Design pattern (computer science)

From Wikipedia, the free encyclopedia

In software engineering, a **design pattern** is a general reusable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Algorithms are not thought of as design patterns, since they solve computational problems rather than design problems.

Not all software patterns are design patterns. Design patterns deal specifically with problems at the level of software *design*. Other kinds of patterns, such as architectural patterns, describe problems and solutions that have alternative scopes.

Contents

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.[1][2] 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 (Gamma et al.). That same year, the first Pattern Languages of Programming Conference was held and the following year, the Portland Pattern Repository was set up for documentation of design patterns. The scope of the term remains a matter

of dispute. Notable books in the design pattern genre include:

- Fowler, Martin (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley. ISBN 978-0321127426.
- 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.

Although the practical application of design patterns is a phenomenon, formalization of the concept of a design pattern languished for several years.[3]

Practice

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

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

By definition, a pattern must be programmed anew into each application that uses it. Since some authors see this as a step backward from software reuse as provided by components, researchers have worked to turn patterns into components. Meyer and Arnout claim a two-thirds success rate in componentizing the best-known patterns.[4]

Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that doesn't 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, 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,[5] Information Visualization [6], Secure Usability[7] and web design.[8]

The Pattern Languages of Programming Conference (annual, 1994—) proceedings (http://hillside.net/plop/pastconferences.html) includes many examples of domain specific patterns.

Classification

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

Name	Description	In *Design Patterns (book)*	In *Code Complete*[9]
Creational patterns			
Abstract factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.	Yes	Yes
Factory method	Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.	Yes	Yes
Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representations.	Yes	No
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
Object pool	Avoid expensive acquisition and release of resources by recycling objects that are no longer in use	No	No
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.	Yes	No
Singleton	Ensure a class has only one instance, and provide a global point of access to it.	Yes	Yes
Structural patterns			
Adapter	Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.	Yes	Yes
Bridge	Decouple an abstraction from its implementation so that the two can vary independently.	Yes	Yes

Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.	Yes	Yes
Decorator	Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.	Yes	Yes
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
Flyweight	Use sharing to support large numbers of fine-grained objects efficiently.	Yes	No
Proxy	Provide a surrogate or placeholder for another object to control access to it.	Yes	No
Behavioral patterns			
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
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
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
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.	Yes	Yes
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
Memento	Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.	Yes	No
Null Object	designed to act as a default value of an object.	No	No
Observer	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.	Yes	Yes
State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.	Yes	No
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
Specification	Recombinable business logic in a boolean fashion	No	No

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
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
Concurrency patterns			
Active Object	The Active Object design pattern 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.	No	No
Event-Based Asynchronous	The event-based asynchronous pattern design pattern addresses problems with the Asychronous Pattern that occur in multithreaded programs.[10]	No	No
Balking	The Balking pattern is a software design pattern that only executes an action on an object when the object is in a particular state.	No	No
Double checked locking	Double-checked locking is a software design pattern also known as "double-checked locking optimization". The pattern is designed to 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. The pattern, when implemented in some language/hardware combinations, can be unsafe. It can therefore sometimes be considered to be an anti-pattern.	No	No
Guarded	In concurrent programming, guarded suspension 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.	No	No
Monitor object	A monitor is an approach to synchronize two or more computer tasks that use a shared resource, usually a hardware device or a set of variables.	No	No
Read write lock	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.	No	No
Scheduler	The scheduler pattern is a software design pattern. It is a concurrency pattern used to explicitly control when threads may execute single-threaded code.	No	No
	In the thread pool pattern in programming, a number of		

Thread pool	threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads.	No	No
Thread-specific storage	Thread-local storage (TLS) is a computer programming method that uses static or global memory local to a thread.	No	No
Reactor	The reactor design pattern is a concurrent programming pattern for handling service requests delivered concurrently to a service handler by one or more inputs. The service handler then demultiplexes the incoming requests and dispatches them synchronously to the associated request handlers.	No	No

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.[11] 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.[12] 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

In the field of computer science, there exist some criticisms regarding the concept of design patterns.

Workarounds for missing language features

Users of dynamic programming languages have discussed many design patterns as workarounds for the limitations of languages such as C++ and Java. For instance, the Visitor pattern need not be implemented in a language that supports multimethods. The purpose of Visitor is to add new operations to existing classes without modifying those classes. In C++, a class is declared as a syntactic structure with a specific and closed set of methods. In a language with multimethods, such as Common Lisp, methods for a class are outside of the class structure, and one may add new methods without modifying it. Similarly, the Decorator pattern amounts to implementing dynamic delegation, as found in Common Lisp, Objective C, Self and JavaScript.

Peter Norvig, in *Design Patterns in Dynamic Programming*, discusses the triviality of implementing various patterns in dynamic languages. [13] Norvig and others have described language features that encapsulate or replace various patterns that a C++ user must implement for themselves.

Does not differ significantly from other abstractions

Some authors allege that design patterns don't differ significantly from other forms of abstraction, and that the use of new terminology (borrowed from the architecture community) to describe existing phenomena in the field of programming is unnecessary. The Model-View-Controller paradigm is cited as an example of a "pattern" which predates the concept of "design patterns" by several years.[14] It is further argued by some that the primary contribution of the Design Patterns community (and the Gang of Four book) was the use of Alexander's pattern language as a form of documentation; a practice which is often ignored in the literature.

See also

- Anti-pattern
- Architectural pattern
- Christopher Alexander
- Distributed design patterns
- GRASP (Object Oriented Design)
- Interaction design pattern
- List of software development philosophies
- List of software engineering topics
- Pattern language
- Pattern theory
- Pedagogical patterns
- Portland Pattern Repository
- Refactoring

References

1. ^ Smith, Reid (October 1987). "Panel on design methodology" in *OOPSLA '87. OOPSLA '87 Addendum to the Proceedings*. 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.*"
2. ^ Beck, Kent; Ward Cunningham (September 1987). "Using Pattern Languages for Object-Oriented Program" in *OOPSLA '87. OOPSLA '87 workshop on* Specification and Design for Object-Oriented Programming'**. Retrieved on 2006-05-26.**
3. ^ Baroni, Aline Lúcia; Yann-Gaël Guéhéneuc and Hervé Albin-Amiot (June 2003). "Design Patterns Formalization". written at Nantes (PDF). École Nationale Supérieure des Techniques Industrielles et des Mines de Nantes. http://www.iro.umontreal.ca/~ptidej/Publications/Documents/Research+report+Metamodeling+June03.doc.pdf. Retrieved on 2007-12-29.
4. ^ Meyer, Bertrand; Karine Arnout (July 2006). "Componentization: The Visitor Example". *IEEE Computer* (IEEE) **39** (7): 23–30. http://se.ethz.ch/~meyer/publications/computer/visitor.pdf.
5. ^ Laakso, Sari A. (2003-09-16). "Collection of User Interface Design Patterns". University of Helsinki, Dept. of Computer Science. http://www.cs.helsinki.fi/u/salaakso/patterns/index.html. Retrieved on 2008-01-31.
6. ^ Heer, J.; M. Agrawala (2006). "Software Design Patterns for Information Visualization". *IEEE Transactions on Visualization and Computer Graphics* **12** (5): 853. doi:10.1109/TVCG.2006.178. http://vis.berkeley.edu/papers/infovis_design_patterns/.
7. ^ Simson L. Garfinkel (2005). *Design Principles and Patterns for Computer Systems That Are Simultaneously Secure and Usable*. http://www.simson.net/thesis/.
8. ^ "Yahoo! Design Pattern Library". http://developer.yahoo.com/ypatterns/. Retrieved on 2008-01-31.
9. ^ McConnell, Steve (June 2004). "Design in Construction". *Code Complete* (2nd edition ed.). Microsoft Press. pp. 104. ISBN 978-0735619678. "Table 5.1 Popular Design Patterns"
10. ^ Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, and Morgan Skinner (2008). "Event-based Asynchronous Pattern". *Professional C# 2008*. Wiley. pp. 570–571. ISBN 0470191376.
11. ^ Gabriel, Dick. "A Pattern Definition". http://hillside.net/patterns/definition.html. Retrieved on 2007-03-06.
12. ^ Fowler, Martin (2006-08-01). "Writing Software Patterns". http://www.martinfowler.com/articles/writingPatterns.html. Retrieved on 2007-03-06.
13. ^ Norvig, Peter (1998-03-17). "Design Patterns in Dynamic Programming". http://norvig.com/design-patterns. Retrieved on 2007-12-29.
14. ^ Reenskaug, Trygve. "MVC XEROX PARC 1978-79". http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html. Retrieved on 2008-06-09.

Further reading

Books

- Alexander, Christopher; Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel (1977). *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press. ISBN 978-0195019193.
- Beck, Kent (October 2007). *Implementation Patterns*. Addison-Wesley. ISBN 978-0321413093.
- Beck, Kent; R. Crocker, G. Meszaros, J.O. Coplien, L. Dominick, F. Paulisch, and J. Vlissides (March 1996). *Proceedings of the 18th International Conference on Software Engineering*. pp. 25–30.
- Borchers, Jan (2001). *A Pattern Approach to Interaction Design*. John Wiley & Sons. ISBN 0-471-49828-9.
- Coplien, James O.; Douglas C. Schmidt (1995). *Pattern Languages of Program Design*.

Addison-Wesley. ISBN 0-201-60734-4.
- Coplien, James O.; John M. Vlissides, and Norman L. Kerth (1996). *Pattern Languages of Program Design 2*. Addison-Wesley. ISBN 0-201-89527-7.
- Fowler, Martin (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley. ISBN 978-0321127426.
- Freeman, Eric; Elisabeth Freeman, Kathy Sierra, and Bert Bates (2004). *Head First Design Patterns*. O'Reilly Media. ISBN 0-596-00712-4.
- Gabriel, Richard (1996) (PDF). *Patterns of Software: Tales From The Software Community*. Oxford University Press. pp. 235. ISBN 0-19-512123-6. http://www.dreamsongs.com/NewFiles/PatternsOfSoftware.pdf.
- 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.
- Kerievsky, Joshua (2004). *Refactoring to Patterns*. Addison-Wesley. ISBN 0-321-21335-1.
- 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.
- 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*. Addison-Wesley. ISBN 0201596075. http://www.cix.co.uk/~smallmemory/.

Web sites

- "History of Patterns". *Portland Pattern Repository*. http://www.c2.com/cgi-bin/wiki?HistoryOfPatterns. Retrieved on 2005-07-28.
- "Are Design Patterns Missing Language Features?". Cunningham & Cunningham, Inc.. http://www.c2.com/cgi/wiki?AreDesignPatternsMissingLanguageFeatures. Retrieved on 2006-01-20.
- "Show Trial of the Gang of Four". Cunningham & Cunningham, Inc.. http://www.c2.com/cgi/wiki?ShowTrialOfTheGangOfFour. Retrieved on 2006-01-20.

External links

- Directory of websites that provide pattern catalogs

(http://hillside.net/patterns/onlinepatterncatalog.htm) at hillside.net.
- Ward Cunningham's *Portland Pattern Repository*.
- Patterns and Anti-Patterns (http://www.dmoz.org/Computers/Programming/Methodologies/Patterns_and_Anti-Patterns//) at the Open Directory Project
- PerfectJPattern Open Source Project (http://perfectjpattern.sourceforge.net) Design Patterns library that aims to provide full or partial componentized version of all known Patterns in Java.
- Jt (http://jt.dev.java.net) J2EE Pattern Oriented Framework
- Printable Design Patterns Quick Reference Cards (http://www.mcdonaldland.info/2007/11/28/40/)
- 101 Design Patterns & Tips for Developers (http://sourcemaking.com/design-patterns-and-tips)
- On Patterns and Pattern Languages (http://media.wiley.com/product_data/excerpt/28/04700590/0470059028.pdf) by Buschmann, Henney, and Schmidt
- Patterns for Scripted Applications (http://web.archive.org/web/20041010125419/www.doc.ic.ac.uk/~np2/patterns/scripting/)

Retrieved from "http://en.wikipedia.org/wiki/Design_pattern_(computer_science)"
Category: Software design patterns
Hidden categories: Cleanup from section | Articles with specifically-marked weasel-worded phrases | All articles with unsourced statements | Articles with unsourced statements since March 2007 | Articles with unsourced statements since February 2007 | Articles needing additional references from November 2008

Creational pattern

From Wikipedia, the free encyclopedia

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.

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: 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

See also

- Behavioral pattern
- Structural pattern
- Concurrency pattern

Abstract factory pattern

From Wikipedia, the free encyclopedia

A software design pattern, the **Abstract Factory Pattern** provides a way to encapsulate a group of individual factories that have a common theme. In normal usage, the client software would create a concrete implementation of the abstract factory and then use the generic interfaces to create the concrete objects that are part of the theme. The client does not know (or care) about 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 its general usage.

An example of this would be an abstract factory class *DocumentCreator* that provides interfaces to create a number of products (eg. *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 instantiation of the *DocumentCreator* and call its factory methods. Each of the resulting objects would be created from the same DocumentCreator implementation and would share a common theme (they would all be fancy or modern objects). The client would need to know how to handle only the abstract *Letter* or *Resume* class, not the specific version that it got from the concrete factory.

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

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

Contents

How to use it

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

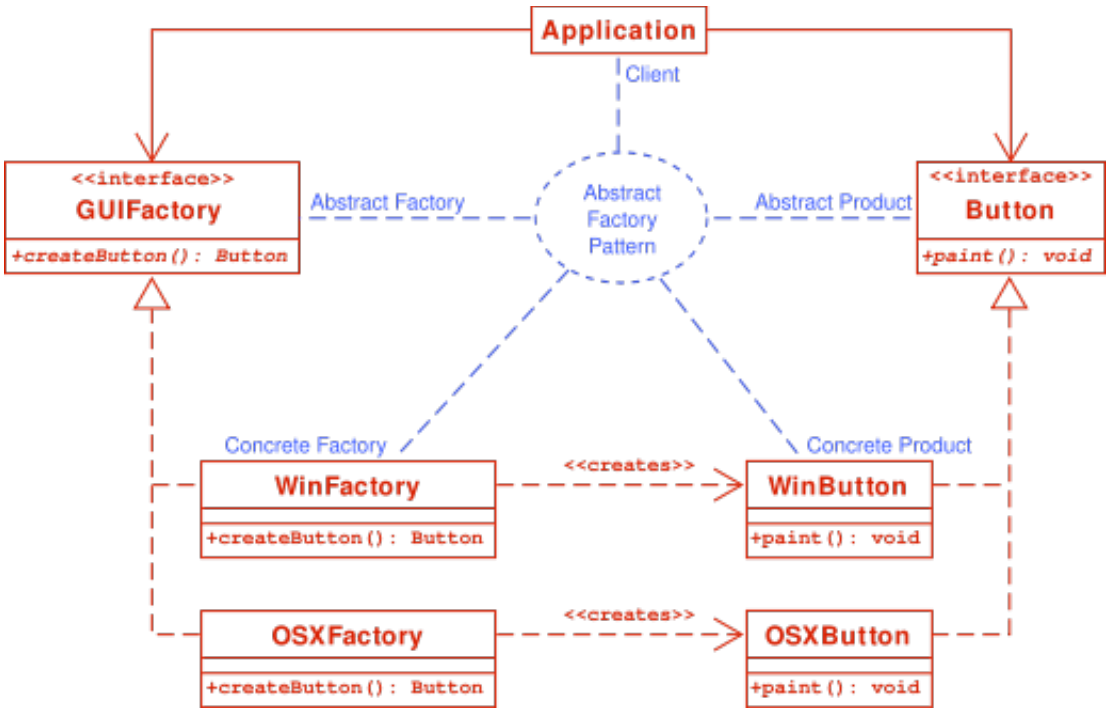
This insulates client code from object creation by having clients ask a factory object to create an object of the desired abstract type and to return an abstract pointer to the object.

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

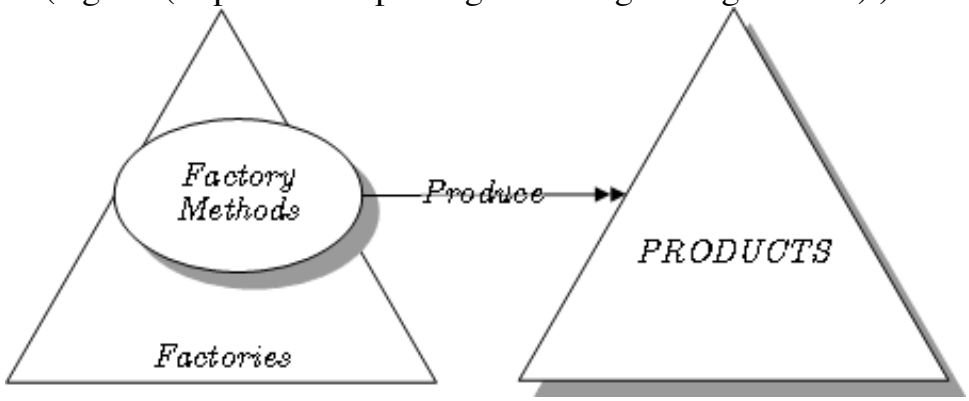
- The client code has no knowledge whatsoever of the concrete type, not needing to include any header files or class declarations relating to the concrete type. The client code deals only with the abstract type. Objects of a concrete type are indeed created by the factory, but the client code accesses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

Structure

The class diagram of this design pattern is as shown here:



The Lepus3 chart (legend (http://www.lepus.org.uk/ref/legend/legend.xml)) of this design pattern is as shown here:



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

Example

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();
        }
    }
}
```

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#

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

namespace Abstract_factory
{
    public interface IButton
    {
        void Paint();
    }
    public class WinButton : IButton
    {
        #region IButton Members

        public void Paint()
        {
```

```
            Console.Out.WriteLine("This is a WinButton");
        }

        #endregion
    }
    public class OSXButton : IButton
    {
        #region IButton Members

        public void Paint()
        {
            Console.Out.WriteLine("This is a OSXButton");
        }

        #endregion
    }
    public interface IGUIFactory
    {
        IButton CreateButton();
    }
    public class WinFactory : IGUIFactory
    {
        #region IGUIFactory Members

        public IButton CreateButton()
        {
            return new WinButton();
        }

        #endregion
    }
    public class OSXFactory : IGUIFactory
    {
        #region IGUIFactory Members

        public IButton CreateButton()
        {
            return new OSXButton();
        }

        #endregion
    }
    public class Application
    {
        public Application(IGUIFactory factory)
        {
            IButton button = factory.CreateButton();
            button.Paint();
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            IGUIFactory factory1 = new WinFactory();
            IGUIFactory factory2 = new OSXFactory();
            Application app1 = new Application(factory1);
            Application app2 = new Application(factory2);
            Console.ReadLine();
        }
    }
}
```

See also

- Object creation
- Concrete class
- Factory method pattern

- Design Pattern

External links

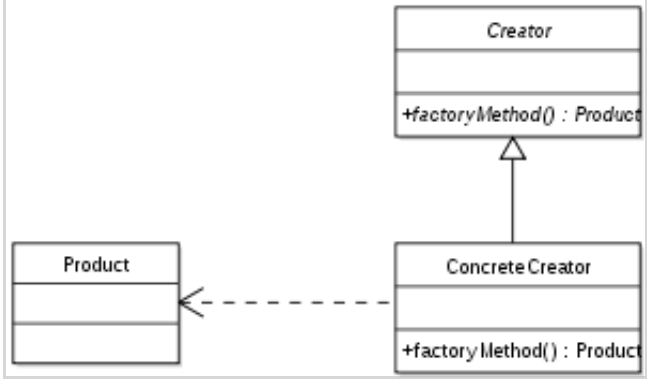
- Abstract Factory (http://c2.com/cgi-bin/wiki?AbstractFactory) in the C2 wiki
- Abstract Factory (http://www.dofactory.com/Patterns/PatternAbstract.aspx) , UML diagram
- Abstract Factory (http://www.netobjectivesrepository.com/TheAbstractFactoryPattern) on the Net Objectives Repository
- Jt (http://www.fsw.com/Jt/Jt.htm#_Toc210319840) Example of use on Jt, a J2EE Pattern Oriented Framework
- Abstract Factory (http://www.lepus.org.uk/ref/companion/AbstractFactory.xml) UML diagram + formal specification in LePUS3 and Class-Z (a Design Description Language)

Categories: Software design patterns | Articles with example Java code
Hidden category: Articles needing additional references from December 2008

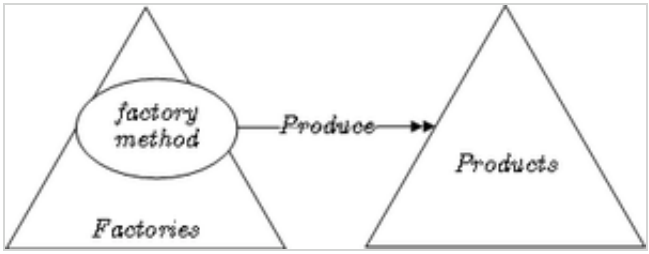
Factory method pattern

From Wikipedia, the free encyclopedia

The **factory method pattern** is an object-oriented design pattern. 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 factory method design pattern handles this problem by defining a separate method for creating the objects, whose subclasses can then override to specify the derived type of product that will be created. More generally, the term *factory method* is often used to refer to any method whose main purpose is creation of objects.



Factory method in UML



Factory Method in LePUS3

Contents

Definition

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

Common usage

- Factory methods are common in **toolkits** and **frameworks** where library code needs to create objects of types which may be subclassed by applications using the framework.
- Parallel class hierarchies often require objects from one hierarchy to be able to create appropriate objects from *another*.

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.

Encapsulation

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

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

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

public class GifReader implements ImageReader
{
     public GifReader( InputStream in )
     {
         // check that it's a gif, throw exception if it's not, then if it is
         // decode it.
     }

     public DecodedImage getDecodedImage()
     {
        return decodedImage;
     }
}

public class JpegReader implements ImageReader
{
     //....
}
```

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

```
public class ImageReaderFactory
{
    public static ImageReader getImageReader( InputStream is )
    {
        int imageType = figureOutImageType( is );

        switch( imageType )
        {
```

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

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

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 inheritance.

- The first limitation is that refactoring an existing class to use factories breaks existing clients. For example, if class Complex were 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 possible), the subclass must provide its own re-implementation of all factory methods with exactly the same signatures. For example, if class StrangeComplex extends Complex, then unless StrangeComplex provides its own version of all factory methods, the call StrangeComplex.fromPolar(1, pi) will yield an instance of *Complex* (the superclass) rather than the expected instance of the subclass.

All three problems could be alleviated by altering the underlying programming language to make factories first-class class members (rather than using the design pattern).

Examples

Java

Pizza example:

```
abstract class Pizza {
    public abstract double getPrice();
}
```

```
class HamAndMushroomPizza extends Pizza {
    private double price = 8.5;

    public double getPrice() {
        return price;
    }
}

class DeluxePizza extends Pizza {
    private double price = 10.5;

    public double getPrice() {
        return price;
    }
}

class HawaiianPizza extends Pizza {
    private double price = 11.5;

    public double getPrice() {
        return price;
    }
}

class PizzaFactory {
    public enum PizzaType {
        HamMushroom,
        Deluxe,
        Hawaiian
    }

    public static Pizza createPizza(PizzaType pizzaType) {
        switch (pizzaType) {
            case HamMushroom:
                return new HamAndMushroomPizza();
            case Deluxe:
                return new DeluxePizza();
            case Hawaiian:
                return new HawaiianPizza();
        }
        throw new IllegalArgumentException("The pizza type " + pizzaType + " is not recognized.");
    }
}

class PizzaLover {
    /**
     * Create all available pizzas and print their prices
     */
    public static void main (String args[]) {
        for (PizzaFactory.PizzaType pizzaType : PizzaFactory.PizzaType.values()) {
            System.out.println("Price of " + pizzaType + " is " + PizzaFactory.createPizza(pizzaType
        }
    }
}
```

C#

Pizza example:

```
public abstract class Pizza
{
    public abstract decimal GetPrice();

    public enum PizzaType
    {
        HamMushroom, Deluxe, Hawaiian
    }
    public static Pizza PizzaFactory(PizzaType pizzaType)
    {
```

```
        switch (pizzaType)
        {
            case PizzaType.HamMushroom:
                return new HamAndMushroomPizza();
                break;

            case PizzaType.Deluxe:
                return new DeluxePizza();
                break;

            case PizzaType.Hawaiian:
                return new HawaiianPizza();
                break;

            default:
                break;
        }

        throw new System.NotSupportedException("The pizza type " + pizzaType.ToString() + " is not r
    }
}
```

```
public class HamAndMushroomPizza : Pizza
{
    private decimal price = 8.5M;
    public override decimal GetPrice() { return price; }
}

public class DeluxePizza : Pizza
{
    private decimal price = 10.5M;
    public override decimal GetPrice() { return price; }
}

public class HawaiianPizza : Pizza
{
    private decimal price = 11.5M;
    public override decimal GetPrice() { return price; }
}

// Somewhere in the code
...
  Console.WriteLine( Pizza.PizzaFactory(Pizza.PizzaType.Hawaiian).GetPrice().ToString("C2") ); // $1
...
<pro>

private class Pro : Pizza

Console.WriteLine( Pizza.PizzaFactory(Pizza.PizzaType.Hawaiian).GetPrice().ToString("C2") ); // $11.
```

C++

Pizza example:

```
#include <string>
#include <iostream>
#include <memory> // std::auto_ptr
class Pizza {
public:
    virtual void get_price() const = 0;
    virtual ~Pizza() {};
};

class HamAndMushroomPizza: public Pizza {
public:
    virtual void get_price() const {
        std::cout << "Ham and Mushroom: $8.5" << std::endl;
```

```
    }
};

class DeluxePizza : public Pizza {
public:
    virtual void get_price() const {
        std::cout << "Deluxe: $10.5" << std::endl;
    }
};

class HawaiianPizza : public Pizza {
public:
    virtual void get_price() const {
        std::cout << "Hawaiian: $11.5" << std::endl;
    }
};

class PizzaFactory {
public:
    static Pizza* create_pizza(const std::string& type) {
        if (type == "Ham and Mushroom")
            return new HamAndMushroomPizza();
        else if (type == "Hawaiian")
            return new HawaiianPizza();
        else
            return new DeluxePizza();
    }
};
//usage
int main() {
    PizzaFactory factory;

    std::auto_ptr<const Pizza> pizza(factory.create_pizza("Default"));
    pizza->get_price();

    pizza.reset(factory.create_pizza("Ham and Mushroom"));
    pizza->get_price();

    pizza.reset(factory.create_pizza("Hawaiian"));
    pizza->get_price();
}
```

JavaScript

Pizza example:

```
//Our pizzas
function HamAndMushroomPizza(){
  var price = 8.50;
  this.getPrice = function(){
    return price;
  }
}

function DeluxePizza(){
  var price = 10.50;
  this.getPrice = function(){
    return price;
  }
}

function HawaiianPizza(){
  var price = 11.50;
  this.getPrice = function(){
    return price;
  }
}

//Pizza Factory
function PizzaFactory(){
  this.createPizza = function(type){
```

```
    switch(type){
     case "Ham and Mushroom":
       return new HamAndMushroomPizza();
     case "Hawaiian":
       return new HawaiianPizza();
     default:
         return new DeluxePizza();
    }
  }
}
```

```
//Usage
var pizzaPrice = new PizzaFactory().createPizza("Ham and Mushroom").getPrice();
alert(pizzaPrice);
```

Ruby

Pizza example:

```
class HamAndMushroomPizza
  def price
    8.50
  end
end

class DeluxePizza
  def price
    10.50
  end
end

class HawaiianPizza
  def price
    11.50
  end
end

class ChunkyBaconPizza
  def price
    19.95
  end
end

class PizzaFactory
  def create_pizza(style='')
    case style
      when 'Ham and Mushroom'
        HamAndMushroomPizza.new
      when 'Deluxe'
        DeluxePizza.new
      when 'Hawaiian'
        HawaiianPizza.new
      when 'WithChunkyBacon'
        ChunkyBaconPizza.new
      else
        DeluxePizza.new
    end
  end
end

# usage
factory = PizzaFactory.new
pizza = factory.create_pizza('Ham and Mushroom')
pizza.price #=> 8.5
# bonus formatting
formatted_price = sprintf("$%.2f", pizza.price) #=> "$8.50"
# one liner
sprintf("$%.2f", PizzaFactory.new.create_pizza('Ham and Mushroom').price) #=> "$8.50"
```

Python

```
# Our Pizzas

class Pizza(object):
    PIZZA_TYPE_HAM_MUSHROOM, PIZZA_TYPE_DELUXE, PIZZA_TYPE_HAWAIIAN = xrange(3)
    def __init__(self): self.price = None
    def getPrice(self): return self.price
```

```
class HamAndMushroomPizza(Pizza):
    def __init__(self): self.price = 8.50

class DeluxePizza(Pizza):
    def __init__(self): self.price = 10.50

class HawaiianPizza(Pizza):
    def __init__(self): self.price = 11.50

class PizzaFactory(object):
    def make(self, pizzaType):
        classByType = {
            Pizza.PIZZA_TYPE_HAM_MUSHROOM: HamAndMushroomPizza,
            Pizza.PIZZA_TYPE_DELUXE: DeluxePizza,
            Pizza.PIZZA_TYPE_HAWAIIAN: HawaiianPizza,
        }
        return classByType[pizzaType]()

# Usage
print '$ %.02f' % PizzaFactory().make(Pizza.PIZZA_TYPE_HAM_MUSHROOM).getPrice()
```

Php

```
// Our Pizzas
abstract class Pizza {
    abstract public function get_price();
}

class HamAndMushroomPizza extends Pizza {
    public function get_price() {
        return 8.5;
    }
}

class DeluxePizza extends Pizza {
    public function get_price() {
        return 10.5;
    }
}

class HawaiianPizza extends Pizza {
    public function get_price() {
        return 11.5;
    }
}

class PizzaFactory {
    public static function create_pizza( $type ) {
        switch( $type ) {
            case 'Ham and Mushroom':
                return new HamAndMushroomPizza();
                break;
            case 'Hawaiian':
                return new HawaiianPizza();
                break;
            default:
                return new DeluxePizza();
        }
```

```
    }
}
// Usage
$pizza = PizzaFactory::create_pizza( 'Hawaiian' );
echo $pizza->get_price();
```

Haskell

Pizza example:

```
class Pizza a where
  price :: a -> Float

data HamMushroom = HamMushroom
data Deluxe      = Deluxe
data Hawaiian    = Hawaiian

instance Pizza HamMushroom where
  price _ = 8.50

instance Pizza Deluxe where
  price _ = 10.50

instance Pizza Hawaiian where
  price _ = 11.50
```

Usage example:

```
main = print (price Hawaiian)
```

Uses

- In ADO.NET, IDbCommand.CreateParameter (http://msdn2.microsoft.com/en-us/library/system.data.idbcommand.createparameter.aspx) is an example of the use of factory method to connect parallel class hierarchies.
- In Qt, QMainWindow::createPopupMenu (http://doc.trolltech.com/4.0/qmainwindow.html#createPopupMenu) is a factory method declared in a framework which can be overridden in application code.
- In Java, several factories are used in the javax.xml.parsers (http://java.sun.com/j2se/1.4.2/docs/api/javax/xml/parsers/package-summary.html) package. e.g. javax.xml.parsers.DocumentBuilderFactory or javax.xml.parsers.SAXParserFactory.

See also

- Design Patterns, the highly-influential book
- Abstract Factory, a pattern often implemented using factory methods
- Builder pattern, another creational pattern
- Template method pattern, which may call factory methods
- Factory object

References

- 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.
- Cohen, Tal; Gil, Joseph (2007). "Better Construction with Factories" (PDF). *Journal of Object Technology* (Bertrand Meyer). http://tal.forum2.org/static/cv/Factories.pdf. Retrieved on 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)
- Description from the Portland Pattern Repository (http://c2.com/cgi/wiki?FactoryMethodPattern)
- Jt (http://www.fsw.com/Jt/Jt.htm) J2EE Pattern Oriented Framework

Categories: Software design patterns | Articles with example Haskell code | Articles with example Java code | Articles with example Python code | Method (computer science)
Hidden categories: All articles with unsourced statements | Articles with unsourced statements

---

Builder pattern

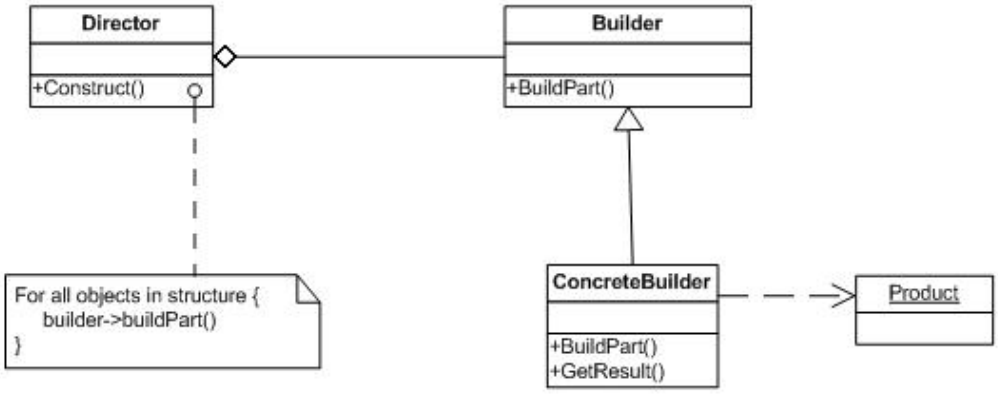
From Wikipedia, the free encyclopedia

The **Builder Pattern** is a software design pattern. The intention is to abstract steps of construction of object so that different implementations of these steps can construct different representations of objects.

Often, the **Builder Pattern** is used to build Products in accordance to the Composite pattern, a structure pattern.

Contents

Class Diagram



Builder

Abstract interface for creating objects (product).

Concrete Builder

Provide implementation for Builder. Construct and assemble parts to build the objects.

Director

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

Product

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

Useful tips

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

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 Cook {
        private PizzaBuilder pizzaBuilder;

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

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

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

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

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

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

C#

```
//Implementation in C#.
class Pizza
{
    string dough;
    string sauce;
    string topping;
    public string Dough    { get { return dough; }    set { dough = value; } }
    public string Sauce    { get { return sauce; }    set { sauce = value; } }
    public string Topping { get { return topping; } set { topping = value; } }
    public override string ToString() {
        return string.Format(
                "Pizza with Dough as {0}, Sauce as {1} and Topping as {2}",
```

```
                Dough,
                Sauce,
                Topping);
    }
}

//Abstract Builder
abstract class PizzaBuilder
{
    protected Pizza pizza;
    public Pizza Pizza { get { return pizza; } }
    public void CreateNewPizza() { pizza = new Pizza(); }

    public abstract void BuildDough();
    public abstract void BuildSauce();
    public abstract void BuildTopping();
}

//Concrete Builder
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
class SpicyPizzaBuilder : PizzaBuilder
{
    public override void BuildDough()   { pizza.Dough   = "pan baked"; }
    public override void BuildSauce()   { pizza.Sauce   = "hot"; }
    public override void BuildTopping() { pizza.Topping = "pepparoni+salami"; }
}

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

    public PizzaBuilder PizzaBuilder { get { return pizzaBuilder; } set { pizzaBuilder = value; } }
    public Pizza Pizza { get { return pizzaBuilder.Pizza; } }

    public void ConstructPizza() {
        pizzaBuilder.CreateNewPizza();
        pizzaBuilder.BuildDough();
        pizzaBuilder.BuildSauce();
        pizzaBuilder.BuildTopping();
    }
}

public class TestPizza {
    private static void BuildAndDisplayPizza(Waiter waiter) {
        waiter.ConstructPizza();
        System.Console.WriteLine(waiter.Pizza);
    }

    public static void Main() {
        Waiter waiter = new Waiter();

        waiter.PizzaBuilder = new HawaiianPizzaBuilder();
        BuildAndDisplayPizza(waiter);

        waiter.PizzaBuilder = new SpicyPizzaBuilder();
        BuildAndDisplayPizza(waiter);
    }
}
```

C++

```
// Implementation in C++.

#include <iostream>
#include <memory>
#include <string>

// Product
class Pizza
{
private:
    std::string dough;
    std::string sauce;
    std::string topping;
```

```
public:
    Pizza() { }
    ~Pizza() { }

    void SetDough(const std::string& d) { dough = d; };
    void SetSauce(const std::string& s) { sauce = s; };
    void SetTopping(const std::string& t) { topping = t; }

    void ShowPizza()
    {
        std::cout << " Yummy !!!" << std::endl
        << "Pizza with Dough as " << dough
        << ", Sauce as " << sauce
        << " and Topping as " << topping
        << " !!! " << std::endl;
    }
};
```

```
// Abstract Builder
class PizzaBuilder
{
protected:
    std::auto_ptr<Pizza> pizza;
public:
    PizzaBuilder() {}
    virtual ~PizzaBuilder() {}
    std::auto_ptr<Pizza> GetPizza() { return pizza; }

    void createNewPizzaProduct() { pizza.reset (new Pizza); }

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

};

// ConcreteBuilder
class HawaiianPizzaBuilder : public PizzaBuilder
{
public:
    HawaiianPizzaBuilder() : PizzaBuilder() {}
    ~HawaiianPizzaBuilder(){}

    void buildDough() { pizza->SetDough("cross"); }
    void buildSauce() { pizza->SetSauce("mild"); }
    void buildTopping() { pizza->SetTopping("ham and pineapple"); }
};
```

```
// ConcreteBuilder
class SpicyPizzaBuilder : public PizzaBuilder
{
public:
    SpicyPizzaBuilder() : PizzaBuilder() {}
    ~SpicyPizzaBuilder() {}

    void buildDough() { pizza->SetDough("pan baked"); }
    void buildSauce() { pizza->SetSauce("hot"); }
    void buildTopping() { pizza->SetTopping("pepperoni and salami"); }
};

// Director
class Waiter
{
private:
    PizzaBuilder* pizzaBuilder;
public:
    Waiter() : pizzaBuilder(NULL) {}
    ~Waiter() { }

    void SetPizzaBuilder(PizzaBuilder* b) { pizzaBuilder = b; }
    std::auto_ptr<Pizza> GetPizza() { return pizzaBuilder->GetPizza(); }
    void ConstructPizza()
    {
        pizzaBuilder->createNewPizzaProduct();
        pizzaBuilder->buildDough();
        pizzaBuilder->buildSauce();
        pizzaBuilder->buildTopping();
    }
};
```

```
// A customer ordering two pizza.
int main()
{
    Waiter waiter;

    HawaiianPizzaBuilder hawaiianPizzaBuilder;
    waiter.SetPizzaBuilder (&hawaiianPizzaBuilder);
    waiter.ConstructPizza();
    std::auto_ptr<Pizza> pizza = waiter.GetPizza();
    pizza->ShowPizza();

    SpicyPizzaBuilder spicyPizzaBuilder;
    waiter.SetPizzaBuilder(&spicyPizzaBuilder);
    waiter.ConstructPizza();
    pizza = waiter.GetPizza();
    pizza->ShowPizza();

    return EXIT_SUCCESS;
}
```

Visual Prolog

Product

```
interface pizza
   predicates
      setDough : (string Dough).
      setSauce : (string Sauce).
      setTopping : (string Topping).
end interface pizza

class pizza : pizza
end class pizza

implement pizza
   facts
      dough : string := "".
      sauce : string := "".
      topping : string := "".
   clauses
      setDough(Dough) :- dough := Dough.
   clauses
      setSauce(Sauce) :- sauce := Sauce.
   clauses
      setTopping(Topping) :- topping := Topping.
end implement pizza
```

Abstract Builder

```
interface pizzaBuilder
   predicates
      getPizza : () -> pizza Pizza.
      createNewPizzaProduct : ().
   predicates
      buildDough : ().
      buildSauce : ().
      buildTopping : ().
end interface pizzaBuilder
```

Visual Prolog does not support abstract classes, but we can create a support class instead:

```
interface pizzaBuilderSupport
   predicates from pizzaBuilder
      getPizza, createNewPizzaProduct
end interface pizzaBuilderSupport

class pizzaBuilderSupport : pizzaBuilderSupport
end class pizzaBuilderSupport

implement pizzaBuilderSupport
```

```
   facts
      pizza : pizza := erroneous.
   clauses
      getPizza() = pizza.
   clauses
      createNewPizzaProduct() :- pizza := pizza::new().
end implement pizzaBuilderSupport
```

ConcreteBuilder #1

```
class hawaiianPizzaBuilder :  pizzaBuilder
end class hawaiianPizzaBuilder

implement hawaiianPizzaBuilder
   inherits pizzaBuilderSupport

   clauses
      buildDough() :- getPizza():setDough("cross").
   clauses
      buildSauce() :- getPizza():setSauce("mild").
   clauses
      buildTopping() :- getPizza():setTopping("ham+pineapple").
end implement hawaiianPizzaBuilder
```

ConcreteBuilder #2

```
class spicyPizzaBuilder :  pizzaBuilder
end class spicyPizzaBuilder

implement spicyPizzaBuilder
   inherits pizzaBuilderSupport

   clauses
      buildDough() :- getPizza():setDough("pan baked").
   clauses
      buildSauce() :- getPizza():setSauce("hot").
   clauses
      buildTopping() :- getPizza():setTopping("pepperoni+salami").
end implement spicyPizzaBuilder
```

Director

```
interface waiter
   predicates
      setPizzaBuilder : (pizzaBuilder PizzaBuilder).
      getPizza : () -> pizza Pizza.
   predicates
      constructPizza : ().
end interface waiter

class waiter : waiter
end class waiter

implement waiter
   facts
      pizzaBuilder : pizzaBuilder := erroneous.
   clauses
      setPizzaBuilder(PizzaBuilder) :- pizzaBuilder := PizzaBuilder.
   clauses
      getPizza() = pizzaBuilder:getPizza().
   clauses
      constructPizza() :-
         pizzaBuilder:createNewPizzaProduct(),
         pizzaBuilder:buildDough(),
         pizzaBuilder:buildSauce(),
         pizzaBuilder:buildTopping().
end implement waiter
```

A customer ordering a pizza.

```
goal
   Hawaiian_pizzabuilder = hawaiianPizzaBuilder::new(),
   Waiter = waiter::new(),
   Waiter:setPizzaBuilder(Hawaiian_pizzabuilder),
   Waiter:constructPizza(),
   Pizza = Waiter:getPizza().
```

perl

```
## Product
package pizza;

sub new {
    return bless {
        dough => undef,
        sauce => undef,
        topping => undef
    }, shift;
}

sub set_dough {
    my( $self, $dough ) = @_;
    $self->{dough} = $dough;
}

sub set_sauce {
    my( $self, $sauce ) = @_;
    $self->{sauce} = $sauce;
}

sub set_topping {
    my( $self, $topping ) = @_;
    $self->{topping} = $topping;
}

1;
```

```
## Abstract builder
package pizza_builder;

sub new {
    return bless {
        pizza => undef
    }, shift;
}

sub get_pizza {
    my( $self ) = @_;
    return $self->{pizza};
}

sub create_new_pizza_product {
    my( $self ) = @_;
    $self->{pizza} = pizza->new;
}
```

```
# This is what an abstract method could look like in perl...

sub build_dough {
    croak("This method must be overridden.");
}

sub build_sauce {
    croak("This method must be overridden.");
}

sub build_topping {
    croak("This method must be overridden.");
}

1;
```

```
## Concrete builder
package hawaiian_pizza_builder;
```

```
use base qw{ pizza_builder };

sub build_dough {
    my( $self ) = @_;
    $self->{pizza}->set_dough("cross");
}

sub build_sauce {
    my( $self ) = @_;
    $self->{pizza}->set_sauce("mild");
}

sub build_topping {
    my( $self ) = @_;
    $self->{pizza}->set_topping("ham+pineapple");
}

1;
```

```
## Concrete builder
package spicy_pizza_builder;

use base qw{ pizza_builder };

sub build_dough {
    my( $self ) = @_;
    $self->{pizza}->set_dough("pan baked");
}

sub build_sauce {
    my( $self ) = @_;
    $self->{pizza}->set_sauce("hot");
}

sub build_topping {
    my( $self ) = @_;
    $self->{pizza}->set_topping("pepperoni+salami");
}

1;
```

```
## Director
package waiter;

sub new {
    return bless {
        pizza_builder => undef
    }, shift;
}

sub set_pizza_builder {
    my( $self, $builder ) = @_;
    $self->{pizza_builder} = $builder;
}

sub get_pizza {
    my( $self ) = @_;
    return $self->{pizza_builder}->get_pizza;
}

sub construct_pizza {
    my( $self ) = @_;
    $self->{pizza_builder}->create_new_pizza_product;
    $self->{pizza_builder}->build_dough;
    $self->{pizza_builder}->build_sauce;
    $self->{pizza_builder}->build_topping;
}

1;
```

```
## Lets order pizza (client of Director/Builder)
package main

my $waiter = waiter->new;
my $hawaiian_pb = hawaiian_pizza_builder->new;
my $spicy_pb = spicy_pizza_builder->new;

$waiter->set_pizza_builder( $hawaiian_pb );
$waiter->construct_pizza;
```

```
my $pizza = $waiter->get_pizza;

print "Serving a nice pizza with:\n";
for (keys %$pizza) {
    print "  $pizza->{$_} $_\n";
}

1;
```

PHP

```
/** Product **/
class Pizza{
  private $dough;
  private $sauce;
  private $topping;
  public function setDough($dough){
    $this->dough = $dough;
  }
  public function setSauce($sauce){
    $this->sauce = $sauce;
  }
  public function setTopping($topping){
    $this->topping = $topping;
  }
}
```

```
/** Abstract builder **/
abstract class PizzaBuilder{
  protected $pizza;
  public function __construct(){
    $this->pizza = new Pizza();
  }
  public function getPizza(){
    return $this->pizza;
  }
  abstract function buildDough();
  abstract function buildSauce();
  abstract function buildTopping();
}

/** Concrete builder  **/
class SpicyPizza extends PizzaBuilder{
  public function buildDough(){
    $this->pizza->setDough('thin');
  }
  public function buildSauce(){
    $this->pizza->setSauce('hot');
  }
  public function buildTopping(){
    $this->pizza->setTopping('pepperoni+salami');
  }
}
```

```
/** Director **/
class Chef{
  private $pizza_builder;
  public function setPizzaBuilder(PizzaBuilder $pizza_builder){
    $this->pizza_builder = $pizza_builder;
  }
  public function cookPizza(){
    $this->pizza_builder->buildDough();
    $this->pizza_builder->buildSauce();
    $this->pizza_builder->buildTopping();
  }
  public function getPizza(){
    return $this->pizza_builder->getPizza();
  }
}
```

```
//Customer orders a Pizza.
$chef = new Chef();

$order = new SpicyPizza();
$chef->setPizzaBuilder($order);
$chef->cookPizza();
```

```
$pizza = $chef->getPizza();

print_r($pizza);
```

External links

- Jt (http://www.fsw.com/Jt/Jt.htm) J2EE Pattern Oriented Framework
- What is the difference between Factory pattern and Builder Pattern ? (http://groups.google.com/group/comp.object/browse_thread/thread/db4b3914ddea5131/25e7e96e2e91983b?lnk=st&q=&rnum=1#25e7e96e2e91983b)

Lazy initialization

From Wikipedia, the free encyclopedia

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.

Contents

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 (compare with a singleton pattern)
- using lazy initialization to instantiate the object the first time it is requested (lazy initialization pattern).

Examples

Java

Here is a dummy example (in Java). The `Fruit` class itself doesn't do anything here, this is just an example to show the architecture. The class variable `types` is a map used to store `Fruit` instances by type.

```
import java.util.*;

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

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

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

C#

Here is the fruit example in C# (based on C++ exmaple)

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

public class Fruit
{
    private static Dictionary<string,Fruit> types = new Dictionary<string,Fruit>();
    private string type;

    /// <summary>
    /// using a private constructor to force use of the factory method.
    /// </summary>
    /// <param name="type">Type of fruit</param>
    private Fruit(String type) {
      this.type = type;
    }

    /// </summary>
    /// <param name="type">Any string that describes a fruit type, e.g. "apple"</param>
    /// <returns>The Fruit instance associated with that type.</returns>
    public static Fruit getFruit(string type) {
        Fruit f;

        if ( ! types.TryGetValue(type, out f) )
        {
          f = new Fruit(type); // lazy initialization
          types.Add(type,f);
        }

      return f;
    }

    public static void printCurrentTypes() {
        if (types.Count > 0) {
            Console.WriteLine("Number of instances made = {0}",types.Count);
            foreach (KeyValuePair<string,Fruit> kvp in types)
            {
                Console.WriteLine(kvp.Key);
            }
            Console.WriteLine();
        }
```

```
    }

}



class Program
{
    static void Main(string[] args)
    {
        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();

        Console.ReadLine();
    }
}
```

C++

Here is how you could do the fruit example in C++

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

using namespace std;

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

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

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

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

/*
 * Lazy Factory method, gets the Fruit instance associated with a
 * certain type. Instantiates new ones as needed.
 * precondition: type. Any string that describes a fruit type, e.g. "apple"
 * postcondition: The Fruit instance associated with that type.
 */
Fruit* Fruit::getFruit(const string& type) {
    Fruit *& f = types[type];   //try to find a pre-existing instance

    if (!f) {
        // couldn't find one, so make a new instance
        f = new Fruit(type); // lazy initialization part
    }
    return f;
}

/*
 * For example purposes to see pattern in action
 */
void Fruit::printCurrentTypes() {
```

```
    if (types.size() > 0) {
        cout << "Number of instances made = " << types.size() << endl;
        for (map<string,Fruit*>::iterator iter = types.begin(); iter != types.end(); ++iter) {
            cout << (*iter).first << endl;
        }
        cout << endl;
    }
}

int main(void) {
    Fruit::getFruit("Banana");
    Fruit::printCurrentTypes();

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

    // returns pre-existing instance from first
    // time Fruit with "Banana" was created
    Fruit::getFruit("Banana");
    Fruit::printCurrentTypes();

    return 0;
}



/*
OUTPUT:
Number of instances made = 1
Banana

Number of instances made = 2
Apple
Banana

Number of instances made = 2
Apple
Banana
*/
```

SmallTalk

The following is an example (in Smalltalk) of a typical accessor method to return the value of a variable using lazy initialization.

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

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

```
    initialize
        height := 2.0

    height
        ^height
```

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

Ruby

The following 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.

```ruby
require 'net/http'
class Blogger
  def auth_token
    return @auth_token if @auth_token

    res = Net::HTTP.post_form(uri, params)
    @auth_token = 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
```

See also

- Proxy pattern
- Singleton pattern
- Double-checked locking

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)

Object pool

From Wikipedia, the free encyclopedia

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

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

However these benefits are mostly true for objects like database connections, socket connections, threads and large graphic objects like fonts or bitmaps. Simple object pooling (which hold no external resources, but only occupy memory) is not as efficient and may even decrease the performance [1] (http://www-128.ibm.com/developerworks/java/library/j-jtp09275.html?ca=dgr-jw22JavaUrbanLegends) .

Contents

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 is not necessary if the pooled objects are immutable or otherwise multithread - safe.

Criticism

Some publications do not recommend using object pooling, especially for objects that only use memory and hold no external resources [2] (http://www.ibm.com/developerworks/java/library/j-jtp11253/) . Opponents usually say that object allocation is relatively fast in modern languages with garbage collectors; while the operator "new" needs only 10 instructions, the classic "new" - "delete" pair found in pooling designs requires hundreds of them as it does more complex work. Also, most garbage collectors scan "live" object references, and not the memory that these objects use for their content. This means that any number of "dead" objects without references can be discarded with little cost. In contrast, keeping a large number of "live" but unused objects increases the duration of garbage collection [3] (http://www-128.ibm.com/developerworks/java/library/j-jtp09275.html?ca=dgr-jw22JavaUrbanLegends)]. In most cases, programs that use garbage collection instead of directly managing memory actually run faster [4] (http://www.cs.ubc.ca/local/reading/proceedings/spe91-95/spe/vol23/issue7/spe836.pdf) .

Examples

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

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

Finally, if this queue may get too long, it can be configured to suspend the requesting thread.

References

- Kircher, Michael; Prashant Jain; (2002-07-04). "Pooling Pattern". *EuroPLoP 2002*. Retrieved on 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/java/ent/article.php/626171)
- 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/)

Prototype pattern

From Wikipedia, the free encyclopedia

A **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-wired 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.

Contents

Examples

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. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

C#

```
/*
 * Base class for all prototypes
 */
public abstract class PrototypeBase : ICloneable {
    // cloning the prototype into a new instance
    public abstract object Clone();
    // some useful action
    public abstract void Action();
}// class PrototypeBase
```

```
/*
 * Place where prototypes are cloning
 */
public class PrototypeManager {
    Dictionary<string, PrototypeBase> prototypes = new Dictionary<string,PrototypeBase>();

    //
    // manage prototype list
    //

    public void AddPrototype(string name, PrototypeBase prototype) {
        prototypes[name] = prototype;
    }// AddPrototype()

    public void DelPrototype(string name) {
        prototypes.Remove(name);
    }// DelPrototype()

    public void DropAllPrototypes() {
        prototypes.Clear();
    }// DropAllPrototypes()

    //
    // main function of class
    //
    public PrototypeBase GetCopy(string name) {
        if (prototypes.ContainsKey(name))
            return (PrototypeBase)prototypes[name].Clone(); else
            return null;
    }// GetCopy()

}// class PrototypeManager
```

```
/*
 * Concrete Prototype First
 */
public class PrototypeFirst : PrototypeBase {
    public List<string> manyStrings = new List<string>();

    //
    // cloning the prototype into a new instance
    //
    public override object Clone() {
        // kind of a deep copy
        PrototypeFirst newObj = new PrototypeFirst();
        foreach(string str in manyStrings)
            newObj.manyStrings.Add( String.Copy(str) );
        return newObj;
    }// Clone()

    //
    // useful action
    //
    public override void Action() {
        foreach (string str in manyStrings)
            Console.WriteLine(str);
    }// Action()
```

```
}// class PrototypeFirst
```

```
/*
 * Concrete Prototype Second
 */
public class PrototypeSecond : PrototypeBase {
    public int value = 0;

    //
    // cloning the prototype into a new instance
    //
    public override object Clone() {
        PrototypeSecond newObj = (PrototypeSecond )this.MemberwiseClone();
        return newObj;
    }// Clone()

    //
    // useful action
    //
    public override void Action() {
        Console.WriteLine(value);
    }// Action()
}// class PrototypeSecond
```

```
/*
 * Usage example
 */
public class Usage {
    public void Foo() {
        PrototypeManager manager = new PrototypeManager();

        //
        // Filling prototype list
        //
        PrototypeFirst first = new PrototypeFirst();
        first.manyStrings.Add("String 1");
        first.manyStrings.Add("String 2");
        manager.AddPrototype("first", first);

        PrototypeSecond second = new PrototypeSecond();
        second.value = 10;
        manager.AddPrototype("second", second);

        //
        // Getting copy by prototype
        //
        PrototypeBase foo1 = manager.GetCopy("first");
        foo1.Action();

    }// Foo()
}// class Usage
```

C++

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

using namespace std;

enum RECORD_TYPE_en
{
  CAR,
  BIKE,
  PERSON
};
```

```
/**
 * Record is the Prototype
 */
class Record
{
  public :

    Record() {}

    virtual ~Record() {}

    virtual Record* Clone() const=0;

    virtual void Print() const=0;
};
```

```
/**
 * CarRecord is Concrete Prototype
 */
class CarRecord : public Record
{
  private :
    string m_oStrCarName;

    uint32_t m_ui32ID;

  public :

    CarRecord(const string& _oStrCarName, uint32_t _ui32ID)
      : Record(), m_oStrCarName(_oStrCarName),
        m_ui32ID(_ui32ID)
    {
    }

    CarRecord(const CarRecord& _oCarRecord)
      : Record()
    {
      m_oStrCarName = _oCarRecord.m_oStrCarName;
      m_ui32ID = _oCarRecord.m_ui32ID;
    }

    ~CarRecord() {}

    CarRecord* Clone() const
    {
      return new CarRecord(*this);
    }

    void Print() const
    {
      cout << "Car Record" << endl
        << "Name   : " << m_oStrCarName << endl
        << "Number: " << m_ui32ID << endl << endl;
    }
};
```

```
/**
 * BikeRecord is the Concrete Prototype
 */
class BikeRecord : public Record
{
  private :
    string m_oStrBikeName;

    uint32_t m_ui32ID;

  public :
    BikeRecord(const string& _oStrBikeName, uint32_t _ui32ID)
      : Record(), m_oStrBikeName(_oStrBikeName),
        m_ui32ID(_ui32ID)
    {
```

```
    }

    BikeRecord(const BikeRecord& _oBikeRecord)
      : Record()
    {
      m_oStrBikeName = _oBikeRecord.m_oStrBikeName;
      m_ui32ID = _oBikeRecord.m_ui32ID;
    }

    ~BikeRecord() {}

    BikeRecord* Clone() const
    {
      return new BikeRecord(*this);
    }

    void Print() const
    {
      cout << "Bike Record" << endl
        << "Name  : " << m_oStrBikeName << endl
        << "Number: " << m_ui32ID << endl << endl;
    }
};
```

```
/**
 * PersonRecord is the Concrete Prototype
 */
class PersonRecord : public Record
{
  private :
    string m_oStrPersonName;

    uint32_t m_ui32Age;

  public :
    PersonRecord(const string& _oStrPersonName, uint32_t _ui32Age)
      : Record(), m_oStrPersonName(_oStrPersonName),
        m_ui32Age(_ui32Age)
    {
    }

    PersonRecord(const PersonRecord& _oPersonRecord)
      : Record()
    {
      m_oStrPersonName = _oPersonRecord.m_oStrPersonName;
      m_ui32Age = _oPersonRecord.m_ui32Age;
    }

    ~PersonRecord() {}

    Record* Clone() const
    {
      return new PersonRecord(*this);
    }

    void Print() const
    {
      cout << "Person Record" << endl
        << "Name : " << m_oStrPersonName << endl
        << "Age  : " << m_ui32Age << endl << endl ;
    }
};
```

```
/**
 * RecordFactory is the client
 */
class RecordFactory
{
  private :
    map<RECORD_TYPE_en, Record* > m_oMapRecordReference;

  public :
```

```
    RecordFactory()
    {
      m_oMapRecordReference[CAR]    = new CarRecord("Ferrari", 5050);
      m_oMapRecordReference[BIKE]   = new BikeRecord("Yamaha", 2525);
      m_oMapRecordReference[PERSON] = new PersonRecord("Tom", 25);
    }

    ~RecordFactory()
    {
      delete m_oMapRecordReference[CAR];
      delete m_oMapRecordReference[BIKE];
      delete m_oMapRecordReference[PERSON];
    }

    Record* CreateRecord(RECORD_TYPE_en enType)
    {
      return m_oMapRecordReference[enType]->Clone();
    }
};
```

```
int main()
{
  RecordFactory* poRecordFactory = new RecordFactory();

  Record* poRecord;
  poRecord = poRecordFactory->CreateRecord(CAR);
  poRecord->Print();
  delete poRecord;

  poRecord = poRecordFactory->CreateRecord(BIKE);
  poRecord->Print();
  delete poRecord;

  poRecord = poRecordFactory->CreateRecord(PERSON);
  poRecord->Print();
  delete poRecord;

  delete poRecordFactory;
  return 0;
}
```

VB.net

```
Public Enum RecordType
    CAR
    PERSON
End Enum

''
'' Record is the Prototype
''
Public MustInherit Class Record
    Public MustOverride Function Clone() As Record
End Class

''
'' PersonRecord is the Concrete Prototype
''
Public Class PersonRecord : Inherits Record
    Private name As String
    Private age As Byte

    Public Overrides Function Clone() As Record
        Return CType(Me.MemberwiseClone(), Record)   ' default shallow copy
    End Function
End Class
```

```
''
'' CarRecord is another Concrete Prototype
```

```
''
Public Class CarRecord : Inherits Record
    Private carname As String
    Private id As Guid

    Public Overrides Function Clone() As Record
        Dim myClone As CarRecord = CType(Me.MemberwiseClone(), Record)   ' default shallow copy
        myClone.id = Guid.NewGuid()   ' always generate new id
        Return myClone
    End Function
End Class
```

```
''
'' RecordFactory is the client
''
Public Class RecordFactory
    Private Shared _prototypes As New Generic.Dictionary(Of RecordType, Record)

    '' Constructor
    Public Sub RecordFactory()
        _prototypes.Add(RecordType.CAR, New CarRecord())
        _prototypes.Add(RecordType.PERSON, New PersonRecord())
    End Sub

    '' The Factory method
    Public Function CreateRecord(ByVal type As RecordType) As Record
        Return _prototypes(type).Clone()
    End Function
End Class
```

Java

```
/**
 * Prototype Class
 */
public class Cookie implements Cloneable {

    @Override
    public Cookie clone() {
        Cookie copy;
        try {
            copy = (Cookie) super.clone();
        } catch (CloneNotSupportedException unexpected) {
            throw new AssertionError(unexpected);
        }

        //In an actual implementation of this pattern you might now change references to
        //the expensive to produce parts from the copies that are held inside the prototype.

        return copy;
    }
}
```

```
/**
 * Concrete Prototypes to clone
 */
public class CoconutCookie extends Cookie { }
```

```
/**
 * Client Class
 */
public class CookieMachine {

    private Cookie cookie; // Could have been a private Cloneable cookie.

    public CookieMachine(Cookie cookie) {
        this.cookie = cookie;
```

```
    }

    public Cookie makeCookie() {
        return (Cookie) cookie.clone();
    }

    public static void main(String args[]) {
        Cookie tempCookie = null;
        Cookie prot = new CoconutCookie();
        CookieMachine cm = new CookieMachine(prot);
        for (int i = 0; i < 100; i++)
            tempCookie = cm.makeCookie();
    }
}
```

Python

```
from copy import deepcopy

class PrototypeManager(object):
    def __init__(self):
        self._objs = {}

    def registerObject(self, name, obj):
        """
        register an object.
        """
        self._objs[name] = obj

    def unregisterObject(self, name):
        """unregister an object"""
        del self._objs[name]

    def clone(self, name, **attr):
        """clone a registered object and add/replace attr"""
        obj = deepcopy(self._objs[name])
        obj.__dict__.update(attr)
        return obj

## Usage
pm = PrototypeManager()

graphic = Graphic() # sample object 1 (expensive to create)
blueGraphic = Graphic() # sample object 2 (modified in some way)
blueGraphic.backgroundColor = "blue"

# Register the created objects
pm.registerObject("standard", graphic)
pm.registerObject("blue", blueGraphic)

# Create new copies of these without instantiation, and pre-customized as needed.
allBlueGraphics = []
for i in range(500):
    # create new, already customized (i.e. blue) graphic objects without instantiating Graphic()
    allBlueGraphics.append(pm.clone("blue"))

# create a new graphic object without instantiating Graphic()
anotherStandardGraphic = pm.clone("standard")
```

```
######## "Without a prototype manager:"
######## "create another instance, w/state, of an existing instance of unknown class/state"

from copy import deepcopy, copy

g = Graphic() # non-cooperative form
shallow_copy_of_g = copy(g)
deep_copy_of_g = deepcopy(g)

from copy import deepcopy, copy
```

```
class Graphic:
    def clone(self):
        return copy(self)
    g = Graphic() # cooperative form
    copy_of_g = g.clone()
```

Cloning in PHP

In PHP 5, unlike previous versions, objects are by default passed by reference. In order to pass by value, use the "magic function" __clone(). This makes the prototype pattern very easy to implement. See object cloning (http://us2.php.net/manual/en/language.oop5.cloning.php) for more information.

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

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

Singleton pattern

From Wikipedia, the free encyclopedia

In software engineering, the **singleton pattern** is a design pattern that is used to restrict instantiation of a class to one object. (This concept is also sometimes generalized to restrict the instance to a specific number of objects - for example, we can restrict the number of instances to five objects.) This is useful when exactly one object is needed to coordinate actions across the system. Sometimes it is generalized to systems that operate more efficiently when only one or a few objects exist. It is also considered an anti-pattern by some people, who feel that it is overused, introducing unnecessary limitations 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]

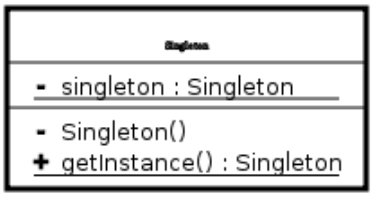
Contents

Common uses

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

will *always* consume resources.

Class diagram



Implementation

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

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

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

Example implementations

Scala

The Scala programming language supports Singleton objects out-of-the-box. The 'object' keyword creates a class and also defines a singleton object of that type. As the concept is implemented natively, the design pattern to implement the concept is not required in this language.

```
object Example extends ArrayList {
    // creates a singleton called Example
}
```

Java

The Java programming language solutions provided here are all thread-safe but differ in supported language versions and lazy-loading.

The solution of Bill Pugh

This is the recommended method. It is known as the initialization on demand holder idiom and is as

lazy as possible. Moreover, it works in all known versions of Java. This solution is the most portable across different Java compilers and virtual machines.

The inner 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 and/or synchronized).

```
public class Singleton {
  // Protected constructor is sufficient to suppress unauthorized calls to the constructor
  protected Singleton() {}

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

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

Traditional simple way

Just like the one above, this solution is thread-safe without requiring special language constructs, but it may lack the laziness. The INSTANCE is created as soon as the Singleton class is initialized (http://java.sun.com/docs/books/jls/third_edition/html/execution.html#12.4) . That might even be long before getInstance() is called. It might be (for example) when some static method of the class is used. If laziness is not needed or the instance needs to be created early in the application's execution, or your class has no other static members or methods that could prompt early initialization (and thus creation of the instance), this (slightly) simpler solution can be used:

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

  // Protected constructor is sufficient to suppress unauthorized calls to the constructor
  protected Singleton() {}
}
```

Sometimes the static final field is made private and a static factory-method is provided to get the instance. This way the underlying implementation may change easily while it has no more performance-issues on modern JVMs.

Java 5 solution

If and only if the compiler used is Java 5 (also known as Java 1.5) or newer, AND all Java virtual machines the application is going to run on fully support the Java 5 memory model, then (and only then) the volatile double checked locking can be used (for a detailed discussion of *why it should never be done before Java 5* see The "Double-Checked Locking is Broken" Declaration (http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html)):

```
public class Singleton {
  private static volatile Singleton INSTANCE;

  // Protected constructor is sufficient to suppress unauthorized calls to the constructor
  protected Singleton() {}

  private static synchronized Singleton tryCreateInstance() {
    if (INSTANCE == null) {
      INSTANCE = new Singleton();
    }
    return INSTANCE;
  }

  public static Singleton getInstance() {
     // use local variable, don't issue 2 reads (memory fences) to 'INSTANCE'
     Singleton s = INSTANCE;
```

```
      if (s == null) {
        //check under lock; move creation logic to a separate method to allow inlining of getInstanc
        s = tryCreateInstance();
      }
      return s;
    }
}
```

Allen Holub (in "Taming Java Threads", Berkeley, CA: Apress, 2000, pp. 176–178) notes that on multi-CPU systems (which are widespread as of 2009), the use of volatile may have an impact on performance approaching to that of synchronization, and raises the possibility of other problems. Thus this solution has little to recommend it over Pugh's solution described above.

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"[8] 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;
}
```

D

Singleton pattern in D programming language

```
import std.stdio;
import std.string;

class Singleton(T) {
    private static T instance;
    public static T opCall() {
        if(instance is null) {
            instance = new T;
        }
        return instance;
    }
}

class Foo {
    public this() {
        writefln("Foo constructor");
    }
}

void main(){
    Foo a = Singleton!(Foo)();
    Foo b = Singleton!(Foo)();
}
```

Or in this manner

```
// this class should be in a package to make private this() not visible
class Singleton {
    private static Singleton instance;

    public static Singleton opCall() {
        if(instance is null) {
            instance = new Singleton();
        }
        return instance;
    }

    private this() {
        writefln("Sigleton constructor");
    }
}

void main(){
    Singleton a = Singleton();
```

```
    Singleton b = Singleton();
}
```

PHP 5

Singleton pattern in PHP 5[9][10]:

```
<?php
class Singleton {

  // object instance
  private static $instance;

  // The protected construct prevents instantiating the class externally.  The construct can be
  // empty, or it can contain additional instructions...
  protected function __construct() {
    ...
  }

  // The clone and wakeup methods prevents external instantiation of copies of the Singleton class,
  // thus eliminating the possibility of duplicate objects.  The methods can be empty, or
  // can contain additional code (most probably generating error messages in response
  // to attempts to call).
  public function __clone() {
    trigger_error('Clone is not allowed.', E_USER_ERROR);
  }

  public function __wakeup() {
    trigger_error('Deserializing is not allowed.', E_USER_ERROR);
  }

  //This method must be static, and must return an instance of the object if the object
  //does not already exist.
  public static function getInstance() {
    if (!self::$instance instanceof self) {
      self::$instance = new self;
    }
    return self::$instance;
  }

  //One or more public methods that grant access to the Singleton object, and its private
  //methods and properties via accessor methods.
  public function doAction() {
    ...
  }
}

//usage
Singleton::getInstance()->doAction();

?>
```

Actionscript 3.0

Private constructors are not available in ActionScript 3.0 - which prevents the use of the ActionScript 2.0 approach to the Singleton Pattern. Many different AS3 Singleton implementations have been published around the web.

```
package {
        public class Singleton  {

                private static var _instance:Singleton = new Singleton();

                public function Singleton () {
                    if (_instance){
                        throw new Error(
                            "Singleton can only be accessed through Singleton.getInstance()"
                        );
                    }
                }

                public static function getInstance():Singleton {
                        return _instance;
                }

        }
}
```

Objective-C

A common way to implement a singleton in Objective-C is the following:

```
@interface MySingleton : NSObject
{
}

+ (MySingleton *)sharedSingleton;
@end

@implementation MySingleton

+ (MySingleton *)sharedSingleton
{
  static MySingleton *sharedSingleton;

  @synchronized(self)
  {
    if (!sharedSingleton)
      sharedSingleton = [[MySingleton alloc] init];

    return sharedSingleton;
  }
}

@end
```

If thread-safety is not required, the synchronization can be left out, leaving the `+sharedSingleton` method like this:

```
+ (MySingleton *)sharedSingleton
{
  static MySingleton *sharedSingleton;

  if (!sharedSingleton)
    sharedSingleton = [[MySingleton alloc] init];

  return sharedSingleton;
}
```

This pattern is widely used in the Cocoa frameworks (see for instance, `NSApplication`, `NSColorPanel`, `NSFontPanel` or `NSWorkspace`, to name but a few).

Some may argue that this is not, strictly speaking, a Singleton, because it is possible to allocate more than one instance of the object. A common way around this is to use assertions or exceptions to prevent this double allocation.

```
@interface MySingleton : NSObject
{
}

+ (MySingleton *)sharedSingleton;
@end

@implementation MySingleton

static MySingleton *sharedSingleton;

+ (MySingleton *)sharedSingleton
{
  @synchronized(self)
  {
    if (!sharedSingleton)
      [[MySingleton alloc] init];

    return sharedSingleton;
  }
}

+(id)alloc
{
  @synchronized(self)
  {
    NSAssert(sharedSingleton == nil, @"Attempted to allocate a second instance of a singleton.");
```

```
    sharedSingleton = [super alloc];
    return sharedSingleton;
  }
}

@end
```

There are alternative ways to express the Singleton pattern in Objective-C, but they are not always as simple or as easily understood, not least because they may rely on the `-init` method returning an object other than `self`. Some of the Cocoa "Class Clusters" (e.g. `NSString`, `NSNumber`) are known to exhibit this type of behaviour.

Note that `@synchronized` is not available in some Objective-C configurations, as it relies on the NeXT/Apple runtime. It is also comparatively slow, because it has to look up the lock based on the object in parentheses. Check the history of this page for a different implementation using an `NSConditionLock`.

C++

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

/* Place holder for thread synchronization mutex */
class Mutex
{   /* placeholder for code to create, use, and free a mutex */
};

/* Place holder for thread synchronization lock */
class Lock
{   public:
        Lock(Mutex& m) : mutex(m) { /* placeholder code to acquire the mutex */ }
        ~Lock() { /* placeholder code to release the mutex */ }
    private:
        Mutex & mutex;
};

class Singleton
{   public:
        static Singleton* GetInstance();
        int a;
        ~Singleton() { cout << "In Dtor" << endl; }

    private:
        Singleton(int _a) : a(_a) { cout << "In Ctor" << endl; }


        static Mutex mutex;

        // Not defined, to prevent copying
        Singleton(const Singleton& );
        Singleton& operator =(const Singleton& other);
};

Mutex Singleton::mutex;

Singleton* Singleton::GetInstance()
{
    Lock lock(mutex);

    cout << "Get Inst" << endl;

    // Initialized during first access
    static Singleton inst(1);

    return &inst;
}

int main()
{
    Singleton* singleton = Singleton::GetInstance();
    cout << "The value of the singleton: " << singleton->a << endl;
    return 0;
}
```

In the above example, the first call to `Singleton::GetInstance` will initialize the singleton instance.

This example is for illustrative purposes only; for anything but a trivial example program, this code contains errors.

C#

The simplest of all is:

```
public class Singleton
{
    // The combination of static and readonly makes the instantiation
    // thread safe.  Plus the constructor being protected (it can be
    // private as well), makes the class sure to not have any other
    // way to instantiate this class than using this member variable.
    public static readonly Singleton Instance = new Singleton();

    // Protected constructor is sufficient to avoid other instantiation
    // This must be present otherwise the compiler provides a default
    // public constructor
    //
    protected Singleton()
    {
    }
}
```

This example is thread-safe with lazy initialization. Note that the explicit static constructor which disables beforefieldinit. See http://www.yoda.arachsys.com/csharp/beforefieldinit.html

```
/// Class implements singleton pattern.

public class Singleton
{
        // Protected constructor is sufficient to avoid other instantiation
        // This must be present otherwise the compiler provides
        // a default public constructor
        protected Singleton()
        {
        }


        /// Return an instance of <see cref="Singleton"/>

        public static Singleton Instance
        {
            get
            {
                /// An instance of Singleton wont be created until the very first
                /// call to the sealed class. This is a CLR optimization that
                /// provides a properly lazy-loading singleton.
                return SingletonCreator.CreatorInstance;
            }
        }


        /// Sealed class to avoid any heritage from this helper class

        private sealed class SingletonCreator
        {
          // Retrieve a single instance of a Singleton
          private static readonly Singleton _instance = new Singleton();

          //explicit static constructor to disable beforefieldinit
          static CreatorInstance() { }

          /// Return an instance of the class <see cref="Singleton"/>

          public static Singleton CreatorInstance
          {
            get { return _instance; }
          }
        }
}
```

Example in C# 2.0 (thread-safe with lazy initialization) Note: This is not a recommended implementation because "TestClass" has a default public constructor, and that violates the definition of a Singleton. A proper Singleton must never be instantiable more than once. More about generic singleton solution in C#: http://www.c-sharpcorner.com/UploadFile/snorrebaard/GenericSingleton11172008110419AM/GenericSingleton.aspx

```
/// Parent for singleton

/// <typeparam name="T">Singleton class</typeparam>
  public class Singleton<T> where T : class, new()
  {

    protected Singleton() { }

    private sealed class SingletonCreator<S> where S : class, new()
    {
      private static readonly S instance = new S();

          //explicit static constructor to disable beforefieldinit
          static CreatorInstance() { }

      public static S CreatorInstance
      {
        get { return instance; }
      }
    }
```

```
    public static T Instance
    {
      get { return SingletonCreator<T>.CreatorInstance; }
    }

  }


/// Concrete Singleton

public class TestClass : Singleton<TestClass>
{
    public string TestProc()
    {
        return "Hello World";
    }
}

// Somewhere in the code
.....
TestClass.Instance.TestProc();
.....
```

Delphi

As described by James Heyworth in a paper[11] presented to the Canberra PC Users Group Delphi SIG on 11/11/1996, there are several examples of the Singleton pattern built into the Delphi Visual Component Library. This unit demonstrates the techniques that were used in order to create both a Singleton component and a Singleton object:

```
unit Singletn;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type

  TCSingleton = class(TComponent)
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  end;

  TOSingleton = class(TObject)
  public
    constructor Create;
    destructor Destroy; override;
  end;

var
  Global_CSingleton: TCSingleton;
  Global_OSingleton: TOSingleton;

procedure Register;

implementation
```

```
procedure Register;
begin
  RegisterComponents('Design Patterns', [TCSingleton]);
end;

{ TCSingleton }

constructor TCSingleton.Create(AOwner: TComponent);
begin
  if Global_CSingleton <> nil then
    {NB could show a message or raise a different exception here}
    Abort
  else begin
    inherited Create(AOwner);
    Global_CSingleton := Self;
  end;
end;

destructor TCSingleton.Destroy;
begin
  if Global_CSingleton = Self then
    Global_CSingleton := nil;
  inherited Destroy;
end;


{ TOSingleton }

constructor TOSingleton.Create;
begin
if Global_OSingleton <> nil then
    {NB could show a message or raise a different exception here}
    Abort
  else
    Global_OSingleton := Self;
end;

destructor TOSingleton.Destroy;
begin
  if Global_OSingleton = Self then
    Global_OSingleton := nil;
  inherited Destroy;
end;

procedure FreeGlobalObjects; far;
begin
  if Global_CSingleton <> nil then
    Global_CSingleton.Free;
  if Global_OSingleton <> nil then
    Global_OSingleton.Free;
end;

begin
  AddExitProc(FreeGlobalObjects);
end.
```

Python

The desired properties of the Singleton pattern can most simply be encapsulated in Python by defining a module, containing module-level variables and functions. To use this modular Singleton, client code merely imports the module to access its attributes and functions in the normal manner. This sidesteps many of the wrinkles in the explicitly-coded versions below, and has the singular advantage of requiring zero lines of code to implement.

According to influential Python programmer Alex Martelli, *The Singleton design pattern (DP) has a catchy name, but the wrong focus—on identity rather than on state. The Borg design pattern has all instances share state instead*.[12] A rough consensus in the Python community is that sharing state among instances is more elegant, at least in Python, than is caching creation of identical instances on class initialization. Coding shared state is nearly transparent:

```
class Borg:
   __shared_state = {}
   def __init__(self):
       self.__dict__ = self.__shared_state
   # and whatever else is needed in the class -- that's all!
```

But with the new style class, this is a better solution, because only one instance is created:

```
class Singleton (object):
    def __new__(cls, *args, **kwargs):
        if not hasattr(cls, 'self'):
            cls.self = object.__new__(cls)
        return cls.self

#Usage
mySingleton1 = Singleton()
mySingleton2 = Singleton()

#mySingleton1 and  mySingleton2 are the same instance.
assert mySingleton1 is mySingleton2
```

Two caveats:

- The *__init__*-method is called every time *Singleton()* is called, unless *cls.__init__* is set to an empty function.
- If it is needed to inherit from the *Singleton*-class, *instance* should probably be a *dictionary* belonging explicitly to the *Singleton*-class.

```
class  InheritableSingleton (object):
    instances = {}
    def __new__(cls, *args, **kwargs):
        if InheritableSingleton.instances.get(cls) is None:
            cls.__original_init__ = cls.__init__
            InheritableSingleton.instances[cls] = object.__new__(cls, *args, **kwargs)
        elif cls.__init__ == cls.__original_init__:
            def nothing(*args, **kwargs):
                pass
            cls.__init__ = nothing
        return InheritableSingleton.instances[cls]
```

To create a singleton that inherits from a non-singleton, multiple inheritance must be used.

```
class  Singleton (NonSingletonClass, object):
    instance = None
    def __new__(cls, *args, **kargs):
        if cls.instance is None:
            cls.instance = object.__new__(cls, *args, **kargs)
        return cls.instance
```

Be sure to call the NonSingletonClass's __init__ function from the Singleton's __init__ function.

A more elegant approach using metaclasses was also suggested.[13]

```
class SingletonType(type):
    def __call__(cls):
        if getattr(cls, '__instance__', None) is None:
            instance = cls.__new__(cls)
            instance.__init__()
            cls.__instance__ = instance
        return cls.__instance__

# Usage
class Singleton(object):
    __metaclass__ = SingletonType

    def __init__(self):
        print '__init__:', self

class OtherSingleton(object):
    __metaclass__ = SingletonType

    def __init__(self):
        print 'OtherSingleton __init__:', self

# Tests
s1 = Singleton()
s2 = Singleton()
assert s1
assert s2
assert s1 is s2

os1 = OtherSingleton()
os2 = OtherSingleton()
assert os1
assert os2
```

```
assert os1 is os2
```

Perl

In a Perl version equal or superior to 5.10 a state variable can be used.

```
package MySingletonClass;
use strict;
use warnings;
use 5.10;

sub new {
    my ($class) = @_;
    state $the_instance;

    if (! defined $the_instance) {
        $the_instance = bless { }, $class;
    }
    return $the_instance;
}
```

In older Perls, just use a closure.

```
package MySingletonClass;
use strict;
use warnings;

my $THE_INSTANCE;
sub new {
    my ($class) = @_;

    if (! defined $THE_INSTANCE) {
        $THE_INSTANCE = bless { }, $class;
    }
    return $THE_INSTANCE;
}
```

If Moose is used, there is the MooseX::Singleton (http://search.cpan.org/perldoc?MooseX::Singleton) extension module.

Ruby

In Ruby, just include the Singleton in the class.

```
require 'singleton'

class Example
  include Singleton
end
```

ABAP Objects

In ABAP Objects, to make instantiation private, add an attribute of type ref to the class, and a static method to control instantiation.

```
program pattern_singleton.

*---------------------------------------------------------------------*

   * Singleton
   * =========
   * Intent

*

   * Ensure a class has only one instance, and provide a global point
   * of access to it.

*---------------------------------------------------------------------*

class lcl_Singleton definition create private.
```

```
  public section.

  class-methods:
    get_Instance returning value(Result) type ref to lcl_Singleton.

  private section.
    class-data:
      fg_Singleton type ref to lcl_Singleton.

endclass.

class lcl_Singleton implementation.

  method get_Instance.
    if ( fg_Singleton is initial ).
      create object fg_Singleton.
    endif.
    Result = fg_Singleton.
  endmethod.

endclass.
```

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 (http://java.sun.com/javase/6/docs/api/java/awt/Toolkit.html)` is an abstract class that binds the various AWT components to particular native toolkit implementations. The `Toolkit` class has a `Toolkit.getDefaultToolkit()` `(http://java.sun.com/javase/6/docs/api/java/awt/Toolkit.html#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()` `(http://java.sun.com/javase/6/docs/api/java/lang/System.html#getProperties())` .

The binding performed by the toolkit allows, for example, the backing implementation of a `java.awt.Window (http://java.sun.com/javase/6/docs/api/java/awt/Window.html)` to bound 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

It should be noted that this pattern makes unit testing far more difficult[14], as it introduces Global state into an application.

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

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. ^ Joshua Bloch: Effective Java 2nd edition, ISBN 978-0-321-35668-0, 2008, p. 18
9. ^ McArthur, Kevin: "Pro PHP: Patterns, Frameworks, Testing and More", pp 22–23. Apress, 2008
10. ^ Zandstra, Matt: "PHP Objects, Patterns and Practice", pp 147–149. Apress, 2008
11. ^ Heyworth, James (1996-11-11). "Introduction to Design Patterns in Delphi". *Canberra PC Users Group Delphi SIG* (Objective Software Technology). http://www.obsof.com/delphi_tips/pattern.html#Singleton. Retrieved on 2008-01-19.
12. ^ Alex Martelli. "Singleton? We don't need no stinkin' singleton: the Borg design pattern". *ASPN Python Cookbook*. http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/66531. Retrieved on 2006-09-07.
13. ^ Timur Izhbulatov. "Singleton in Python". *timka.org - Programming*. http://timka.org/programming/2008/12/17/singleton-in-python/. Retrieved on 2009-01-05.
14. ^ http://googletesting.blogspot.com/2008/11/clean-code-talks-global-state-and.html

- "C++ and the Perils of Double-Checked Locking" (http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf#search=%22meyers%20double%20checked%20locking%22) 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

- Singletons are Pathological Liars (http://misko.hevery.com/2008/08/17/singletons-are-pathological-liars/) by Miško Hevery
- Java Singleton Design Pattern (http://java4all.info/designpattern/interview-questions/design-pattern-interview-questions.html)
- The "Double-Checked Locking is Broken" Declaration (http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html) (Java)
- Java Singleton Pattern (http://www.javabeginner.com/singleton.htm)
- A Pattern Enforcing Compiler (http://pec.dev.java.net/) that enforces the Singleton pattern amongst other patterns
- Description from the Portland Pattern Repository (http://c2.com/cgi/wiki?SingletonPattern)
- Implementing the Singleton Pattern in C# (http://www.yoda.arachsys.com/csharp/singleton.html) by Jon Skeet
- A Threadsafe C++ Template Singleton Pattern for Windows Platforms (http://www.opbarnes.com/blog/Programming/OPB/Snippets/Singleton.html) by O. Patrick Barnes
- Implementing the Inheritable Singleton Pattern in PHP5 (http://svn.shadanakar.org/filedetails.php?repname=onPHP&path=%2Ftrunk%2Fcore%2FBase%2FSingleton.class.php&rev=0&sc=0)
- Singleton Pattern and Thread Safety (http://www.oaklib.org/docs/oak/singleton.html)
- PHP patterns (http://www.php.net/manual/en/language.oop5.patterns.php)
- Javascript implementation of a Singleton Pattern (http://prototyp.ical.ly/index.php/2007/03/01/javascript-design-patterns-1-the-singleton/) by Christian Schaefer
- Singletons Cause Cancer (http://www.prestonlee.com/archives/22) by Preston Lee
- Singleton examples (http://www.oodesign.com/oo_design_patterns/creational_patterns/singleton.html)

- Article "Double-checked locking and the Singleton pattern (http://www-128.ibm.com/developerworks/java/library/j-dcl.html?loc=j) " by Peter Haggar
- Article "Use your singletons wisely (http://www-106.ibm.com/developerworks/library/co-single.html) " by J. B. Rainsberger
- Article "Simply Singleton (http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns.html) " by David Geary
- Article "Description of Singleton (http://www.dofactory.com/Patterns/PatternSingleton.aspx) " by Aruna
- Article "Why Singletons Are Controversial (http://code.google.com/p/google-singleton-detector/wiki/WhySingletonsAreControversial) "
- The Google Singleton Detector (http://code.google.com/p/google-singleton-detector/) analyzes Java bytecode to detect singletons, so that their usefulness can be evaluated.
- Jt (http://www.fsw.com/Jt/Jt.htm) J2EE Pattern Oriented Framework
- Serialization of Singleton in Java (http://www.exampledepot.com/egs/java.io/DeserSingle.html?l=rel)
- Singleton at Microsoft patterns & practices Developer Center (http://msdn.microsoft.com/en-us/library/ms998426.aspx)
- Using a Singleton variation (Registry) with FlashVars in actionscript 3 (http://www.dpdk.nl/opensource/using-a-flashvars-flash-parameters-registry-configure-your-flash-files-externally)
- Singleton Pattern in Cairngorm 2.1 with Actionscript 3 (http://tomschober.blogspot.com/2007/01/singleton-pattern-in-cairngorm-21-with.html)
- More about Generic Singleton in C#: [1] (http://www.c-sharpcorner.com/UploadFile/snorrebaard/GenericSingleton11172008110419AM/GenericSingleton.aspx)

Structural pattern

From Wikipedia, the free encyclopedia

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
- 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
- Composite pattern: a tree structure of objects where every object has the same interface
- Decorator pattern: add additional functionality to a class at runtime where subclassing would result in an exponential rise of new classes
- Extensibility pattern: aka. Framework - hide complex code behind a simple interface
- Facade pattern: create a simplified interface of an existing interface to ease usage for common tasks
- Flyweight pattern: a high quantity of objects share a common properties object to save space
- Proxy pattern: a class functioning as an interface to another thing
- Pipes and filters: a chain of processes where the output of each process is the input of the next
- Private class data pattern: restrict accessor/mutator access

See also

- Behavioral pattern
- Creational pattern
- Concurrency pattern

Adapter pattern

From Wikipedia, the free encyclopedia

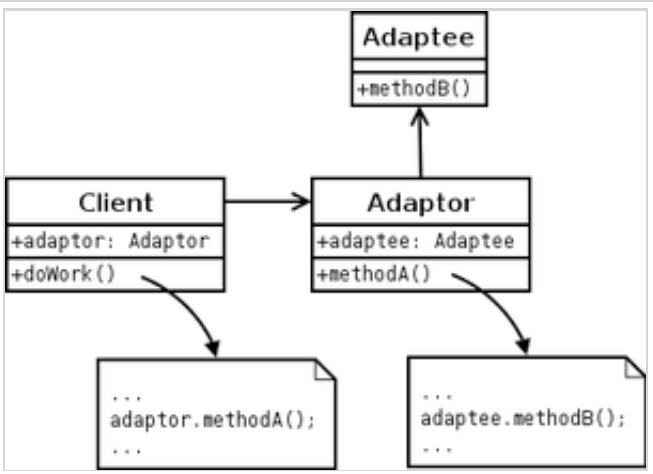
In computer programming, the **adapter design pattern** (often referred to as the **wrapper pattern** or simply a **wrapper**) translates one interface for a class into a compatible interface. An *adapter* allows classes to work together that normally could not because of incompatible interfaces, by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

Contents

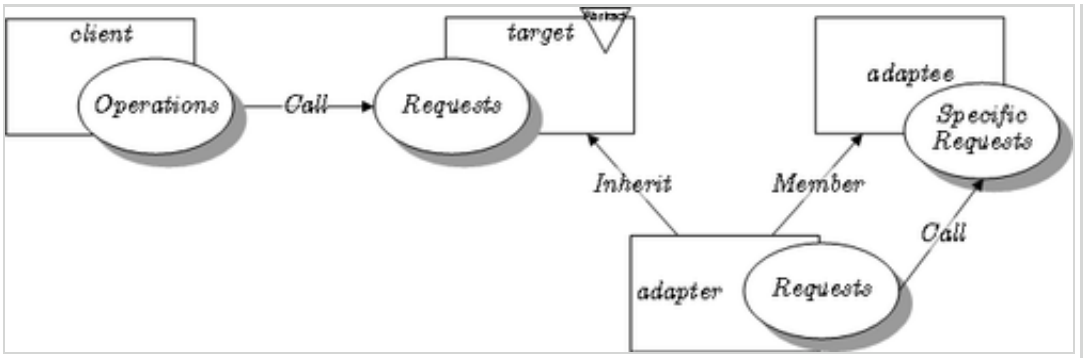
Structure

There are two types of adapter patterns:

Object 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.



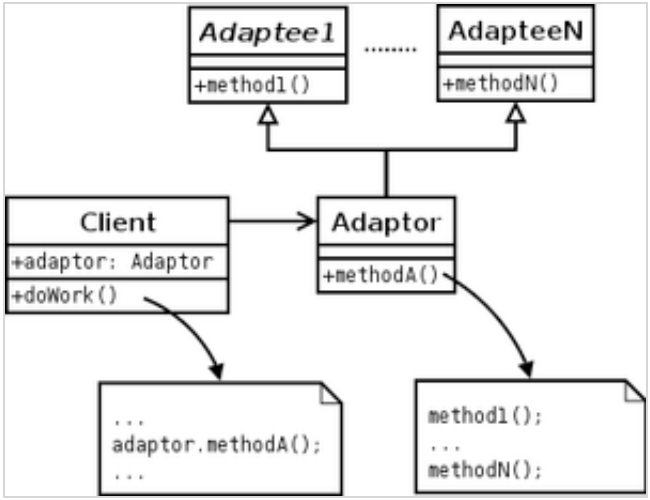
The object adapter pattern expressed in UML. The adapter *hides* the adaptee's interface from the client.



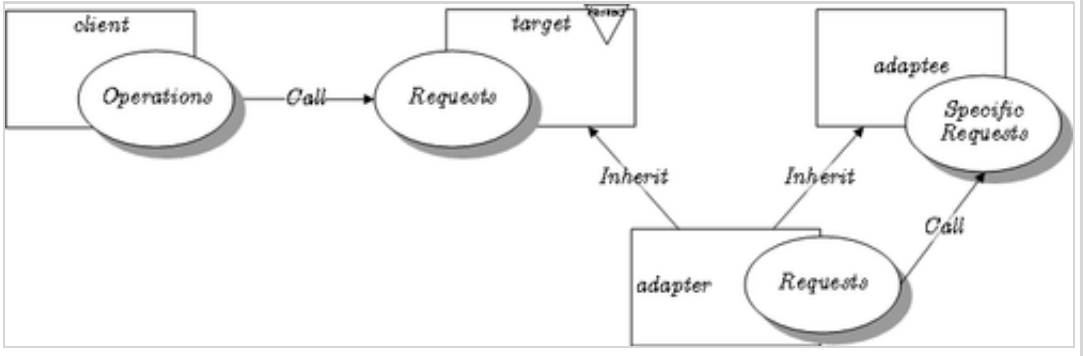
The object adapter pattern expressed in LePUS3.

Class Adapter pattern

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



The 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.

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

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

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

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

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

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

A solution using "adapters" proceeds as follows:

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

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

public class ClassAFormat1 implements StringProvider {
 ClassA classA;
 public ClassAFormat1(ClassA classA) {
 this.classA = classA;
  }
 public String getStringData() {
  return format(classA.toString());
}
}
```

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

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

 public boolean isAdapterFor(Class c) {

  return c.equals(StringProvider.class);
 }

}
```

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

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

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

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

StringProvider provider = (StringProvider) adapter.adapt(classA);

String string = provider.getStringData();

classB.setStringData(string);
```

or more concisely:

```
classB.setStringData((StringProvider) AdapterFactory.getInstance().getAdapterFromTo(ClassA.class, St
```

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

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

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

```
Adapter adapter = AdapterFactory.getInstance().getAdapterFromTo(ClassA.class, ImageProvider.class, '

ImageProvider provider = (ImageProvider) adapter.adapt(classA);

classC.setImage(provider.getImage());
```

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

Sample - Object Adapter

```
# Python code sample

class Target:
    def request(self):
        pass

class Adaptee:
    def specific_request(self):
        return 'Hello Adapter Pattern!'

class Adapter(Target):
    def __init__(self, adaptee):
        self.adaptee = adaptee

    def request(self):
        return self.adaptee.specific_request()

client = Adapter(Adaptee())
print client.request()
```

Sample - Class Adapter

```
/**
 * Java code sample
 */

/* the client class should instantiate adapter objects */
/* by using a reference of this type */
interface Stack<T>
{
  void push (T o);
  T pop ();
  T top ();
}

/* DoubleLinkedList is the adaptee class */
class DList<T>
{
  public void insert (DNode pos, T o) { ... }
  public void remove (DNode pos) { ... }

  public void insertHead (T o) { ... }
  public void insertTail (T o) { ... }

  public T removeHead () { ... }
  public T removeTail () { ... }

  public T getHead () { ... }
  public T getTail () { ... }
}

/* Adapt DList class to Stack interface is the adapter class */
class DListImpStack<T> extends DList<T> implements Stack<T>
{
  public void push (T o) {
    insertTail (o);
  }

  public T pop () {
    return removeTail ();
  }

  public T top () {
    return getTail ();
  }
}
```

External links

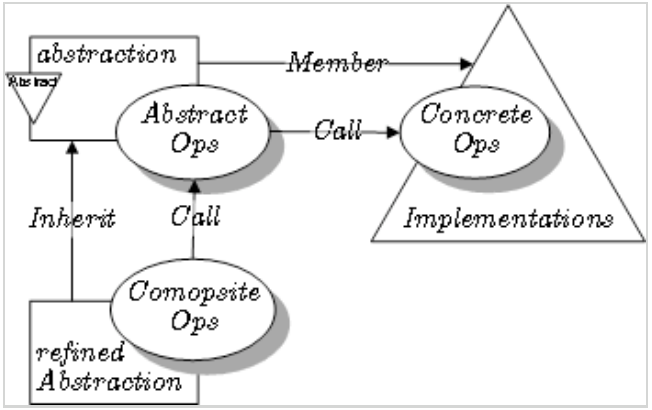
- Adapter in UML and in LePUS3 (a formal modelling language) (http://www.lepus.org.uk/ref/companion/Adapter(Class).xml)
- Description in Portland Pattern Repository's Wiki (http://www.c2.com/cgi/wiki?AdapterPattern)
- Java Tutorial on the Document Object Model (http://java.sun.com/webservices/jaxp/dist/1.1/docs/tutorial/dom/) (uses the adapter pattern)
- Citations from CiteSeer (http://citeseer.org/cs?q=Adapter+pattern)
- Jt (http://www.fsw.com/Jt/Jt.htm) J2EE Pattern Oriented Framework
- Adapter Design Pattern in C# and VB.NET (http://www.dofactory.com/Patterns/PatternAdapter.aspx)
- A generic implementation of The Adapter Pattern in C++ (http://www.ddj.com/architect/199204099)

- PerfectJPattern Open Source Project (http://perfectjpattern.sourceforge.net/dp-adapter.html) , Provides componentized implementation of the Adapter Pattern in Java

Retrieved from "http://en.wikipedia.org/wiki/Adapter_pattern"
Categories: Software design patterns | Articles with example Java code

Bridge pattern

From Wikipedia, the free encyclopedia



Bridge in LePUS3 (legend (http://lepus.org.uk/ref/legend/legend.xml))

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 not only the class itself varies often, but the class does also. The class itself can be thought of as the *implementation* and what the class can do as the *abstraction*.

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

Contents

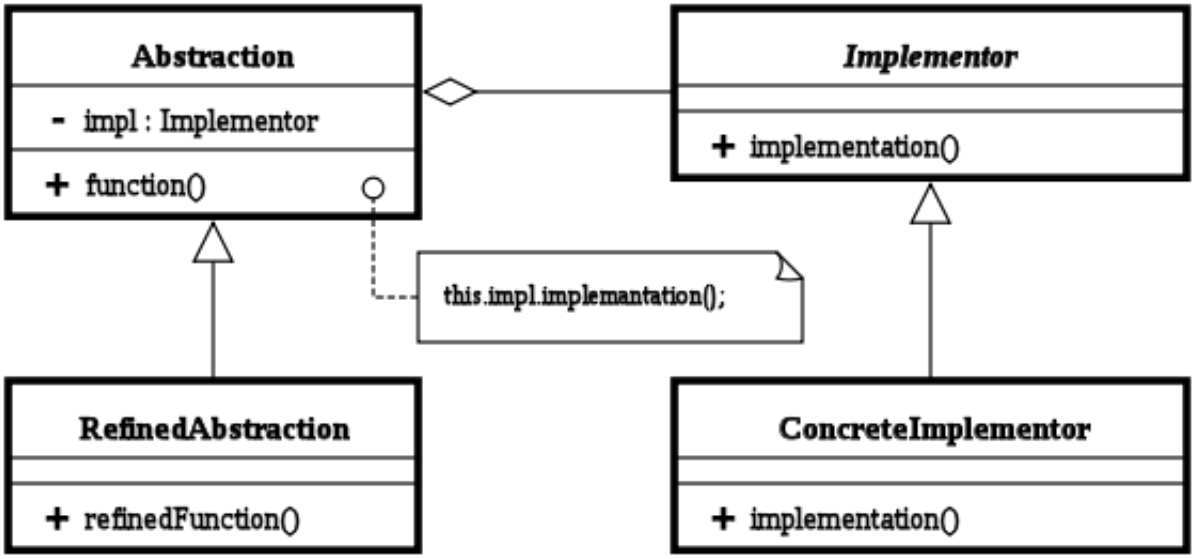
Examples

Shape abstraction

When the abstraction and implementation are separated, they can vary independently. Consider the abstraction of shapes. There are many types of shapes, each with its own properties. And there are

things that all shapes do. One thing all shapes can do is draw themselves. However, drawing graphics to a screen can sometimes be dependent on different graphics implementations or operating systems. Shapes have to be able to be drawn on many types of operating systems. Having the shape itself implement them all, or modifying the shape class to work with different architectures is not practical. The bridge helps by allowing the creation of new implementation classes that provide the drawing implementation. The abstraction class, shape, provides methods for getting the size or properties of a shape. The implementation class, drawing, provides an interface for drawing graphics. If a new shape needs to be created or there is a new graphics API to be drawn on, then it is very easy to add a new implementation class that implements the needed features.[2]

Structure



Abstraction
: defines the abstract interface
: maintains the Implementor reference

Refined Abstraction
: extends the interface defined by Abstraction

Implementor
: defines the interface for implementation classes

ConcreteImplementor
: implements the Implementor interface

Code examples

Java

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

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

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

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

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

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

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

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

/** "Client" */
class BridgePattern {
   public static void main(String[] args) {
       Shape[] shapes = new Shape[2];
       shapes[0] = new CircleShape(1, 2, 3, new DrawingAPI1());
       shapes[1] = new CircleShape(5, 7, 11, new DrawingAPI2());

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

C#

The following C# program illustrates the "shape" example given above and will output:

```
API1.circle at 1:2 radius 7.5
API2.circle at 5:7 radius 27.5
```

```
using System;
```

```
/** "Implementor" */
interface IDrawingAPI {
   void DrawCircle(double x, double y, double radius);
}

/** "ConcreteImplementor" 1/2 */
class DrawingAPI1 : IDrawingAPI {
   public void DrawCircle(double x, double y, double radius)
   {
       System.Console.WriteLine("API1.circle at {0}:{1} radius {2}", x, y, radius);
   }
}

/** "ConcreteImplementor" 2/2 */
class DrawingAPI2 : IDrawingAPI
{
   public void DrawCircle(double x, double y, double radius)
   {
       System.Console.WriteLine("API2.circle at {0}:{1} radius {2}", x, y, radius);
   }
}

/** "Abstraction" */
interface IShape {
   void Draw();                              // low-level (i.e. Implementation-specific)
   void ResizeByPercentage(double pct);      // high-level (i.e. Abstraction-specific)
}

/** "Refined Abstraction" */
class CircleShape : IShape {
   private double x, y, radius;
   private IDrawingAPI drawingAPI;
   public CircleShape(double x, double y, double radius, IDrawingAPI drawingAPI)
   {
       this.x = x;  this.y = y;  this.radius = radius;
       this.drawingAPI = drawingAPI;
   }
   // low-level (i.e. Implementation-specific)
   public void Draw() { drawingAPI.DrawCircle(x, y, radius); }
   // high-level (i.e. Abstraction-specific)
   public void ResizeByPercentage(double pct) { radius *= pct; }
}

/** "Client" */
class BridgePattern {
   public static void Main(string[] args) {
       IShape[] shapes = new IShape[2];
       shapes[0] = new CircleShape(1, 2, 3, new DrawingAPI1());
       shapes[1] = new CircleShape(5, 7, 11, new DrawingAPI2());

       foreach (IShape shape in shapes) {
           shape.ResizeByPercentage(2.5);
           shape.Draw();
       }
   }
}
```

C# using generics

The following C# program illustrates the "shape" example given above and will output:

```
API1.circle at 1:2 radius 7.5
API2.circle at 5:7 radius 27.5
```

```
using System;

/** "Implementor" */
interface IDrawingAPI {
   void DrawCircle(double x, double y, double radius);
```

```
}

/** "ConcreteImplementor" 1/2 */
struct DrawingAPI1 : IDrawingAPI {
    public void DrawCircle(double x, double y, double radius)
    {
        System.Console.WriteLine("API1.circle at {0}:{1} radius {2}", x, y, radius);
    }
}

/** "ConcreteImplementor" 2/2 */
struct DrawingAPI2 : IDrawingAPI
{
    public void DrawCircle(double x, double y, double radius)
    {
        System.Console.WriteLine("API2.circle at {0}:{1} radius {2}", x, y, radius);
    }
}

/** "Abstraction" */
interface IShape {
    void Draw();                                // low-level (i.e. Implementation-specific)
    void ResizeByPercentage(double pct);     // high-level (i.e. Abstraction-specific)
}

/** "Refined Abstraction" */
class CircleShape<T> : IShape
    where T : struct, IDrawingAPI
{
    private double x, y, radius;
    private IDrawingAPI drawingAPI = new T();
    public CircleShape(double x, double y, double radius)
    {
        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) {
        IShape[] shapes = new IShape[2];
        shapes[0] = new CircleShape<DrawingAPI1>(1, 2, 3);
        shapes[1] = new CircleShape<DrawingAPI2>(5, 7, 11);

        foreach (IShape shape in shapes) {
            shape.ResizeByPercentage(2.5);
            shape.Draw();
        }
    }
}
```

C++

The following C++ program illustrates the "shape" example given above and will output:

```
API1.circle at 1:2 7.5
API2.circle at 5:7 27.5
```

```
#include <iostream>

using namespace std;

/* Implementor*/
class DrawingAPI {
  public:
```

```
   virtual void drawCircle(double x, double y, double radius) = 0;
   virtual ~DrawingAPI() {}
};

/* Concrete ImplementorA*/
class DrawingAPI1 : public DrawingAPI {
  public:
   void drawCircle(double x, double y, double radius) {
      cout << "API1.circle at " << x << ':' << y << ' ' << radius << endl;
   }
};

/* Concrete ImplementorB*/
class DrawingAPI2 : public DrawingAPI {
public:
   void drawCircle(double x, double y, double radius) {
      cout << "API2.circle at " << x << ':' << y << ' ' <<  radius << endl;
   }
};

/* Abstraction*/
class Shape {
  public:
   virtual ~Shape() {}
   virtual void draw() = 0;
   virtual void resizeByPercentage(double pct) = 0;
};

/* Refined Abstraction*/
class CircleShape : public Shape {
  public:
   CircleShape(double x, double y,double radius, DrawingAPI *drawingAPI) :
           m_x(x), m_y(y), m_radius(radius), m_drawingAPI(drawingAPI)
   {}
   void draw() {
      m_drawingAPI->drawCircle(m_x, m_y, m_radius);
   }
   void resizeByPercentage(double pct) {
      m_radius *= pct;
   }
  private:
   double m_x, m_y, m_radius;
   DrawingAPI *m_drawingAPI;
};

int main(void) {
   CircleShape circle1(1,2,3,new DrawingAPI1());
   CircleShape circle2(5,7,11,new DrawingAPI2());
   circle1.resizeByPercentage(2.5);
   circle2.resizeByPercentage(2.5);
   circle1.draw();
   circle2.draw();
   return 0;
}
```

Python

The following Python program illustrates the "shape" example given above and will output:

```
API1.circle at 1:2 7.5
API2.circle at 5:7 27.5
```

```
# Implementor
class DrawingAPI:
        def drawCircle(x, y, radius):
                pass

# ConcreteImplementor 1/2
class DrawingAPI1(DrawingAPI):
```

```
        def drawCircle(self, x, y, radius):
                print "API1.circle at %f:%f radius %f" % (x, y, radius)

# ConcreteImplementor 2/2
class DrawingAPI2(DrawingAPI):
        def drawCircle(self, x, y, radius):
                print "API2.circle at %f:%f radius %f" % (x, y, radius)

# Abstraction
class Shape:
        # low-level
        def draw(self):
                pass

        # high-level
        def resizeByPercentage(self, pct):
                pass

# Refined Abstraction
class CircleShape(Shape):
        def __init__(self, x, y, radius, drawingAPI):
                self.__x = x
                self.__y = y
                self.__radius = radius
                self.__drawingAPI = drawingAPI

        # low-level i.e. Implementation specific
        def draw(self):
                self.__drawingAPI.drawCircle(self.__x, self.__y, self.__radius)

        # high-level i.e. Abstraction specific
        def resizeByPercentage(self, pct):
                self.__radius *= pct

def main():
        shapes = [
                CircleShape(1, 2, 3, DrawingAPI1()),
                CircleShape(5, 7, 11, DrawingAPI2())
        ]

        for shape in shapes:
                shape.resizeByPercentage(2.5)
                shape.draw()

if __name__ == "__main__":
        main()
```

See also

- Template method pattern
- Strategy pattern

References

1. ^ Gamma, E, Helm, R, Johnson, R, Vlissides, J: *Design Patterns*, page 151. Addison-Wesley, 1995
2. ^ Shalloway; Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*---.

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". *Sample Chapter*.

http://www.informit.com/articles/article.aspx?p=30297. From: James W. Cooper. *C# Design Patterns: A Tutorial*. Addison-Wesley. ISBN 0201844532. http://www.informit.com/store/product.aspx?isbn=0201844532.

- Example of using Bridge pattern (http://www.fsw.com/Jt/Jt.htm#_Toc186545248) in Jt, J2EE Pattern Oriented Framework (http://www.fsw.com/Jt/Jt.htm)

Retrieved from "http://en.wikipedia.org/wiki/Bridge_pattern"
Categories: Software design patterns | Articles with example C Sharp code | Articles with example Java code | Articles with example C++ code
Hidden category: Wikipedia articles needing clarification from August 2007

Composite pattern

From Wikipedia, the free encyclopedia

In computer science, the **composite pattern** is a partitioning design pattern. Composite allows a group of objects to be treated in the same way as a single instance of an object. The intent of composite is to "compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions uniformly."[1]

Contents

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 (e.g., a shape) designed as a composition of one-or-more similar objects (other kinds of shapes/geometries), all exhibiting similar functionality. This is known as a "has-a" relationship between objects. The key concept is that you can manipulate a single instance of the object just as you would 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.

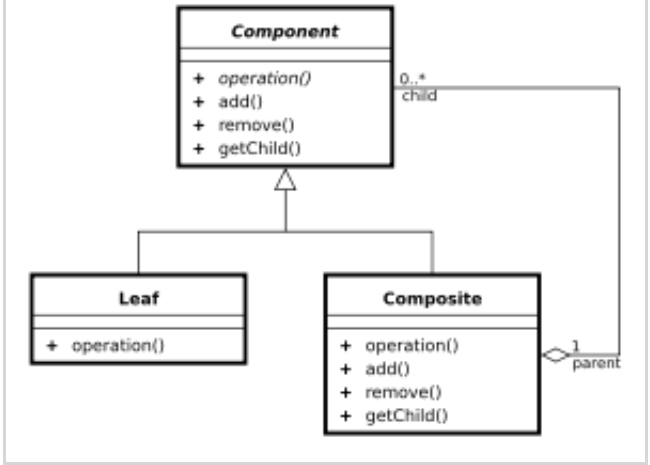
When to use

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

Component

- is the abstraction for all components, including composite ones
- declares the interface for objects in the composition
- implements default behavior for the interface common to all classes, as appropriate
- declares an interface for accessing and managing its child components
- (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate



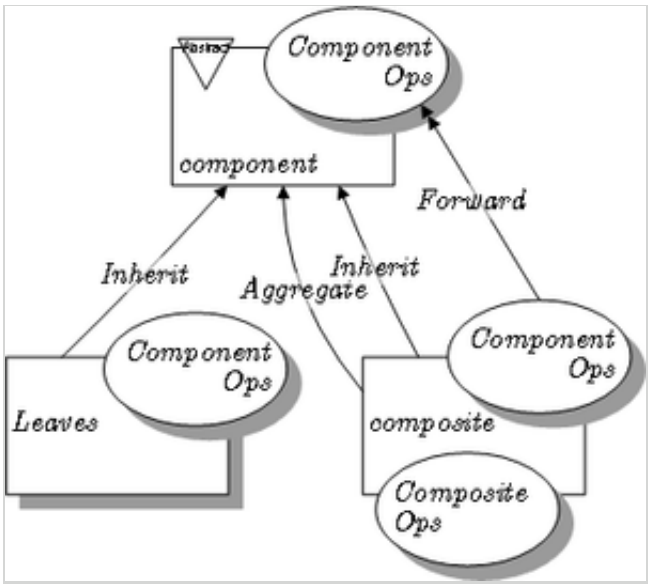
Composite pattern in UML.

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.

Example

Common Lisp

The following example, written in Common Lisp, and translated directly from the Java example below it, implements a method named *print-graphic*, which can be used on either an *ellipse*, or a list whose elements are either lists or *ellipses*.

```
(defstruct ellipse) ;; An empty struct.

;; For the method definitions, "object" is the variable,
;; and the following word is the type.
```

```
(defmethod print-graphic ((object null))
  NIL)

(defmethod print-graphic ((object cons))
  (print-graphic (first object))
  (print-graphic (rest object)))

(defmethod print-graphic ((object ellipse))
  (print 'ELLIPSE))

(let* ((ellipse-1 (make-ellipse))
       (ellipse-2 (make-ellipse))
       (ellipse-3 (make-ellipse))
       (ellipse-4 (make-ellipse)))

  (print-graphic (cons (list ellipse-1 (list ellipse-2 ellipse-3)) ellipse-4)))
```

Java

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 | ellipse GraphicList
```

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

List = empty_list | atom List | List List

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

/** "Component" */
interface Graphic {

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

}

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

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

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

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

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

}
```

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

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

}


/** Client */
public class Program {

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

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

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

        graphic2.add(ellipse4);

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

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

Python Example

```
# Component
class Graphic:
        # Prints the graphic
        def printGraphic(self):
                pass

# Composite
class CompositeGraphic(Graphic):

        def __init__(self):
                # Collection of child graphics
                self.__mChildGraphics = []

        # Prints the graphic.
        def printGraphic(self):
                for graphic in self.__mChildGraphics:
                        graphic.printGraphic()

        # Adds the graphic to the composition.
        def add(self, graphic):
                self.__mChildGraphics.append(graphic)

        # Removes the graphic from the composition.
        def remove(self, graphic):
                self.__mChildGraphics.remove(graphic)

# Leaf
```

```
class Ellipse(Graphic):

        def printGraphic(self):
                print "Ellipse"

def main():
        # Initialize four ellipses
        ellipse1 = Ellipse()
        ellipse2 = Ellipse()
        ellipse3 = Ellipse()
        ellipse4 = Ellipse()

        # Initialize three composite graphics
        graphic = CompositeGraphic()
        graphic1 = CompositeGraphic()
        graphic2 = 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.printGraphic()

if __name__ == "__main__":
        main()
```

C++ Example

```
#include <vector>
#include <iostream> // std::cout
#include <memory> // std::auto_ptr
#include <algorithm> // std::for_each
#include <functional> // std::mem_fun
using namespace std;

class Graphic
{
public:
  virtual void print() const = 0;
  virtual ~Graphic() {}
};

class Ellipse : public Graphic
{
public:
  void print() const {
    cout << "Ellipse \n";
  }
};

class CompositeGraphic : public Graphic
{
public:
  void print() const {
    // for each element in graphicList_, call the print member function
    for_each(graphicList_.begin(), graphicList_.end(), mem_fun(&Graphic::print));
  }

  void add(Graphic *aGraphic) {
    graphicList_.push_back(aGraphic);
  }

private:
  vector<Graphic*>  graphicList_;
```

```
};

int main()
{
  // Initialize four ellipses
  const auto_ptr<Ellipse> ellipse1(new Ellipse());
  const auto_ptr<Ellipse> ellipse2(new Ellipse());
  const auto_ptr<Ellipse> ellipse3(new Ellipse());
  const auto_ptr<Ellipse> ellipse4(new Ellipse());

  // Initialize three composite graphics
  const auto_ptr<CompositeGraphic> graphic(new CompositeGraphic());
  const auto_ptr<CompositeGraphic> graphic1(new CompositeGraphic());
  const auto_ptr<CompositeGraphic> graphic2(new CompositeGraphic());

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

  graphic2->add(ellipse4.get());

  graphic->add(graphic1.get());
  graphic->add(graphic2.get());

  // Prints the complete graphic (four times the string "Ellipse")
  graphic->print();
  return 0;
}
```

C# Example

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Composite
{
    /// <summary>
    /// Component
    /// </summary>
    public interface IGraphic
    {
        void print();
    }
    /// <summary>
    /// Composite
    /// </summary>
    public class CompositeGraphic:IGraphic
    {
        private List<IGraphic> m_ChildGraphics = new List<IGraphic>();
        #region Graphic Members

        public void print()
        {
            foreach (IGraphic graphic in m_ChildGraphics)
                graphic.print();
        }
        //Adds the graphic to the composition.
        public void add(IGraphic graphic)
        {
            m_ChildGraphics.Add(graphic);
        }

        //Removes the graphic from the composition.
        public void remove(IGraphic graphic)
        {
            m_ChildGraphics.Remove(graphic);
        }
```

```
        #endregion
    }
    /// <summary>
    /// Leaf
    /// </summary>
    public class Ellipse : IGraphic
    {
        #region IGraphic Members

        public void print()
        {
            Console.WriteLine("Ellipse");
        }

        #endregion
    }

    class Program
    {
        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();
            Console.ReadLine();
        }
    }
}
```

See also

- Design Patterns: The book that started it all.
- Mixin
- Facade pattern
- Decorator pattern
- Law of Demeter
- Delegation pattern
- Builder pattern
- Abstract factory pattern

External links

- Composite pattern description from the Portland Pattern Repository (http://c2.com/cgi/wiki?

CompositePattern)
- Composite pattern in UML and in LePUS3, a formal modelling language (http://www.lepus.org.uk/ref/companion/Composite.xml)
- Class::Delegation on CPAN (http://search.cpan.org/dist/Class-Delegation/lib/Class/Delegation.pm)
- Chinese Ring Puzzle Applet (http://www.cs.oberlin.edu/~jwalker/puzzle/)
- "The End of Inheritance: Automatic Run-time Interface Building for Aggregated Objects" (http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/149878) by Paul Baranowski
- PerfectJPattern Open Source Project (http://perfectjpattern.sourceforge.net/dp-composite.html) , Provides componentized implementation of the Composite Pattern in Java

References

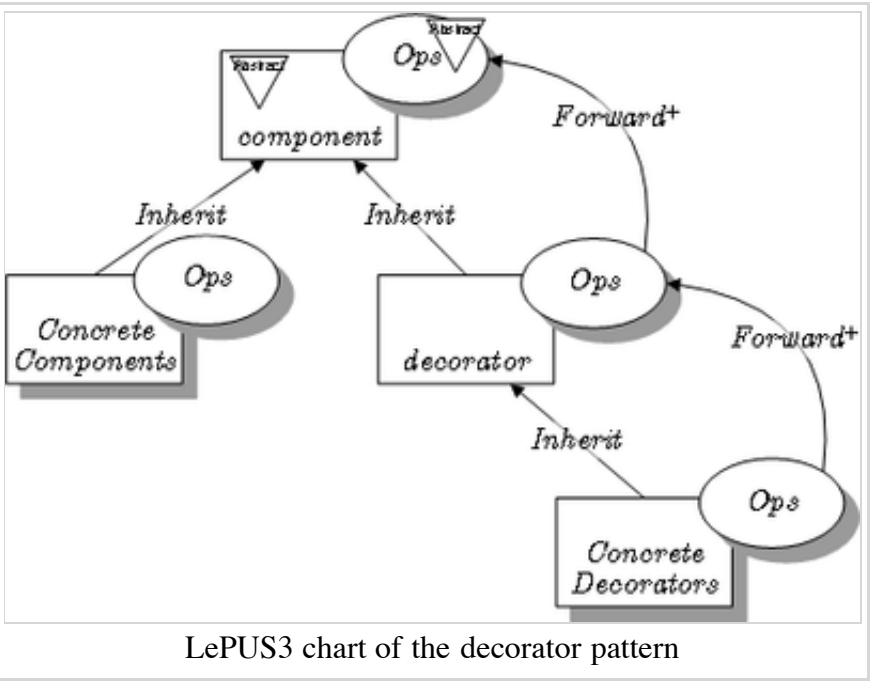
1. ^ ***a*** ***b*** Gamma, Erich; Richard Helm, Ralph Johnson, John M. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. pp. 395. ISBN 0201633612.

Parts of this article originated from the Perl Design Patterns Book

Decorator pattern

From Wikipedia, the free encyclopedia

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



LePUS3 chart of the decorator pattern

Contents

Introduction

The decorator pattern can be used to make it possible to extend (decorate) the functionality of a class at runtime. This works by adding a new *decorator* class that wraps the original class. This wrapping is achieved by

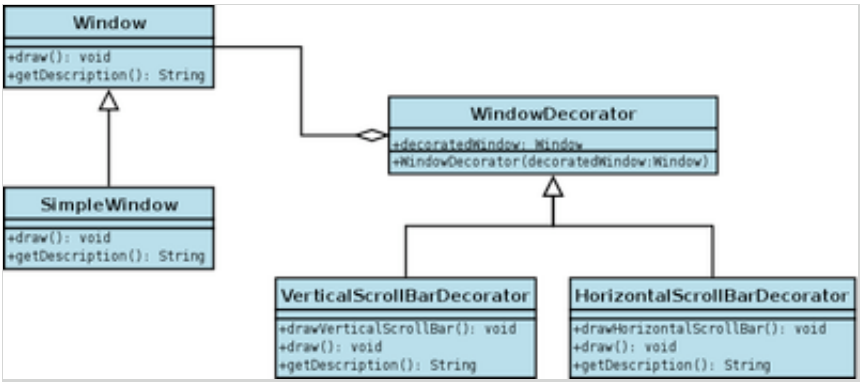
1. Subclass the original "Component" class into a "Decorator" class (see UML diagram)
2. In class Decorator, add a Component pointer as a field
3. Pass a Decorator pointer to the Decorator constructor to initialize the Component pointer.
4. In class Decorator, redirect all "Component" methods to the "Component" pointer. This implies that all Decorator fields coming from the Component motherclass will never be used and their memory space will be wasted. That is an accepted drawback of the decorator pattern.
5. In class Decorator, override any Component method which 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.

The decorator pattern is an alternative to subclassing. Subclassing adds behaviour at compile time whereas decorating can provide new behaviour at runtime.

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 what combinations of extensions will be needed at design time. 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. An example of the decorator pattern is the Java I/O Streams implementation.

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

Now let's assume we also wish the option 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 need merely 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.

Code examples

The following examples illustrate the use of decorators using the window/scrolling scenario.

Java

```
// 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;
    }
}


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

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

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

    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.

C++

```
#include <iostream>

using namespace std;

/* Component (interface) */
class Widget {

public:
  virtual void draw() = 0;
  virtual ~Widget() {}
};

/* ConcreteComponent */
class TextField : public Widget {

private:
   int width, height;

public:
   TextField( int w, int h ){
      width  = w;
      height = h;
   }

   void draw() {
      cout << "TextField: " << width << ", " << height << endl;
   }
};

/* Decorator (interface) */
class Decorator : public Widget {

private:
   Widget* wid;       // reference to Widget

public:
   Decorator( Widget* w )  {
     wid = w;
   }

   void draw() {
     wid->draw();
   }

   ~Decorator() {
     delete wid;
   }
};
```

```
/* ConcreteDecoratorA */
class BorderDecorator : public Decorator {

public:
   BorderDecorator( Widget* w ) : Decorator( w ) { }
   void draw() {
      Decorator::draw();
      cout << "   BorderDecorator" << endl;
   }
};

/* ConcreteDecoratorB */
class ScrollDecorator : public Decorator {
public:
   ScrollDecorator( Widget* w ) : Decorator( w ) { }
   void draw() {
      Decorator::draw();
      cout << "   ScrollDecorator" << endl;
   }
};

int main( void ) {

   Widget* aWidget = new BorderDecorator(
                     new ScrollDecorator(
                     new TextField( 80, 24 )));
   aWidget->draw();
   delete aWidget;
}
```

C#

```
namespace GSL_Decorator_pattern
{
    interface IWindowObject
    {
        void Draw(); // draws the object
        String GetDescription(); // returns a description of the object
    }


    class ControlComponent : IWindowObject
    {
        public ControlComponent()
        {
        }

        public void Draw() // draws the object
        {
            Console.WriteLine("ControlComponent::draw()");
        }

        public String GetDescription() // returns a description of the object
        {
            return "ControlComponent::getDescription()";
        }
    }

    abstract class Decorator : IWindowObject
    {
        protected IWindowObject _decoratedWindow = null; // the object being decorated

        public Decorator(IWindowObject decoratedWindow)
        {
            _decoratedWindow = decoratedWindow;
        }

        public virtual void Draw()
        {
            _decoratedWindow.Draw();
            Console.WriteLine("\tDecorator::draw() ");
```

```
    }

    public virtual String GetDescription() // returns a description of the object
    {
        return _decoratedWindow.GetDescription() + "\n\t" + "Decorator::getDescription() ";
    }
}

// the first decorator
class DecorationA : Decorator
{
    public DecorationA(IWindowObject decoratedWindow)
        : base(decoratedWindow)
    {
    }

    public override void Draw()
    {
        base.Draw();
        DecorationAStuff();
    }

    private void DecorationAStuff()
    {
        Console.WriteLine("\t\tdoing DecorationA things");
    }

    public override String GetDescription()
    {
        return base.GetDescription() + "\n\t\tincluding " + this.ToString();
    }

}// end  class ConcreteDecoratorA : Decorator

class DecorationB : Decorator
{
    public DecorationB(IWindowObject decoratedWindow)
        : base(decoratedWindow)
    {
    }

    public override void Draw()
    {
        base.Draw();
        DecorationBStuff();
    }

    private void DecorationBStuff()
    {
        Console.WriteLine("\t\tdoing DecorationB things");
    }

    public override String GetDescription()
    {
        return base.GetDescription() + "\n\t\tincluding " + this.ToString();
    }

}// end  class DecorationB : Decorator

class DecorationC : Decorator
{
    public DecorationC(IWindowObject decoratedWindow)
        : base(decoratedWindow)
    {
    }

    public override void Draw()
    {
        base.Draw();
        DecorationCStuff();
    }

    private void DecorationCStuff()
    {
        Console.WriteLine("\t\tdoing DecorationC things");
```

```
        }

        public override String GetDescription()
        {
            return base.GetDescription() + "\n\t\tincluding " + this.ToString();
        }

    }// end  class DecorationC : Decorator

    class Program
    {
        static void Main(string[] args)
        {
            IWindowObject window = new DecorationA(new ControlComponent());
            window = new DecorationB(window);
            window = new DecorationC(window);
            window.Draw();
            window.GetDescription();
            Console.ReadLine();
        }
    }
}// end of namespace GSL_Decorator_pattern
```

Python

```
class Coffee:
    def cost(self):
        return 1

class Milk:
    def __init__(self, coffee):
        self.coffee = coffee

    def cost(self):
        return self.coffee.cost() + .5

class Whip:
    def __init__(self, coffee):
        self.coffee = coffee

    def cost(self):
        return self.coffee.cost() + .7

class Sprinkles:
    def __init__(self, coffee):
        self.coffee = coffee

    def cost(self):
        return self.coffee.cost() + .2

# Example 1
coffee = Milk(Coffee())
print "Coffee with milk : "+str(coffee.cost())

# Example 2
coffee = Sprinkles(Whip(Milk(Coffee())))
print "Coffee with milk, whip and sprinkles : "+str(coffee.cost())
```

PHP

```
// Component
interface MenuItem{
        function cost();
}

// Concrete Component
class Coffee implements MenuItem{
        function cost(){
```

```
                return 1;
        }
}

// Decorator
abstract class MenuDecorator{

        protected $component;

        function __construct($component){

                $this->component = $component;

        }

        abstract function cost();

}

// Concrete Decorator
class CoffeeMilkDecorator extends MenuDecorator{

        function cost(){

                return $this->component->cost() + 0.5;

        }

}

// Concrete Decorator
class CoffeeWhipDecorator extends MenuDecorator{

        function cost(){

                return $this->component->cost() + 0.7;

        }

}

// Concrete Decorator
class CoffeeSprinklesDecorator extends MenuDecorator{

        function cost(){

                return $this->component->cost() + 0.2;

        }

}


// Example 1
$coffee = new CoffeeMilkDecorator(new Coffee());
print "Coffee with milk : ".$coffee->cost();

// Example 2
$coffee = new CoffeeWhipDecorator(new CoffeeMilkDecorator(new CoffeeSprinklesDecorator(new Coffee())
print "Coffee with milk, whip and sprinkles : ".$coffee->cost();
```

Dynamic Languages

The decorator pattern can also be implemented in dynamic languages with no interfaces or traditional OOP inheritance.

JavaScript coffee shop:

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

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

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

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

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

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

See also

- Composite pattern
- Adapter pattern
- Abstract class
- Abstract factory
- Immutable object

External links

- Decorator pattern description from the Portland Pattern Repository (http://c2.com/cgi/wiki?DecoratorPattern)
- Article "The Decorator Design Pattern (http://exciton.cs.rice.edu/JavaResources/DesignPatterns/DecoratorPattern.htm) " (Java) by Antonio García and Stephen Wong
- Article "Using the Decorator Pattern (http://www.onjava.com/pub/a/onjava/2003/02/05/decorator.html) " (Java) by Budi Kurniawan
- Decorator in UML and in LePUS3 (a formal modelling language) (http://www.lepus.org.uk/ref/companion/Decorator.xml)
- Sample Chapter "C# Design Patterns: The Decorator Pattern (http://www.informit.com/articles/article.aspx?p=31350) " (C#) by James W. Cooper

- A PHP approach (http://phppatterns.com/index.php/article/articleview/30/1/1) (PHP)
- A Delphi approach (http://www.castle-cadenza.demon.co.uk/decorate.htm) (Delphi)
- A JavaScript implementation (http://www.phpied.com/a-javascript-implementation-of-the-decorator-pattern/)
- Jt (http://www.fsw.com/Jt/Jt.htm) J2EE Pattern Oriented Framework
- Decorator Pattern with Ruby in 8 Lines (http://www.lukeredpath.co.uk/2006/9/6/decorator-pattern-with-ruby-in-8-lines) (Ruby)
- PerfectJPattern Open Source Project (http://perfectjpattern.sourceforge.net/dp-decorator.html) , Provides componentized implementation of the Decorator Pattern in Java

Facade pattern

From Wikipedia, the free encyclopedia

The **Facade pattern** is a software engineering design pattern commonly used with Object-oriented programming.

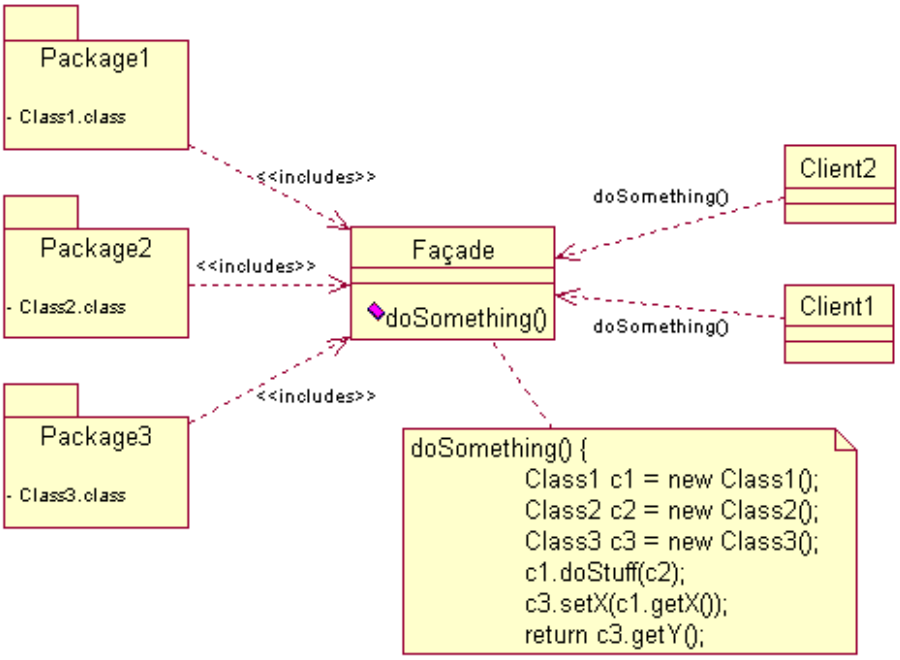
A facade is an object that provides a simplified interface to a larger body of code, such as a class library. A facade can:

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

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

Contents

Structure



Facade
: The facade class interacts Packages 1, 2, and 3 with the rest of the application.

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

Packages
: Software library / API collection accessed through the Facade Class.

Examples

Java

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) {
                ...
        }
}

/* Façade */

class Computer {
        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();
        }
}
```

C#

```
// Facade pattern -- Structural example

using System;

namespace DoFactory.GangOfFour.Facade.Structural
{
  // Mainapp test application

  class MainApp
  {
    public static void Main()
    {
      Facade facade = new Facade();

      facade.MethodA();
      facade.MethodB();

      // Wait for user
      Console.Read();
    }
  }

  // "Subsystem ClassA"

  class SubSystemOne
  {
    public void MethodOne()
    {
      Console.WriteLine(" SubSystemOne Method");
    }
  }

  // Subsystem ClassB"

  class SubSystemTwo
  {
    public void MethodTwo()
    {
      Console.WriteLine(" SubSystemTwo Method");
    }
  }

  // Subsystem ClassC"

  class SubSystemThree
  {
    public void MethodThree()
    {
      Console.WriteLine(" SubSystemThree Method");
    }
  }

  // Subsystem ClassD"

  class SubSystemFour
  {
    public void MethodFour()
    {
      Console.WriteLine(" SubSystemFour Method");
    }
  }

  // "Facade"
```

```
  class Facade
  {
    SubSystemOne one;
    SubSystemTwo two;
    SubSystemThree three;
    SubSystemFour four;

    public Facade()
    {
      one = new SubSystemOne();
      two = new SubSystemTwo();
      three = new SubSystemThree();
      four = new SubSystemFour();
    }

    public void MethodA()
    {
      Console.WriteLine("\nMethodA() ---- ");
      one.MethodOne();
      two.MethodTwo();
      four.MethodFour();
    }

    public void MethodB()
    {
      Console.WriteLine("\nMethodB() ---- ");
      two.MethodTwo();
      three.MethodThree();
    }
  }
}
```

External links

- Description from the Portland Pattern Repository (http://c2.com/cgi/wiki?FacadePattern)
- Description from the Net Objectives Repository (http://www.netobjectivesrepository.com/TheFacadePattern)
- Jt (http://www.fsw.com/Jt/Jt.htm) J2EE Pattern Oriented Framework

Flyweight pattern

From Wikipedia, the free encyclopedia

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's common to put them in external data structures and pass them to the flyweight objects temporarily when they are used.

A classic example usage of the flyweight pattern are the data structures for graphical representation of characters in a word processor. It would be nice to have, for each character in a document, a glyph object containing its font outline, font metrics, and other formatting data, but it 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 externally.

Contents

Examples

The following programs illustrate the document example given above: the flyweights are called FontData in the Java example and GraphicChar in the C# example.

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

Java

```
public enum FontEffect {
    BOLD, ITALIC, SUPERSCRIPT, SUBSCRIPT, STRIKETHROUGH
}

public final class FontData {
    /**
     * A weak hash map will drop unused references to FontData.
     * Values have to be wrapped in WeakReferences,
     * because value objects in weak hash map are held by strong references.
     */
    private static final WeakHashMap<FontData, WeakReference<FontData>> flyweightData =
        new WeakHashMap<FontData, WeakReference<FontData>>();
```

```
    private final int pointSize;
    private final String fontFace;
    private final Color color;
    private final Set<FontEffect> effects;

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

    public static FontData create(int pointSize, String fontFace, Color color,
        FontEffect... effects) {
        EnumSet<FontEffect> effectsSet = EnumSet.noneOf(FontEffect.class);
        for (FontEffect fontEffect : effects) {
            effectsSet.add(fontEffect);
        }
        // We are unconcerned with object creation cost, we are reducing overall memory consumption
        FontData data = new FontData(pointSize, fontFace, color, effectsSet);
        if (!flyweightData.containsKey(data)) {
            flyweightData.put(data, new WeakReference<FontData> (data));
        }
        // return the single immutable copy with the given values
        return flyweightData.get(data).get();
    }

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

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

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

C#

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


class GraphicChar
{
    char c;
    string fontFace;
    public GraphicChar(char c, string fontFace) { this.c = c; this.fontFace = fontFace; }
    public static void printAtPosition(GraphicChar c, int x, int y)
    {
        Console.WriteLine("Printing '{0}' in '{1}' at position {2}:{3}.", c.c, c.fontFace, x, y);
    }
}

class GraphicCharFactory
{
    Dictionary<string, GraphicChar> pool = new Dictionary<string, GraphicChar>();

    public int getNum() { return pool.Count; }

    public GraphicChar get(char c, string fontFace)
    {
        GraphicChar gc;
```

```
        string key = c.ToString() + fontFace;
        if (!pool.TryGetValue(key, out gc)) {
            gc = new GraphicChar(c, fontFace);
            pool.Add(key, gc);
        }
        return gc;
    }
}

class FlyWeightExample
{
    public static void Main(string[] args)
    {
        GraphicCharFactory cf = new GraphicCharFactory();

        // Compose the text by storing the characters as objects.
        List<GraphicChar> text = new List<GraphicChar>();
        text.Add(cf.get('H', "Arial"));    // 'H' and "Arial" are called intrinsic information
        text.Add(cf.get('e', "Arial"));    // because it is stored in the object itself.
        text.Add(cf.get('l', "Arial"));
        text.Add(cf.get('l', "Arial"));
        text.Add(cf.get('o', "Times"));
        text.Add(cf.get(' ', "Times"));
        text.Add(cf.get('w', "Times"));
        text.Add(cf.get('o', "Times"));
        text.Add(cf.get('r', "Times"));
        text.Add(cf.get('l', "Times"));
        text.Add(cf.get('d', "Times"));

        // See how the Flyweight approach is beginning to save space:
        Console.WriteLine("CharFactory created only {0} objects for {1} characters.", cf.getNum(), t

        int x = 0, y = 0;
        foreach (GraphicChar c in text)
        {             // Passing position as extrinsic information to the objects,
            GraphicChar.printAtPosition(c, x++, y);   // as a top-left 'A' is not different from a t
        }

        Console.ReadLine();
    }
}
```

C++

Please note this is just a simple demo of flyweight concept in c++. You could easily use the Boost Flyweight library instead.

```
#include <iostream>
#include <string>
#include <map>
#include <list>
#include <memory>
#include <algorithm>
using namespace std;

class GraphicChar
{
public:
        friend class Character;
        GraphicChar(char c, const string fontFace):m_c(c), m_fontFace(fontFace){}
protected:
private:
        char    m_c;
        string  m_fontFace;
};

class GraphicCharFactory
{
        typedef std::map<string, GraphicChar*>::iterator poolIter;
public:
        virtual ~GraphicCharFactory()
        {
                for (poolIter beg = m_pool.begin(); beg != m_pool.end(); ++beg)
```

```
                        delete beg->second;
                m_pool.clear();
        }

        GraphicChar* get(char c, const string fontFace)
        {
                string key = c + fontFace;
                poolIter posit;
                if (( posit = m_pool.find(key))!= m_pool.end())
                        return posit->second;
                else
                        return (*((m_pool.insert(make_pair(key, new GraphicChar(c, fontFace))))).firs
        }

        int count()
        {
                return m_pool.size();
        }

protected:
private:
        std::map<string, GraphicChar*>  m_pool;
};

class Character
{
public:
        Character(int a,int b,const GraphicChar* gra):m_x(a),m_y(b),m_Grap(gra){}

        void print()
        {
                cout << "Printing " << m_Grap->m_c << " in " << m_Grap->m_fontFace << " at position"
        }

protected:
private:
        int                                 m_x, m_y;
        const GraphicChar* m_Grap;
};

int _tmain(int argc, _TCHAR* argv[])
{
        typedef std::list<Character> article;
        article simpArticle;
        std::auto_ptr<GraphicCharFactory> cf(new GraphicCharFactory);
        int x = 0, y = 0;
        simpArticle.push_back(Character(x,   y, cf->get('H', "Arial")) );
        simpArticle.push_back(Character(++x, y, cf->get('e', "Arial")) );
        simpArticle.push_back(Character(++x, y, cf->get('l', "Arial")) );
        simpArticle.push_back(Character(++x, y, cf->get('l', "Arial")) );
        simpArticle.push_back(Character(++x, y, cf->get('o', "Arial")) );
        simpArticle.push_back(Character(++x, y, cf->get(' ', "Arial")) );
        simpArticle.push_back(Character(++x, y, cf->get('w', "Times")) );
        simpArticle.push_back(Character(++x, y, cf->get('o', "Arial")) );
        simpArticle.push_back(Character(++x, y, cf->get('r', "Times")) );
        simpArticle.push_back(Character(++x, y, cf->get('l', "Arial")) );
        simpArticle.push_back(Character(++x, y, cf->get('d', "Times")) );
        std::for_each(simpArticle.begin(),simpArticle.end(), std::mem_fun_ref(&Character::print));
        cout <<"CharFactory created only "<< cf->count() <<" objects for " << simpArticle.size() <<"
        return 0;
}
```

Console output:

```
Printing H in Arial at position0 : 0
Printing e in Arial at position1 : 0
Printing l in Arial at position2 : 0
Printing l in Arial at position3 : 0
Printing o in Arial at position4 : 0
Printing   in Arial at position5 : 0
Printing w in Times at position6 : 0
Printing o in Arial at position7 : 0
Printing r in Times at position8 : 0
Printing l in Arial at position9 : 0
Printing d in Times at position10 : 0
```

```
CharFactory created only 8 objects for 11 characters.
```

External links

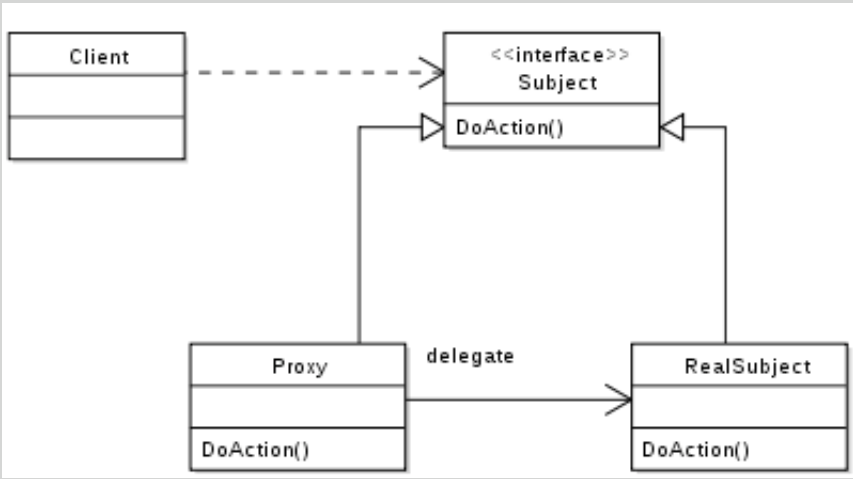
- Flyweight in UML and in LePUS3 (a formal modelling language) (http://www.lepus.org.uk/ref/companion/Flyweight.xml)
- Article "Make your apps fly - Implement Flyweight to improve performance (http://www.javaworld.com/javaworld/jw-07-2003/jw-0725-designpatterns.html) " by David Geary
- Article "Enhancing Web Application Performance with Caching (http://theserverside.com/articles/article.tss?l=Caching) " by Neal Ford
- Article "The Flyweight Pattern (http://codeproject.com/gen/design/testvalidators.asp) " by Alberto Bar-Noy
- Sample Chapter "C# Design Patterns: The Flyweight Pattern (http://www.informit.com/articles/article.aspx?p=31563) " by James W. Cooper
- 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
- Overview (http://dofactory.com/Patterns/PatternFlyweight.aspx)
- Sourdough Design (http://sourdough.phpee.com/index.php?node=18)
- Structural Patterns - Flyweight Pattern (http://www.allapplabs.com/java_design_patterns/flyweight_pattern.htm)
- Class::Flyweight - implement the flyweight pattern in OO perl (http://perlmonks.thepen.com/94783.html)
- Boost.Flyweight - A generic C++ implemention (http://boost.org/libs/flyweight/index.html)

Proxy pattern

From Wikipedia, the free encyclopedia

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

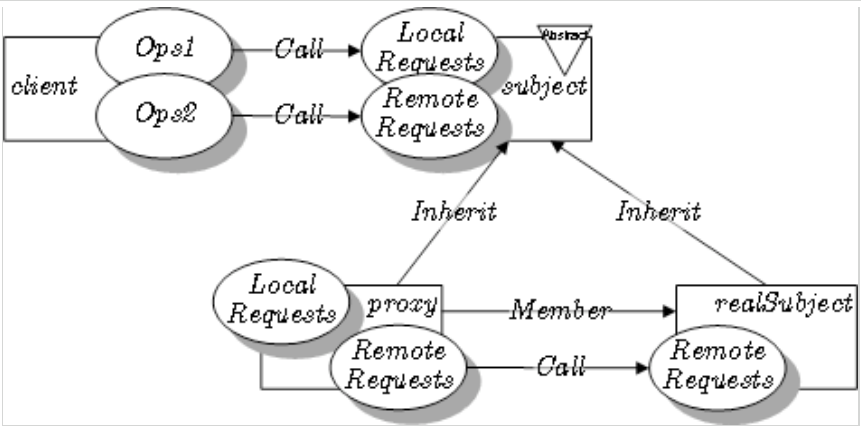
A proxy, in its most general form, is a class functioning as an interface to something else. The proxy could interface to anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate.



Proxy in UML

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

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



Proxy in LePUS3 (legend (http://lepus.org.uk/ref/legend/legend.xml))

Contents

Examples

Java

The following Java example illustrates the "virtual proxy" pattern. The `ProxyImage` class is used to delay the expensive operation of loading a file from disk until the result of that operation is actually needed. If the file is never needed, then the expensive load has been totally eliminated.

```
import java.util.*;

interface Image {
    public void displayImage();
}

class RealImage implements Image {
    private String filename;
    public RealImage(String filename) {
        this.filename = filename;
        loadImageFromDisk();
    }

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

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

class ProxyImage implements Image {
    private String filename;
    private Image image;

    public ProxyImage(String filename) { this.filename = filename; }
    public void displayImage() {
        if (image == null) {
            image = new RealImage(filename); // load only on demand
        }
        image.displayImage();
    }
}

class ProxyExample {
    public static void main(String[] args) {
        Image image1 = new ProxyImage("HiRes_10MB_Photo1");
        Image image2 = new ProxyImage("HiRes_10MB_Photo2");
        Image image3 = new ProxyImage("HiRes_10MB_Photo3");

        image1.displayImage(); // loading necessary
        image2.displayImage(); // loading necessary
        image1.displayImage(); // no loading necessary; already done
        // the third image will never be loaded - time saved!
    }
}
```

The program's output is:

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

Python

```python
#!/opt/env/python

class Image(object):
        def display(self):
                pass

class RealImage(Image):
        def __init__(self, name):
                self.name = name
                self.load()
        def load(self):
                print "Loading...", self.name
        def display(self):
                print "Displaying...", self.name

class ProxyImage(Image):
        def __init__(self, name):
                self.name = name
                self.test = None
        def display(self):
                if not self.test:
                        self.test = RealImage(self.name)
                self.test.display()

def main():
        t1 = ProxyImage("big_image1.png")
        t2 = ProxyImage("big_image2.png")
        t3 = ProxyImage("big_image3.png")

        t1.display() # loads, displays
        t2.display() # loads, displays
        t1.display() # displays
#       t3.display() # no display, no load
        return

if __name__ == "__main__":
        main()
```

C#

In this C# example, the `RealClient` stores an account number. Only users who know a valid password can access this account number. The `RealClient` is protected by a `ProtectionProxy` which knows the password. If a user wants to get an account number, first the proxy asks the user to authenticate; only if the user entered a correct password does the proxy invoke the `RealClient` to get an account number for the user.

In this example, **thePassword** is the correct password.

```
using System;

namespace ConsoleApplicationTest.FundamentalPatterns.ProtectionProxyPattern
{
    public interface IClient {
        string GetAccountNo();
    }

    public class RealClient : IClient {
        private string accountNo = "12345";
        public RealClient() {
            Console.WriteLine("RealClient: Initialized");
        }
        public string GetAccountNo() {
            Console.WriteLine("RealClient's AccountNo: " + accountNo);
```

```
            return accountNo;
        }
    }


    public class ProtectionProxy : IClient
    {
        private string password;  //password to get secret
        RealClient client;

        public ProtectionProxy(string pwd) {
            Console.WriteLine("ProtectionProxy: Initialized");
            password = pwd;
            client = new RealClient();
        }

        // Authenticate the user and return the Account Number
        public String GetAccountNo() {
            Console.Write("Password: ");
            string tmpPwd = Console.ReadLine();

            if (tmpPwd == password) {
                return client.GetAccountNo();
            } else {
                Console.WriteLine("ProtectionProxy: Illegal password!");
                return "";
            }
        }
    }

    class ProtectionProxyExample
    {
        [STAThread]
        public static void Main(string[] args) {
            IClient client = new ProtectionProxy("thePassword");
            Console.WriteLine();
            Console.WriteLine("main received: " + client.GetAccountNo());
            Console.WriteLine("\nPress any key to continue . . .");
            Console.Read();
        }
    }
}
```

C++

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

class Image //abstract
{
protected:
   Image() {}

public:
   virtual ~Image() {}

   virtual void displayImage() = 0;
};

class RealImage : public Image
{
public:
   explicit RealImage(std::string filename)
   {
      this->filename = filename;
      loadImageFromDisk();
   }


   void displayImage()
   {
```

```
        std::cout<<"Displaying "<<filename<<std::endl;
    }

private:
    void loadImageFromDisk()
    {
        // Potentially expensive operation
        // ...
        std::cout<<"Loading    "<<filename<<std::endl;
    }

    std::string filename;
};

class ProxyImage : public Image
{
public:
    explicit ProxyImage(std::string filename)
    {
        this->filename = filename;
        this->image = nullptr;
    }
    ~ProxyImage() { delete image; }

    void displayImage()
    {
        if (image == nullptr) {
            image = new RealImage(filename); // load only on demand
        }

        image->displayImage();
    }

private:
    std::string filename;
    Image* image;
};

int main(int argc, char* argv[])
{
    std::cout<<"main"<<std::endl;

    Image* image1 = new ProxyImage("HiRes_10MB_Photo1");
    Image* image2 = new ProxyImage("HiRes_10MB_Photo2");
    Image* image3 = new ProxyImage("HiRes_10MB_Photo3");

    image1->displayImage(); // loading necessary
    image2->displayImage(); // loading necessary
    image1->displayImage(); // no loading necessary; already done
    // the third image will never be loaded - time saved!

    delete image1;
    delete image2;
    delete image3;

    return 0;
}
```

Perl

The Perl with Moose implementation. The "lazy" option is used to delay the expensive operation. The "handles" option is used to create the delegation to proxied class.

```
use MooseX::Declare;

role Image {
    requires 'display_image';
}

class RealImage with Image {
```

```
    has 'filename' => (
        isa => 'Str',
    );

    # Moose's constructor
    method BUILD ($params) {
        $self->_load_image_from_disk;
    }

    method _load_image_from_disk () {
        # Expensive operation
        print "Loading   " . $self->{filename} . "\n";
    }

    method display_image () {
        print "Displaying " . $self->{filename} . "\n";
    }
}

class ProxyImage with Image {
    has 'filename' => (
        isa => 'Str',
    );
    has 'image' => (
        does => 'Image',
        lazy => 1,  # default value is loaded only on demand
        default => sub { RealImage->new( filename => $_[0]->{filename} ) },
        handles => [ 'display_image' ],  # automatic delegation
    );
}

class ProxyExample {
    # Moose's constructor
    method BUILD ($params) {
        my $image1 = ProxyImage->new( filename=>'HiRes_10MB_Photo1' );
        my $image2 = ProxyImage->new( filename=>'HiRes_10MB_Photo2' );
        my $image3 = ProxyImage->new( filename=>'HiRes_10MB_Photo3' );

        $image1->display_image;  # loading necessary
        $image2->display_image;  # loading necessary
        $image1->display_image;  # already done
        # the third image will never be loaded - time saved!
    }
}

ProxyExample->new;
```

ActionScript

```
package
{
        import flash.display.Sprite;

        public class ProxyExample extends Sprite
        {
                public function ProxyExample()
                {
                        var image1:Image = new ProxyImage("HiRes_10MB_Photo1");
                        var image2:Image = new ProxyImage("HiRes_10MB_Photo2");
                        var image3:Image = new ProxyImage("HiRes_10MB_Photo3");

                        image1.displayImage();// loading necessary
                        image2.displayImage();// loading necessary
                        image1.displayImage();// no loading necessary; already done
                        // the third image will never be loaded - time saved!
                }
        }
}
interface Image
```

```
{
        function displayImage():void;
}

class RealImage implements Image
{
        private var filename:String;
        public function RealImage(filename:String)
        {
                this.filename = filename;
                loadImageFromDisk();
        }
        private function loadImageFromDisk():void
        {
                // Potentially expensive operation
                // ...
                trace("Loading  ", filename);
        }

        public function displayImage():void
        {
                trace("Displaying       ", filename);
        }
}

class ProxyImage implements Image
{
        private var filename:String;
        private var image:Image;

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

        public function displayImage():void
        {
                if ( image == null )
                {
                        image = new RealImage(filename); // load only on demand
                }
                image.displayImage();
        }
}
```

See also

- Composite pattern
- Decorator pattern
- Lazy initialization

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
- Jt (http://www.fsw.com/Jt/Jt.htm) J2EE Pattern Oriented Framework
- PerfectJPattern Open Source Project (http://perfectjpattern.sourceforge.net/dp-proxy.html) , Provides componentized implementation of the Proxy Pattern in Java

Categories: Software design patterns | Articles with example Java code | Articles with example C Sharp code
Hidden categories: Cleanup from February 2009 | All pages needing cleanup

Behavioral pattern

From Wikipedia, the free encyclopedia

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
- 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 which may be raised by another object
- State pattern: A clean way for an object to partially change its type at runtime
- Strategy pattern: Algorithms can be selected on the fly
- Specification pattern: Recombinable Business logic in a boolean fashion
- Template method pattern: Describes the program skeleton of a program
- Visitor pattern: A way to separate an algorithm from an object
- Single-serving visitor pattern: Optimise the implementation of a visitor that is allocated, used only once, and then deleted
- Hierarchical visitor pattern: Provide a way to visit every node in a hierarchical data structure such as a tree.

See also

- Structural pattern
- Creational pattern
- Concurrency pattern

Chain-of-responsibility pattern

From Wikipedia, the free encyclopedia
(Redirected from Chain of responsibility pattern)

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

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

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

Contents

Examples

Java

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

```
Writing to stdout:   Entering function y.
Writing to stdout:   Step1 completed.
```

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

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

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.

```
import java.util.*;

abstract class Logger
{
    public static int ERR = 3;
    public static int NOTICE = 5;
    public static int DEBUG = 7;
    protected int mask;

    // The next element in the chain of responsibility
    protected Logger next;
    public Logger setNext( Logger l)
    {
        next = l;
        return l;
    }

    public void message( String msg, int priority )
    {
        if ( priority <= mask )
        {
            writeMessage( msg );
            if ( next != null )
            {
                next.message( msg, priority );
            }
        }
    }

    abstract protected void writeMessage( String msg );

}

class StdoutLogger extends Logger
{

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

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


class EmailLogger extends Logger
{

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

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

class StderrLogger extends Logger
{
```

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

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

public class ChainOfResponsibilityExample
{
    public static void main( String[] args )
    {
        // Build the chain of responsibility
        Logger l,l1;
        l1 = l = new StdoutLogger( Logger.DEBUG );
        l1 = l1.setNext(new EmailLogger( Logger.NOTICE ));
        l1 = l1.setNext(new StderrLogger( Logger.ERR ));

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

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

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

C#, Abstract class based implementation

```
using System;

namespace Chain_of_responsibility{
        public abstract class Chain{
                private Chain _next;

                public Chain Next{
                        get{return _next;}
                        set{_next = value;}
                }

                public void Message(object command){
                        if ( Process(command) == false && _next != null ){
                                _next.Message(command);
                        }
                }

                public static Chain operator +(Chain lhs, Chain rhs){
                        Chain last = lhs;
                        while ( last.Next != null ){
                                last = last.Next;
                        }
                        last.Next = rhs;
                        return lhs;
                }

                protected abstract bool Process(object command);
        }

        public class StringHandler : Chain{
                protected override bool Process(object command) {
                        if ( command is string ){
                                Console.WriteLine("StringHandler can handle this message : {0}",(str
                                return true;
                        }
                        return false;
                }
        }

        public class IntegerHandler : Chain{
```

```
                protected override bool Process(object command){
                        if ( command is int ){
                                Console.WriteLine("IntegerHandler can handle this message : {0}",(in
                                return true;
                        }
                        return false;
                }
        }

        public class NullHandler : Chain{
                protected override bool Process(object command){
                        if ( command == null ){
                                Console.WriteLine("NullHandler can handle this message.");
                                return true;
                        }
                        return false;
                }
        }

        public class IntegerBypassHandler : Chain{
                protected override bool Process(object command){
                        if ( command is int ){
                                Console.WriteLine("IntegerBypassHandler can handle this message : {0
                                return false; // Always pass to next handler
                        }
                        return false; // Always pass to next handler
                }
        }

        class TestMain{
                static void Main(string[] args){
                        Chain chain = new StringHandler();
                        chain += new IntegerBypassHandler();
                        chain += new IntegerHandler();
                        chain += new IntegerHandler(); // Never reached
                        chain += new NullHandler();

                        chain.Message("1st string value");
                        chain.Message(100);
                        chain.Message("2nd string value");
                        chain.Message(4.7f); // not handled
                        chain.Message(null);
                }
        }
}
```

C#, Interface based implementation

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

namespace Chain_of_responsibility{
        public interface IChain{
            bool Process(object command);
        }

        public class Chain{
            private List<IChain> list;

            public List<IChain> List{
                get{
                    return this.list;
                }
            }

            public Chain(){
                this.list = new List<IChain>();
            }
```

```
        public void Message(object command){
            foreach (IChain item in this.list){
                bool result = item.Process(command);

                if (result == true)
                    break;
            }
        }

        public void Add(IChain handler){
            this.list.Add(handler);
        }
    }

    public class StringHandler : IChain{
        public bool Process(object command){
            if (command is string){
                Console.WriteLine("StringHandler can handle this message : {0}", (string)commanc
                return true;
            }
            return false;
        }
    }

    public class IntegerHandler : IChain{
        public bool Process(object command){
            if (command is int){
                Console.WriteLine("IntegerHandler can handle this message : {0}", (int)command);
                return true;
            }
            return false;
        }
    }

    public class NullHandler : IChain{
        public bool Process(object command){
            if (command == null){
                Console.WriteLine("NullHandler can handle this message.");
                return true;
            }
            return false;
        }
    }

    public class IntegerBypassHandler : IChain{
        public bool Process(object command){
            if (command is int){
                Console.WriteLine("IntegerBypassHandler can handle this message : {0}", (int)com
                return false; // Always pass to next handler
            }
            return false; // Always pass to next handler
        }
    }

    class TestMain{
            static void Main(string[] args){
                    Chain chain = new Chain();

                    chain.Add(new StringHandler());
                    chain.Add(new IntegerBypassHandler());
                    chain.Add(new IntegerHandler());
                    chain.Add(new IntegerHandler()); // Never reached
                    chain.Add(new NullHandler());

                    chain.Message("1st string value");
                    chain.Message(100);
                    chain.Message("2nd string value");
                    chain.Message(4.7f); // not handled
                    chain.Message(null);
            }
    }
}
```

C#, Delegate based implementation

```
using System;

namespace Chain_of_responsibility{
        public static class StaticState{
                private static int count;

                public static bool StringHandler(object command){
                        if ( command is string ){
                                string th;
                                count++;
                                if ( count % 10 == 1 ) th = "st";
                                else if ( count % 10 == 2 ) th = "nd";
                                else if ( count % 10 == 3 ) th = "rd";
                                else th = "th";
                                Console.WriteLine("StringHandler can handle this {0}{1} message : {2
                                return true;
                        }
                        return false;
                }
        }

        public static class NoState
        {
                public static bool StringHandler2(object command)
                {
                        if ( command is string ){
                                Console.WriteLine("StringHandler2 can handle this message : {0}",(st
                                return true;
                        }
                        return false;
                }

                public static bool IntegerHandler(object command){
                        if ( command is int ){
                                Console.WriteLine("IntegerHandler can handle this message : {0}",(in
                                return true;
                        }
                        return false;
                }
        }

        public static class Chain{
                public delegate bool MessageHandler(object message);
                public static event MessageHandler Message;

                public static void Process(object message){
                        foreach(MessageHandler handler in Message.GetInvocationList()){
                                if(handler(message))
                                        break;
                        }
                }
        }

        class TestMain{
                static void Main(string[] args){
                        Chain.Message += StaticState.StringHandler;
                        Chain.Message += NoState.StringHandler2;
                        Chain.Message += NoState.IntegerHandler;
                        Chain.Message += NoState.IntegerHandler;

                        Chain.Process("1st string value");
                        Chain.Process(100);
                        Chain.Process("2nd string value");
                        Chain.Process(4.7f); // not handled
                }
        }
}
```

PHP

```php
<?php
abstract class Logger {
        const ERR = 3;
        const NOTICE = 5;
        const DEBUG = 7;

        protected $mask;
        protected $next; // The next element in the chain of responsibility

        public function setNext(Logger $l) {
                $this->next = $l;
                return $this;
        }

        public function message($msg, $priority) {

                if ($priority <= $this->mask) {
                        $this->writeMessage( $msg );
                }

                if (false == is_null($this->next)) {
                        $this->next->message($msg, $priority);
                }
        }

        abstract public function writeMessage($msg );
}

class DebugLogger extends Logger {
        public function __construct($mask) {
                $this->mask = $mask;
                return $this;
        }

        public function writeMessage($msg ) {
                        echo "Writing to debug output: {$msg}\n";
        }
}

class EmailLogger extends Logger {
        public function __construct($mask) {
                $this->mask = $mask;
                return $this;
        }

        public function writeMessage($msg ) {
                        echo "Sending via email: {$msg}\n";
        }
}

class StderrLogger extends Logger {
        public function __construct($mask) {
                $this->mask = $mask;
                return $this;
        }

        public function writeMessage($msg ) {
                        echo "Writing to stderr: {$msg}\n";
        }
}

class ChainOfResponsibilityExample {
        public function __construct() {
                // build the chain of responsibility
                $l = new DebugLogger(Logger::DEBUG);
                $e = new EmailLogger(Logger::NOTICE);
                $s = new StderrLogger(Logger::ERR);
                $l->setNext( $e->setNext( $s ) );

                $l->message("Entering function y.",             Logger::DEBUG);         // handled b
                $l->message("Step1 completed.",                 Logger::NOTICE);        // handled b
```

```
                $1->message("An error has occurred.",   Logger::ERR);           // handled by all th
        }
}

new ChainOfResponsibilityExample();
```

Visual Prolog

This is a realistic example of protecting the operations on a stream with a named mutex.

First the outputStream interface (a simplified version of the real one):

```
interface outputStream
   predicates
      write : (...).
      writef : (string FormatString [formatString], ...).
end interface outputStream
```

This class encapsulates each of the stream operations the mutex. The **try-finally** construction is used to ensure that the mutex is released no matter how the operation goes (i.e. also in case of exceptions). The underlying mutex object is released by a *finalizer* in the mutex class, and for this example we leave it at that.

```
class outputStream_protected : outputStream
   constructors
      new : (string MutexName, outputStream Stream).
end class outputStream_protected

#include @"pfc\multiThread\multiThread.ph"

implement outputStream_protected
   facts
      mutex : mutex.
      stream : outputStream.

   clauses
      new(MutexName, Stream) :-
         mutex := mutex::createNamed(MutexName, false), % do not take ownership
         stream := Stream.

   clauses
      write(...) :-
         _ = mutex:wait(),  % ignore wait code in this simplified example
         try
            stream:write(...)
        finally
            mutex:release()
        end try.

   clauses
      writef(FormatString, ...) :-
         _ = mutex:wait(),  % ignore wait code in this simplified example
         try
            stream:writef(FormatString, ...)
         finally
            mutex:release()
         end try.
end implement outputStream_protected
```

Usage example.

The **client** uses an **outputStream**. Instead of receiving the **pipeStream** directly, it gets the protected version of it.

```
#include @"pfc\pipe\pipe.ph"

goal
   Stream = pipeStream::openClient("TestPipe"),
   Protected = outputStream_protected::new("PipeMutex", Stream),
   client(Protected),
   Stream:close().
```

ColdFusion

```
<cfcomponent name="AbstractLogger" hint="Base Logging Class">

        <cfset this['ERROR'] = 3 />
        <cfset this['NOTICE'] = 5 />
        <cfset this['DEBUG'] = 7 />
        <cfset mask = 0 />

        <cffunction name="init" access="public" returntype="AbstractLogger" hint="Constructor.">
                <cfreturn this />
        </cffunction>

        <cffunction name="setMask" access="public" returntype="AbstractLogger" output="false" hint="
                <cfargument name="maskName" type="string" required="true" />
                <cfset mask = this[arguments.maskName] />
                <cfreturn this />
        </cffunction>

        <cffunction name="setNext" access="public" returntype="AbstractLogger" output="false" hint="
                <cfargument name="logger" type="AbstractLogger" required="true" />
                <cfset next = arguments.logger />
                <cfreturn this />
        </cffunction>

        <cffunction name="message" access="public" returntype="void" output="true" hint="I generate
                <cfargument name="msg" type="string" required="true" />
                <cfargument name="priority" type="numeric" required="true" />
                <cfset var local = StructNew() />
                <cfif arguments.priority lte mask>
                        <cfset writeMessage(arguments.msg) />
                        <cfif StructKeyExists(variables, 'next')>
                                <cfset next.message(arguments.msg, arguments.priority) />
                        </cfif>
                </cfif>
        </cffunction>

</cfcomponent>

<cfcomponent name="DebugLogger" extends="AbstractLogger" hint="Standard Logger">

        <cffunction name="init" access="public" returntype="DebugLogger" hint="Constructor.">
                <cfreturn this />
        </cffunction>

        <cffunction name="writeMessage" access="public" returntype="void" output="true" hint="Genera
                <cfargument name="msg" type="string" required="true" />
                <cfoutput>Writing to debug: #arguments.msg#</cfoutput>
        </cffunction>

</cfcomponent>

<cfcomponent name="EmailLogger" extends="AbstractLogger" hint="Email Logger">

        <cffunction name="init" access="public" returntype="EmailLogger" hint="Constructor.">
                <cfreturn this />
        </cffunction>
```

```
        <cffunction name="writeMessage" access="public" returntype="void" output="true" hint="Genera
                <cfargument name="msg" type="string" required="true" />
                <cfoutput>Sending via email: #arguments.msg#</cfoutput>
        </cffunction>

</cfcomponent>

<cfcomponent name="ErrorLogger" extends="AbstractLogger" hint="Error Logger">

        <cffunction name="init" access="public" returntype="ErrorLogger" hint="Constructor.">
                <cfreturn this />
        </cffunction>

        <cffunction name="writeMessage" access="public" returntype="void" output="true" hint="Genera
                <cfargument name="msg" type="string" required="true" />
                <cfoutput>Sending to error: #arguments.msg#</cfoutput>
        </cffunction>

</cfcomponent>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html>
<head>
<title>ColdFusion Example of Chain of Responsibility Pattern</title>
</head>

<body>
<h1>ColdFusion Example of Chain of Responsibility Pattern</h1>

<cfset logger = CreateObject('component', 'DebugLogger').init().setMask('DEBUG') />
<cfset emailLogger = CreateObject('component', 'EmailLogger').init().setMask('NOTICE') />
<cfset errorLogger = CreateObject('component', 'ErrorLogger').init().setMask('ERROR') />
<cfset logger.setNext(emailLogger.setNext(errorLogger)) />

<!--- Handled by StdoutLogger --->
<cfset logger.message("Entering function y.", logger['DEBUG']) /><br /><br />
<!--- Handled by StdoutLogger and EmailLogger --->
<cfset logger.message("Step1 completed.", logger['NOTICE']) /><br /><br />
<!--- Handled by all three loggers --->
<cfset logger.message("An error has occurred.", logger['ERROR']) /><br /><br />

</body>
</html>
```

See also

- Single responsibility principle

External links

- Article "The Chain of Responsibility pattern's pitfalls and improvements (http://www.javaworld.com/javaworld/jw-08-2004/jw-0816-chain.html) " by Michael Xinsheng Huang
- Article "Follow the Chain of Responsibility (http://www.javaworld.com/javaworld/jw-08-2003/jw-0829-designpatterns.html) " by David Geary
- Article "Pattern Summaries: Chain of Responsibility (http://developer.com/java/other/article.php/631261) " by Mark Grand
- CoR overview (http://dofactory.com/Patterns/PatternChain.aspx)
- Behavioral Patterns - Chain of Responsibility Pattern (http://allapplabs.com/java_design_patterns/chain_of_responsibility_pattern.htm)

- Descriptions from Portland Pattern Repository (http://c2.com/cgi/wiki?ChainOfResponsibilityPattern)
- Apache Jakarta Commons Chain (http://jakarta.apache.org/commons/chain/)
- PerfectJPattern Open Source Project (http://perfectjpattern.sourceforge.net/dp-chainofresponsibility.html) , Provides a context-free and type-safe implementation of the Chain of Responsibility Pattern in Java
- Chain.NET(NChain) (http://nchain.sourceforge.net) - Ready-to-use, generic and lightweight implementation of the Chain of Responsibility pattern for .NET and Mono

Retrieved from "http://en.wikipedia.org/wiki/Chain-of-responsibility_pattern"
Categories: Software design patterns | Articles with example Java code

---

Command pattern

From Wikipedia, the free encyclopedia

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.

Contents

Uses for the Command pattern

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
: Undo is perhaps even more essential when it's called *rollback* and happens automatically when an operation fails partway through. Installers need this and so do databases. Command objects can also be used to implement two-phase commit.

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` (`http://java.sun.com/javase/6/docs/api/javax/swing/Action.html`) 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.

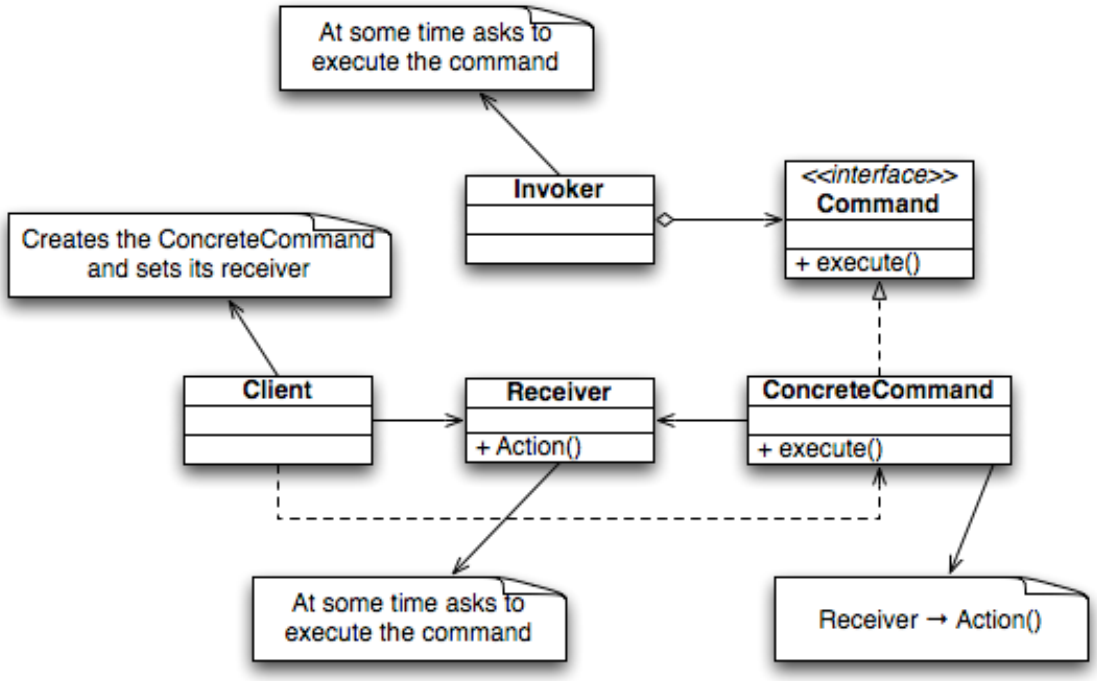
Macro recording
: If all user actions are represented by command objects, a program can record a sequence of actions simply by keeping a list of the command objects as they are executed. It can then "play back" the same actions by executing the same command objects again in sequence. If the program embeds 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



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. The C++ example below is an example of this interpretation. 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. At 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 (http://msdn.microsoft.com/en-us/library/ms752308.aspx) (WPF), introduces routed commands, which combine the command pattern with event processing. As a result the command object no longer contains a reference to the target object nor a reference to the application code. Instead, invoking the command object's *execute* command results in a so called *Executed Routed Event* which during the event's tunneling or bubbling may encounter a so called *binding* object which identifies the target and the application code, which is executed at that point.

Examples

C++

Consider a simple recipe making program. Every command could be represented as an object. Then if the user makes a mistake then he/she can undo the command by removing that object from the stack

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

class Command{
  public:
```

```
   virtual void execute(void) =0;
   virtual ~Command(void){};
};

class Ingredient : public Command {
  public:
   Ingredient(string amount, string ingredient){
      _ingredient = ingredient;
      _amount = amount;
   }
   void execute(void){
      cout << " *Add " << _amount << " of " << _ingredient << endl;
   }
  private:
   string _ingredient;
   string _amount;
};

class Step : public Command {
  public:
   Step(string action, string time){
      _action= action;
      _time= time;
   }
   void execute(void){
      cout << " *" << _action << " for " << _time << endl;
   }
  private:
   string _time;
   string _action;
};

class CmdStack{
  public:
   void add(Command *c) {
      commands.push_back(c);
   }
   void createRecipe(void){
      for(vector<Command*>::size_type x=0;x<commands.size();x++){
         commands[x]->execute();
      }
   }
   void undo(void){
      if(commands.size() > 0) {
         commands.pop_back();
      }
      else {
         cout << "Can't undo" << endl;
      }
   }
  private:
   vector<Command*> commands;
};

int main(void) {
   CmdStack list;

   //Create ingredients
   Ingredient first("2 tablespoons", "vegetable oil");
   Ingredient second("3 cups", "rice");
   Ingredient third("1 bottle","Ketchup");
   Ingredient fourth("4 ounces", "peas");
   Ingredient fifth("1 teaspoon", "soy sauce");

   //Create Step
   Step step("Stir-fry","3-4 minutes");

   //Create Recipe
   cout << "Recipe for simple Fried Rice" << endl;
   list.add(&first);
   list.add(&second);
   list.add(&step);
   list.add(&third);
   list.undo();
   list.add(&fourth);
   list.add(&fifth);
```

```
    list.createRecipe();
    cout << "Enjoy!" << endl;
    return 0;
}
```

Java

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.

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

    private Command flipUpCommand;
    private Command flipDownCommand;

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

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

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

}

/*Receiver class*/

public class Light{

     public Light(){  }

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

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

}


/*the Command interface*/

public interface Command{
    void execute();
}


/*the Command for turning on the light*/

public class TurnOnLightCommand implements Command{

   private Light theLight;

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

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

}

/*the Command for turning off the light*/

public class TurnOffLightCommand implements Command{

   private Light theLight;

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

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

}

/*The test class*/
public class TestCommand{

   public static void main(String[] args){
       Light l=new Light();
       Command switchUp=new TurnOnLightCommand(l);
       Command switchDown=new TurnOffLightCommand(l);

       Switch s=new Switch(switchUp,switchDown);

       s.flipUp();
       s.flipDown();
   }

}
```

See also

- Closure
- Model-view-controller
- Function object

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.martinfowler.com/eaaCatalog/unitOfWork.html
- http://www.javaworld.com/javaworld/javatips/jw-javatip68.html
- http://www.microsoft.com/belux/msdn/nl/community/columns/jdruyts/wpf_commandpattern.mspx (Windows_Presentation_Foundation)
- Microsoft's Windows Presentation Foundation Commanding OverView (http://windowssdk.msdn.microsoft.com/en-us/library/ms752308.aspx)
- PerfectJPattern Open Source Project (http://perfectjpattern.sourceforge.net/dp-command.html) ,

Provides a componentized i.e. context-free and type-safe implementation of the Command Pattern in Java
- Jt (http://www.fsw.com/Jt/Jt.htm) J2EE Pattern Oriented Framework

Retrieved from "http://en.wikipedia.org/wiki/Command_pattern"
Category: Software design patterns

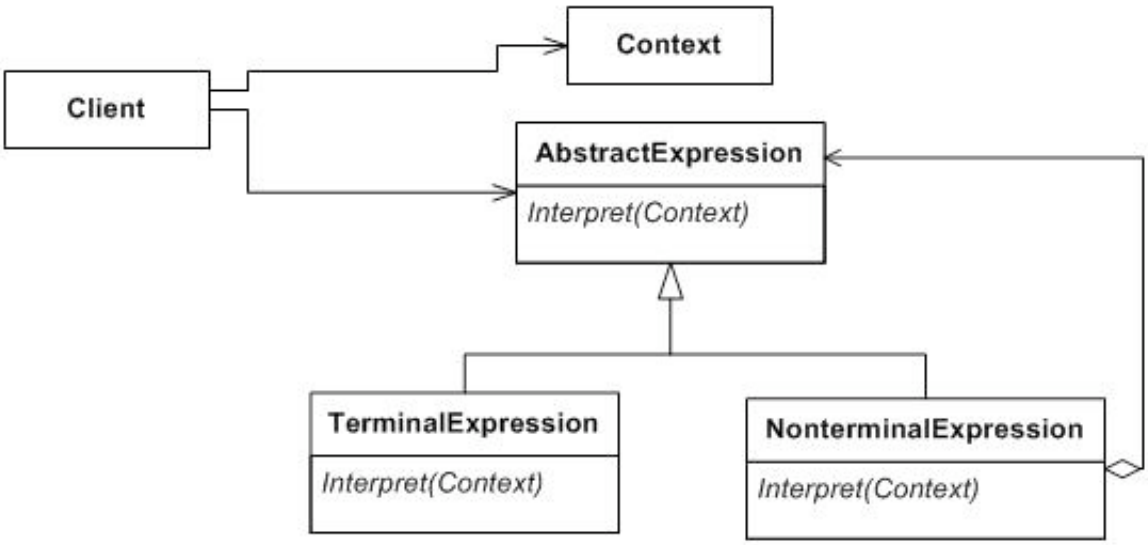
Interpreter pattern

From Wikipedia, the free encyclopedia

In computer programming, the **interpreter pattern** is a particular design pattern. The basic idea is to have a class for each symbol (terminal or nonterminal) in a specialized computer language, so that the language's syntax tree is an instance of the composite pattern. The interpreter pattern specifies how to evaluate language constructs.

Contents

Structure



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.

Examples

Java

The following Java example illustrates how a general purpose language would interpret a more specialized language, here the Reverse Polish notation. The output is:

```
'42 4 2 - +' equals 44
```

```
import java.util.*;

interface Expression {
    public void interpret(Stack<Integer> s);
}

class TerminalExpression_Number implements Expression {
    private int number;
    public TerminalExpression_Number(int number)       { this.number = number; }
    public void interpret(Stack<Integer> s)  { s.push(number); }
}

class TerminalExpression_Plus implements Expression {
    public void interpret(Stack<Integer> s)  { s.push( s.pop() + s.pop() ); }
}

class TerminalExpression_Minus implements Expression {
    public void interpret(Stack<Integer> s)  { s.push( - s.pop() + s.pop() ); }
}

class Parser {
    private ArrayList<Expression> parseTree = new ArrayList<Expression>(); // only one NonTerminal E

    public Parser(String s) {
        for (String token : s.split(" ")) {
            if      (token.equals("+")) parseTree.add( new TerminalExpression_Plus() );
            else if (token.equals("-")) parseTree.add( new TerminalExpression_Minus() );
            // ...
            else                        parseTree.add( new TerminalExpression_Number(Integer.valueOf
        }
    }

    public int evaluate() {
        Stack<Integer> context = new Stack<Integer>();
        for (Expression e : parseTree) e.interpret(context);
        return context.pop();
    }
}

class InterpreterExample {
    public static void main(String[] args) {
        String expression = "42 4 2 - +";
        Parser p = new Parser(expression);
        System.out.println("'" + expression +"' equals " + p.evaluate());
    }
}
```

See also

- Backus-Naur form
- Domain specific languages
- *Design Patterns* p. 243

Categories: Software design patterns | Computer science stubs | Articles with example Java code
Hidden categories: Articles lacking sources from November 2008 | All articles lacking sources

Iterator pattern

From Wikipedia, the free encyclopedia

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

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

Contents

Examples

PHP 5

As a default behavior in PHP 5, using an object in a foreach structure will traverse all public values. Multiple Iterator classes are available with PHP to allow you to iterate through common lists, such as directories, XML structures and recursive arrays.

It's possible to define your own Iterator classes by implementing the Iterator interface, which will override the default behavior.

The Iterator interface definition:

```
interface Iterator
{
    // Returns the current value
    function current();

    // Returns the current key
    function key();

    // Moves the internal pointer to the next element
```

```
    function next();

    // Moves the internal pointer to the first element
    function rewind();

    // If the current element is not at all valid (boolean)
    function valid();
}
```

These methods are all being used in a complete `foreach( $object as $key=>$value )` sequence. The methods are executed in the following order:

```
rewind()
while valid() {
    current() in $value
    key() in $key
    next()
}
End of Loop
```

According to Zend, the `current()` method is called before and after the valid() method.

C++

The following C++ program gives the implementation of iterator design pattern with generic C++ template:

```
/*************************************************************************/
/* Iterator.h                                                            */
/*************************************************************************/
#ifndef MY_ITERATOR_HEADER
#define MY_ITERATOR_HEADER

////////////////////////////////////////////////////////////////////////
template<class T,class U>
class Iterator
{
public:
        typedef std::vector<T>::iterator iter_type;
        Iterator(U*  pData):m_pData(pData){
                m_it = m_pData->m_data.begin();
        }

        void first()
        {
                m_it = m_pData->m_data.begin();
        }

        void next()
        {
                m_it++;
        }

        bool isDone()
        {
                return (m_it == m_pData->m_data.end());
        }

        iter_type current()
        {
                return m_it;
        }
private:
        U* m_pData;
        iter_type m_it;
};
```

```
template<class T,class U,class A>
class setIterator
{
public:
        typedef std::set<T,U>::iterator iter_type;

        setIterator(A* pData):m_pData(pData)
        {
                m_it = m_pData->m_data.begin();
        }

        void first()
        {
                m_it = m_pData->m_data.begin();
        }

        void next()
        {
                m_it++;
        }

        bool isDone()
        {
                return (m_it == m_pData->m_data.end());
        }

        iter_type current()
        {
                return m_it;
        }

private:
        A*                              m_pData;
        iter_type               m_it;
};
#endif
```

```
/*****************************************************************************/
/* Aggregate.h                                                               */
/*****************************************************************************/
#ifndef MY_DATACOLLECTION_HEADER
#define MY_DATACOLLECTION_HEADER

#include "MyIterator.h"

template <class T>
class aggregate
{
        friend Iterator<T,aggregate>;
public:
        void add(T a)
        {
                m_data.push_back(a);
        }

        Iterator<T,aggregate>* create_iterator()
        {
                return new Iterator<T,aggregate>(this);
        }


private:
        std::vector<T> m_data;
};
template <class T,class U>
class aggregateSet
{
        friend setIterator<T,U,aggregateSet>;
public:
        void add(T a)
        {
                m_data.insert(a);
```

```
        }

        setIterator<T,U,aggregateSet>* create_iterator()
        {
                return new setIterator<T,U,aggregateSet>(this);
        }

        void Print()
        {
                copy(m_data.begin(),m_data.end(),ostream_iterator<T>(cout,"\r\n"));
        }

protected:
private:
        std::set<T,U> m_data;
};

#endif
```

```
/*************************************************************************/
/* Iterator Test.cpp                                                     */
/*************************************************************************/
#include <iostream>
#include <vector>
using namespace std;

class Money
{
public:
        Money(int a=0):m_data(a){}

        void SetMoney(int a){
                m_data = a;
        }

        int GetMoney()
        {
                return m_data;
        }

protected:
private:
        int m_data;
};

class Name
{
public:
        Name(string name):m_name(name){}

        const string& GetName() const{
                return m_name;
        }

        friend ostream operator<<(ostream cout,Name name)
        {
                cout<<name.GetName();
                return cout;
        }

private:
        string m_name;
};

struct NameLess
{
        bool operator() ( const Name& lhs, const Name& rhs) const
        {
                return lhs.GetName() < rhs.GetName();
        }
};

int main( void ) {
```

```
		//sample 1
	cout<<"_________________Iterator with int_____________________________________"<<endl;
	aggregate<int> agg;

	for(int i=0;i<10;i++)
	{
		agg.add(i);
	}


	Iterator< int,aggregate<int> > *it = agg.create_iterator();
	for(it->first();!it->isDone();it->next())
	{
		cout<<*it->current()<<endl;
	}


	//sample 2
	aggregate<Money> agg2;
	Money a(100),b(1000),c(10000);
	agg2.add(a);
	agg2.add(b);
	agg2.add(c);

	cout<<"_________________Iterator with Class Money_____________________________"<<endl;
	Iterator< Money,aggregate<Money> > *it2 = agg2.create_iterator();
	it2->first();
	while (!it2->isDone()) {
		cout<<((Money*)(it2->current()))->GetMoney()<<endl;
		it2->next();
	}

	//sample 3
	cout<<"_________________Set Iterator with Class Name_____________________________"<<endl;

	aggregateSet<Name,NameLess> aset;
	aset.add(Name("Qmt"));
	aset.add(Name("Bmt"));
	aset.add(Name("Cmt"));
	aset.add(Name("Amt"));


	setIterator<Name,NameLess,aggregateSet<Name,NameLess>  > * it3 = aset.create_iterator();
	for(it3->first();!it3->isDone();it3->next())
	{
		cout<<(*it3->current())<<endl;
	}
}
```

Console output:

```
_________________Iterator with int_____________________________________
0
1
2
3
4
5
6
7
8
9
_________________Iterator with Class Money_____________________________
100
1000
10000
_________________Set Iterator with Class Name_____________________________
Amt
Bmt
Cmt
Qmt
```

Ruby

Iterate over the members of an array.

```
numbers = [1,2,3,4,5,6]
numbers.each {|i| puts "I got #{i}"}
```

Ouputs:

```
I got 1
I got 2
I got 3
I got 4
I got 5
I got 6
```

See also

- Iterator
- Composite Pattern
- Design Pattern

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#
- Jt (http://www.fsw.com/Jt/Jt.htm) J2EE Pattern Oriented Framework
- Iterator pattern in UML and in LePUS3 (a formal modelling language) (http://www.lepus.org.uk/ref/companion/Iterator.xml)

Mediator pattern

From Wikipedia, the free encyclopedia

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

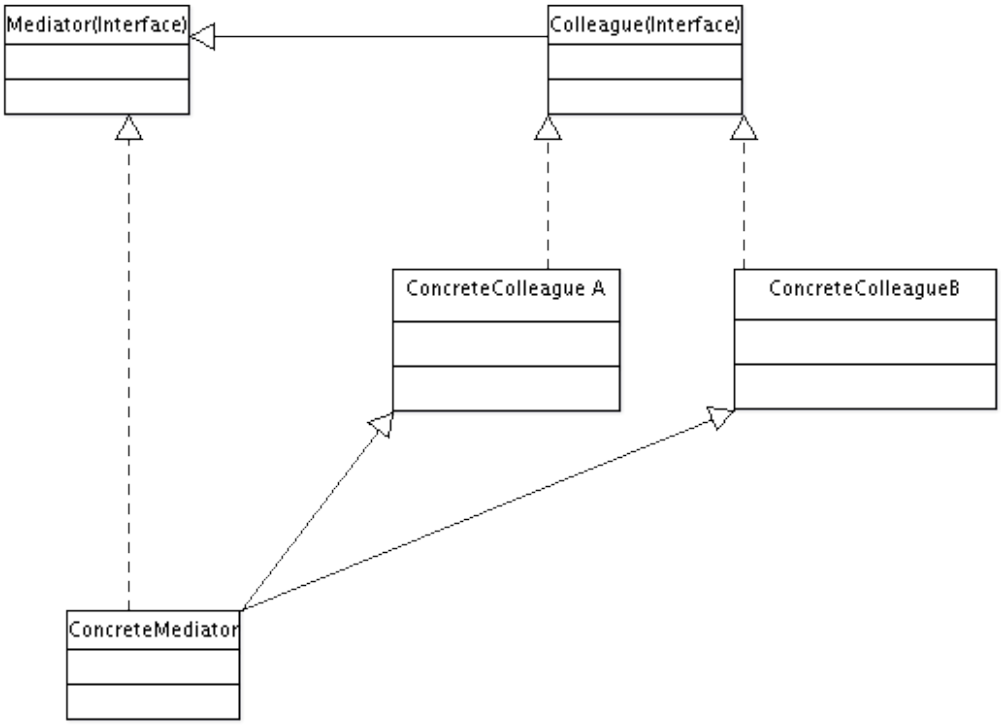
Usually a program is made up of a (sometimes large) number of classes. So the logic and computation is distributed among these classes. However, as more classes are developed in a program, especially during maintenance and/or refactoring, the problem of communication between these classes may become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

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

Contents

Structure

The class diagram of this design pattern is as shown below:



Note: the arrows pointing from the mediator to the concrete colleagues should be association arrows, not inheritance arrows. See the source code below.

Code Examples

C#

```
using System;
using System.Collections;

namespace DoFactory.GangOfFour.Mediator.Structural
{

  // Mainapp test application

  class MainApp
  {
    static void Main()
    {
      ConcreteMediator m = new ConcreteMediator();

      ConcreteColleague1 c1 = new ConcreteColleague1(m);
      ConcreteColleague2 c2 = new ConcreteColleague2(m);

      m.Colleague1 = c1;
      m.Colleague2 = c2;

      c1.Send("How are you?");
      c2.Send("Fine, thanks");

      // Wait for user
      Console.Read();
    }
  }

  // "Mediator"

  abstract class Mediator
  {
    public abstract void Send(string message,
      Colleague colleague);
  }

  // "ConcreteMediator"

  class ConcreteMediator : Mediator
```

```
{
  private ConcreteColleague1 colleague1;
  private ConcreteColleague2 colleague2;

  public ConcreteColleague1 Colleague1
  {
    set{ colleague1 = value; }
  }

  public ConcreteColleague2 Colleague2
  {
    set{ colleague2 = value; }
  }

  public override void Send(string message,
    Colleague colleague)
  {
    if (colleague == colleague1)
    {
      colleague2.Notify(message);
    }
    else
    {
      colleague1.Notify(message);
    }
  }
}

// "Colleague"

abstract class Colleague
{
  protected Mediator mediator;

  // Constructor
  public Colleague(Mediator mediator)
  {
    this.mediator = mediator;
  }
}

// "ConcreteColleague1"

class ConcreteColleague1 : Colleague
{
  // Constructor
  public ConcreteColleague1(Mediator mediator)
    : base(mediator)
  {
  }

  public void Send(string message)
  {
    mediator.Send(message, this);
  }

  public void Notify(string message)
  {
    Console.WriteLine("Colleague1 gets message: "
      + message);
  }
}

// "ConcreteColleague2"

class ConcreteColleague2 : Colleague
{
  // Constructor
  public ConcreteColleague2(Mediator mediator)
    : base(mediator)
  {
  }

  public void Send(string message)
  {
    mediator.Send(message, this);
  }

  public void Notify(string message)
  {
    Console.WriteLine("Colleague2 gets message: "
      + message);
```

```
    }
  }
}
```

Java

```
interface Command {
    void execute();
}

class Mediator {

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

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

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

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

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

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

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

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

}

class BtnView extends JButton implements Command {

    Mediator med;

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

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

}

class BtnSearch extends JButton implements Command {

    Mediator med;

    BtnSearch(ActionListener al, Mediator m) {
```

```
        super("Search");
        addActionListener(al);
        med = m;
        med.registerSearch(this);
    }

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

}

class BtnBook extends JButton implements Command {

    Mediator med;

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

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

}

class LblDisplay extends JLabel {

    Mediator med;

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

}

class MediatorDemo extends JFrame implements ActionListener {

    Mediator med = new Mediator();

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

    public void actionPerformed(ActionEvent ae) {
        Command 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*

See also

- Data mediation
- Design Patterns, the book which gave rise to the study of design patterns in computer science
- Design pattern (computer science), a standard solution to common problems in software design

External links

- Mediator Pattern (http://www.javacamp.org/designPattern/mediator.html) in Java
- Mediator Pattern (http://www.dofactory.com/Patterns/PatternMediator.aspx) in C#
- Jt (http://www.fsw.com/Jt/Jt.htm) JEE Pattern Oriented Framework

Memento pattern

From Wikipedia, the free encyclopedia

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

The memento pattern is used by two objects: the *originator* and a *caretaker*. The originator is some object that has an internal state. The caretaker is going to do something to the originator, but wants to be able to undo the change. The caretaker first asks the originator for a memento object. Then it does whatever operation (or sequence of operations) it was going to do. To roll back to the state before the operations, it returns the memento object to the originator. The memento object itself is an opaque object (one which the caretaker can not, 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 and the state in a finite state machine.

Contents

Examples

Actionscript 3

The following Actionscript 3 program illustrates the Memento Pattern.

```
package
{
	import flash.display.Sprite;

	/**
	* Memento
	*/
	public class As3Memento extends Sprite
	{
		function As3Memento()
		{
			var ct:CareTaker = new CareTaker();
			var originator:Originator = new Originator();
```

```
                originator.A = 'letter A';
                originator.B = 'letter B';
                originator.C = 'letter C';

                ct.addMementoFrom(originator);

                originator.A = 'anything...';
                originator.B = 'blah blah...';
                originator.C = 'etc...';
                ct.addMementoFrom(originator);

                originator.restoreFromMemento(ct.getMemento(0));

                trace(originator.A, originator.B, originator.C);

                originator.restoreFromMemento(ct.getMemento(1));

                trace(originator.A, originator.B, originator.C);

        }
    }

}

interface IMemento
{
    function getMemento():Object;
    function restoreFromMemento(obj:Object):void;
}

class Originator implements IMemento
{
    public var A:String;
    public var B:String;
    public var C:String;

    function Originator()
    { }

    /* INTERFACE IMemento */

    public function getMemento():Object
    {
        return { _A:A, _B:B, _C:C };
    }

    public function restoreFromMemento(obj:Object):void
    {
        A = obj._A;
        B = obj._B;
        C = obj._C;
    }

}

class CareTaker
{
    private var mementos:Array;
    function CareTaker()
    {
        mementos = new Array();
    }

    public function addMementoFrom( object:IMemento ):void
    {
        mementos.push( object.getMemento() );
    }

    public function getMemento(index:uint):Object
    {
        return mementos[index];
    }
}
```

Java

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

```
class Originator {

    private String state;
    /* lots of memory consumptive private data that is not necessary to define the
     * state and should thus not be saved. Hence the small memento object. */

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

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

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

    private static class Memento {

        private String state;

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

        public String getSavedState() {
            return state;
        }

    }

}

import java.util.*;

class Caretaker {

    private List<Object> savedStates = new ArrayList<Object>();

    public void addMemento(Object m) {
        savedStates.add(m);
    }

    public Object getMemento(int index) {
        return savedStates.get(index);
    }

}

class MementoExample {

    public static void main(String[] args) {

        Caretaker caretaker = new Caretaker();

        Originator originator = new Originator();
        originator.set("State1");
        originator.set("State2");
        caretaker.addMemento(originator.saveToMemento());
        originator.set("State3");
```

```
        caretaker.addMemento(originator.saveToMemento());
        originator.set("State4");

        originator.restoreFromMemento(caretaker.getMemento(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
```

Ruby

The following Ruby program illustrates the same pattern.

```
#!/usr/bin/env ruby -KU

require 'rubygems'
require 'spec'

class Originator
  class Memento
    def initialize(state)
      # dup required so that munging of the Originator's original state doesn't mess up
      # this Memento for first or subsequent restore
      @state = state.dup
    end

    def state
      # dup required so that munging of the Originator's restored state doesn't mess up
      # this Memento for a second restore
      @state.dup
    end
  end

  attr_accessor :state

  # pretend there's lots of additional memory-heavy data, which can be reconstructed
  # from state
  def save_to_memento
    Memento.new(@state)
  end

  def restore_from_memento(m)
    @state = m.state
  end

end

class Caretaker < Array; end

describe Originator do
  before(:all) do
    @caretaker = Caretaker.new
    @originator = Originator.new

    @originator.state = "State1"
  end

  it "should have original state" do
```

```
    @originator.state.should == 'State1'
  end

  it "should update state" do
    @originator.state = "State2"
    @originator.state.should == 'State2'
  end

  it "should save memento" do
    @caretaker << @originator.save_to_memento
    @caretaker.size.should == 1
  end

  it "should update state after save to memento" do
    @originator.state = "State3"
    @originator.state.should == 'State3'
  end

  it "should save to memento again" do
    @caretaker << @originator.save_to_memento
    @caretaker.size.should == 2
  end

  it "should update state after save to memento again" do
    @originator.state = "State4";
    @originator.state.should == 'State4'
  end

  it "should restore to original save point" do
    @originator.restore_from_memento @caretaker[0]
    @originator.state.should == 'State2'
  end

  it "should restore to second save point" do
    @originator.restore_from_memento @caretaker[1]
    @originator.state.should == 'State3'
  end

  it "should restore after pathological munging of restored state" do
    @originator.state[-1] = '5'
    @originator.state.should == 'State5'
    @originator.restore_from_memento @caretaker[1]
    @originator.state.should == 'State3'
  end

  it "should restore after pathological munging of original state" do
    @originator.state = "State6"
    @originator.state.should == 'State6'
    @caretaker << @originator.save_to_memento
    @originator.state[-1] = '7'
    @originator.state.should == 'State7'
    @originator.restore_from_memento @caretaker[2]
    @originator.state.should == 'State6'
  end
end
```

See also

- Facade pattern

External links

- Description (http://adapower.com/index.php?Command=Class&ClassID=Patterns&CID=271) by Matthew Heaney
- Memento UML Class Diagram (http://dofactory.com/Patterns/PatternMemento.aspx) with C# and .NET code samples

---

Null Object pattern

From Wikipedia, the free encyclopedia

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].

Contents

Motivation

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

Alternate Patterns

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, the processing of binary search trees, including operations such as insertion, deletion, and lookup, involves many nullity check of the encountered pointers. These checks can be implemented within the special object pattern.

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

Relation to other patterns

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

It is not a pattern from Design Patterns, but is mentioned in Martin Fowler's *Refactoring*[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]

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.

External links

- Jeffrey Walkers' account of the Null Object Pattern (http://www.cs.oberlin.edu/~jwalker/nullObjPattern/)
- Antonio Garcias' account of the Null Object Pattern (http://exciton.cs.rice.edu/javaresources/DesignPatterns/NullPattern.htm)
- Martin Fowlers' description of Special Case, a slightly more general pattern (http://martinfowler.com/eaaCatalog/specialCase.html)
- Null Object Pattern Revisited (http://www.owlnet.rice.edu/~comp212/00-spring/handouts/week06/null_object_revisited.htm)
- Introduce Null Object refactoring (http://www.refactoring.com/catalog/introduceNullObject.html)
- Jt (http://www.fsw.com/Jt/Jt.htm) J2EE Pattern Oriented Framework

Observer pattern

From Wikipedia, the free encyclopedia

The **observer pattern** (sometimes known as publish/subscribe) 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.

Contents

Participant classes

The participants of the pattern are detailed below. Member functions are listed with bullets.

Subject

This abstract class provides an interface for attaching and detaching observers. Subject class also holds a private list of observers. Contains these functions (methods):

- *Attach* - Adds a new observer to the list of observers observing the subject.
- *Detach* - Removes an existing observer from the list of observers observing the subject
- *Notify* - Notifies each observer by calling the *Update* function in the observer, when a change

occurs.

The *Attach* function has an observer as argument. This may be either a virtual function of the observer class (*Update* in this description) or a function pointer (more generally a function object or functor) in a non object oriented setting.

ConcreteSubject

The class provides the state of interest to observers. It also sends a notification to all observers, by calling the Notify function in its superclass or base class (i.e., in the Subject class). Contains this function:

- *GetState* - Returns the state of the subject.

Observer

This class defines an updating interface for all observers, to receive update notification from the subject. The Observer class is used as an abstract class to implement concrete observers. Contains this function:

- *Update* - An abstract function, to be overridden by concrete observers.

ConcreteObserver

This class maintains a reference with the ConcreteSubject, to receive the state of the subject when a notification is received. Contains this function:

- *Update* - This is the overridden function in the concrete class. When this function is called by the subject, the ConcreteObserver calls the GetState function of the subject to update the information it has about the subject's state.

Each concrete observer implements the update function and as a consequence defines its own behavior when the notification occurs.

When a change occurs to the (concrete) subject, it sends a notification to all observers, by calling the *Notify* function. The *Notify* function then calls the *Update* function of all attached (concrete) observers. The *Notify* and *Update* functions may have parameters indicating what kind of change has occurred to the subject. This allows for optimizations in the observer (only updating those parts that changed).

Typical usages

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects allows the programmer to vary and reuse them independently.
- When a change to one object requires changing others, and it's not known in advance how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are.

The observer pattern is also very often associated with the model-view-controller (MVC) paradigm. In MVC, the observer pattern is used to create a loose coupling between the model and the view. Typically, a modification in the model triggers the notification of model observers which are actually the views.

An example is Java Swing, in which the model is expected to issue change notifications to the views via the *PropertyChangeNotification* infrastructure. Model classes are Java beans that behave as the subject, described above. View classes are associated with some visible item on the GUI and behave as the observers, described above. As the application runs, changes are made to the model. The user sees these changes because the views are updated accordingly.

Examples

Python

The observer pattern in Python:

```
class AbstractSubject:
    def register(self, listener):
        raise NotImplementedError("Must subclass me")

    def unregister(self, listener):
        raise NotImplementedError("Must subclass me")

    def notify_listeners(self, event):
        raise NotImplementedError("Must subclass me")

class Listener:
    def __init__(self, name, subject):
        self.name = name
        subject.register(self)

    def notify(self, event):
        print self.name, "received event", event

class Subject(AbstractSubject):
    def __init__(self):
        self.listeners = []
        self.data = None

    def getUserAction(self):
        self.data = raw_input('Enter something to do:')
        return self.data

    # Implement abstract Class AbstractSubject

    def register(self, listener):
        self.listeners.append(listener)

    def unregister(self, listener):
        self.listeners.remove(listener)

    def notify_listeners(self, event):
        for listener in self.listeners:
            listener.notify(event)


if __name__=="__main__":
    # make a subject object to spy on
    subject = Subject()

    # register two listeners to monitor it.
    listenerA = Listener("<listener A>", subject)
    listenerB = Listener("<listener B>", subject)
```

```
    # simulated event
    subject.notify_listeners ("<event 1>")
    # outputs:
    #     <listener A> received event <event 1>
    #     <listener B> received event <event 1>

    action = subject.getUserAction()
    subject.notify_listeners(action)
    #Enter something to do:hello
    # outputs:
    #     <listener A> received event hello
    #     <listener B> received event hello
```

The observer pattern can be implemented more succinctly in Python using function decorators.

Java

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` and `java.util.Observable`. When a string is supplied from System.in, the method `notifyObserver` is then called, in order to notify all observers of the event's occurrence, in the form of an invocation of their 'update' methods - in our example, `ResponseHandler.update(...)`.

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

```
/* File Name : EventSource.java */

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

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

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

package obs;

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

public class ResponseHandler implements Observer
{
```

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

```
/* Filename : myapp.java */
/* This is main program */

package obs;

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

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

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

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

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

C# - Traditional Method

C# and the other .NET Framework languages do not typically require a full implementation of the Observer pattern using interfaces and concrete objects. Here is an example of using them, however.

```
using System;
using System.Collections;

namespace Wikipedia.Patterns.Observer
{
        // IObserver --> interface for the observer
        public interface IObserver
        {
                // called by the subject to update the observer of any change
                // The method parameters can be modified to fit certain criteria
                void Update(string message);
        }

        public class Subject
        {
                // use array list implementation for collection of observers
                private ArrayList observers;

                // constructor
                public Subject()
                {
                        observers = new ArrayList();
                }

                public void Register(IObserver observer)
```

```
        {
                // if list does not contain observer, add
                if (!observers.Contains(observer))
                {
                        observers.Add(observer);
                }
        }

        public void Unregister(IObserver observer)
        {
                // if observer is in the list, remove
                if (observers.Contains(observer))
                {
                        observers.Remove(observer);
                }
        }

        public void Notify(string message)
        {
                // call update method for every observer
                foreach (IObserver observer in observers)
                {
                        observer.Update(message);
                }
        }
}

// Observer1 --> Implements the IObserver
public class Observer1 : IObserver
{
        public void Update(string message)
        {
                Console.WriteLine("Observer1:" + message);
        }
}

// Observer2 --> Implements the IObserver
public class Observer2 : IObserver
{
        public void Update(string message)
        {
                Console.WriteLine("Observer2:" + message);
        }
}

// Test class
public class ObserverTester
{
        [STAThread]
        public static void Main()
        {
                Subject mySubject = new Subject();
                IObserver myObserver1 = new Observer1();
                IObserver myObserver2 = new Observer2();

                // register observers
                mySubject.Register(myObserver1);
                mySubject.Register(myObserver2);

                mySubject.Notify("message 1");
                mySubject.Notify("message 2");
        }
}
}
```

C# - Using Events

The alternative to using concrete and abstract observers and publishers in C# and other .NET Framework languages, such as Visual Basic, is to use events. The event model is supported via delegates that define the method signature that should be used to capture events. Consequently,

delegates provide the mediation otherwise provided by the abstract observer, the methods themselves provide the concrete observer, the concrete subject is the class defining the event, and the subject is the event system built into the base class library. It is the preferred method of accomplishing the Observer pattern in .NET applications.

```
using System;

// First, declare a delegate type that will be used to fire events.
//  This is the same delegate as System.EventHandler.
//  This delegate serves as the abstract observer.
//  It does not provide the implementation, but merely the contract.
public delegate void EventHandler(object sender, EventArgs e);

// Next, declare a published event.  This serves as the concrete subject.
//  Note that the abstract subject is handled implicitly by the runtime.
public class Button
{
    // The EventHandler contract is part of the event declaration.
    public event EventHandler Clicked;

    // By convention, .NET events are fired from descendant classes by a virtual method,
    //  allowing descendant classes to handle the event invocation without subscribing
    //  to the event itself.
    protected virtual void OnClicked(EventArgs e)
    {
        if (Clicked != null)
            Clicked(this, e); // implicitly calls all observers/subscribers
    }
}

// Then in an observing class, you are able to attach and detach from the events:
public class Window
{
    private Button okButton;

    public Window()
    {
        okButton = new Button();
        // This is an attach function.  Detaching is accomplished with -=.
        // Note that it is invalid to use the assignment operator - events are multicast
        //  and can have multiple observers.
        okButton.Clicked += new EventHandler(okButton_Clicked);
    }

    private void okButton_Clicked(object sender, EventArgs e)
    {
        // This method is called when Clicked(this, e) is called within the Button class
        //  unless it has been detached.
    }
}
```

C++

```
#include <list>
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

// The Abstract Observer
class ObserverBoardInterface
{
public:
    virtual void update(float a,float b,float c) = 0;
};

// Abstract Interface for Displays
class DisplayBoardInterface
```

```
{
public:
    virtual void show() = 0;
};

// The Abstract Subject
class WeatherDataInterface
{
public:
    virtual void registerOb(ObserverBoardInterface* ob) = 0;
    virtual void removeOb(ObserverBoardInterface* ob) = 0;
    virtual void notifyOb() = 0;
};

// The Concrete Subject
class ParaWeatherData: public WeatherDataInterface
{
public:
    void SensorDataChange(float a,float b,float c)
    {
        m_humidity = a;
        m_temperature = b;
        m_pressure = c;
        notifyOb();
    }

    void registerOb(ObserverBoardInterface* ob)
    {
        m_obs.push_back(ob);
    }

    void removeOb(ObserverBoardInterface* ob)
    {
        m_obs.remove(ob);
    }
protected:
    void notifyOb()
    {
        list<ObserverBoardInterface*>::iterator pos = m_obs.begin();
        while (pos != m_obs.end())
        {
            ((ObserverBoardInterface* )(*pos))->update(m_humidity,m_temperature,m_pressure);
            (dynamic_cast<DisplayBoardInterface*>(*pos))->show();
            ++pos;
        }
    }

private:
    float        m_humidity;
    float        m_temperature;
    float        m_pressure;
    list<ObserverBoardInterface* > m_obs;
};

// A Concrete Observer
class CurrentConditionBoard : public ObserverBoardInterface, public DisplayBoardInterface
{
public:
    CurrentConditionBoard(ParaWeatherData& a):m_data(a)
    {
        m_data.registerOb(this);
    }
    void show()
    {
        cout<<"_____CurrentConditionBoard_____"<<endl;
        cout<<"humidity: "<<m_h<<endl;
        cout<<"temperature: "<<m_t<<endl;
        cout<<"pressure: "<<m_p<<endl;
        cout<<"_____________________________"<<endl;
    }

    void update(float h, float t, float p)
    {
        m_h = h;
        m_t = t;
```

```
        m_p = p;
    }

private:
    float m_h;
    float m_t;
    float m_p;
    ParaWeatherData& m_data;
};

// A Concrete Observer
class StatisticBoard : public ObserverBoardInterface, public DisplayBoardInterface
{
public:
    StatisticBoard(ParaWeatherData& a):m_maxt(-1000),m_mint(1000),m_avet(0),m_count(0),m_data(a)
    {
        m_data.registerOb(this);
    }

    void show()
    {
        cout<<"________StatisticBoard_________"<<endl;
        cout<<"lowest  temperature: "<<m_mint<<endl;
        cout<<"highest temperature: "<<m_maxt<<endl;
        cout<<"average temperature: "<<m_avet<<endl;
        cout<<"_____________________________"<<endl;
    }

    void update(float h, float t, float p)
    {
        ++m_count;
        if (t>m_maxt)
        {
            m_maxt = t;
        }
        if (t<m_mint)
        {
            m_mint = t;
        }
        m_avet = (m_avet * (m_count-1) + t)/m_count;
    }

private:
    float m_maxt;
    float  m_mint;
    float m_avet;
    int m_count;
    ParaWeatherData& m_data;
};


int main(int argc, char *argv[])
{

    ParaWeatherData * wdata = new ParaWeatherData;
    CurrentConditionBoard* currentB = new CurrentConditionBoard(*wdata);
    StatisticBoard* statisticB = new StatisticBoard(*wdata);

    wdata->SensorDataChange(10.2, 28.2, 1001);
    wdata->SensorDataChange(12, 30.12, 1003);
    wdata->SensorDataChange(10.2, 26, 806);
    wdata->SensorDataChange(10.3, 35.9, 900);

    wdata->removeOb(currentB);

    wdata->SensorDataChange(100, 40, 1900);

    delete statisticB;
    delete currentB;
    delete wdata;

    return 0;
}
```

AS3

```
// Main Class
package {
    import flash.display.MovieClip;

    public class Main extends MovieClip {
        private var _cs:ConcreteSubject = new ConcreteSubject();
        private var _co1:ConcreteObserver1 = new ConcreteObserver1();
        private var _co2:ConcreteObserver2 = new ConcreteObserver2();

        public function Main() {
            _cs.registerObserver(_co1);
            _cs.registerObserver(_co2);

            _cs.changeState(10);
            _cs.changeState(99);

            _cs.unRegisterObserver(_co1);

            _cs.changeState(17);

            _co1 = null;
        }
    }
}

// Interface Subject
package {
    public interface ISubject {
        function registerObserver(o:IObserver):void;

        function unRegisterObserver(o:IObserver):void;

        function updateObservers():void;

        function changeState(newState:uint):void;
    }
}

// Interface Observer
package {
    public interface IObserver {
        function update(newState:uint):void;
    }
}

// Concrete Subject
package {
    import flash.utils.setInterval;

    public class ConcreteSubject implements ISubject {
        private var _observersList:Array = new Array();
        private var _currentState:uint;

        public function ConcreteSubject() {
        }

        public function registerObserver(o:IObserver):void {
            _observersList.push( o );
            _observersList[_observersList.length-1].update(_currentState); // update newly registere
        }

        public function unRegisterObserver(o:IObserver):void {
            _observersList.splice( _observersList.indexOf( o ), 1 );
        }

        public function updateObservers():void {
            for( var i:uint = 0; i<_observersList.length; i++) {
                _observersList[i].update(_currentState);
            }
        }
```

```
        public function changeState(newState:uint):void {
            _currentState = newState;
            updateObservers();
        }
    }
}

// Concrete Observer 1
package {
    public class ConcreteObserver1 implements IObserver {
        public function ConcreteObserver1() {
        }

        public function update(newState:uint):void {
            trace( "co1: "+newState );
        }

        // other Observer specific methods
    }
}

// Concrete Observer 2
package {
    public class ConcreteObserver2 implements IObserver {
        public function ConcreteObserver2() {
        }

        public function update(newState:uint):void {
            trace( "co2: "+newState );
        }

        // other Observer specific methods
    }
}
```

PHP

class STUDENT

```
<?php
class Student implements SplObserver {

        protected $type = "Student";
        private   $name;
        private   $address;
        private   $telephone;
        private   $email;
        private   $_classes = array();

        public function __construct($name)
        {
                $this->name = $name;
        }

        public function GET_type()
        {
                return $this->type;
        }

        public function GET_name()
        {
                return $this->name;
        }

        public function GET_email()
        {
                return $this->email;
        }
```

```
        public function GET_telephone()
        {
                return $this->telephone;
        }

        public function update(SplSubject $object)
        {
                $object->SET_log("Comes from ".$this->name.": I'm a student of ".$object->GET_materi
        }

}

?>
```

class TEACHER

```
<?php
class Teacher implements SplObserver {

        protected $type = "Teacher";
        private   $name;
        private   $address;
        private   $telephone;
        private   $email;
        private   $_classes = array();

        public function __construct($name)
        {
                $this->name = $name;
        }

        public function GET_type()
        {
                return $this->type;
        }

        public function GET_name()
        {
                return $this->name;
        }

        public function GET_email()
        {
                return $this->email;
        }

        public function GET_telephone()
        {
                return $this->name;
        }

        public function update(SplSubject $object)
        {
                $object->SET_log("Comes from ".$this->name.": I teach in ".$object->GET_materia());
        }

}

?>
```

Class SUBJECT

```
<?php

class Subject implements SplSubject {

        private $name_materia;
        private $_observers = array();
```

```
        private $_log = array();

        function __construct($name)
        {
                $this->name_materia = $name;
                $this->_log[]       = "Subject $name was included";
        }

        /* Add an observer */
        public function attach(SplObserver $classes) {
                $this->_classes[] = $classes;
                $this->_log[]     = " The ".$classes->GET_type()." ".$classes->GET_name()." was incl
        }

        /* Remove an observer */
        public function detach(SplObserver $classes) {
                foreach ($this->_classes as $key => $obj) {
                if ($obj == $classes) {
                        unset($this->_classes[$key]);
                        $this->_log[] = " The ".$classes->GET_type()." ".$classes->GET_name(
                }
                }
        }

        /* Notificate an observer */
        public function notify(){
                foreach ($this->_classes as $classes){
                $classes->update($this);
                }
        }

        public function GET_materia()
        {
                return $this->name_materia;
        }

        function SET_log($valor)
        {
                $this->_log[] = $valor ;
        }

        function GET_log()
        {
                return $this->_log;
        }


}
?>
```

Application

```
<?php
require_once("teacher.class.php");
require_once("student.class.php");
require_once("subject.class.php");

$subject  = new Subject("Math");
$marcus   = new Teacher("Marcus Brasizza");
$rafael   = new Student("Rafael");
$vinicius = new Student("Vinicius");

// Include observers in the math Subject
$subject->attach($rafael);
$subject->attach($vinicius);
$subject->attach($marcus);

$subject2 = new Subject("English");
$renato   = new Teacher("Renato");
$fabio    = new Student("Fabio");
$tiago    = new Student("Tiago");
```

```php
// Include observers in the english Subject
$subject2->attach($renato);
$subject2->attach($vinicius);
$subject2->attach($fabio);
$subject2->attach($tiago);

// Remove the instance "Rafael from subject"
$subject->detach($rafael);

// Notify both subjects
$subject->notify();
$subject2->notify();

echo "First Subject <br>";
echo "<pre>";
print_r($subject->GET_log());
echo "</pre>";
echo "<hr>";
echo "Second Subject <br>";
echo "<pre>";
print_r($subject2->GET_log());
echo "</pre>";
?>
```

OUTPUT

First Subject

```
Array
(
    [0] =>  Subject Math was included
    [1] =>  The Student Rafael was included
    [2] =>  The Student Vinicius was included
    [3] =>  The Teacher Marcus Brasizza was included
    [4] =>  The Student Rafael was removed
    [5] => Comes from Vinicius: I'm a student of Math
    [6] => Comes from Marcus Brasizza: I teach in Math
)
```

Second Subject

```
Array
(
    [0] =>  Subject English was included
    [1] =>  The Teacher Renato was included
    [2] =>  The Student Vinicius was included
    [3] =>  The Student Fabio was included
    [4] =>  The Student Tiago was included
    [5] => Comes from Renato: I teach in English
    [6] => Comes from Vinicius: I'm a student of English
    [7] => Comes from Fabio: I'm a student of English
    [8] => Comes from Tiago: I'm a student of English
)
```

Ruby

In Ruby, use the standard Observable mixin. For documentation and an example, see http://www.ruby-doc.org/stdlib/libdoc/observer/rdoc/index.html

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:

- The Java Swing library makes extensive use of the observer pattern for event management
- Boost.Signals (http://www.boost.org/doc/html/signals.html) , an extension of the C++ STL providing a signal/slot model
- The Qt C++ framework's signal/slot model
- libsigc++ (http://libsigc.sourceforge.net) - the C++ signalling template library.
- sigslot (http://sigslot.sourceforge.net/) - C++ Signal/Slot Library
- XLObject (http://xlobject.sourceforge.net/) - Template-based C++ signal/slot model patterned after Qt.
- libevent (http://www.monkey.org/~provos/libevent/) - Multi-threaded Crossplatform Signal/Slot C++ Library
- GObject, in GLib - an implementation of objects and signals/callbacks in C. (This library has many bindings to other programming languages.)
- Exploring the Observer Design Pattern (http://msdn2.microsoft.com/en-us/library/ms954621.aspx) - the C# and Visual Basic .NET implementation, using delegates and the Event pattern
- Using the Observer Pattern (http://ramblings.aaronballman.com/2005/04/Using_the_Observer_Pattern_in_REALbasic.html) , a discussion and implementation in REALbasic
- flash.events (http://livedocs.macromedia.com/flex/2/langref/flash/events/package-detail.html) , a package in ActionScript 3.0 (following from the mx.events package in ActionScript 2.0).
- CSP (http://ptolemy.eecs.berkeley.edu/presentations/06/FutureOfEmbeddedSoftware_Lee_Graz.ppt) - *Observer Pattern* using *CSP-like Rendezvous* (each actor is a process, communication is via rendezvous).
- YUI Event utility (http://developer.yahoo.com/yui/event/) implements custom events through the observer pattern
- Py-notify (http://home.gna.org/py-notify/) , a Python implementation
- Event_Dispatcher (http://pear.php.net/package/Event_Dispatcher) , a PHP implementation
- Delphi Observer Pattern (http://blogs.teamb.com/joannacarter/articles/690.aspx) , a Delphi implementation
- .NET Remoting (http://www.codeproject.com/csharp/c_sharp_remoting.asp) , Applying the Observer Pattern in .NET Remoting (using C#)
- PerfectJPattern Open Source Project (http://perfectjpattern.sourceforge.net/dp-observer.html) , Provides a context-free and type-safe implementation of the Observer Pattern in Java.
- Cells (http://common-lisp.net/project/cells/) , a dataflow extension to Common Lisp that uses meta-programming to hide some of the details of Observer pattern implementation.
- Publish/Subscribe with LabVIEW (http://www.labviewtutorial.eu/viewtopic.php?f=19&t=9) , Implementation example of Observer or Publish/Subscribe using G.
- SPL (http://www.php.net/~helly/php/ext/spl/main.html) , the Standard PHP Library

References

- http://www.research.ibm.com/designpatterns/example.htm

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

See also

- Design Patterns, the book which gave rise to the study of design patterns in computer science
- Design pattern (computer science), a standard solution to common problems in software design
- implicit invocation

External links

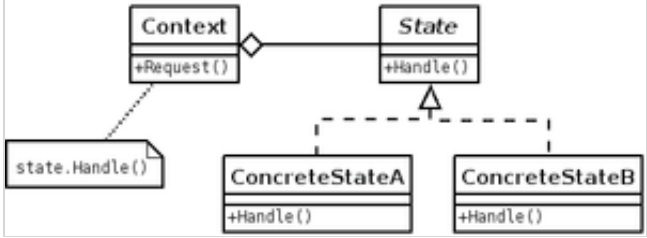
- Observer pattern in PHP (http://www.a-scripts.com/object-oriented-php/2009/02/21/the-observer-pattern/)
- A sample implementation in .NET (http://www.codeproject.com/gen/design/applyingpatterns.asp)
- Observer Pattern in Java (http://www.javaworld.com/javaworld/javaqa/2001-05/04-qa-0525-observer.html)
- Observer Pattern implementation in JDK 1.4 (http://java.sun.com/j2se/1.4.2/docs/api/java/util/Observable.html)
- Definition & UML diagram (http://www.dofactory.com/Patterns/PatternObserver.aspx)
- Jt (http://www.fsw.com/Jt/Jt.htm) J2EE Pattern Oriented Framework
- Discussion of multiple observer application. (http://mooph.cz/index.php?less_fun&mvc)

State pattern

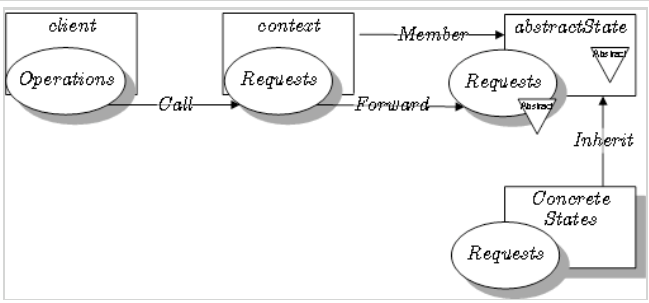
From Wikipedia, the free encyclopedia

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

Take for example, a drawing program, in which there could be an abstract interface representing a tool, then concrete instances of that class could each represent a kind of tool. When the user selects a different tool, the appropriate tool would be instantiated.



State in UML



State in LePUS3 (legend (http://lepus.org.uk/ref/legend/legend.xml))

Contents

Interface Example

For example, an interface to a drawing tool could be

```
class AbstractTool {
public:
  virtual void MoveTo(const Point& inP) = 0;
  virtual void MouseDown(const Point& inP) = 0;
  virtual void MouseUp(const Point& inP) = 0;
};
```

Then a simple pen tool could be

```
class PenTool : public AbstractTool {
public:
  PenTool() : mMouseIsDown(false) {}
  virtual void MoveTo(const Point& inP) {
```

```
    if(mMouseIsDown) {
      DrawLine(mLastP, inP);
    }
    mLastP = inP;
  }
  virtual void MouseDown(const Point& inP) {
    mMouseIsDown = true;
    mLastP = inP;
  }
  virtual void MouseUp(const Point& inP) {
    mMouseIsDown = false;
  }
private:
  bool mMouseIsDown;
  Point mLastP;
};

class SelectionTool : public AbstractTool {
public:
  SelectionTool() : mMouseIsDown(false) {}
  virtual void MoveTo(const Point& inP) {
    if(mMouseIsDown) {
      mSelection.Set(mLastP, inP);
    }
  }
  virtual void MouseDown(const Point& inP) {
    mMouseIsDown = true;
    mLastP = inP;
    mSelection.Set(mLastP, inP);
  }
  virtual void MouseUp(const Point& inP) {
    mMouseIsDown = false;
  }
private:
  bool mMouseIsDown;
  Point mLastP;
  Rectangle mSelection;
};
```

A client using the state pattern above could look like this:

```
class DrawingController {
 public:
  DrawingController() { selectPenTool(); } // Start with some tool.
  void MoveTo(const Point& inP) {currentTool->MoveTo(inP); }
  void MouseDown(const Point& inP) {currentTool->MouseDown(inP); }
  void MouseUp(const Point& inP) {currentTool->MouseUp(inP); }

  void selectPenTool() {
    currentTool.reset(new PenTool);
  }

  void selectSelectionTool() {
    currentTool.reset(new SelectionTool);
  }

 private:
  std::auto_ptr<AbstractTool> currentTool;
};
```

The state of the drawing tool is thus represented entirely by an instance of AbstractTool. This makes it easy to add more tools and to keep their behavior localized to that subclass of `AbstractTool`.

As opposed to using switch

The state pattern can be used to replace `switch()` statements and `if {}` statements which can be

difficult to maintain and are less type-safe. For example, the following is similar to the above but adding a new tool type to this version would be much more difficult.

```
class Tool {
public:
  Tool() : mMouseIsDown(false) {}
  virtual void MoveTo(const Point& inP);
  virtual void MouseDown(const Point& inP);
  virtual void MouseUp(const Point& inP);
private:
  enum Mode { Pen, Selection };
  Mode mMode;
  Point mLastP;
  bool mMouseIsDown;
  Rectangle mSelection;
};

void Tool::MoveTo(const Point& inP) {
  switch(mMode) {
  case Pen:
    if(mMouseIsDown) {
      DrawLine(mLastP, inP);
    }
    mLastP = inP;
    break;
  case Selection:
    if(mMouseIsDown) {
      mSelection.Set(mLastP, inP);
    }
    break;
  default:
    throw std::exception();
  }
}

void Tool::MouseDown(const Point& inP) {
  switch(mMode) {
  case Pen:
    mMouseIsDown = true;
    mLastP = inP;
    break;
  case Selection:
    mMouseIsDown = true;
    mLastP = inP;
    mSelection.Set(mLastP, inP);
    break;
  default:
    throw std::exception();
  }
}

void Tool::MouseUp(const Point& inP) {
  mMouseIsDown = false;
}
```

See also

- Finite State Machine
- Strategy pattern
- Dynamic classification

External links

- Jt (http://www.fsw.com/Jt/Jt.htm) J2EE Pattern Oriented Framework

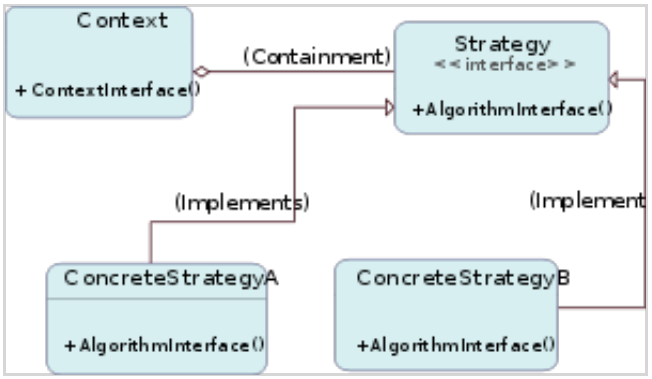
- State pattern in UML and in LePUS3 (http://www.lepus.org.uk/ref/companion/State.xml) (a formal modelling language)
- State Pattern using Java : A Different Approach (http://blogs.sun.com/jkumaran/entry/state_design_pattern_using_java)

References

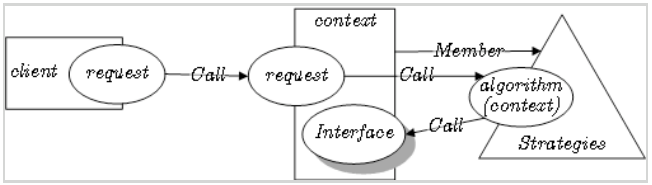
1. **^** Gamma, Erich; Richard Helm, Ralph Johnson, John M. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. pp. 395. ISBN 0201633612.

Strategy pattern

From Wikipedia, the free encyclopedia



Strategy Pattern in UML



Strategy pattern in LePUS3 (legend (http://lepus.org.uk/ref/legend/legend.xml))

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.

In some programming languages, such as those without polymorphism, the issues addressed by this pattern are handled through forms of reflection, such as the native function pointer or function delegate syntax.

This pattern is invisible in languages with first-class functions. See the Python code for an example.

The strategy pattern is useful for situations where it is necessary to dynamically swap the algorithms used in an application. The strategy pattern is intended to provide a means to define a family of algorithms, encapsulate each one as an object, and make them interchangeable. The strategy pattern lets the algorithms vary independently from clients that use them.

Contents

Code Examples

C++

```cpp
#include <iostream>

using namespace std;

class StrategyInterface
{
    public:
        virtual void execute() = 0;
};

class ConcreteStrategyA: public StrategyInterface
{
    public:
        virtual void execute()
        {
            cout << "Called ConcreteStrategyA execute method" << endl;
        }
};

class ConcreteStrategyB: public StrategyInterface
{
    public:
        virtual void execute()
        {
            cout << "Called ConcreteStrategyB execute method" << endl;
        }
};

class ConcreteStrategyC: public StrategyInterface
{
    public:
        virtual void execute()
        {
            cout << "Called ConcreteStrategyC execute method" << endl;
        }
};

class Context
{
    private:
        StrategyInterface *_strategy;

    public:
        Context(StrategyInterface *strategy):_strategy(strategy)
        {
        }

        void set_strategy(StrategyInterface *strategy)
        {
            _strategy = strategy;
        }

        void execute()
        {
            _strategy->execute();
        }
};

int main(int argc, char *argv[])
{
    ConcreteStrategyA concreteStrategyA;
    ConcreteStrategyB concreteStrategyB;
    ConcreteStrategyC concreteStrategyC;

    Context contextA(&concreteStrategyA);
    Context contextB(&concreteStrategyB);
    Context contextC(&concreteStrategyC);

    contextA.execute();
    contextB.execute();
```

```
    contextC.execute();

    contextA.set_strategy(&concreteStrategyB);
    contextA.execute();
    contextA.set_strategy(&concreteStrategyC);
    contextA.execute();

    return 0;
}
```

Java

```
//StrategyExample test application

class StrategyExample {

    public static void main(String[] args) {

        Context context;

        // Three contexts following different strategies
        context = new Context(new ConcreteStrategyA());
        context.execute();

        context = new Context(new ConcreteStrategyB());
        context.execute();

        context = new Context(new ConcreteStrategyC());
        context.execute();

    }

}

// The classes that implement a concrete strategy should implement this

// The context class uses this to call the concrete strategy
interface Strategy {

    void execute();

}

// Implements the algorithm using the strategy interface
class ConcreteStrategyA implements Strategy {

    public void execute() {
        System.out.println("Called ConcreteStrategyA's execute()");
    }

}

class ConcreteStrategyB implements Strategy {

    public void execute() {
        System.out.println("Called ConcreteStrategyB's execute()");
    }

}

class ConcreteStrategyC implements Strategy {

    public void execute() {
        System.out.println("Called ConcreteStrategyC's execute()");
    }

}

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

```
    Strategy strategy;

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

    public void execute() {
        this.strategy.execute();
    }

}
```

Python

Python has first-class functions, so there's no need to implement this pattern explicitly. However one loses information because the interface of the strategy is not made explicit. Here's an example you might encounter in GUI programming, using a callback function:

```
class Button:
    """A very basic button widget."""
    def __init__(self, submit_func, label):
        self.on_submit = submit_func   # Set the strategy function directly
        self.label = label

# Create two instances with different strategies
button1 = Button(sum, "Add 'em")
button2 = Button(lambda nums: " ".join(map(str, nums)), "Join 'em")

# Test each button
numbers = range(1, 10)   # A list of numbers 1 through 9
print button1.on_submit(numbers)   # displays "45"
print button2.on_submit(numbers)   # displays "1 2 3 4 5 6 7 8 9"
```

C# 3.0

In C# 3.0, with lambda expressions we can do something similar to the Python example above.

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

class Program
{
  static void Main(string[] args)
  {
    var button1 = new MyButton((x) => x.Sum().ToString(), "Add them");
    var button2 = new MyButton((x) => string.Join(" ", x.Select(y => y.ToString()).ToArray()), "Joi

    var numbers = Enumerable.Range(1, 10);
    Console.WriteLine(button1.Submit(numbers));
    Console.WriteLine(button2.Submit(numbers));

    Console.ReadLine();
  }

  public class MyButton
  {
    private readonly Func<IEnumerable<int>, string> submitFunction;
    public string Label { get; private set; }

    public MyButton(Func<IEnumerable<int>, string> submitFunction, string label)
    {
        this.submitFunction = submitFunction;
        Label = label;
```

```
    }

    public string Submit(IEnumerable<int> data)
    {
        return submitFunction(data);
    }
  }
}
```

C#

```
using System;

namespace Wikipedia.Patterns.Strategy
{
  // MainApp test application
  class MainApp
  {
    static void Main()
    {
      Context anObject;

      // Three contexts following different strategies
      anObject= new Context(new ConcreteStrategyA());
      anObject.Execute();

      anObject.UpdateContext(new ConcreteStrategyB());
      anObject.Execute();

      anObject.UpdateContext(new ConcreteStrategyC());
      anObject.Execute();

    }
  }

  // The classes that implement a concrete strategy must implement this Execute function.
  // The context class uses this to call the concrete strategy
  interface IStrategy
  {
    void Execute();
  }

  // Implements the algorithm using the strategy interface
  class ConcreteStrategyA : IStrategy
  {
    public void Execute()
    {
      Console.WriteLine( "Called ConcreteStrategyA.Execute()" );
    }
  }

  class ConcreteStrategyB : IStrategy
  {
    public void Execute()
    {
      Console.WriteLine( "Called ConcreteStrategyB.Execute()" );
    }
  }

  class ConcreteStrategyC : IStrategy
  {
    public void Execute()
    {
      Console.WriteLine( "Called ConcreteStrategyC.Execute()" );
    }
  }

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

```
    IStrategy strategy;

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

    public UpdateContext(IStrategy strategy)
    {
      this.strategy = strategy;
    }

    public void Execute()
    {
      strategy.Execute();
    }
  }
}
```

ActionScript 3

```
//invoked from application.initialize
private function init() : void
{
    var context:Context;

    context = new Context( new ConcreteStrategyA() );
    context.execute();

    context = new Context( new ConcreteStrategyB() );
    context.execute();

    context = new Context( new ConcreteStrategyC() );
    context.execute();
}

package org.wikipedia.patterns.strategy
{
    public interface IStrategy
    {
        function execute() : void ;
    }
}

package org.wikipedia.patterns.strategy
{
    public final class ConcreteStrategyA implements IStrategy
    {
        public function execute():void
        {
             trace( "ConcreteStrategyA.execute(); invoked" );
        }
    }
}

package org.wikipedia.patterns.strategy
{
    public final class ConcreteStrategyB implements IStrategy
    {
        public function execute():void
        {
             trace( "ConcreteStrategyB.execute(); invoked" );
        }
    }
}

package org.wikipedia.patterns.strategy
{
    public final class ConcreteStrategyC implements IStrategy
    {
```

```
        public function execute():void
        {
             trace( "ConcreteStrategyC.execute(); invoked" );
        }
    }
}

package org.wikipedia.patterns.strategy
{
   public class Context
   {
        private var strategy:IStrategy;

        public function Context(strategy:IStrategy)
        {
             this.strategy = strategy;
        }

        public function execute() : void
        {
             strategy.execute();
        }
    }
}
```

PHP

```
<?php
class StrategyExample {
    public function __construct() {
        $context = new Context(new ConcreteStrategyA());
        $context->execute();

        $context = new Context(new ConcreteStrategyB());
        $context->execute();

        $context = new Context(new ConcreteStrategyC());
        $context->execute();
    }
}

interface IStrategy {
    public function execute();
}

class ConcreteStrategyA implements IStrategy {
    public function execute() {
        echo "Called ConcreteStrategyA execute method\n";
    }
}

class ConcreteStrategyB implements IStrategy {
    public function execute() {
        echo "Called ConcreteStrategyB execute method\n";
    }
}

class ConcreteStrategyC implements IStrategy {
    public function execute() {
        echo "Called ConcreteStrategyC execute method\n";
    }
}

class Context {
    var $strategy;

    public function __construct(IStrategy $strategy) {
        $this->strategy = $strategy;
    }

    public function execute() {
```

```
        $this->strategy->execute();
    }
}

new StrategyExample;
?>
```

Perl

Perl has first-class functions, so it doesn't require this pattern coded explicitly:

```
sort { lc($a) cmp lc($b) } @items
```

The strategy pattern can be formally implemented with Moose:

```
package Strategy;
use Moose::Role;
requires 'execute';


package FirstStrategy;
use Moose;
with 'Strategy';

sub execute {
    print "Called FirstStrategy->execute()\n";
}


package SecondStrategy;
use Moose;
with 'Strategy';

sub execute {
    print "Called SecondStrategy->execute()\n";
}


package ThirdStrategy;
use Moose;
with 'Strategy';

sub execute {
    print "Called ThirdStrategy->execute()\n";
}


package Context;
use Moose;

has 'strategy' => (
    is => 'rw',
    does => 'Strategy',
    handles => [ 'execute' ],  # automatic delegation
);


package StrategyExample;
use Moose;

# Moose's constructor
sub BUILD {
    my $context;

    $context = Context->new(strategy => 'FirstStrategy');
    $context->execute;

    $context = Context->new(strategy => 'SecondStrategy');
```

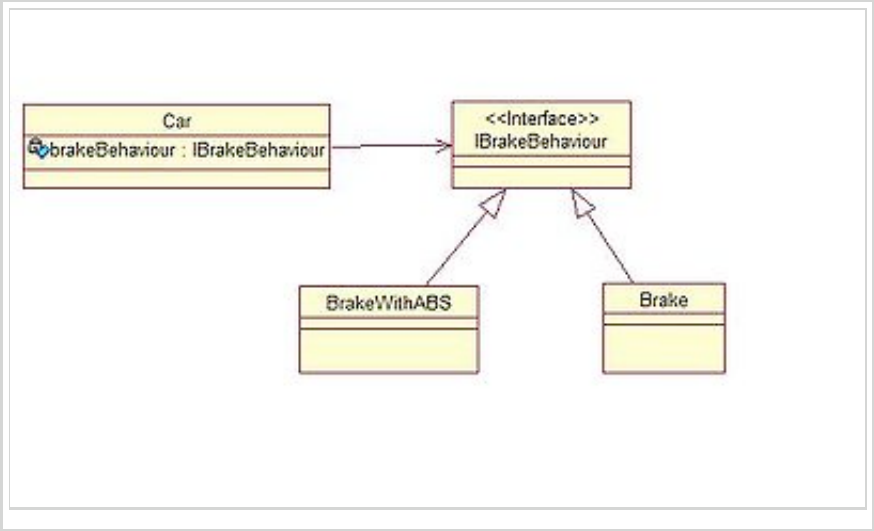
```
    $context->execute;

    $context = Context->new(strategy => 'ThirdStrategy');
    $context->execute;
}


package main;

StrategyExample->new;
```

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 Pattern and Open Closed Principle



According to Strategy pattern, the behaviors of a class should not be inherited, instead they should be encapsulated using interfaces. As an example, consider a car class. Two possible behaviors of 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. This may not be a concern when there are only a small number of models, but the work of managing these behaviors increases greatly as the number of models increases, and requires code to be duplicated across models. Additionally, it is not easy to determine the exact nature of the behavior for each model without investigating the code in each.

The strategy pattern uses composition instead of inheritance. In the strategy pattern behaviors are defined as separate interfaces and specific classes that implement these interfaces. Specific classes encapsulate these interfaces. This allows better decoupling between the behavior and the class that uses the behavior. The behavior can be changed without breaking the classes that use it, and the classes can switch between behaviors by changing the specific implementation used without requiring any significant code changes. Behaviors can also be changed at run-time as well as at design-time. For instance, a car object's brake behavior can be changed from `BrakeWithABS()` to `Brake()` by changing the `brakeBehavior` member to:

```
```

```
brakeBehavior = new Brake();
```

This gives greater flexibility in design and is in harmony with the Open/closed principle (OCP) that states classes should be open for extension but closed for modification.

See also

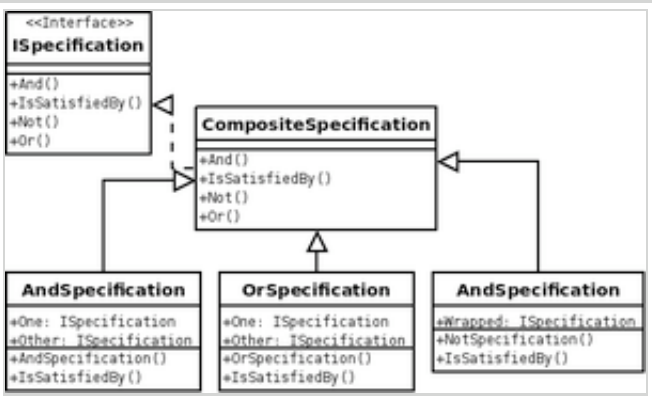
- Mixin
- Policy-based design
- First-class function
- Template method pattern
- Bridge pattern
- Open/closed principle
- Factory Pattern
- List of object-oriented programming terms

External links

- The Strategy Pattern from the Net Objectives Repository (http://www.netobjectivesrepository.com/TheStrategyPattern)
- Strategy Pattern for Java article (http://www.javaworld.com/javaworld/jw-04-2002/jw-0426-designpatterns.html)
- Strategy pattern in UML and in LePUS3 (http://www.lepus.org.uk/ref/companion/Strategy.xml) (a formal modelling notation)
- Data & object factory (http://www.dofactory.com/Patterns/PatternStrategy.aspx)
- Refactoring: Replace Type Code with State/Strategy (http://www.refactoring.com/catalog/replaceTypeCodeWithStateStrategy.html)
- Jt (http://www.fsw.com/Jt/Jt.htm) J2EE Pattern Oriented Framework
- Strategy Pattern with a twist! (http://anirudhvyas.com/root/2008/04/02/a-much-better-strategy-pattern/)

Specification pattern

From Wikipedia, the free encyclopedia



Specification Pattern in UML

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

A Specification pattern outlines a unit of business logic that is combinable with other business logic units. In this pattern, a unit of Business Logic inherits its functionality from the abstract aggregate Composite Specification class. The Composite Specification class has one function called IsSatisfiedBy that returns a boolean value. After instantiation, the specification is 'chained' with other specifications, making new specifications easily maintainable, yet highly customizable business logic. Furthermore upon instantiation the business logic may, through method invocation or Inversion of Control, have its state altered in order to become a delegate of other classes such as a persistence repository.

Contents

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++

```
#include <iostream>
```

```
using namespace std;

    class Operand
    {
    };

    class ISpecification
    {
        public:
            virtual bool IsSatisfiedBy(Operand *candidate) = 0;

    };

    class ICompSpecification : public ISpecification
    {
        public:

            virtual ISpecification * And(ISpecification *other) = 0;

            virtual ISpecification * Or(ISpecification *other) = 0;

            virtual ISpecification * Not() = 0;
    };


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

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

            virtual bool IsSatisfiedBy(Operand *candidate)
            {
                return One->IsSatisfiedBy(candidate) && Other->IsSatisfiedBy(candidate);
            }
    };


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

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

          virtual bool IsSatisfiedBy(Operand *candidate)
          {
              return One->IsSatisfiedBy(candidate) || Other->IsSatisfiedBy(candidate);
          }
   };

   class NotSpecification : public  ISpecification
   {
       private:
           ISpecification *Wrapped;

       public:
       NotSpecification(ISpecification *x)
       {
           Wrapped = x;
       }
```

```
        virtual bool IsSatisfiedBy(Operand *candidate)
        {
            return !Wrapped->IsSatisfiedBy(candidate);
        }
    };

    class RealSpecification : public ISpecification
    {
        public:
        virtual bool IsSatisfiedBy(Operand *candidate)
        {
           cout << "real specification satisfied" << endl;
        }
    };

     class CompositeSpecification : public ICompSpecification
     {
         public:
         virtual bool IsSatisfiedBy(Operand *candidate)
         {
             cout << "Not implemented" << endl;
         }

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

         virtual ISpecification * Or(ISpecification *other)
         {
             return new OrSpecification(this, other);
         }

         virtual ISpecification * Not()
         {
            return new NotSpecification(this);
         }
     };


int main()
{
    CompositeSpecification *compSpec = new CompositeSpecification();
    RealSpecification *realSpec = new RealSpecification();
    ISpecification * iSpec = compSpec->And(realSpec);
    Operand cand;
    iSpec->IsSatisfiedBy(&cand);
    realSpec->IsSatisfiedBy(&cand);
    return 0;
}
```

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

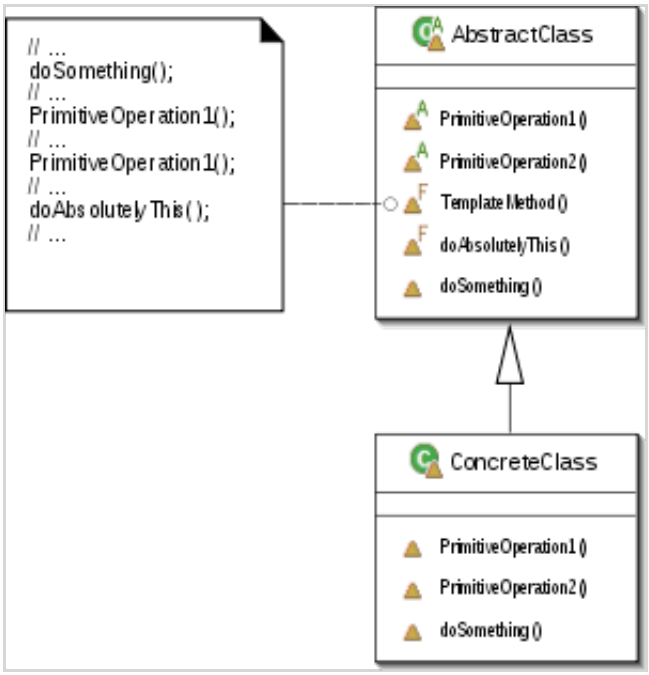
- Specifications (http://www.martinfowler.com/apsupp/spec.pdf) by Eric Evans and Martin Fowler
- The Specification Pattern: A Primer (http://www.mattberther.com/2005/03/25/the-specification-pattern-a-primer/) by Matt Berther
- specification pattern in flash actionscript 3 (http://www.dpdk.nl/opensource/specification-pattern-for-selection-on-lists) by Rolf Vreijdenberger

Template method pattern

From Wikipedia, the free encyclopedia

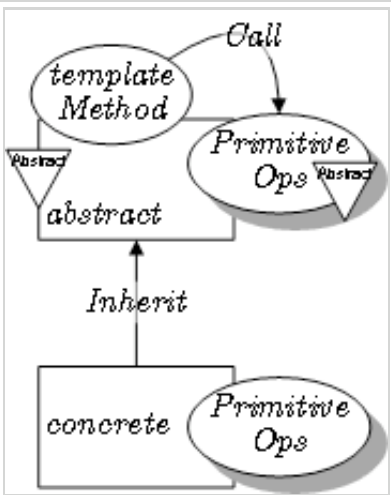
In software engineering, the **template method pattern** is a design pattern. It is a so-called behavioral pattern, and is unrelated to C++ templates.

Contents



Template method: UML class diagram.

Introduction



Template Method in LePUS3 (legend (http://lepus.org.uk/ref/legend/))

In a template pattern, the model (data such as XML, or a set of APIs) has no inherent knowledge of how it would be utilized. The actual algorithm is delegated to the views, i.e. templates. Different templates could be applied to the same set of data or APIs and produce different results. Thus, it is a subset of model-view-controller patterns without the controller. The pattern does not have to be limited to the object-oriented programming. For example, different XSLT templates could render the same XML data and produce different outputs. C++ templates also make it possible to code the algorithms (i.e. views) without having to use abstract classes or interfaces.

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. The expressive power and degrees of freedom occur in abstract methods that may be implemented in subclasses. Some or all of the abstract methods can be specialized in the subclass, the abstract method is the smallest unit of granularity, allowing the writer of the subclass to provide

particular behavior with minimal modifications to the larger semantics. In contrast the template method need not be changed and is not an abstract operation and thus may guarantee required steps before and after the abstract operations. Thus the template method is invoked and as a consequence the subordinate non-abstract methods and abstract methods are called in the correct 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 is strongly related to the NVI (Non-Virtual Interface) 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 frameworks employ the NVI pattern.

Usage

The template method is used to:

- let subclasses implement (through method overriding) behaviour that can vary
- avoid duplication in the code: you look for the general code in the algorithm, and implement the variants in the subclasses
- control at what point(s) subclassing is allowed.

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 example:

Example (in Java)

```
/**
 * An abstract class that is common to several games in
 * which players play against the others, but only one is
 * playing at a given time.
 */

abstract class Game {

    protected int playersCount;

    abstract void initializeGame();

    abstract void makePlay(int player);

    abstract boolean endOfGame();

    abstract void printWinner();

    /* A template method : */
    final void playOneGame(int playersCount) {
        this.playersCount = playersCount;
```

```
        initializeGame();
        int j = 0;
        while (!endOfGame()) {
            makePlay(j);
            j = (j + 1) % playersCount;
        }
        printWinner();
    }
}

//Now we can extend this class in order to implement actual games:

class Monopoly extends Game {

    /* Implementation of necessary concrete methods */

    void initializeGame() {
        // ...
    }

    void makePlay(int player) {
        // ...
    }

    boolean endOfGame() {
        // ...
    }

    void printWinner() {
        // ...
    }

    /* Specific declarations for the Monopoly game. */

    // ...

}

class Chess extends Game {

    /* Implementation of necessary concrete methods */

    void initializeGame() {
        // ...
    }

    void makePlay(int player) {
        // ...
    }

    boolean endOfGame() {
        // ...
    }

    void printWinner() {
        // ...
    }

    /* Specific declarations for the chess game. */

    // ...

}
```

Example (in C++)

In C++ one has to use virtual methods to be able to call a method in a subclass. This is used in the example below. Some may not like the use of virtual methods, and prefer the use of C++-templates (as a language feature) instead. This is related to the Template design pattern, but different.

The example below, does follow the concept as stated above: a subclass implements details which are used at the abstract level.

Note: There always will be a discussion about the cost of 'virtual functions'. Some may argue a 'runtime; is needed; or at least an indirection through a vtable. Others may say modern compilers, processors and on-chip cache does compensate for this.

For that reason this pattern may be applicable (used) in some cases, and at other places it may not.

In the example, we assume the template is applicable for the given application.

```
/**
 * A class that is common to several games in
 * which players play against the others, but only one is
 * playing at a given time.
 * T will implement this interface:
 *
 *  void initializeGame();
 *  void makePlay(int player);
 *  boolean endOfGame();
 *  void printWinner();
*/

class Game  {
  int playersCount;

public:
    /* A template method: valid for all games, when subclasses implement details : */
    void playOneGame(int playersCount) {
        playersCount = playersCount;
        initializeGame();
        int j = 0;
        while (!endOfGame()) {
            makePlay(j);
            j = (j + 1) % playersCount;
        }
        printWinner();
    }
protected:
    virtual void initializeGame();                      // Note:
    virtual void makePlay(int player);                  //      Some may implement with as "=0",
    virtual bool endOfGame();                           //      some throw an exception
    virtual void printWinner();                 //      Basic: the methods exist in subclass
};
// Now we can provide Game policies that implement the actual games:

class Monopoly : public Game {
protected:
    /* Implementation of necessary concrete methods */

    void initializeGame() {
        // ...
    }

    void makePlay(int player) {
        // ...
    }

    bool endOfGame() {
        // ...
      return false;                                         // to get rid of g++ warning
    }

    void printWinner() {
        // ...
    }
```

```
    /* Specific declarations for the Monopoly game. */

    // ...

};

class Chess : public Game {
protected:
    /* Implementation of necessary concrete methods */

    void initializeGame() {
        // ...
    }

    void makePlay(int player) {
        // ...
    }

    bool endOfGame() {
        // ...
      return false;                                          // to get rid of g++ warning
    }

    void printWinner() {
        // ...
    }

    /* Specific declarations for the chess game. */

    // ...

};

// example instantiating a chess game
void EXAMPLE(void)
{

  Chess g1;
  Monopoly g2;

  // ...
  g1.playOneGame(2);
  g2.playOneGame(7);
}
```

See also

- Inheritance (computer science)
- Method overriding (programming)

External links

- Template design pattern in C# and VB.NET (http://www.dofactory.com/Patterns/PatternTemplate.aspx)
- Working with Template Classes in PHP 5 (http://www.devshed.com/c/a/PHP/Working-with-Template-Classes-in-PHP-5/)
- Template Method pattern in UML and in LePUS3 (http://www.lepus.org.uk/ref/companion/TemplateMethod.xml) (a formal modelling language)

Categories: Software design patterns | Articles with example Java code | Method (computer science)

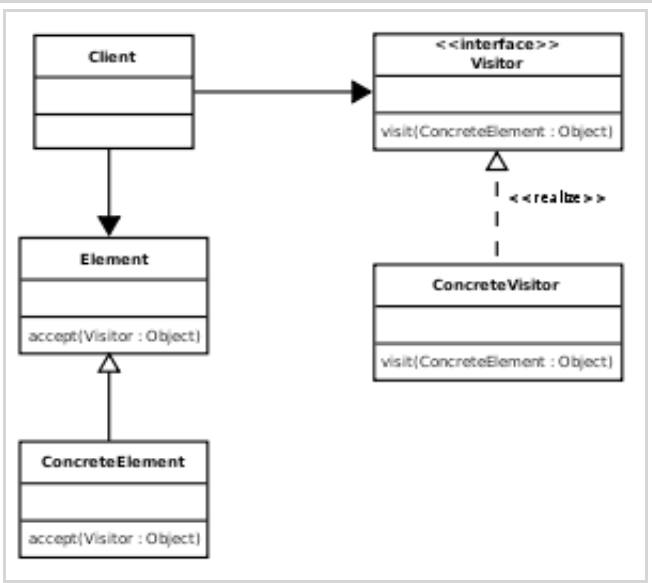
Visitor pattern

From Wikipedia, the free encyclopedia

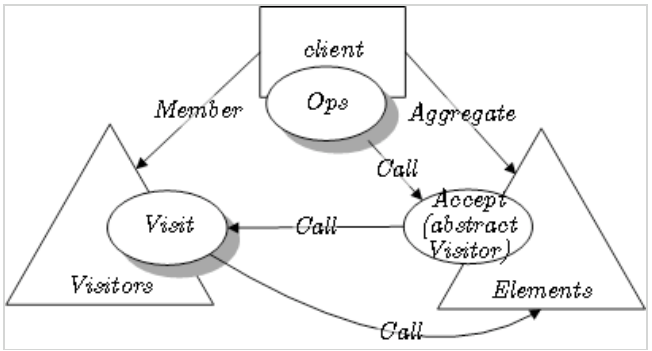
In object-oriented programming and software engineering, the **visitor** design pattern is a way of separating an algorithm from an object structure upon which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures. Thus, using the visitor pattern helps conformance with the open/closed principle.

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

While powerful, the visitor pattern is more limited than conventional virtual functions. It is not possible to create visitors for objects without adding a small callback method inside each class and the callback method in each of the classes is not inheritable.



Visitor in UML



Visitor in LePUS3 (legend (http://lepus.org.uk/ref/legend/legend.xml))

Contents

Elaborated

The idea is to use a structure of element classes, each of which has an `accept()` method that takes a `visitor` object as an argument. `Visitor` is an interface that has a `visit()` method for each element class. The `accept()` method of an element class calls back the `visit()` method for its class. Separate concrete `visitor` classes can then be written that perform some particular operations, by

implementing these operations in their respective `visit()` methods.

One of these `visit()` methods of a concrete `visitor` can be thought of as a method not of a single class, but rather a method of a pair of classes: the concrete visitor and the particular element class. Thus the visitor pattern simulates double dispatch in a conventional single-dispatch object-oriented language such as Java, Smalltalk, and C++. For an explanation of how double dispatch differs from function overloading, see Double dispatch is more than function overloading in the double dispatch article. In the Java language, two techniques have been documented which use reflection to simplify the mechanics of double dispatch simulation in the visitor pattern: getting rid of accept() methods (http://www.cs.ucla.edu/~palsberg/paper/compsac98.pdf) (the Walkabout variation), and getting rid of extra visit() methods (http://www.javaworld.com/javaworld/javatips/jw-javatip98.html) .

The visitor pattern also specifies how iteration occurs over the object structure. In the simplest version, where each algorithm needs to iterate in the same way, the `accept()` method of a container element, in addition to calling back the `visit()` method of the `visitor`, also passes the `visitor` object to the `accept()` method of all its constituent child elements.

Because the Visitor object has one principal function (manifested in a plurality of specialized methods) and that function is called `visit()`, the Visitor can be readily identified as a potential function object or functor. Likewise, the `accept()` function can be identified as a function applicator, a mapper, which knows how to traverse a particular type of object and apply a function to its elements. Lisp's object system with its multiple dispatch does not replace the Visitor pattern, but merely provides a more concise implementation of it in which the pattern all but disappears.

Example in Java

The following example is in the Java programming language:

```
interface CarElementVisitor {
    void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Body body);
    void visitCar(Car car);
}
interface CarElement{
    void accept(CarElementVisitor visitor);
}
class Wheel implements CarElement{
    private String name;
    Wheel(String name) {
        this.name = name;
    }
    String getName() {
        return this.name;
    }
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Engine implements CarElement{
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}
```

```
class Body implements CarElement{
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Car  {
    CarElement[] elements;
    public CarElement [] getElements(){
        return elements.clone();
    }
    public Car() {
        this.elements = new CarElement[]
          { new Wheel("front left"), new Wheel("front right"),
            new Wheel("back left") , new Wheel("back right"),
            new Body(), new Engine()};
    }
}

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 visitCar(Car car) {
        System.out.println("\nVisiting car");
        for(CarElement element : car.getElements()) {
            element.accept(this);
        }
        System.out.println("Visited car");
    }
}

class CarElementDoVisitor implements CarElementVisitor {
    public void visit(Wheel wheel) {
        System.out.println("Kicking my "+ wheel.getName());
    }
    public void visit(Engine engine) {
        System.out.println("Starting my engine");
    }
    public void visit(Body body) {
        System.out.println("Moving my body");
    }
    public void visitCar(Car car) {
        System.out.println("\nStarting my car");
        for(CarElement carElement : car.getElements()) {
            carElement.accept(this);
        }
        System.out.println("Started car");
    }

}

public class VisitorDemo {
    static public void main(String[] args){
        Car car = new Car();
        CarElementVisitor printVisitor = new CarElementPrintVisitor();
        CarElementVisitor doVisitor = new CarElementDoVisitor();
        printVisitor.visitCar(car);
        doVisitor.visitCar(car);
    }
}
```

Example in D

The following example is in the D programming language:

```
import std.stdio;
import std.string;

interface CarElementVisitor {
    void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Body bod);
    void visitCar(Car car);
}

interface CarElement{
    void accept(CarElementVisitor visitor);
}

class Wheel : CarElement {
    private string name;
    this(string name) {
        this.name = name;
    }
    string getName() {
        return name;
    }
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Engine : CarElement {
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Body : CarElement {
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Car {
    CarElement[] elements;
    public CarElement[] getElements(){
        return elements;
    }
    public this() {
        elements =
        [
            cast(CarElement) new Wheel("front left"),
            cast(CarElement) new Wheel("front right"),
            cast(CarElement) new Wheel("back left"),
            cast(CarElement) new Wheel("back right"),
            cast(CarElement) new Body(),
            cast(CarElement) new Engine()
        ];
    }
}

class CarElementPrintVisitor : CarElementVisitor {
    public void visit(Wheel wheel) {
        writefln("Visiting "~ wheel.name ~ " wheel");
    }
    public void visit(Engine engine) {
        writefln("Visiting engine");
    }
    public void visit(Body bod) {
        writefln("Visiting body");
    }
```

```
    public void visitCar(Car car) {
        writefln("\nVisiting car");
        foreach(CarElement element ; car.elements) {
            element.accept(this);
        }
        writefln("Visited car");
    }
}

class CarElementDoVisitor : CarElementVisitor {
    public void visit(Wheel wheel) {
        writefln("Kicking my "~ wheel.name);
    }
    public void visit(Engine engine) {
        writefln("Starting my engine");
    }
    public void visit(Body bod) {
        writefln("Moving my body");
    }
    public void visitCar(Car car) {
        writefln("\nStarting my car");
        foreach(CarElement carElement ; car.getElements()) {
            carElement.accept(this);
        }
        writefln("Started car");
    }
}

void main(){
    Car car = new Car;
    CarElementVisitor printVisitor = new CarElementPrintVisitor;
    CarElementVisitor doVisitor = new CarElementDoVisitor;
    printVisitor.visitCar(car);
    doVisitor.visitCar(car);
}
```

Example in C++

The following example is an example in the C++ programming language:

```
#include <string>
#include <iostream>
#include <vector>

using namespace std;

class Wheel;
class Engine;
class Body;
class Car;

// interface to all car 'parts'
struct CarElementVisitor
{
  virtual void visit(Wheel& wheel) const = 0;
  virtual void visit(Engine& engine) const = 0;
  virtual void visit(Body& body) const = 0;

  virtual void visitCar(Car& car) const = 0;
  virtual ~CarElementVisitor() {}
};

// interface to one part
struct CarElement
{
  virtual void accept(const CarElementVisitor& visitor) = 0;
  virtual ~CarElement() {}
};

// wheel element, there are four wheels with unique names
```

```
class Wheel : public CarElement
{
public:
  explicit Wheel(const string& name) :
    name_(name)
  {
  }
  const string& getName() const
  {
    return name_;
  }
  void accept(const CarElementVisitor& visitor)
  {
    visitor.visit(*this);
  }
private:
    string name_;
};

// engine
class Engine : public CarElement
{
public:
  void accept(const CarElementVisitor& visitor)
  {
    visitor.visit(*this);
  }
};

// body
class Body : public CarElement
{
public:
  void accept(const CarElementVisitor& visitor)
  {
    visitor.visit(*this);
  }
};

// car, all car elements(parts) together
class Car
{
public:
  vector<CarElement*>& getElements()
  {
    return elements_;
  }
  Car()
  {
    // assume that neither push_back nor Wheel(const string&) may throw
    elements_.push_back( new Wheel("front left") );
    elements_.push_back( new Wheel("front right") );
    elements_.push_back( new Wheel("back left") );
    elements_.push_back( new Wheel("back right") );
    elements_.push_back( new Body() );
    elements_.push_back( new Engine() );
  }
  ~Car()
  {
    for(vector<CarElement*>::iterator it = elements_.begin();
      it != elements_.end(); ++it)
    {
      delete *it;
    }
  }
private:
  vector<CarElement*> elements_;
};

// PrintVisitor and DoVisitor show by using a different implementation the Car class is unchanged
// even though the algorithm is different in PrintVisitor and DoVisitor.
class CarElementPrintVisitor : public CarElementVisitor
{
public:
  void visit(Wheel& wheel) const
```

```
  {
    cout << "Visiting " << wheel.getName() << " wheel" << endl;
  }
  void visit(Engine& engine) const
  {
    cout << "Visiting engine" << endl;
  }
  void visit(Body& body) const
  {
    cout << "Visiting body" << endl;
  }
  void visitCar(Car& car) const
  {
    cout << endl << "Visiting car" << endl;
    vector<CarElement*>& elems = car.getElements();
    for(vector<CarElement*>::iterator it = elems.begin();
      it != elems.end(); ++it )
    {
      (*it)->accept(*this);     // this issues the callback i.e. to this from the element
    }
    cout << "Visited car" << endl;
  }
};

class CarElementDoVisitor : public CarElementVisitor
{
public:
  // these are specific implementations added to the original object without modifying the original
  void visit(Wheel& wheel) const
  {
    cout << "Kicking my " << wheel.getName() << " wheel" << endl;
  }
  void visit(Engine& engine) const
  {
    cout << "Starting my engine" << endl;
  }
  void visit(Body& body) const
  {
    cout << "Moving my body" << endl;
  }
  void visitCar(Car& car) const
  {
    cout << endl << "Starting my car" << endl;
    vector<CarElement*>& elems = car.getElements();
    for(vector<CarElement*>::iterator it = elems.begin();
      it != elems.end(); ++it )
    {
      (*it)->accept(*this);     // this issues the callback i.e. to this from the element
    }
    cout << "Stopped car" << endl;
  }
};

int main()
{
  Car car;
  CarElementPrintVisitor printVisitor;
  CarElementDoVisitor doVisitor;

  printVisitor.visitCar(car);
  doVisitor.visitCar(car);

  return 0;
}
```

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. In order 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.

See also

- Double and multiple dispatch
- Composite pattern
- Hierarchical visitor pattern
- Strategy pattern
- Function object

External links

- The Visitor Family of Design Patterns (http://objectmentor.com/resources/articles/visitor.pdf) 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 (http://www.lepus.org.uk/ref/companion/Visitor.xml) (a Design Description Language)
- Article "Componentization: the Visitor Example (http://se.ethz.ch/~meyer/publications/computer/visitor.pdf) by Bertrand Meyer and Karine Arnout, *Computer* (IEEE), vol. 39, no. 7, July 2006, pages 23-30.
- Article "Domain Searching Using Visitors (http://www.onjava.com/pub/a/onjava/2005/06/01/searchvisitor.html) " by Paul Mukherjee
- Article "Parsing Conditions using Interpreter and Visitor Pattern (http://codeproject.com/cpp/ConditionInterpreter.asp) "
- Article A Type-theoretic Reconstruction of the Visitor Pattern (http://www.cs.bham.ac.uk/~hxt/research/mfps-visitors.pdf)
- Article "The Essence of the Visitor Pattern (http://citeseer.ist.psu.edu/palsberg97essence.html) " 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 "Eliminate accept() methods from your Visitor pattern (http://www.polyglotinc.com/articles.html#reflectVisitor) " by Bruce Wallace
- Article "Cooperative Visitor: A Template Technique for Visitor Creation (http://www.artima.com/cppsource/cooperative_visitor.html) " by Anand Shankar Krishnamoorthi
- Visitor Patterns (http://goblin.colourcountry.net/apt1002/Visitor%20patterns) as a universal

model of terminating computation.
- Visitor Pattern (http://www.oodesign.com/oo_design_patterns/behavioral_patterns/visitor_pattern.html) using reflection(java).
- PerfectJPattern Open Source Project (http://perfectjpattern.sourceforge.net/dp-visitor.html) , Provides a context-free and type-safe implementation of the Visitor Pattern in Java based on Delegates.

Categories: Software design patterns | Articles with example Java code | Articles with example C++ code

---

Concurrency pattern

From Wikipedia, the free encyclopedia

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

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

External links

Recordings about concurrency patterns from Software Engineering Radio:

- Episode 12: Concurrency Pt. 1 (http://www.se-radio.net/index.php?post_id=81083)
- Episode 19: Concurrency Pt. 2 (http://www.se-radio.net/index.php?post_id=99079)
- Episode 29: Concurrency Pt. 3 (http://www.se-radio.net/index.php?post_id=126370)

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

Active object

From Wikipedia, the free encyclopedia

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.

See also

- Actor model

References

1. **^** D, Schmidt; M., Rohnert, H., Buschmann, F. (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" (PDF). http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf. Retrieved on 2007-02-02.

Categories: Software design patterns | Computer programming stubs

---

Asynchronous method invocation

From Wikipedia, the free encyclopedia
(Redirected from Event-Based Asynchronous Pattern)

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`, 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". *Distributed Programming with Ice*. ZeroC, Inc.. http://www.zeroc.com/doc/Ice-3.2.1/manual/Async.34.2.html#71139. Retrieved on 22 November 2008.
2. ^ Vermeulen, Allan (June 1996). "An Asynchronous Design Pattern". *Dr. Dobb's Journal*. http://www.ddj.com/184409898. Retrieved on 22 November 2008.
3. ^ Nash, Trey (2007). "Threading in C#". *Accelerated C# 2008*. Apress. ISBN 9781590598733.
4. ^ Lavender, R. Greg; Douglas C. Schmidt (PDF). *Active Object*. http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf. Retrieved on 22 November 2008.
5. ^ "Asynchronous Programming Design Patterns". *.NET Framework Developer's Guide*. Microsoft Developer Network. http://msdn.microsoft.com/en-us/library/ms228969.aspx. Retrieved on 22 November 2008.
6. ^ "Asynchronous Programming Overview". *.NET Framework Developer's Guide*. Microsoft Developer Network. http://msdn.microsoft.com/en-us/library/ms228963.aspx. Retrieved on 22 November 2008.
7. ^ "Using an AsyncCallback Delegate to End an Asynchronous Operation". *.NET Framework Developer's Guide*. Microsoft Developer Network. http://msdn.microsoft.com/en-us/library/ms228972.aspx. Retrieved on 22 November 2008.
8. ^ "Concurrency Issues". *Distributed Programming with Ice*. ZeroC, Inc.. http://www.zeroc.com/doc/Ice-3.2.1/manual/Async.34.3.html#76161. Retrieved on 22 November 2008.
9. ^ Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, and Morgan Skinner (2008). "Event-based Asynchronous Pattern". *Professional C# 2008*. Wiley. pp. 570–571. ISBN 0470191376.
10. ^ "Multithreaded Programming with the Event-based Asynchronous Pattern". *.NET Framework Developer's Guide*. Microsoft Developer Network. http://msdn.microsoft.com/en-us/library/hkasytyf.aspx. Retrieved on 22 November 2008.
11. ^ "Deciding When to Implement the Event-based Asynchronous Pattern". *.NET Framework Developer's Guide*. Microsoft Developer Network. http://msdn.microsoft.com/en-

us/library/ms228966.aspx. Retrieved on 22 November 2008.

Further reading

- Chris Sells and Ian Griffiths (2007). "Appendix C.3: The Event-Based Asynchronous Pattern". *Programming WPF*. O'Reilly. pp. 747–749. ISBN 0596510373.
- Using asynchronous method calls in C# (http://articles.techrepublic.com.com/5100-10878_11-1044325.html)

Balking pattern

From Wikipedia, the free encyclopedia

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 in this field that 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 it should …

- be created in a sane state
- not make itself available until it is in a sane state
- become a façade and answer back an object that is in a sane state

… so that the call can be made without limitation.

Contents

When to Use

This pattern is only to be used when an object is in a particular state, however, the developer must determine whether or not an object will remain in a balking state for an indefinite period of time. If that is the case, then the balking pattern is recommended when designing an object. However, if the object will return to an appropriate state within a known period of finite time, then the guarded suspension pattern would serve better use.

Implementation

Below is a general, simple example for an implementation of the balking pattern as originally seen in Grand (2002). As demonstrated by the definition above, notice how the "synchronized" line is utilized. If there are multiple calls to the job method, only one will proceed while the other calls will return with nothing. Another thing to note is the `jobCompleted()` method. The reason it is not synchronized is that there will never be multiple calls to it at the same time, as only one job will

execute at a time.

```
public class Example{
    private boolean jobInProgress = false;

    public void job(){
        synchronized(this){
           if(jobInProgress){
               return;
           }
           jobInProgress = true;
        }
        //Code to execute job goes here
        ...
    }

    void jobCompleted() {
        jobInProgress = false;
    }

}
```

References

- Grand, Mark (2002), *Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML, Second Edition*, Indianapolis, Ind: John Wiley & Sons.

See also

- Read and write lock pattern
- Guarded suspension pattern

Double-checked locking

From Wikipedia, the free encyclopedia
(Redirected from Double checked locking pattern)

In software engineering, **double-checked locking** is a software design pattern also known as "double-checked locking optimization[1]". The pattern is designed to 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.

The pattern, when implemented in some language/hardware combinations, can be unsafe. It can therefore sometimes be considered to be an anti-pattern.

It is typically used to reduce locking overhead when implementing "lazy initialization" in a multi-threaded environment, especially as part of the Singleton pattern. Lazy initialization avoids initializing a value until the first time it is accessed.

Contents

Usage in Java

Consider, for example, this code segment in the Java programming language as given by [1] (http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html) (as well as all other Java code segments):

```
// Single threaded version
class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null)
            helper = new Helper();
        return helper;
    }

    // other functions and members...
}
```

The problem is that this does not work when using multiple threads. A lock must be obtained in case two threads call `getHelper()` simultaneously. Otherwise, either they may both try to create the object at the same time, or one may wind up getting a reference to an incompletely initialized object. This is

done by 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, the overhead of acquiring and releasing a lock every time this method is called seems unnecessary: once the initialization has been completed, acquiring and releasing the locks would appear unnecessary. Many programmers have attempted to optimize this situation in the following manner:

1. Check that the variable is initialized (without obtaining the lock). If it is initialized, return it immediately.
2. Obtain the lock.
3. Double-check whether the variable has already been initialized: if another thread acquired the lock first, it may have already done the initialization. If so, return the initialized variable.
4. Otherwise, initialize and return the variable.

```
// Broken multithreaded version
// "Double-Checked Locking" idiom
class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null) {
                    helper = new Helper();
                }
            }
        }
        return helper;
    }

    // other functions and members...
}
```

Intuitively, this algorithm seems like an efficient solution to the problem. However, this technique has many subtle problems and should usually be avoided. For example, consider the following sequence of events:

1. Thread *A* notices that the value is not initialized, so it obtains the lock and begins to initialize the value.
2. Due to the semantics of some programming languages, the code generated by the compiler is allowed to update the shared variable to point to a partially constructed object before *A* has finished performing the initialization.
3. Thread *B* notices that the shared variable has been initialized (or so it appears), and returns its value. Because thread *B* believes the value is already initialized, it does not acquire the lock. If

the variable is used before *A* finishes initializing it, the program will likely crash.

One of the dangers of using double-checked locking in J2SE 1.4 (and earlier versions) is that it will often appear to work: it is not easy to distinguish between a correct implementation of the technique and one that has subtle problems. Depending on the compiler, the interleaving of threads by the scheduler and the nature of other concurrent system activity, failures resulting from an incorrect implementation of double-checked locking may only occur intermittently. Reproducing the failures can be difficult.

As of J2SE 5.0, this problem has been fixed. The volatile keyword now ensures that multiple threads handle the singleton instance correctly. This new idiom is described in [2] (http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html) :

```
// 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() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null)
                    helper = new Helper();
            }
        }
        return helper;
    }

    // other functions and members...
}
```

Many versions of the double-checked locking idiom that do not use explicit methods such as volatile or synchronization to communicate the construction of the object have been proposed, and all of them are wrong[3] (http://jeremymanson.blogspot.com/2007/05/double-checked-locking-and-problem-with.html) [4] (http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html) .

Usage in Microsoft Visual C++

Double-checked locking can be implemented in Visual C++ 2005 if the pointer to the resource is marked as being volatile. Visual C++ 2005 guarantees that volatile variables will behave as fences, as in J2SE 5.0, preventing both compiler and CPU arrangement of reads and writes. 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.

See also

- The Test and Test-and-set idiom for a low-level locking mechanism.
- Initialization on demand holder idiom for a thread-safe replacement in Java.

References

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

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/?p=35) 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
- The "Double-Checked Locking is Broken" Declaration (http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html) ; David Bacon et al.
- Double-checked locking and the Singleton pattern (http://www-106.ibm.com/developerworks/java/library/j-dcl.html)
- Singleton Pattern and Thread Safety (http://www.oaklib.org/docs/oak/singleton.html)
- volatile keyword in VC++ 2005 (http://msdn2.microsoft.com/en-us/library/12a04hfd.aspx)
- Java Examples and timing of double check locking solutions (http://blogs.sun.com/cwebster/entry/double_check_locking)

Categories: Anti-patterns | Concurrency control | Programming constructs | Software design patterns
Hidden categories: All articles with unsourced statements | Articles with unsourced statements since August 2008

Guarded suspension

From Wikipedia, the free encyclopedia

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.

Contents

When to Use

Due to its nature, the guarded suspension pattern should only be used when the developer knows that a method call will be suspended for a finite period of time. If a method call is suspended for too long, then the overall program wrapped around the method call will slow it down. If developers know that the method call suspension will be indefinite, then one would be better off using the balking pattern (see links below).

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.

See also

- Read write lock pattern
- Balking pattern is an alternative pattern for dealing with a precondition
- Guarded commands includes a similar language construct

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.

Monitor (synchronization)

From Wikipedia, the free encyclopedia

In concurrent programming, a **monitor** is an object intended to be used safely by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion. That is, at each point in time, at most one thread may be executing any of its methods. This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared with code that may be executed in parallel.

Monitors also provide a mechanism for threads to temporarily give up exclusive access, in order to wait for some condition to be met, before regaining exclusive access and resuming their task. Monitors also have a mechanism for signaling other threads that such conditions have been met.

Monitors were invented by C.A.R. Hoare [1] and Per Brinch Hansen, [2] and were first implemented in Brinch Hansen's Concurrent Pascal language.

Contents

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)
  {
    if amount < 0 then error "Amount may not be negative"
    else if balance < amount then return false
    else { balance := balance - amount ; return true }
  }

  public method deposit(int amount) {
    if amount < 0 then error "Amount may not be negative"
    else 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 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 without error while causing the balance to drop by only 1000.

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 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 assertion 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, upon which a thread may wait for some assertion to become true. Thus each condition variable c is associated with some assertion P_c. While a thread is waiting upon a condition variable, that thread is not considered to occupy the monitor, and so other threads may enter the monitor to changes 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.

Thus there are two main operations on conditions variables:

- **`wait c`** is called by a thread that needs to wait until the assertion P_c to be true before proceeding.
- **`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 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*. Monitors using blocking condition variables are often called *Hoare style* monitors. With a blocking condition variable, the signaling thread must wait outside the monitor (at least) until the signaled thread relinquishes occupancy of the monitor by either returning or by again waiting on a condition.

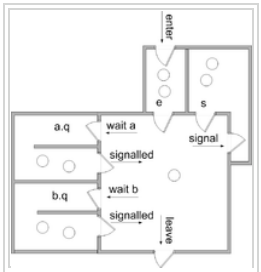
We assume there are two queues of threads associated with each monitor object

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

In addition we assume that for each condition c, there is a queue

- c`.q`, which is a queue for threads waiting on condition c

All queues are typically guaranteed to be fair (i.e. each thread that enters the queue will not be not chosen an infinite number of times) and, in some implementations, may be guaranteed to be first in first out.



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

The implementation of each operation is as follows. (We assume that each operation runs in mutual exclusion to the others; thus restarted threads do not begin executing until the operation is complete.)

```
enter the monitor:
   enter the method
   if the monitor is locked
       add this thread to e
       block this thread
   else
       lock the monitor
```

```
leave the monitor:
   schedule
   return from the method
```

```
wait c :
   add this thread to c.q
   schedule
   block this thread
```

```
signal c :
   if there is a thread waiting on c.q
       select and remove one such thread t from c.q
       (t is called "the signaled thread")
       add this thread to s
       restart t
       (so t will occupy the monitor next)
       block this thread
```

```
 schedule :
   if there is a thread on s
     select and remove one thread from s and restart it
     (this thread will occupy the monitor next)
   else if there is a thread on e
     select and remove one thread from e and restart it
     (this thread will occupy the monitor next)
   else
     unlock the monitor
```

```
    (the monitor will become unoccupied)
```

The `schedule` routine selects the next thread to occupy the monitor or, in the absence of any candidate threads, unlocks the monitor.

The resulting signaling discipline is known a "*signal and urgent wait,*" as the signaler must wait, but is given priority over threads on the entrance queue. An alternative is "*signal and wait,*" in which there is no `s` queue and signaler waits on the `e` queue instead.

Some implementations provide a **signal and return** operation that combines signaling with returning from a procedure.

```
signal c and return :
   if there is a thread waiting on c.q
       select and remove one such thread t from c.q
       (t is called "the signaled thread")
       restart t
       (so t will occupy the monitor next)
   else
       schedule
   return from the method
```

In either case ("signal and urgent wait" or "signal and wait"), when a condition is signaled and there is at least one thread on waiting on the condition, 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.

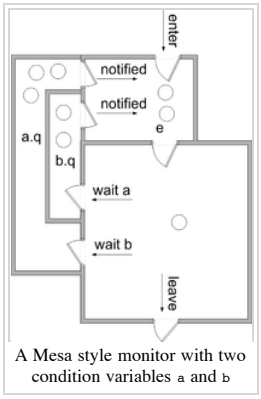
```
monitor class SharedStack {
  private const capacity := 10
  private int[capacity] A
  private int size := 0
  invariant 0 <= size and size <= capacity
  private BlockingCondition theStackIsNotEmpty /* associated with 0 < size and size <= capacity */
  private BlockingCondition theStackIsNotFull /* 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 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 by moved to a queue called `w`, which has priority over `e`. See Howard[5] and Burh *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 is 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 assertion stronger than P_c. The operations **`notify`** c and **`notify all`** c operations 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.

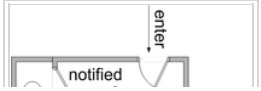
As an example of "hinting" consider a bank account in which a withdrawing thread will wait until the account has sufficient funds before proceeding

```
monitor class Account {
  private int balance := 0
  invariant balance >= 0
  private NonblockingCondition balanceMayBeBigEnough

  public method withdraw(int amount)
  {
    if amount < 0 then error "Amount may not be negative"
    else {
       while balance < amount do wait balanceMayBeBigEnough
       assert balance >= amount
       balance := balance - amount }
  }

  public method deposit(int amount) {
    if amount < 0 then error "Amount may not be negative"
    else {
        balance := balance + amount
        notify all balanceMayBeBigEnough }
  }
}
```

In this example, the assertion being waited for is a function of the amount to be withdrawn, so it is impossible for a depositing thread to be sure that it has established the assertion. It makes sense in this case to allow each waiting thread into the monitor (one at a time) to check if itsassertion is true.

Implicit condition monitors

In the Java programming language each object may be used as a monitor. (However, methods that require mutual exclusion must be explicitly marked as **synchronized**). Rather than having explicit condition variables, each monitor (i.e. object) is equipped with a



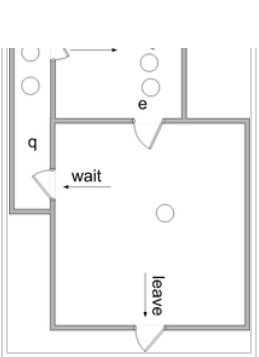
single wait queue, in addition to its entrance queue. All waiting is done on this single wait queue and all **notify** and **notify all** operations apply to this queue.

This approach has also been adopted in other languages such as C#.

Implicit signaling

Another approach to signaling is to omit the **signal** operation. Whenever a thread leaves the monitor (by returning or waiting) the assertions of all waiting threads are evaluated until one is found to be true. In such a system, condition variables are not needed, but the assertions must be explicitly coded. The contract for wait is

```
wait P:
    precondition I
    modifies the state of the monitor
    postcondition P and I
```



A Java style monitor

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 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 2005 (as protected objects)
- C# (and other languages that use the .NET Framework)
- Concurrent Euclid
- Concurrent Pascal
- D programming language
- Java (via the synchronized keyword)
- Mesa
- Modula-3
- 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.

See also

- Mutual exclusion
- Communicating sequential processes - a later development of monitors by C. A. R. Hoare
- Semaphore (programming)

Bibliography

- Monitors: an operating system structuring concept, C. A. R. Hoare - Communications of the ACM, v.17 n.10, p.549-557, Oct. 1974 [5] (http://doi.acm.org/10.1145/355620.361161)
- Monitor classification P.A. Buhr, M. Fortier, M.H. Coffin - ACM Computing Surveys (CSUR), 1995 [6] (http://doi.acm.org/10.1145/214037.214100)

External links

- "Monitors: An Operating System Structuring Concept (http://www.acm.org/classics/feb96/) " by Charles Antony Richard Hoare
- "Signalling in Monitors (http://portal.acm.org/citation.cfm?id=807647) " by John H. Howard (Computer Scientist)
- "Proving Monitors (http://doi.acm.org/10.1145/360051.360079) " by John H. Howard (Computer Scientist)
- "Experience with Processes and Monitors in Mesa (http://portal.acm.org/citation.cfm?id=358824) " by Butler W. Lampson and David D. Redell
- pthread_cond_wait (http://www.opengroup.org/onlinepubs/009695399/functions/pthread_cond_wait.html) - description from the Open Group Base Specifications Issue 6, IEEE Std 1003.1
- "Block on a Condition Variable (http://gd.tuwien.ac.at/languages/c/programming-dmarshall/node31.html#SECTION00312500000000000000) " by Dave Marshall (Computer Scientist)
- "Strategies for Implementing POSIX Condition Variables on Win32 (http://www.cs.wustl.edu/~schmidt/win32-cv-1.html) " by Douglas C. Schmidt and Irfan Pyarali
- Condition Variable Routines (http://apr.apache.org/docs/apr/group__apr__thread__cond.html) from the Apache Portable Runtime Library
- wxCondition description (http://wxwidgets.org/manuals/2.6.3/wx_wxcondition.html)
- Boost Condition Variables Reference

(http://www.boost.org/doc/html/thread/synchronization.html#thread.synchronization.condvar_ref)
- ZThread Condition Class Reference (http://zthread.sourceforge.net/html/classZThread_1_1Condition.html)
- Wefts::Condition Class Reference (http://wefts.sourceforge.net/wefts-apidoc-0.99c/classWefts_1_1Condition.html)
- ACE_Condition Class Template Reference (http://www.dre.vanderbilt.edu/Doxygen/Stable/ace/classACE__Condition.html)
- QWaitCondition Class Reference (http://doc.trolltech.com/latest/qwaitcondition.html)
- Common C++ Conditional Class Reference (http://www.gnu.org/software/commoncpp/docs/refman/html/class_conditional.html)
- at::ConditionalMutex Class Reference (http://austria.sourceforge.net/dox/html/classat_1_1ConditionalMutex.html)
- threads::shared (http://perldoc.perl.org/threads/shared.html) - Perl extension for sharing data structures between threads
- Tutorial multiprocessing traps (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)
- http://msdn.microsoft.com/en-us/library/ms682052(VS.85).aspx

Notes

1. **^** Hoare, C. A. R. (1974), "Monitors: an operating system structuring concept". Comm. A.C.M. **17(10)**, 549–57. [1] (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. [2] (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. [3] (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. [4] (http://doi.acm.org/10.1145/154766.155361)

Read/write lock pattern

From Wikipedia, the free encyclopedia
(Redirected from Read write lock pattern)

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.

The current edition of the POSIX standard includes a read-write lock in the form of `pthread_rwlock_t` and the associated operations (http://www.opengroup.org/onlinepubs/009695399/functions/pthread_rwlock_init.html) .

Java version 5 or above includes an interface named `java.util.concurrent.locks.ReadWriteLock` that allows the use of this pattern.

Also, the Boost C++ Libraries include a read/write lock in the form of `boost::shared_mutex` (http://www.boost.org/doc/libs/1_36_0/doc/html/thread/synchronization.html#thread.synchronization.mutex_types.shared_mutex) .

See also

- Lock pattern
- Semaphore
- Mutex
- Scheduler pattern
- Balking pattern
- Lock (software engineering)

External links

- Good explanation of the implementation of a read/write lock (http://doc.trolltech.com/qq/qq11-mutex.html)

Scheduler pattern

From Wikipedia, the free encyclopedia

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.

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.

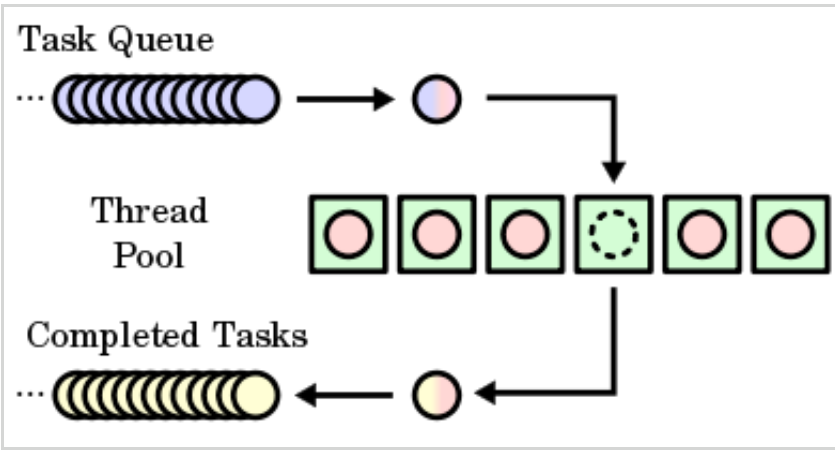
The scheduler pattern is not quite the same as the scheduled-task pattern.

See also

- Mediator pattern

Thread pool pattern

From Wikipedia, the free encyclopedia



A sample thread pool (green boxes) with waiting tasks (blue) and completed tasks (yellow)

In the **thread pool pattern** in programming, a number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads. As soon as a thread completes its task, it will request the next task from the queue until all tasks have been completed. The thread can then terminate, or sleep until there are new tasks available.

The number of threads used is a parameter that can be tuned to provide the best performance. Additionally, the number of threads can be dynamic based on the number of waiting tasks. For example, a web server can add threads if numerous web page requests come in and can remove threads when those requests taper down. The cost of having a larger thread pool is increased resource usage. The algorithm that determines when creating or destroying threads will have an impact on the overall performance:

- create too many threads and resources are wasted and time also wasted creating the 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.

The advantage of using a thread pool over creating a new thread for each task, is that thread creation and destruction overhead is negated, which may result in better performance and better system stability. Also, if the number of tasks is very large, then creating a thread for each one may be impractical.

When implementing this pattern, the programmer should ensure thread-safety of the queue.

Typically, a thread pool executes on a single processor. However, thread pools are conceptually related to server farms in which a master process distributes tasks to worker processes on different computers, in order to increase the overall throughput. Embarrassingly parallel problems are highly amenable to this approach.

See also

- Concurrency pattern

- Parallelization
- Server farm

External links

- Article "Query by Slice, Parallel Execute, and Join: A Thread Pool Pattern in Java (http://today.java.net/pub/a/today/2008/01/31/query-by-slice-parallel-execute-join-thread-pool-pattern.html) " by Binildas C. A.
- Article "Thread pools and work queues (http://www-128.ibm.com/developerworks/java/library/j-jtp0730.html) " by Brian Goetz
- Article "A Method of Worker Thread Pooling (http://www.codeproject.com/threads/thread_pooling.asp) " by Pradeep Kumar Sahu
- Article "Work Queue (http://codeproject.com/threads/work_queue.asp) " by Uri Twig
- Article "Windows Thread Pooling and Execution Chaining (http://codeproject.com/threads/Joshthreadpool.asp) "
- Article "Smart Thread Pool (http://www.codeproject.com/cs/threads/smartthreadpool.asp) " by Ami Bar
- Article "Programming the Thread Pool in the .NET Framework (http://msdn.microsoft.com/en-us/library/ms973903.aspx) " by David Carmona
- Article "The Thread Pool and Asynchronous Methods (http://www.yoda.arachsys.com/csharp/threads/threadpool.shtml) " by Jon Skeet
- Article "Creating a Notifying Blocking Thread Pool in Java (http://java.net/pub/a/today/2008/10/23/creating-a-notifying-blocking-thread-pool-executor.html) " by Amir Kirsh
- Article "Practical Threaded Programming with Python: Thread Pools and Queues (http://www.ibm.com/developerworks/aix/library/au-threadingpython/) " by Noah Gift
- Paper "Optimizing Thread-Pool Strategies for Real-Time CORBA (http://www.cs.wustl.edu/~schmidt/PDF/OM-01.pdf) " by Irfan Pyarali, Marina Spivak, Douglas C. Schmidt and Ron Cytron
- Code "Thread Pool CookBook (http://code.activestate.com/recipes/tags/thread%20pool/) "
- Code "Pool CookBook (http://code.activestate.com/recipes/tags/pool/) "

Retrieved from "http://en.wikipedia.org/wiki/Thread_pool_pattern"
Categories: Threads | Software design patterns

Thread-local storage

From Wikipedia, the free encyclopedia
(Redirected from Thread-Specific 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 all threads in a process share the same address space. In other words, data in a static or global variable is normally always located at the same memory location, when referred to by threads from the same process. Variables on the stack however are local to threads, because each thread has its own stack, residing in a different memory location.

Sometimes it is desirable that two threads referring to the same static or global variable are actually referring to different memory locations, thereby making the variable thread local, a canonical example being the C error code variable `errno`.

If it is possible to make at least a memory address sized variable thread local, it is in principle possible to make arbitrarily sized memory blocks thread local, by allocating such a memory block and storing the memory address of that block in a thread local variable.

Contents

Windows implementation

The API function **TlsAlloc** can be used to obtain an unused *TLS slot index*; the *TLS slot index* will then be considered ‘used’.

The **TlsGetValue** and **TlsSetValue** functions can then be used to read and write a memory address to a thread local variable identified by the *TLS slot index*. **TlsSetValue** can only affect the variable for

the current thread.

The **TlsFree** function can be called to release the *TLS slot index*; the index will then be considered 'unused' and a new call to **TlsAlloc** can return it again.

Pthreads implementation

TLS with Pthreads (*Thread-Specific Data* in Pthreads nomenclature) is similar to **TlsAlloc** and related functionality for Windows. **pthread_key_create** creates a *key*, with an optional *destructor*, that can later be associated with thread specific data via **pthread_setspecific**. The data can be retrieved using **pthread_getspecific**. If the thread specific value is not *NULL*, the *destructor* will be called when the thread exits. Additionally, *key* must be destroyed with **pthread_key_delete**.

Language-specific implementation

Apart from relying on programmers to call the appropriate API functions, it is also possible to extend the programming language to support TLS.

Object Pascal

In Delphi or Free Pascal you can use the 'threadvar' reserved keyword instead of 'var' to declare variables using the thread-local storage.

```
var
   mydata_process: integer;
threadvar
   mydata_threadlocal: integer;
```

Java

In Java thread local variables are implemented by the `ThreadLocal` (`http://java.sun.com/javase/6/docs/api/java/lang/ThreadLocal.html`) class. A `ThreadLocal` object maintains a separate instance of the variable for each thread that calls the object's `get` or `set` method. The following example (for J2SE 5.0 or later version of Java) illustrates using a `ThreadLocal` that holds an `Integer` object:

```
ThreadLocal<Integer> local = new ThreadLocal<Integer>(){
    @Override protected Integer initialValue(){
        return 1;
    }
};
```

the preceding code declares and instantiates a `ThreadLocal` object **`local`** which returns an initial value of 1 if no other value has been stored for the calling thread. The following code increments the value for the currently executing thread.

```
local.set( local.get()+1 );
```

`local.get()` returns the current `Integer` object associated with the current thread, or 1 if no object has yet been associated with the thread. The code calls **`local.set()`** to set a new value associated

with the thread. (Note that the above example uses both generics and autoboxing—features added to Java in J2SE 5.0.)

Sun Studio C/C++, IBM XL C/C++, GNU C & Intel C/C++(Linux systems)

The keyword **__thread** is used like this:

```
__thread int number;
```

- **__thread** defines *number* to be a thread local variable.
- **int** defines the type of *number* to be of type **int**.

Visual C++, Intel C/C++(Windows systems)

In Visual C++ the keywords **declspec(thread)** are used like this:

```
__declspec(thread) int number;
```

- **__declspec(thread)** defines *number* to be a thread local variable.
- **int** defines the type of *number* to be of type **int**.
- On operating systems prior to Vista and Server 2008 **__declspec(thread)** works in DLLs only when those DLLs are bound to the executable, and will *not* work for those loaded with **LoadLibrary()** (a protection fault will occur)
- There are additional rules: "Rules and Limitations for TLS" (http://msdn2.microsoft.com/en-us/library/2s9wt68x.aspx) in MSDN.

Digital Mars C++

In Digital Mars C++ the keywords **declspec(thread)** are used like this:

```
__declspec(thread) int number;
```

- **__declspec(thread)** defines *number* to be a thread local variable.
- **int** defines the type of *number* to be of type **int**.

Borland C++ Builder

In Borland C++ Builder the keywords **__declspec(thread)** are used like this:

```
__declspec(thread) int number;
```

the same in a more elegant way:

```
int __thread number;
```

- **__declspec(thread)** defines *number* to be a thread local variable. **__thread** is a synonym for **__declspec(thread)**.

- **int** defines the type of *number* to be of type **int**.

GCC

GCC implements __thread as above.

The initialiser must be a compile-time constant, even in C++. E.g.

```
__thread int number = 1;
```

but not

```
void f(int number)
{
 static __thread int number_copy = number;
```

or (C++)

```
__thread int number = calculate_number();
```

C# and other .NET languages

Static fields can be marked with ThreadStaticAttribute (http://msdn2.microsoft.com/en-us/library/system.threadstaticattribute(vs.80).aspx) :

```
class FooBar
{
  [ThreadStatic] static int foo;
}
```

Also an API (http://msdn2.microsoft.com/en-us/library/system.threading.thread.getnameddataslot(vs.80).aspx) is available for dynamically allocating thread local variables.

Python

In Python version 2.4 or later **local** class in **threading** module can be used to create thread-local storage.

```
import threading
mydata = threading.local()
mydata.x = 1
```

Ruby

In Ruby thread local variables can be created/accessed using []=/[] methods.

```
Thread.current[:user_id] = 1
```

Perl

In Perl threads were added late in the evolution of the language, after a large body of existing code was already present on the Comprehensive Perl Archive Network. As a result, threads in Perl by default take their own local storage for all variables, to minimise the impact of threads on existing non-thread-aware code. In Perl, a thread-shared variable can be created using an attribute:

```
use threads;
use threads::shared;

my $localvar;
my $sharedvar :shared;
```

External links

- ELF Handling For Thread-Local Storage (http://people.redhat.com/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)

Retrieved from "http://en.wikipedia.org/wiki/Thread-local_storage"
Categories: Storage | Threads | Variable (computer programming)

Reactor pattern

From Wikipedia, the free encyclopedia

The reactor design pattern is a concurrent programming pattern for handling service requests delivered concurrently to a service handler by one or more inputs. The service handler then demultiplexes the incoming requests and dispatches them synchronously to the associated request handlers.

Contents

Structure

- **Resources:** Any resource that can provide input 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 multi threaded 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

- The ADAPTIVE Communication Environment (http://www.cs.wustl.edu/~schmidt/ACE.html) (C++)
- xSocket (http://xsocket.org/) (Java)
- Apache MINA (http://mina.apache.org/) (Java)
- POE (Perl)
- Twisted (Python)
- EventMachine (http://rubyeventmachine.com/) (Ruby)

References

1. **^** Kegel, Dan, *The C10K problem*, http://www.kegel.com/c10k.html#nb.select, retrieved on 2007-07-28

External links

- An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events by Douglas C. Schmidt (http://www.cs.wustl.edu/~schmidt/PDF/reactor-siemens.pdf)
- APR Networking & the Reactor Pattern (http://www.ddj.com/cpp/193101548)
- Architecture of a Highly Scalable NIO-Based Server (http://today.java.net/cs/user/print/a/350)

Categories: Software design patterns | Concurrent computing
