Timenkov Fedor, Tobias Bergemann, Versus22, Vghuston, Vinz83, Wiki alf, Wlievens, Woohookitty, Zozakral, Ржавый Хомяк, 242 anonymous edits

Concurrency pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=508428985 *Contributors*: AshleySt, Atreys, Casbah, Enlilos, Fredrik, GregorB, Ipsign, Jogerh, Jopincar, Khalid, Mati22081979, RJFJR, RedWolf, Sae1962, TheParanoidOne, Torinthiel, Vinod.pahuja, 8 anonymous edits

Active object *Source*: http://en.wikipedia.org/w/index.php?oldid=506491603 *Contributors*: Beland, Gingerjoos, Hans Adler, Ipsign, JennyRad, Kku, Kobrabones, Nixeagle, RHaworth, ShakespeareFan00, Silesianus, Stimpy, Tarcieri, Tim@, Vegarwe, Wigy, Zyx, 9 anonymous edits

Balking pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=441330740 *Contributors*: Abdull, Amiceli, Beland, Crystallina, DavidBrooks, Demonkoryu, GregorB, Loopy48, RedWolf, Spliffy, TakuyaMurata, Viperidaenz, 17 anonymous edits

Messaging pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=464329115 *Contributors*: 1000Faces, 4th-otaku, Abdull, Anrie Nord, Beland, Djmackenzie, JerryLerman, Lambda Magician, Munahaf, Prickus, Rwwww, Tinucherian, 12 anonymous edits

Double-checked locking *Source*: http://en.wikipedia.org/w/index.php?oldid=499648503 *Contributors*: (, Andreas Kaufmann, Bdawson, BeeOnRope, Beland, Bultro, Cryptic, Demonkoryu, Dhvik, Digulla, Doug Bell, Dripp, Dubwai, Flyingped, Frecklefoot, Ghettoblaster, Gracefool, Haakon, Hairy Dude, Initworks, Javawizard, Jczeus, Jm307, Julesd, Jwir3, Jyke, Karl80, Marudubshinki, Michaeldsuarez, MikeZar, Msundman, Mìthrandir, Neilc, Nlu, PJTraill, Quercus basaseachicensis, RedWolf, Rich Farmbrough, RossPatterson, ShelfSkewed, SimonTrew, StarryGrandma, TakuyaMurata, Taw, ToastyKen, 89 anonymous edits

Asynchronous method invocation *Source*: http://en.wikipedia.org/w/index.php?oldid=497234472 *Contributors*: Aervanath, Andreas Kaufmann, Cander0000, Christian75, Davhdavh, Hairy Dude, Hans Adler, Ipsign, JulesH, JzG, Pnm, 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 *Source*: http://en.wikipedia.org/w/index.php?oldid=506397008 *Contributors*: (, Allan McInnes, Altenmann, Amorken, AnObfuscator, Andahazy, Beland, CanisRufus, Chozan, Clausen, Craig Stuntz, CyborgTosser, Danim, DavidCary, Demonkoryu, Dicklyon, Dori, Długosz, Edward, ErkinBatu, Fuhghettaboutit, Furrykef, Gazpacho, Gslin, Hathawayc, Hervegirod, JCLately, Jacobolus, Jay, Jerryobject, JimD, Kaycee srk, KeyStroke, Liao, Mark Renier, Maxal, Michael Slone, Michael Suess, Moonwolf14, Neilc, Omikronuk, OrangeDog, Radik.khisamov, Rjwilmsi, Roman12345, Ross bencina, Rsocol, SBunce, SJP, Sae1962, Saikrishna Chunchu, Shivamohan, StanContributor, Sun Creator, Suruena, Svick, TakuyaMurata, Taw, Togamaru, Torc2, Tothwolf, Vald, VictorianMutant, Wmbolle, Wrp103, Wykypydya, YB3, Zeroflag, Zigger, Zoicon5, Zundark, ²¹², 99 anonymous edits

Monitor *Source*: http://en.wikipedia.org/w/index.php?oldid=506753036 *Contributors*: AKEB, Abdull, Anna Lincoln, Asafshelly, Assimil8or, AxelBoldt, Charlesb, Chruck, Cleared as filed, Craig Pemberton, DARTH SIDIOUS 2, Dcoetzee, Delldot, Flex, Franl, Gazpacho, Gennaro Prota, Gonnet, Herrturtur, Ianb1469, Intgr, Ipatrol, JLaTondre, Jacosi, Jeroen74, Jerryobject, Jfmantis, John Vandenberg, Joy, Kislay kishore2003, Leszek Jańczuk, Marudubshinki, Miym, Mormegil, Moshewe, Moulaali, Musiphil, Niyue, Pacman128, Parklandspanaway, Pburka, Pdcook, Raul654, Rich Farmbrough, RickBeton, Rv74, TPReal, Tcncv, Theodore.norvell, Thiagomacieira, Thomas Linder Puls, Torc2, Voskanyan, Waqas1987, WikHead, Yukoba, 107 anonymous edits

Reactor pattern *Source*: http://en.wikipedia.org/w/index.php?oldid=506455447 *Contributors*: Alejolp, Apoc2400, Athaba, Benjamin.booth, Bigcheesegs, Chocolateboy, Danio, DavidBourguignon, Distalzou, Ehames, Hinrik, Ipsign, JCarlos, JLaTondre, Jaxelrod, Jmg.utn, Krauss, M4gnum0n, Obankston, Praxis, Tedickey, Tomerfiliba, UlrichAAB, 48 anonymous edits

Readers-writer lock *Source*: http://en.wikipedia.org/w/index.php?oldid=469789715 *Contributors*: Andreas Kaufmann, Babobibo, Ber, Darth Panda, Download, Dublet, Dvyukov, Fuzzbox, Greensburger, JC Chu, Jakarr, Jeenuv, KindDragon33, Loopy48, Msnicki, Ohconfucius, Peepeedia, Pnm, Quuxplusone, RandyFischer, Ronklein, Spoonboy42, Vald, ZeroOne, 21 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=492700058 *Contributors*: APerson241, Adrian hipgeek, Andreas Kaufmann, Arkanosis, Asimahme tx1, Bezenek, Caitlinhalstead, CanisRufus, Cburnett, Charles Matthews, Check, Cybercobra, Denny-cn, Doug Bell, Ghettoblaster, Iceman42, Jdavidw13, JonHarder, LarryJeff, Leuk he, Psm, Ptrb, Red Thrush, RedWolf, RickBeton, Soumyasch, Stokestack, Swguy3, VadimIppolitov, Ysangkok, 28 anonymous edits

Thread-local storage *Source*: http://en.wikipedia.org/w/index.php?oldid=496968263 *Contributors*: AKGhetto, Andareed, Andreas Kaufmann, AndrewHowse, Antithesisw, Asafshelly, B4hand, BevinBrett, Brianski, Christian75, CyberShadow, D.scain.farenzena, Demonkoryu, Doug Bell, ENeville, F. Saerdna, Frap, Gerbrant, Heron, Hmains, Hutteman, Jengelh, Jerryobject, Jorend, KamasamaK, KazKylheku, Lanzkron, LeoNerd, Mais oui!, Mangotang, Marudubshinki, NathanBeach, Nealcardwell, Orderud, Parklandspanaway, Pavel Vozenilek, Pjvpjv, Qutezuce, Raul654, Raysonho, Richard1962, Rvalles, SJFriedl, Scottielad, SimonTrew, Suruena, Svick, Tackline, Thomerow, TimBentley, Tobias Bergemann, Toddintr, William Avery, 78 anonymous edits

Image Sources, Licenses and Contributors

File:Creational Pattern Simple Structure.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Creational_Pattern_Simple_Structure.png *License*: Creative Commons Attribution 3.0 *Contributors*: Yungtrang

Image:Abstract Factory in LePUS3.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Abstract_Factory_in_LePUS3.png *License*: Public Domain *Contributors*: Amnon Eden (User:Edenphd)

Image:Abstract factory.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Abstract_factory.svg *License*: GNU Free Documentation License *Contributors*: DoktorMandrake

Image:Builder UML class diagram.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Builder_UML_class_diagram.svg *License*: Public Domain *Contributors*: Trashtoy

Image:FactoryMethod.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:FactoryMethod.svg *License*: Public Domain *Contributors*: Traced by 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 *License*: Public Domain *Contributors*: Amnon Eden (User:Edenphd)

File:Prototype UML.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Prototype_UML.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: User:Giacomo Ritucci

Image:Singleton UML class diagram.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Singleton_UML_class_diagram.svg *License*: Public Domain *Contributors*: Trashtoy

Image:ObjectAdapter.png *Source*: http://en.wikipedia.org/w/index.php?title=File:ObjectAdapter.png *License*: unknown *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 *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 *License*: GNU Free Documentation License *Contributors*: Edenphd

Image:Bridge UML class diagram.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Bridge_UML_class_diagram.svg *License*: Public Domain *Contributors*: Trashtoy

Image:Bridge pattern in LePUS3.1.gif *Source*: 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:Composite UML class diagram (fixed).svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Composite_UML_class_diagram_(fixed).svg *License*: Public Domain *Contributors*: Composite_UML_class_diagram.svg: Trashtoy derivative work: « Aaron Rotenberg « Talk «

Image:Composite pattern in LePUS3.png *Source*: 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 *License*: Public Domain *Contributors*: Trashtoy

Image:UML2 Decorator Pattern.png *Source*: http://en.wikipedia.org/w/index.php?title=File:UML2_Decorator_Pattern.png *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Nheirbaut (talk)

Image:FacadeDesignPattern.png *Source*: http://en.wikipedia.org/w/index.php?title=File:FacadeDesignPattern.png *License*: GNU Free Documentation License *Contributors*: Original uploader was 에멜무지로 at en.wikipedia

File:Flyweight UML class diagram.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Flyweight_UML_class_diagram.svg *License*: Public Domain *Contributors*: Trashtoy

Image:proxy_pattern_diagram.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Proxy_pattern_diagram.svg *License*: GNU Free Documentation License *Contributors*: Traced by User:Stannered, created by en:User:TravisHein

Image:Proxy pattern in LePUS3.gif *Source*: 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 *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 *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*: 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_v2.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Specification_UML_v2.png *License*: GNU Free Documentation License *Contributors*: 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 *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: State_Design_Pattern_UML_Class_Diagram.png: JoaoTrindade (talk) Original uploader was JoaoTrindade at en.wikipedia derivative work: Ertwroc (talk)

Image:State pattern in LePUS3.gif *Source*: http://en.wikipedia.org/w/index.php?title=File:State_pattern_in_LePUS3.gif *License*: Public Domain *Contributors*: Amnon Eden (User:Edenphd)

Image:StrategyPatternClassDiagram.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:StrategyPatternClassDiagram.svg *License*: Creative Commons Zero *Contributors*: Bocsika

Image:Strategy pattern in LePUS3.gif *Source*: http://en.wikipedia.org/w/index.php?title=File:Strategy_pattern_in_LePUS3.gif *License*: Public Domain *Contributors*: Amnon Eden (User:Edenphd)

Image:StrategyPattern IBrakeBehavior.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:StrategyPattern_IBrakeBehavior.svg *License*: Creative Commons Zero *Contributors*: Ptoonen

Image:Template Method UML.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Template_Method_UML.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *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 *License*: Public Domain *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 *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Amnon Eden (User:Edenphd)

Image:VisitorPatternUML.png *Source*: http://en.wikipedia.org/w/index.php?title=File:VisitorPatternUML.png *License*: Public domain *Contributors*: User:SreeBot

Image:Monitor (synchronization)-SU.png *Source*: 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 *License*: Creative Commons Attribution 3.0 *Contributors*: Theodore.norvell (talk)

Image:Monitor (synchronization)-Java.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Monitor_(synchronization)-Java.png *License*: Creative Commons Attribution 3.0 *Contributors*: Theodore.norvell (talk)

Image:Thread pool.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Thread_pool.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: en:User:Cburnett

License