

Software design pattern

In software engineering, a **software design pattern** or **design pattern** is a general, reusable solution to a commonly occurring problem in many contexts in software design.^[1] A design pattern is not a rigid structure to be transplanted directly into source code. Rather, it is a description or a template for solving a particular type of problem that can be deployed in many different situations.^[2] Design patterns can be viewed as formalized best practices that the programmer may use to solve common problems when designing a software application or system.

Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply mutable state may be unsuited for functional programming languages. Some patterns can be rendered unnecessary in languages that have built-in support for solving the problem they are trying to solve, and object-oriented patterns are not necessarily suitable for non-object-oriented languages.

Design patterns may be viewed as a structured approach to computer programming intermediate between the levels of a programming paradigm and a concrete algorithm.

History

Patterns originated as an architectural concept by Christopher Alexander as early as 1977 in A Pattern Language (cf. his article, "The Pattern of Streets," JOURNAL OF THE AIP, September, 1966, Vol. 32, No. 5, pp. 273–278). In 1987, Kent Beck and Ward Cunningham began experimenting with the idea of applying patterns to programming – specifically pattern languages – and presented their results at the OOPSLA conference that year.^{[3][4]} In the following years, Beck, Cunningham and others followed up on this work.

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

- Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 978-0-201-63361-0.
- Brinch Hansen, Per (1995). *Studies in Computational Science: Parallel Programming Paradigms*. Prentice Hall. ISBN 978-0-13-439324-7.
- Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter (1996). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons. ISBN 978-0-471-95869-7.
- Beck, Kent (1997). *Smalltalk Best Practice Patterns*. Prentice Hall. ISBN 978-0134769042.
- Schmidt, Douglas C.; Stal, Michael; Rohnert, Hans; Buschmann, Frank (2000). *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons. ISBN 978-0-471-60695-6.
- Fowler, Martin (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley. ISBN 978-0-321-12742-6.
- Hohpe, Gregor; Woolf, Bobby (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley. ISBN 978-0-321-20068-6.
- Freeman, Eric T.; Robson, Elisabeth; Bates, Bert; Sierra, Kathy (2004). *Head First Design Patterns*. O'Reilly Media. ISBN 978-0-596-00712-6.
- Larman, Craig (2004). *Applying UML and Patterns (3rd Ed, 1st Ed 1995)*. Pearson. ISBN 978-0131489066.

Although design patterns have been applied practically for a long time, formalization of the concept of design patterns languished for several years.^[5]

Practice

Design patterns can speed up the development process by providing proven development paradigms.^[6] Effective software design requires considering issues that may not become apparent until later in the implementation. Freshly written code can often have hidden, subtle issues that take time to be detected; issues that sometimes can cause major problems down the road. Reusing design patterns can help to prevent such issues,^[7] and enhance code readability for those familiar with the patterns.

Software design techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that does not require specifics tied to a particular problem.

In 1996, Christopher Alexander was invited to give a Keynote Speech (<https://www.patternlanguage.com/archive/ieee.html>) to the 1996 OOPSLA Convention. Here he reflected on how his work on Patterns in Architecture had developed and his hopes for how the Software Design community could help Architecture extend Patterns to create living structures that use generative

schemes that are more like computer code.

Motif

A pattern describes a *design motif*, a.k.a. *prototypical micro-architecture*, as a set of program constituents (e.g., classes, methods...) and their relationships. A developer adapts the motif to their codebase to solve the problem described by the pattern. The resulting code has structure and organization similar to the chosen motif.

Domain-specific patterns

Efforts have also been made to codify design patterns in particular domains, including the use of existing design patterns as well as domain-specific design patterns. Examples include user interface design patterns,^[8] information visualization,^[9] secure design,^[10] "secure usability",^[11] Web design ^[12] and business model design.^[13]

The annual Pattern Languages of Programming Conference proceedings ^[14] include many examples of domain-specific patterns.

Object-oriented programming

Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply mutable state may be unsuited for functional programming languages. Some patterns can be rendered unnecessary in languages that have built-in support for solving the problem they are trying to solve, and object-oriented patterns are not necessarily suitable for non-object-oriented languages.

Examples

Design patterns can be organized into groups based on what kind of problem they solve. Creational patterns create objects. Structural patterns organize classes and objects to form larger structures that provide new functionality. Behavioral patterns describe collaboration between objects.

Creational patterns

Name	Description	In <i>Design Patterns</i>	In <i>Code Complete</i> ^[15]	Other
<u>Abstract factory</u>	Provide an interface for creating <i>families</i> of related or dependent objects without specifying their concrete classes.	Yes	Yes	—
<u>Builder</u>	Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.	Yes	Yes	—
<u>Dependency Injection</u>	A class accepts the objects it requires from an injector instead of creating the objects directly.	—	Yes	—
<u>Factory method</u>	Define an interface for creating a <i>single</i> object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.	Yes	Yes	—
<u>Lazy initialization</u>	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 pattern appears in the GoF catalog as "virtual proxy", an implementation strategy for the <u>Proxy</u> pattern.	Yes	Yes	<u>PoEAA</u> ^[16]
<u>Multiton</u>	Ensure a class has only named instances, and provide a global point of access to them.	Yes	Yes	Yes
<u>Object pool</u>	Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalisation of <u>connection pool</u> and <u>thread pool</u> patterns.	Yes	Yes	Yes
<u>Prototype</u>	Specify the kinds of objects to create using a prototypical instance, and create new objects from the 'skeleton' of an existing object, thus boosting performance and keeping memory footprints to a minimum.	Yes	Yes	Yes
<u>Resource acquisition is initialization (RAII)</u>	Ensure that resources are properly released by tying them to the lifespan of suitable objects.	Yes	Yes	Yes
<u>Singleton</u>	Ensure a class has only one instance, and provide a global point of access to it.	Yes	Yes	Yes

Structural patterns

Name	Description	In <i>Design Patterns</i>	In <i>Code Complete</i> ^[15]	Other
<u>Adapter</u> , <u>Wrapper</u> , or <u>Translator</u>	Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces. The enterprise integration pattern equivalent is the translator.	Yes	Yes	Yes
<u>Bridge</u>	Decouple an abstraction from its implementation allowing the two to vary independently.	Yes	Yes	Yes
<u>Composite</u>	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.	Yes	Yes	Yes
<u>Decorator</u>	Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.	Yes	Yes	Yes
<u>Delegation</u>	Extend a class by composition instead of subclassing. The object handles a request by delegating to a second object (the delegate)	Yes	Yes	Yes
<u>Extension object</u>	Adding functionality to a hierarchy without changing the hierarchy.	Yes	Yes	Yes
<u>Facade</u>	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	Yes
<u>Flyweight</u>	Use sharing to support large numbers of similar objects efficiently.	Yes	Yes	Yes
<u>Front controller</u>	The pattern relates to the design of Web applications. It provides a centralized entry point for handling requests.	Yes	Yes	J2EE Patterns ^[17] PoEAA ^[18]
<u>Marker</u>	Empty interface to associate metadata with a class.	Yes	Yes	Effective Java ^[19]
<u>Module</u>	Group several related elements, such as classes, singletons, methods, globally used, into a single conceptual entity.	Yes	Yes	Yes
<u>Proxy</u>	Provide a surrogate or placeholder for another object to control access to it.	Yes	Yes	Yes
<u>Twin</u> ^[20]	Twin allows modeling of multiple inheritance in programming languages that do not support this feature.	Yes	Yes	Yes

Behavioral patterns

Name	Description	In <u>Design Patterns</u>	In <u>Code Complete</u> ^[15]	Other
<u>Blackboard</u>	Artificial intelligence pattern for combining disparate sources of data (see <u>blackboard system</u>)	Yes	Yes	Yes
<u>Chain of responsibility</u>	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	Yes	Yes
<u>Command</u>	Encapsulate a request as an object, thereby allowing for the parameterization of clients with different requests, and the queuing or logging of requests. It also allows for the support of undoable operations.	Yes	Yes	Yes
<u>Fluent interface</u>	Design an API to be method chained so that it reads like a DSL. Each method call returns a context through which the next logical method call(s) are made available.	Yes	Yes	Yes
<u>Interpreter</u>	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.	Yes	Yes	Yes
<u>Iterator</u>	Provide a way to access the elements of an <u>aggregate</u> object sequentially without exposing its underlying representation.	Yes	Yes	Yes
<u>Mediator</u>	Define an object that encapsulates how a set of objects interact. Mediator promotes <u>loose coupling</u> by keeping objects from referring to each other explicitly, and it allows their interaction to vary independently.	Yes	Yes	Yes
<u>Memento</u>	Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later.	Yes	Yes	Yes
<u>Null object</u>	Avoid null references by providing a default object.	Yes	Yes	Yes
<u>Observer or Publish/subscribe</u>	Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.	Yes	Yes	Yes
<u>Servant</u>	Define common functionality for a group of classes. The servant pattern is also frequently called helper class or utility class implementation for a given set of classes. The helper classes generally have no objects hence they have all static methods that act upon different kinds of class objects.	Yes	Yes	Yes
<u>Specification</u>	Recombinable <u>business logic</u> in a <u>Boolean</u> fashion.	Yes	Yes	Yes
<u>State</u>	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.	Yes	Yes	Yes
<u>Strategy</u>	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	Yes
<u>Template method</u>	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	Yes
<u>Visitor</u>	Represent an operation to be performed on instances of a set of classes. Visitor lets a new operation be defined without changing the classes of the elements on which it operates.	Yes	Yes	Yes

Concurrency patterns

Name	Description	In <i>POSA2</i> ^[21]	Other
<u>Active Object</u>	Decouples method execution from method invocation that reside in their own thread of control. The goal is to introduce concurrency, by using <u>asynchronous method invocation</u> and a <u>scheduler</u> for handling requests.	Yes	—
<u>Balking</u>	Only execute an action on an object when the object is in a particular state.	No	—
<u>Binding properties</u>	Combining multiple observers to force properties in different objects to be synchronized or coordinated in some way. ^[22]	No	—
<u>Compute kernel</u>	The same calculation many times in parallel, differing by integer parameters used with non-branching pointer math into shared arrays, such as <u>GPU-optimized Matrix multiplication</u> or <u>Convolutional neural network</u> .	No	—
<u>Double-checked locking</u>	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 locking logic proceed. Can be unsafe when implemented in some language/hardware combinations. It can therefore sometimes be considered an <u>anti-pattern</u> .	Yes	—
<u>Event-based asynchronous</u>	Addresses problems with the asynchronous pattern that occur in multithreaded programs. ^[23]	No	—
<u>Guarded suspension</u>	Manages operations that require both a lock to be acquired and a precondition to be satisfied before the operation can be executed.	No	—
<u>Join</u>	Join-pattern provides a way to write concurrent, parallel and distributed programs by message passing. Compared to the use of threads and locks, this is a high-level programming model.	No	—
<u>Lock</u>	One thread puts a "lock" on a resource, preventing other threads from accessing or modifying it. ^[24]	No	PoEAA ^[16]
<u>Messaging design pattern (MDP)</u>	Allows the interchange of information (i.e. messages) between components and applications.	No	—
<u>Monitor object</u>	An object whose methods are subject to <u>mutual exclusion</u> , thus preventing multiple objects from erroneously trying to use it at the same time.	Yes	—
<u>Reactor</u>	A reactor object provides an asynchronous interface to resources that must be handled synchronously.	Yes	—
<u>Read-write lock</u>	Allows concurrent read access to an object, but requires exclusive access for write operations. An underlying semaphore might be used for writing, and a <u>Copy-on-write</u> mechanism may or may not be used.	No	—
<u>Scheduler</u>	Explicitly control when threads may execute single-threaded code.	No	—
<u>Service handler pattern</u>	For each request, a server spawns a dedicated client handler to handle a request. ^[25] Also referred to as <i>thread-per-session</i> . ^[26]	No	—
<u>Thread pool</u>	A number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads. Can be considered a special case of the <u>object pool</u> pattern.	No	—
<u>Thread-specific storage</u>	Static or "global" memory local to a thread.	Yes	—
<u>Safe Concurrency with Exclusive Ownership</u>	Avoiding the need for runtime concurrent mechanisms, because exclusive ownership can be proven. This is a notable capability of the Rust language, but compile-time checking isn't the only means, a programmer will often manually design such patterns into code - omitting the use of locking mechanism because the programmer assesses that a given variable is never going to be concurrently accessed.	No	—
<u>CPU atomic operation</u>	x86 and other CPU architectures support a range of atomic instructions that guarantee memory safety for modifying and accessing primitive values (integers). For example, two threads may both increment a counter safely. These capabilities can also be used to implement the mechanisms for other concurrency patterns as above. The C# language uses the Interlocked (https://docs.microsoft.com/en-us/dotnet/api/system.threading.interlocked?view=net-5.0) class for these capabilities.	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.^[27] 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.^[28] One example of a commonly used documentation format is the one used by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides 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

Some suggest that design patterns may be a sign that features are missing in a given programming language (Java or C++ for instance). [Peter Norvig](#) demonstrates that 16 out of the 23 patterns in the *Design Patterns* book (which is primarily focused on C++) are simplified or eliminated (via direct language support) in [Lisp](#) or [Dylan](#).^[29] Related observations were made by [Hannemann](#) and [Kiczales](#) who implemented several of the 23 design patterns using an [aspect-oriented programming language](#) (AspectJ) and showed that code-level dependencies were removed from the implementations of 17 of the 23 design patterns and that aspect-oriented programming could simplify the implementations of design patterns.^[30] See also [Paul Graham's](#) essay "Revenge of the Nerds".^[31]

Inappropriate use of patterns may unnecessarily increase complexity.^[32] [FizzBuzzEnterpriseEdition](#) (<https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition>) offers a humorous example of over-complexity introduced by design patterns.^[33]

By definition, a pattern must be programmed anew into each application that uses it. Since some authors see this as a step backward from software reuse as provided by [components](#), researchers have worked to turn patterns into components. [Meyer](#) and [Arnout](#) were able to provide full or partial componentization of two-thirds of the patterns they attempted.^[34]

In order to achieve flexibility, design patterns may introduce additional levels of [indirection](#), which may complicate the resulting design and decrease [runtime](#) performance.

Relationship to other topics

Software design patterns offer finer granularity compared to software architecture patterns and software architecture styles, as design patterns focus on solving detailed, low-level design problems within individual components or subsystems. Examples include [Singleton](#), [Factory Method](#), and [Observer](#).^{[35][36][37]}

[Software Architecture Pattern](#) refers to a reusable, proven solution to a recurring problem at the system level, addressing concerns related to the overall structure, component interactions, and quality attributes of the system. Software architecture patterns operate at a higher level of abstraction than design patterns, solving broader system-level challenges. While these patterns typically affect system-level concerns, the distinction between architectural patterns and architectural styles can sometimes be blurry. Examples include [Circuit Breaker](#).^{[35][36][37]}

[Software Architecture Style](#) refers to a high-level structural organization that defines the overall system organization, specifying how components are organized, how they interact, and the constraints on those interactions. Architecture styles typically include a vocabulary of component and connector types, as well as semantic models for interpreting the system's properties. These styles represent the most coarse-grained level of system organization. Examples include [Layered Architecture](#), [Microservices](#), and [Event-Driven Architecture](#).^{[35][36][37]}

See also

- [Abstraction principle](#)
- [Algorithmic skeleton](#)
- [Anti-pattern](#)
- [Architectural pattern](#)
- [Canonical protocol pattern](#)
- [Debugging patterns](#)
- [Design pattern](#)
- [Distributed design patterns](#)
- [Enterprise Architecture framework](#)

- [GRASP \(object-oriented design\)](#)
- [Helper class](#)
- [Idiom in programming](#)
- [Interaction design pattern](#)
- [List of software architecture styles and patterns](#)
- [List of software development philosophies](#)
- [List of software engineering topics](#)
- [Pattern language](#)
- [Pattern theory](#)
- [Pedagogical patterns](#)
- [Portland Pattern Repository](#)
- [Refactoring](#)
- [Software development methodology](#)

References

1. Alexandrescu, Andrei (2001). *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley. p. xviii. ISBN 978-0-201-70431-0.
2. Horner, Mark (2005). "9". *Pro .NET 2.0 Code and Design Standards in C#*. Apress. p. 171. ISBN 978-1-59059-560-2.
3. Smith, Reid (October 1987). *Panel on design methodology*. OOPSLA '87 Addendum to the Proceedings. doi:10.1145/62138.62151 (<https://doi.org/10.1145/62138.62151>). "Ward cautioned against requiring too much programming at, what he termed, 'the high level of wizards.' He pointed out that a written 'pattern language' can significantly improve the selection and application of abstractions. He proposed a 'radical shift in the burden of design and implementation' basing the new methodology on an adaptation of Christopher Alexander's work in pattern languages and that programming-oriented pattern languages developed at Tektronix has significantly aided their software development efforts."
4. Beck, Kent; Cunningham, Ward (September 1987). *Using Pattern Languages for Object-Oriented Program* (<http://c2.com/doc/oopsla87.html>). OOPSLA '87 workshop on *Specification and Design for Object-Oriented Programming*. Retrieved 2006-05-26.
5. Baroni, Aline Lúcia; Guéhéneuc, Yann-Gaël; Albin-Amiot, Hervé (June 2003). *Design Patterns Formalization* (<https://www.researchgate.net/publication/277282980>) (Report). EMN Technical Report. Nantes: École Nationale Supérieure des Techniques Industrielles et des Mines de Nantes. CiteSeerX 10.1.1.62.6466 (<https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.62.6466>). S2CID 624834 (<https://api.semanticscholar.org/CorpusID:624834>) – via ResearchGate.
6. Bishop, Judith. "C# 3.0 Design Patterns: Use the Power of C# 3.0 to Solve Real-World Problems" (<http://msdn.microsoft.com/en-us/vstudio/ff729657>). C# Books from O'Reilly Media. Retrieved 2012-05-15. "If you want to speed up the development of your .NET applications, you're ready for C# design patterns -- elegant, accepted and proven ways to tackle common programming problems."
7. Tiako, Pierre F. (31 March 2009). "Formal Modeling and Specification of Design Patterns Using RTPA" (https://books.google.com/books?id=_SkIfgSidxQC&q=Reusing+design+patterns+helps+to+prevent+such+subtle+issues&pg=PA636). In Tiako, Pierre F (ed.). *Software Applications: Concepts, Methodologies, Tools, and Applications: Concepts, Methodologies, Tools, and Applications*. p. 636. doi:10.4018/978-1-60566-060-8 (<https://doi.org/10.4018/978-1-60566-060-8>). ISBN 9781605660615.
8. Laakso, Sari A. (2003-09-16). "Collection of User Interface Design Patterns" (<http://www.cs.helsinki.fi/u/salaakso/patterns/index.html>). University of Helsinki, Dept. of Computer Science. Retrieved 2008-01-31.
9. Heer, J.; Agrawala, M. (2006). "Software Design Patterns for Information Visualization" (http://vis.berkeley.edu/papers/infovis_design_patterns/). *IEEE Transactions on Visualization and Computer Graphics*. **12** (5): 853–60. CiteSeerX 10.1.1.121.4534 (<https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.121.4534>). doi:10.1109/TVCG.2006.178 (<https://doi.org/10.1109/TVCG.2006.178>). PMID 17080809 (<https://pubmed.ncbi.nlm.nih.gov/17080809/>). S2CID 11634997 (<https://api.semanticscholar.org/CorpusID:11634997>).
10. Dougherty, Chad; Sayre, Kirk; Seacord, Robert C.; Svoboda, David; Togashi, Kazuya (2009). *Secure Design Patterns* (<http://www.cert.org/archive/pdf/09tr010.pdf>) (PDF). Software Engineering Institute.
11. Garfinkel, Simson L. (2005). *Design Principles and Patterns for Computer Systems That Are Simultaneously Secure and Usable* (<http://www.simson.net/thesis/>) (Ph.D. thesis).
12. "Yahoo! Design Pattern Library" (<https://web.archive.org/web/20080229011119/http://developer.yahoo.com/ypatterns/>). Archived from the original (<http://developer.yahoo.com/ypatterns/>) on 2008-02-29. Retrieved 2008-01-31.
13. "How to design your Business Model as a Lean Startup?" (<http://torgronsund.wordpress.com/2010/01/06/lean-startup-business-model-pattern/>). 2010-01-06. Retrieved 2010-01-06.
14. Pattern Languages of Programming, Conference proceedings (annual, 1994—) [1] (<http://hillside.net/plop/pastconferences.html>)
15. McConnell, Steve (June 2004). "Design in Construction". *Code Complete* (2nd ed.). Microsoft Press. p. 104 (<https://archive.org/details/CodeCompleteSteveMcConnell>).

ails/codecomplete0000mcco/page/104). ISBN 978-0-7356-1967-8. "Table 5.1 Popular Design Patterns"

16. Fowler, Martin (2002). *Patterns of Enterprise Application Architecture* (<http://martinfowler.com/books.html#eaa>). Addison-Wesley. ISBN 978-0-321-12742-6.
17. Alur, Deepak; Crupi, John; Malks, Dan (2003). *Core J2EE Patterns: Best Practices and Design Strategies* (<http://www.corej2eepatterns.com>). Prentice Hall. p. 166. ISBN 978-0-13-142246-9.
18. Fowler, Martin (2002). *Patterns of Enterprise Application Architecture* (<http://martinfowler.com/books.html#eaa>). Addison-Wesley. p. 344. ISBN 978-0-321-12742-6.
19. Bloch, Joshua (2008). "Item 37: Use marker interfaces to define types" (https://archive.org/details/effectivejava00bloc_0/page/179). *Effective Java* (Second ed.). Addison-Wesley. p. 179 (https://archive.org/details/effectivejava00bloc_0/page/179). ISBN 978-0-321-35668-0.
20. "Twin – A Design Pattern for Modeling Multiple Inheritance" (<http://www.ssw.jku.at/Research/Papers/Moe99/Paper.pdf>) (PDF).
21. Schmidt, Douglas C.; Stal, Michael; Rohnert, Hans; Buschmann, Frank (2000). *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons. ISBN 978-0-471-60695-6.
22. Binding Properties (<http://c2.com/cgi/wiki?BindingProperties>)
23. Nagel, Christian; Evjen, Bill; Glynn, Jay; Watson, Karli; Skinner, Morgan (2008). "Event-based Asynchronous Pattern". *Professional C# 2008*. Wiley. pp. 570–571. ISBN 978-0-470-19137-8.
24. Lock Pattern (<http://c2.com/cgi/wiki?LockPattern>)
25. Francalanza, Adrian; Tabone, Gerard (October 2023). "ElixirST: A session-based type system for Elixir modules". *Journal of Logical and Algebraic Methods in Programming*. **135**. doi:10.1016/j.jlamp.2023.100891 (<https://doi.org/10.1016%2Fj.jlamp.2023.100891>). S2CID 251442539 (<https://api.semanticscholar.org/CorpusID:251442539>).
26. Schmidt, Douglas C.; Vinoski, Steve (July–August 1996). "Object Interconnections: Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers (Column 7)" (<https://www.dre.vanderbilt.edu/~schmidt/PDF/C++-report-col7.pdf>) (PDF). *SIGS C++ Report*. S2CID 2654843 (<https://api.semanticscholar.org/CorpusID:2654843>).
27. Gabriel, Dick. "A Pattern Definition" (<https://web.archive.org/web/20070209224120/http://hillside.net/patterns/definition.html>). Archived from the original (<http://hillside.net/patterns/definition.html>) on 2007-02-09. Retrieved 2007-03-06.
28. Fowler, Martin (2006-08-01). "Writing Software Patterns" (<http://www.martinfowler.com/articles/writingPatterns.html>). Retrieved 2007-03-06.
29. Norvig, Peter (1998). *Design Patterns in Dynamic Languages* (<http://www.norvig.com/design-patterns/>).
30. Hannemann, Jan; Kiczales, Gregor (2002). "Design pattern implementation in Java and AspectJ". *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '02*. OOPSLA '02. p. 161. doi:10.1145/582419.582436 (<https://doi.org/10.1145%2F582419.582436>). ISBN 1581134711.
31. Graham, Paul (2002). "Revenge of the Nerds" (<http://www.paulgraham.com/icad.html>). Retrieved 2012-08-11.
32. McConnell, Steve (2004). *Code Complete: A Practical Handbook of Software Construction, 2nd Edition* (<https://archive.org/details/codecomplete0000mcco>). Pearson Education. p. 105 (<https://archive.org/details/codecomplete0000mcco/page/105>). ISBN 9780735619678.
33. Kragbæk, Mikael. "FizzBuzzEnterpriseEdition" (<https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition>). Retrieved 2024-11-19.
34. Meyer, Bertrand; Arnout, Karine (July 2006). "Componentization: The Visitor Example" (<http://se.ethz.ch/~meyer/publications/computer/visitor.pdf>) (PDF). *IEEE Computer*. **39** (7): 23–30. CiteSeerX 10.1.1.62.6082 (<https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.62.6082>). doi:10.1109/MC.2006.227 (<https://doi.org/10.1109%2FMC.2006.227>). S2CID 15328522 (<https://api.semanticscholar.org/CorpusID:15328522>).
35. *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media. 2020. ISBN 978-1492043454.
36. *Design Patterns: Elements of Reusable Object-Oriented Software*. ISBN 978-0201633610.
37. *Patterns of Enterprise Application Architecture*. ISBN 978-0321127426.

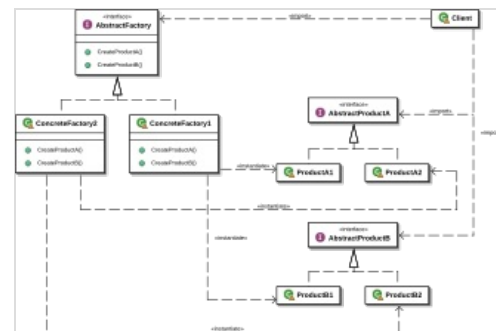
Further reading

- Alexander, Christopher; Ishikawa, Sara; Silverstein, Murray; Jacobson, Max; Fiksdahl-King, Ingrid; Angel, Shlomo (1977). *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press. ISBN 978-0-19-501919-3.
- Alur, Deepak; Crupi, John; Malks, Dan (May 2003). *Core J2EE Patterns: Best Practices and Design Strategies* (2nd ed.). Prentice Hall. ISBN 978-0-13-142246-9.
- Beck, Kent (October 2007). *Implementation Patterns*. Addison-Wesley. ISBN 978-0-321-41309-3.
- Beck, Kent; Crocker, R.; Meszaros, G.; Coplien, J. O.; Dominick, L.; Paulisch, F.; Vlissides, J. (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 978-0-471-49828-5.
- Coplien, James O.; Schmidt, Douglas C. (1995). *Pattern Languages of Program Design*. Addison-Wesley. ISBN 978-0-201-60734-5.

- Coplien, James O.; Vlissides, John M.; Kerth, Norman L. (1996). *Pattern Languages of Program Design 2*. Addison-Wesley. ISBN 978-0-201-89527-8.
- Eloranta, Veli-Pekka; Koskinen, Johannes; Leppänen, Marko; Reijonen, Ville (2014). *Designing Distributed Control Systems: A Pattern Language Approach*. Wiley. ISBN 978-1118694152.
- Fowler, Martin (1997). *Analysis Patterns: Reusable Object Models*. Addison-Wesley. ISBN 978-0-201-89542-1.
- Fowler, Martin (2003). *Patterns of Enterprise Application Architecture*. Addison-Wesley. ISBN 978-0-321-12742-6.
- Freeman, Eric; Freeman, Elisabeth; Sierra, Kathy; Bates, Bert (2004). *Head First Design Patterns*. O'Reilly Media. ISBN 978-0-596-00712-6.
- Hohmann, Luke; Fowler, Martin; Kawasaki, Guy (2003). *Beyond Software Architecture*. Addison-Wesley. ISBN 978-0-201-77594-5.
- Gabriel, Richard (1996). *Patterns of Software: Tales From The Software Community* (<https://web.archive.org/web/20030801111358/http://dreamsongs.com/NewFiles/PatternsOfSoftware.pdf>) (PDF). Oxford University Press. p. 235. ISBN 978-0-19-512123-0. Archived from the original (<http://www.dreamsongs.com/NewFiles/PatternsOfSoftware.pdf>) (PDF) on 2003-08-01.
- Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 978-0-201-63361-0.
- Hohpe, Gregor; Woolf, Bobby (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley. ISBN 978-0-321-20068-6.
- Holub, Allen (2004). *Holub on Patterns*. Apress. ISBN 978-1-59059-388-2.
- Kircher, Michael; Völter, Markus; Zdun, Uwe (2005). *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware* (<https://archive.org/details/remotingpatterns0000volt>). John Wiley & Sons. ISBN 978-0-470-85662-8.
- Larman, Craig (2005). *Applying UML and Patterns*. Prentice Hall. ISBN 978-0-13-148906-6.
- Liskov, Barbara; Guttag, John (2000). *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley. ISBN 978-0-201-65768-5.
- Manolescu, Dragos; Voelter, Markus; Noble, James (2006). *Pattern Languages of Program Design 5*. Addison-Wesley. ISBN 978-0-321-32194-7.
- Marinescu, Floyd (2002). *EJB Design Patterns: Advanced Patterns, Processes and Idioms* (<https://archive.org/details/ejbdesignpatterns00mari>). John Wiley & Sons. ISBN 978-0-471-20831-0.
- Martin, Robert Cecil; Riehle, Dirk; Buschmann, Frank (1997). *Pattern Languages of Program Design 3*. Addison-Wesley. ISBN 978-0-201-31011-5.
- Mattson, Timothy G.; Sanders, Beverly A.; Massingill, Berna L. (2005). *Patterns for Parallel Programming* (<https://archive.org/details/patternsforparallel0000matt>). Addison-Wesley. ISBN 978-0-321-22811-6.
- Shalloway, Alan; Trott, James R. (2001). *Design Patterns Explained, Second Edition: A New Perspective on Object-Oriented Design* (https://archive.org/details/isbn_9780321247148). Addison-Wesley. ISBN 978-0-321-24714-8.
- Vlissides, John M. (1998). *Pattern Hatching: Design Patterns Applied*. Addison-Wesley. ISBN 978-0-201-43293-0.
- Weir, Charles; Noble, James (2000). *Small Memory Software: Patterns for systems with limited memory* (<https://web.archive.org/web/20070617114432/http://www.cix.co.uk/~smallmemory>). Addison-Wesley. ISBN 978-0-201-59607-6. Archived from the original (<http://www.cix.co.uk/~smallmemory/>) on 2007-06-17.

Abstract factory pattern

The **abstract factory pattern** in software engineering is a design pattern that provides a way to create families of related objects without imposing their concrete classes, by encapsulating a group of individual factories that have a common theme without specifying their concrete classes.^[1] According to this pattern, a client software component creates a concrete implementation of the abstract factory and then uses the generic interface of the factory to create the concrete objects that are part of the family. The client does not know which concrete objects it receives from each of these internal factories, as it uses only the generic interfaces of their products.^[1] This pattern separates the details of implementation of a set of objects from their general usage and relies on object composition, as object creation is implemented in methods exposed in the factory interface.^[2]



UML class diagram

Use of this pattern enables interchangeable concrete implementations without changing the code that uses them, even at runtime. However, employment of this pattern, as with similar design patterns, may result in unnecessary complexity and extra work in the initial writing of code. Additionally, higher levels of separation and abstraction can result in systems that are more difficult to debug and maintain.

Overview

The abstract factory design pattern is one of the 23 patterns described in the 1994 *Design Patterns* book. It may be used to solve problems such as:^[3]

- How can an application be independent of how its objects are created?
- How can a class be independent of how the objects that it requires are created?
- How can families of related or dependent objects be created?

Creating objects directly within the class that requires the objects is inflexible. Doing so commits the class to particular objects and makes it impossible to change the instantiation later without changing the class. It prevents the class from being reusable if other objects are required, and it makes the class difficult to test because real objects cannot be replaced with mock objects.

A factory is the location of a concrete class in the code at which objects are constructed. Implementation of the pattern intends to insulate the creation of objects from their usage and to create families of related objects without depending on their concrete classes.^[2] This allows for new derived types to be introduced with no change to the code that uses the base class.

The pattern describes how to solve such problems:

- Encapsulate object creation in a separate (factory) object by defining and implementing an interface for creating objects.
- Delegate object creation to a factory object instead of creating objects directly.

This makes a class independent of how its objects are created. A class may be configured with a factory object, which it uses to create objects, and the factory object can be exchanged at runtime.

Definition

Design Patterns describes the abstract factory pattern as "an interface for creating families of related or dependent objects without specifying their concrete classes."^[4]

Usage

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

This insulates client code from object creation by having clients request that a factory object create an object of the desired abstract type and return an abstract pointer to the object.^[5]

An example is an abstract factory class `DocumentCreator` that provides interfaces to create a number of products (e.g., `createLetter()` and `createResume()`). The system would have any number of derived concrete versions of the `DocumentCreator` class such as `FancyDocumentCreator` or `ModernDocumentCreator`, each with a different implementation of `createLetter()` and `createResume()` that would create corresponding objects such as `FancyLetter` or `ModernResume`.

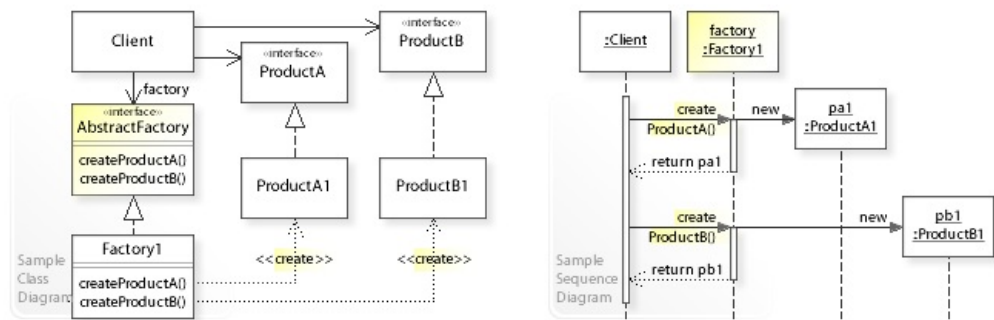
Each of these products is derived from a simple abstract class such as `Letter` or `Resume` of which the client is aware. The client code would acquire an appropriate instance of the `DocumentCreator` and call its factory methods. Each of the resulting objects would be created from the same `DocumentCreator` implementation and would share a common theme. The client would only need to know how to handle the abstract `Letter` or `Resume` class, not the specific version that was created by the concrete factory.

As the factory only returns a reference or a pointer to an abstract type, the client code that requested the object from the factory is not aware of—and is not burdened by—the actual concrete type of the object that was created. However, the abstract factory knows the type of a concrete object (and hence a concrete factory). For instance, the factory may read the object's type from a configuration file. The client has no need to specify the type, as the type has already been specified in the configuration file. In particular, this means:

- The client code has no knowledge of the concrete type, not needing to include any header files or class declarations related to it. The client code deals only with the abstract type. Objects of a concrete type are indeed created by the factory, but the client code accesses such objects only through their abstract interfaces.^[6]
- Adding new concrete types is performed by modifying the client code to use a different factory, a modification that is typically one line in one file. The different factory then creates objects of a different concrete type but still returns a pointer of the same abstract type as before, thus insulating the client code from change. This is significantly easier than modifying the client code to instantiate a new type. Doing so would require changing every location in the code where a new object is created as well as ensuring that all such code locations have knowledge of the new concrete type, for example, by including a concrete class header file. If all factory objects are stored globally in a singleton object, and all client code passes through the singleton to access the proper factory for object creation, then changing factories is as easy as changing the singleton object.^[6]

Structure

UML diagram



A sample UML class and sequence diagram for the abstract factory design pattern.^[7]

In the above UML class diagram, the `Client` class that requires `ProductA` and `ProductB` objects does not instantiate the `ProductA1` and `ProductB1` classes directly. Instead, the `Client` refers to the `AbstractFactory` interface for creating objects, which makes the `Client` independent of how the objects are created (which concrete classes are instantiated). The `Factory1` class implements the `AbstractFactory` interface by instantiating the `ProductA1` and `ProductB1` classes.

The UML sequence diagram shows the runtime interactions. The `Client` object calls `createProductA()` on the `Factory1` object, which creates and returns a `ProductA1` object. Thereafter, the `Client` calls `createProductB()` on `Factory1`, which creates and returns a `ProductB1` object.

Variants

The original structure of the abstract factory pattern, as defined in 1994 in *Design Patterns*, is based on abstract classes for the abstract factory and the abstract products to be created. The concrete factories and products are classes that specialize the abstract classes using inheritance.^[4]

A more recent structure of the pattern is based on interfaces that define the abstract factory and the abstract products to be created. This design uses native support for interfaces or protocols in mainstream programming languages to avoid inheritance. In this case, the concrete factories and products are classes that realize the interface by implementing it.^[1]

Example

This C++23 implementation is based on the pre-C++98 implementation in the book.

```
import std;
```

```

using std::array;
using std::shared_ptr;
using std::unique_ptr;
using std::vector;

enum class Direction {
    North,
    South,
    East,
    West
};

class MapSite {
public:
    virtual void enter() = 0;
    virtual ~MapSite() = default;
};

class Room: public MapSite {
private:
    int roomNumber;
    shared_ptr<array<MapSite, 4>> sides;
public:
    Room():
        roomNumber{0} {}

    explicit Room(int n):
        roomNumber{n} {}

    Room& setSide(Direction d, MapSite* ms) {
        sides[static_cast<size_t>(d)] = std::move(ms);
        std::println("Room::setSide {} ms", d);
        return *this;
    }

    virtual void enter() override = 0;

    Room(const Room&) = delete;
    Room& operator=(const Room&) = delete;
};

class Wall: public MapSite {
public:
    Wall():
        MapSite() {}

    explicit Wall(int n):
        MapSite(n) {}

    void enter() override {
        // ...
    }
};

class Door: public MapSite {
private:
    shared_ptr<Room> room1;
    shared_ptr<Room> room2;
public:
    Door(shared_ptr<Room> r1 = nullptr, shared_ptr<Room> r2 = nullptr):
        MapSite(), room1{std::move(r1)}, room2{std::move(r2)} {}

    explicit Door(int n, shared_ptr<Room> r1 = nullptr, shared_ptr<Room> r2 = nullptr):
        MapSite(n), room1{std::move(r1)}, room2{std::move(r2)} {}

    void enter() override {
        // ...
    }

    Door(const Door&) = delete;
    Door& operator=(const Door&) = delete;
};

class Maze {
private:
    vector<shared_ptr<Room>> rooms;
public:
    Maze& addRoom(shared_ptr<Room> r) {
        std::println("Maze::addRoom {}", reinterpret_cast<void*>(r.get()));
        rooms.push_back(std::move(r));
        return *this;
    }

    shared_ptr<Room> roomNo(int n) const {
        for (const Room& r: rooms) {
            // actual lookup logic here...
        }
        return nullptr;
    }
};

class MazeFactory {
public:
    MazeFactory() = default;

    virtual ~MazeFactory() = default;

    [[nodiscard]]
    unique_ptr<Maze> makeMaze() const {
        return std::make_unique<Maze>();
    }

    [[nodiscard]]
    shared_ptr<Wall> makeWall() const {
        return std::make_shared<Wall>();
    }

    [[nodiscard]]

```

```

    shared_ptr<Room> makeRoom(int n) const {
        return std::make_shared<Room>(new Room(n));
    }

    [[nodiscard]]
    shared_ptr<Door> makeDoor(shared_ptr<Room> r1, shared_ptr<Room> r2) const {
        return std::make_shared<Door>(std::move(r1), std::move(r2));
    }
};

// If createMaze is passed an object as a parameter to use to create rooms, walls, and doors, then you can change the classes of rooms, walls,
// and doors by passing a different parameter. This is an example of the Abstract Factory (99) pattern.

class MazeGame {
public:
    [[nodiscard]]
    unique_ptr<Maze> createMaze(MazeFactory& factory) {
        unique_ptr<Maze> maze = factory.makeMaze();
        shared_ptr<Room> r1 = factory.makeRoom(1);
        shared_ptr<Room> r2 = factory.makeRoom(2);
        shared_ptr<Door> door = factory.makeDoor(r1, r2);
        maze->addRoom(r1)
            .addRoom(r2)
            .setSide(Direction::North, factory.makeWall())
            .setSide(Direction::East, door)
            .setSide(Direction::South, factory.makeWall())
            .setSide(Direction::West, factory.makeWall())
            .setSide(Direction::North, factory.makeWall())
            .setSide(Direction::East, factory.makeWall())
            .setSide(Direction::South, factory.makeWall())
            .setSide(Direction::West, door);
        return maze;
    }
};

int main(int argc, char* argv[]) {
    MazeGame game;
    unique_ptr<Maze> maze = game.createMaze(MazeFactory());
}

```

The program output is:

```

Maze::addRoom 0x1317ed0
Maze::addRoom 0x1317ef0
Room::setSide 0 0x1318340
Room::setSide 2 0x1317f10
Room::setSide 1 0x1318360
Room::setSide 3 0x1318380
Room::setSide 0 0x13183a0
Room::setSide 2 0x13183c0
Room::setSide 1 0x13183e0
Room::setSide 3 0x1317f10

```

See also

- [Concrete class](#)
- [Factory method pattern](#)
- [Object creation](#)
- [Software design pattern](#)


References

- Freeman, Eric; Robson, Elisabeth; Sierra, Kathy; Bates, Bert (2004). Hendrickson, Mike; Loukides, Mike (eds.). *Head First Design Patterns* (<http://shop.oreilly.com/product/9780596007126.do>) (paperback). Vol. 1. O'REILLY. p. 156. ISBN 978-0-596-00712-6. Retrieved 2012-09-12.
- Freeman, Eric; Robson, Elisabeth; Sierra, Kathy; Bates, Bert (2004). Hendrickson, Mike; Loukides, Mike (eds.). *Head First Design Patterns* (<http://shop.oreilly.com/product/9780596007126.do>) (paperback). Vol. 1. O'REILLY. p. 162. ISBN 978-0-596-00712-6. Retrieved 2012-09-12.
- "The Abstract Factory design pattern - Problem, Solution, and Applicability" (<http://w3sdesign.com/?gr=c01&ugr=proble>). *w3sDesign.com*. Retrieved 2017-08-11.
- Gamma, Erich; Richard Helm; Ralph Johnson; John M. Vlissides (2009-10-23). "Design Patterns: Abstract Factory" (<https://web.archive.org/web/20120516213805/http://www.informit.com/>). informIT. Archived from the original on 2012-05-16. Retrieved 2012-05-16. "Object Creational: Abstract Factory: Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes."
- Veeneman, David (2009-10-23). "Object Design for the Perplexed" (<https://web.archive.org/web/20110221224616/http://www.codeproject.com/>). The Code Project. Archived from the original on 2011-02-21. Retrieved 2012-05-16. "The factory insulates the client from changes to the product or how it is created, and it can provide this insulation across objects derived from very different abstract interfaces."
- "Abstract Factory: Implementation" (<http://www.oodeesign.com/abstract-factory-pattern.html>). OODesign.com. Retrieved

2012-05-16.

7. "The Abstract Factory design pattern - Structure and Collaboration" (<http://w3sdesign.com/?gr=c01&ugr=struct>). *w3sDesign.com*. Retrieved 2017-08-12.

External links

-  Media related to Abstract factory at Wikimedia Commons
 - [Abstract Factory \(https://web.archive.org/web/20151101110755/http://www.patterns.pl/abstractfactory.html\)](https://web.archive.org/web/20151101110755/http://www.patterns.pl/abstractfactory.html) Abstract Factory implementation example
-

Retrieved from "https://en.wikipedia.org/w/index.php?title=Abstract_factory_pattern&oldid=1318448817"

Builder pattern

The **builder pattern** is a design pattern that provides a flexible solution to various object creation problems in object-oriented programming. The builder pattern separates the construction of a complex object from its representation. It is one of the 23 classic design patterns described in the book *Design Patterns* and is sub-categorized as a creational pattern.^[1]

Overview

The builder design pattern solves problems like:^[2]

- How can a class (the same construction process) create different representations of a complex object?
- How can a class that includes creating a complex object be simplified?

Creating and assembling the parts of a complex object directly within a class is inflexible. It commits the class to creating a particular representation of the complex object and makes it impossible to change the representation later independently from (without having to change) the class.

The builder design pattern describes how to solve such problems:

- Encapsulate creating and assembling the parts of a complex object in a separate Builder object.
- A class delegates object creation to a Builder object instead of creating the objects directly.

A class (the same construction process) can delegate to different Builder objects to create different representations of a complex object.

Definition

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

Advantages

Advantages of the builder pattern include:^[3]

- Allows you to vary a product's internal representation.
- Encapsulates code for construction and representation.
- Provides control over the steps of the construction process.

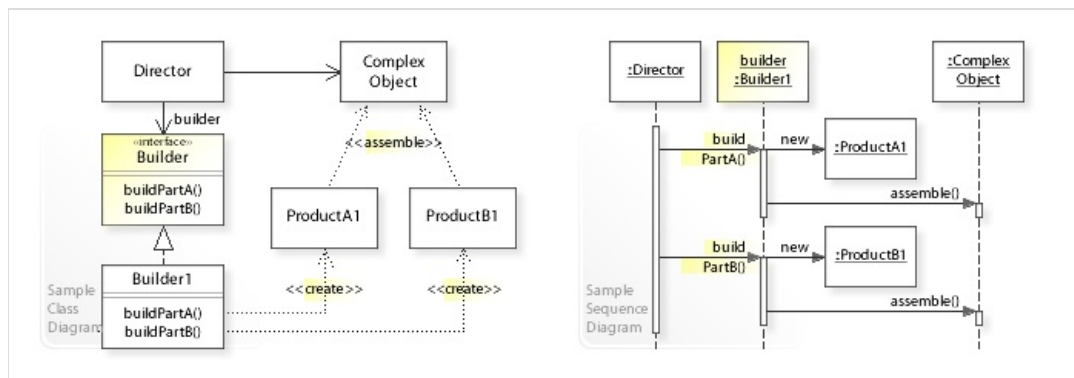
Disadvantages

Disadvantages of the builder pattern include:

- A distinct ConcreteBuilder must be created for each type of product.^[3]
- Builder classes must be mutable.
- May hamper/complicate dependency injection.
- In many null-safe languages, the builder pattern defers compile-time errors for unset fields to runtime.

Structure

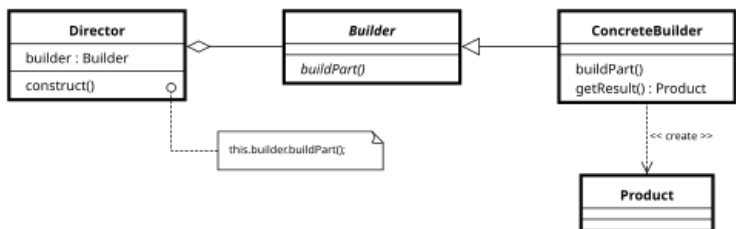
UML class and sequence diagram



A sample UML class and sequence diagram for the builder design pattern.^[4]

In the above UML class diagram, the Director class doesn't create and assemble the ProductA1 and ProductB1 objects directly. Instead, the Director refers to the Builder interface for building (creating and assembling) the parts of a complex object, which makes the Director independent of which concrete classes are instantiated (which representation is created). The Builder1 class implements the Builder interface by creating and assembling the ProductA1 and ProductB1 objects. The UML sequence diagram shows the run-time interactions: The Director object calls buildPartA() on the Builder1 object, which creates and assembles the ProductA1 object. Thereafter, the Director calls buildPartB() on Builder1, which creates and assembles the ProductB1 object.

Class diagram



Builder

Abstract interface for creating objects (product).

ConcreteBuilder

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

Examples

A C# example:

```

/// <summary>
/// Represents a product created by the builder.
/// </summary>
public class Bicycle
{
    public Bicycle(string make, string model, string colour, int height)
    {
        Make = make;
        Model = model;
        Colour = colour;
        Height = height;
    }

    public string Make { get; set; }
    public string Model { get; set; }
    public int Height { get; set; }
    public string Colour { get; set; }
}

/// <summary>
/// The builder abstraction.
/// </summary>
public interface IBicycleBuilder
{
    Bicycle GetResult();

    string Colour { get; set; }
    int Height { get; set; }
}

/// <summary>
/// Concrete builder implementation.
/// </summary>
public class GTBuilder : IBicycleBuilder
{
    public Bicycle GetResult()
    {

```

```
        return Height == 29 ? new Bicycle("GT", "Avalanche", Colour, Height) : null;
    }

    public string Colour { get; set; }
    public int Height { get; set; }
}

/// <summary>
/// The director.
/// </summary>
public class MountainBikeBuildDirector
{
    private IBicycleBuilder _builder;

    public MountainBikeBuildDirector(IBicycleBuilder builder)
    {
        _builder = builder;
    }

    public void Construct()
    {
        _builder.Colour = "Red";
        _builder.Height = 29;
    }

    public Bicycle GetResult()
    {
        return this._builder.GetResult();
    }
}

public class Client
{
    public void DoSomethingWithBicycles()
    {
        MountainBikeBuildDirector director = new(new GTBuilder());
        // Director controls the stepwise creation of product and returns the result.
        director.Construct();
        Bicycle myMountainBike = director.GetResult();
    }
}
```

The Director assembles a bicycle instance in the example above, delegating the construction to a separate builder object that has been given to the Director by the Client.

See also

- Currying

Notes

- [Gamma et al. 1994](#), p. 97.
- "The Builder design pattern - Problem, Solution, and Applicability" (<http://w3sdesign.com/?gr=c02&ugr=proble>). *w3sDesign.com*. Retrieved 2017-08-13.
- "Index of /archive/2010/winter/51023-1/presentations" (https://www.classes.cs.uchicago.edu/archive/2010/winter/51023-1/presentations/ricetj_builder.pdf) (PDF). *www.classes.cs.uchicago.edu*. Retrieved 2016-03-03.
- "The Builder design pattern - Structure and Collaboration" (<http://w3sdesign.com/?gr=c02&ugr=struct>). *w3sDesign.com*. Retrieved 2017-08-12.

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.

External links

Retrieved from "https://en.wikipedia.org/w/index.php?title=Builder_pattern&oldid=1309339296"

Dependency injection

In software engineering, **dependency injection** is a programming technique in which an object or function receives other objects or functions that it requires, as opposed to creating them internally. Dependency injection aims to separate the concerns of constructing objects and using them, leading to loosely coupled programs. The pattern ensures that an object or function that wants to use a given service should not have to know how to construct those services. Instead, the receiving "client" (object or function) is provided with its dependencies by external code (an "injector"), which it is not aware of. Dependency injection makes implicit dependencies explicit and helps solve the following problems:

- How can a class be independent from the creation of the objects it depends on?
- How can an application and the objects it uses support different configurations?

Dependency injection is often used to keep code in-line with the dependency inversion principle.

In statically typed languages using dependency injection means that a client only needs to declare the interfaces of the services it uses, rather than their concrete implementations, making it easier to change which services are used at runtime without recompiling.

Application frameworks often combine dependency injection with inversion of control. Under inversion of control, the framework first constructs an object (such as a controller), and then passes control flow to it. With dependency injection, the framework also instantiates the dependencies declared by the application object (often in the constructor method's parameters), and passes the dependencies into the object.

Dependency injection implements the idea of "inverting control over the implementations of dependencies", which is why certain Java frameworks generically name the concept "inversion of control" (not to be confused with inversion of control flow).

Roles

Dependency injection involves four roles: services, clients, interfaces and injectors.

Services and clients

A service is any class which contains useful functionality. In turn, a client is any class which uses services. The services that a client requires are the client's dependencies.

Any object can be a service or a client; the names relate only to the role the objects play in an injection. The same object may even be both a client (it uses injected services) and a service (it is injected into other objects). Upon injection, the service is made part of the client's state, available for use.

Interfaces

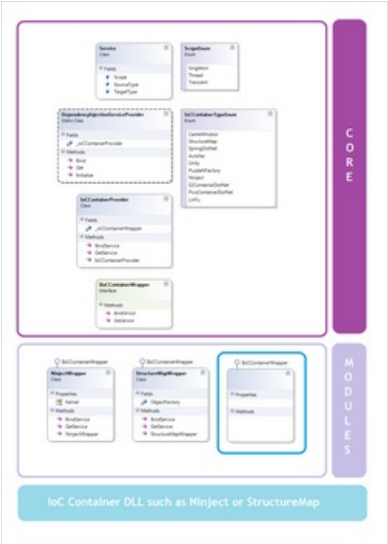
Clients should not know how their dependencies are implemented, only their names and API. A service which retrieves emails, for instance, may use the IMAP or POP3 protocols behind the scenes, but this detail is likely irrelevant to calling code that merely wants an email retrieved. By ignoring implementation details, clients do not need to change when their dependencies do.

Injectors

The injector, sometimes also called an assembler, container, provider or factory, introduces services to the client.

The role of injectors is to construct and connect complex object graphs, where objects may be both clients and services. The injector itself may be many objects working together, but must not be the client, as this would create a circular dependency.

Because dependency injection separates how objects are constructed from how they are used, it often diminishes the importance of the new keyword found in most object-oriented languages. Because the framework handles creating services, the programmer tends to only directly construct value objects which represents entities in the program's domain (such as an Employee object in a business app or an Order object in a shopping app).



Dependency injection is often used alongside specialized frameworks, known as 'containers', to facilitate program composition.



PetManager gets injected into PetController and PetRepository gets injected into PetManager

Dependency injection for five-year-olds

When you go and get things out of the refrigerator for yourself, you can cause problems. You might leave the door open, you might get something Mommy or Daddy don't want you to have. You might even be looking for something we don't even have or which has expired.

What you should be doing is stating a need, "I need something to drink with lunch," and then we will make sure you have something when you sit down to eat something.

John Munsch, 28 October 2009.

Analogy

As an analogy, cars can be thought of as services which perform the useful work of transporting people from one place to another. Car engines can require gas, diesel or electricity, but this detail is unimportant to the client—a passenger—who only cares if it can get them to their destination.

Cars present a uniform interface through their pedals, steering wheels and other controls. As such, which engine they were 'injected' with on the factory line ceases to matter and drivers can switch between any kind of car as needed.

Advantages and disadvantages

Advantages

A basic benefit of dependency injection is decreased coupling between classes and their dependencies.^{[17][18]}

By removing a client's knowledge of how its dependencies are implemented, programs become more reusable, testable and maintainable.^[19]

This also results in increased flexibility: a client may act on anything that supports the intrinsic interface the client expects.^[20]

More generally, dependency injection reduces boilerplate code, since all dependency creation is handled by a singular component.^[19]

Finally, dependency injection allows concurrent development. Two developers can independently develop classes that use each other, while only needing to know the interface the classes will communicate through. Plugins are often developed by third-parties that never even talk to developers of the original product.^[21]

Testing

Many of dependency injection's benefits are particularly relevant to unit-testing.

For example, dependency injection can be used to externalize a system's configuration details into configuration files, allowing the system to be reconfigured without recompilation. Separate configurations can be written for different situations that require different implementations of components.^[22]

Similarly, because dependency injection does not require any change in code behavior, it can be applied to legacy code as a refactoring. This makes clients more independent and are easier to unit test in isolation, using stubs or mock objects, that simulate other objects not under test.

This ease of testing is often the first benefit noticed when using dependency injection.^[23]

Disadvantages

Critics of dependency injection argue that it:

- Creates clients that demand configuration details, which can be onerous when obvious defaults are available.^[21]
- Makes code difficult to trace because it separates behavior from construction.^[21]
- Is typically implemented with reflection or dynamic programming, hindering IDE automation.^[24]
- Typically requires more upfront development effort.^[25]
- Encourages dependence on a framework.^{[26][27][28]}

Types of dependency injection

There are several ways in which a client can receive injected services:^[29]

- Constructor injection, where dependencies are provided through a client's class constructor.
- Method Injection, where dependencies are provided to a method only when required for specific functionality.
- Setter injection, where the client exposes a setter method which accepts the dependency.
- Interface injection, where the dependency's interface provides an injector method that will inject the dependency into any client passed to it.

In some frameworks, clients do not need to actively accept dependency injection at all. In Java, for example, reflection can make private attributes public when testing and inject services directly.^[30]

Without dependency injection

In the following Java example, the `Client` class contains a `Service` member variable initialized in the constructor. The client directly constructs and controls which service it uses, creating a hard-coded dependency.

```
public class Client {
    private Service service;

    Client() {
        // The dependency is hard-coded.
        this.service = new ExampleService();
    }
}
```

Constructor injection

The most common form of dependency injection is for a class to request its dependencies through its constructor. This ensures the client is always in a valid state, since it cannot be instantiated without its necessary dependencies.

```
public class Client {
    private Service service;

    // The dependency is injected through a constructor.
    Client(final Service service) {
        if (service == null) {
            throw new IllegalArgumentException("service must not be null");
        }
        this.service = service;
    }
}
```

Method Injection

Dependencies are passed as arguments to a specific method, allowing them to be used only during that method's execution without maintaining a long-term reference. This approach is particularly useful for temporary dependencies or when different implementations are needed for various method calls.

```
public class Client {
    public void performAction(Service service) {
        if (service == null) {
            throw new IllegalArgumentException("service must not be null");
        }
        service.execute();
    }
}
```

Setter injection

By accepting dependencies through a setter method, rather than a constructor, clients can allow injectors to manipulate their dependencies at any time. This offers flexibility, but makes it difficult to ensure that all dependencies are injected and valid before the client is used.

```
public class Client {
    private Service service;

    // The dependency is injected through a setter method.
    public void setService(final Service service) {
        if (service == null) {
            throw new IllegalArgumentException("service must not be null");
        }
        this.service = service;
    }
}
```

Interface injection

With interface injection, dependencies are completely ignorant of their clients, yet still send and receive references to new clients.

In this way, the dependencies become injectors. The key is that the injecting method is provided through an interface.

An assembler is still needed to introduce the client and its dependencies. The assembler takes a reference to the client, casts it to the setter interface that sets that dependency, and passes it to that dependency object which in turn passes a reference to itself back to the client.

For interface injection to have value, the dependency must do something in addition to simply passing back a reference to itself. This could be acting as a factory or sub-assembler to resolve other dependencies, thus abstracting some details from the main assembler. It could be reference-counting so that the dependency knows how many clients are using it. If the dependency maintains a collection of clients, it could later inject them all with a different instance of itself.

```
public interface ServiceSetter {
    void setService(Service service);
}
```



```

}

public class Client implements ServiceSetter {
    private Service service;

    @Override
    public void setService(final Service service) {
        if (service == null) {
            throw new IllegalArgumentException("service must not be null");
        }
        this.service = service;
    }
}

public class ServiceInjector {
    private final Set<ServiceSetter> clients = new HashSet<>();

    public void inject(final ServiceSetter client) {
        this.clients.add(client);
        client.setService(new ExampleService());
    }

    public void switch() {
        for (final Client client : this.clients) {
            client.setService(new AnotherExampleService());
        }
    }
}

public class ExampleService implements Service {}

public class AnotherExampleService implements Service {}

```

Assembly

The simplest way of implementing dependency injection is to manually arrange services and clients, typically done at the program's root, where execution begins.

```

public class Program {

    public static void main(final String[] args) {
        // Build the service.
        final Service service = new ExampleService();

        // Inject the service into the client.
        final Client client = new Client(service);

        // Use the objects.
        System.out.println(client.greet());
    }
}

```

Manual construction may be more complex and involve builders, factories, or other construction patterns.

Frameworks

Manual dependency injection is often tedious and error-prone for larger projects, promoting the use of frameworks which automate the process. Manual dependency injection becomes a dependency injection framework once the constructing code is no longer custom to the application and is instead universal.^[31] While useful, these tools are not required in order to perform dependency injection.^{[32][33]}

Some frameworks, like Spring, can use external configuration files to plan program composition:

```

import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Injector {

    public static void main(final String[] args) {
        // Details about which concrete service to use are stored in configuration separate from the
        program itself.
        final BeanFactory beanfactory = new ClassPathXmlApplicationContext("Beans.xml");
        final Client client = (Client) beanfactory.getBean("client");
        System.out.println(client.greet());
    }
}

```

Even with a potentially long and complex object graph, the only class mentioned in code is the entry point, in this case `Client`.`Client` has not undergone any changes to work with Spring and remains a POJO.^{[34][35][36]} By keeping Spring-specific annotations and calls from spreading out among many classes, the system stays only loosely dependent on Spring.^[27]

Examples



StructureMap are commonly used in object-oriented programming languages to achieve Dependency Injection and inversion of control.

AngularJS

The following example shows an AngularJS component receiving a greeting service through dependency injection.

```
function SomeClass(greeter) {
    this.greeter = greeter;
}

SomeClass.prototype.doSomething = function(name) {
    this.greeter.greet(name);
}
```

Each AngularJS application contains a service locator responsible for the construction and look-up of dependencies.

```
// Provide the wiring information in a module
var myModule = angular.module('myModule', []);

// Teach the injector how to build a greeter service.
// greeter is dependent on the $window service.
myModule.factory('greeter', function($window) {
    return {
        greet: function(text) {
            $window.alert(text);
        }
    };
});
```

We can then create a new injector that provides components defined in the `myModule` module, including the greeter service.

```
var injector = angular.injector(['myModule', 'ng']);
var greeter = injector.get('greeter');
```

To avoid the service locator antipattern, AngularJS allows declarative notation in HTML templates which delegates creating components to the injector.

```
<div ng-controller="MyController">
  <button ng-click="sayHello()">Hello</button>
</div>
```

```
function MyController($scope, greeter) {
    $scope.sayHello = function() {
        greeter.greet('Hello World');
    };
}
```

The `ng-controller` directive triggers the injector to create an instance of the controller and its dependencies.

C++

This sample provides an example of constructor injection in C++.

```
import std;

class DatabaseConnection {
public:
    void connect() {
        std::println("Connecting to database...");
    }
};

class DatabaseService {
private:
    DatabaseConnection& dbConn;
public:
    explicit DatabaseService(DatabaseConnection& db):
        dbConn{db} {}

    void execute() {
        dbConn.connect();
        std::println("Executing database service...");
    }
};

int main(int argc, char* argv[]) {
    DatabaseConnection db;
    DatabaseService sv(db);

    sv.execute();
}
```

This sample provides an example of interface injection in C++.

```
import std;

using std::expected;
using std::shared_ptr;
using std::unexpected;
```

```

enum class DatabaseConnectionError {
    NoConnection,
    // more errors here
};

class IConnection {
public:
    virtual void connect() = 0;
    virtual ~IConnection() = default;
};

class DatabaseConnection: public IConnection {
public:
    DatabaseConnection() = default;

    void connect() override {
        std::println("Connecting to database...");
    }
};

class DatabaseService {
private:
    shared_ptr<IConnection> conn;
public:
    DatabaseService() = default;

    void setConnection(shared_ptr<IConnection> nextConn) noexcept {
        conn = nextConn;
    }

    expected<void, DatabaseConnectionError> execute() {
        if (conn) {
            conn->connect();
            std::println("Executing database service...");
        } else {
            return unexpected(DatabaseConnectionError::NoConnection);
        }
    }
};

int main(int argc, char* argv[]) {
    shared_ptr<DatabaseConnection> db = std::make_shared<DatabaseConnection>();
    DatabaseService sv;
    sv.setConnection(db);
    sv.execute();
}

```

C#

This sample provides an example of constructor injection in [C#](#).

```

using System;

namespace DependencyInjection;

// Our client will only know about this interface, not which specific gamepad it is using.
interface IGamePadFunctionality
{
    string GetGamePadName();
    void SetVibrationPower(float power);
}

// The following services provide concrete implementations of the above interface.

class XboxGamePad : IGamePadFunctionality
{
    float vibrationPower = 1.0f;

    public string GetGamePadName() => "Xbox controller";

    public void SetVibrationPower(float power) => this.vibrationPower = Math.Clamp(power, 0.0f, 1.0f);
}

class PlayStationJoystick : IGamePadFunctionality
{
    float vibratingPower = 100.0f;

    public string GetGamePadName() => "PlayStation controller";

    public void SetVibrationPower(float power) => this.vibratingPower = Math.Clamp(power * 100.0f, 0.0f, 100.0f);
}

class SteamController : IGamePadFunctionality
{
    double vibrating = 1.0;

    public string GetGamePadName() => "Steam controller";

    public void SetVibrationPower(float power) => this.vibrating = Convert.ToDouble(Math.Clamp(power, 0.0f, 1.0f));
}

// This class is the client which receives a service.
class GamePad
{
    IGamePadFunctionality gamePadFunctionality;

    // The service is injected through the constructor and stored in the above field.
    public GamePad(IGamePadFunctionality gamePadFunctionality) => this.gamePadFunctionality = gamePadFunctionality;

    public void Showcase()
    {
        // The injected service is used.
        string gamePadName = this.gamePadFunctionality.GetGamePadName();
        string message = $"We're using the {gamePadName} right now, do you want to change the vibrating power?";
        Console.WriteLine(message);
    }
}

```

```

    }
}

class Program {
    static void Main(string[] args)
    {
        SteamController steamController = new SteamController();

        // We could have also passed in an XboxController, PlayStationJoystick, etc.
        // The gamepad doesn't know what it's using and doesn't need to.
        GamePad gamepad = new GamePad(steamController);

        gamepad.Showcase();
    }
}

```

Go

Go does not support classes and usually dependency injection is either abstracted by a dedicated library that utilizes reflection or generics (the latter being supported since Go 1.18^[37]).^[38] A simpler example without using dependency injection libraries is illustrated by the following example of an MVC web application.

First, pass the necessary dependencies to a router and then from the router to the controllers:

```

package router

import (
    "database/sql"
    "net/http"

    "example/controllers/users"

    "github.com/go-chi/chi/v5"
    "github.com/go-chi/chi/v5/middleware"

    "github.com/redis/go-redis/v9"
    "github.com/rs/zerolog"
)

type RoutingHandler struct {
    // passing the values by pointer further down the call stack
    // means we won't create a new copy, saving memory
    log      *zerolog.Logger
    db       *sql.DB
    cache    *redis.Client
    router   chi.Router
}

// connection, logger and cache initialized usually in the main function
func NewRouter(
    log *zerolog.Logger,
    db *sql.DB,
    cache *redis.Client,
) (r *RoutingHandler) {
    rtr := chi.NewRouter()

    return &RoutingHandler{
        log:    log,
        db:     db,
        cache:  cache,
        router: rtr,
    }
}

func (r *RoutingHandler) SetupUsersRoutes() {
    uc := users.NewController(r.log, r.db, r.cache)

    r.router.Get("/users/:name", func(w http.ResponseWriter, r *http.Request) {
        uc.Get(w, r)
    })
}

```

Then, you can access the private fields of the struct in any method that is its pointer receiver, without violating encapsulation.

```

package users

import (
    "database/sql"
    "net/http"

    "example/models"

    "github.com/go-chi/chi/v5"
    "github.com/redis/go-redis/v9"
    "github.com/rs/zerolog"
)

type Controller struct {
    log      *zerolog.Logger
    storage  models.UserStorage
    cache    *redis.Client
}

func NewController(log *zerolog.Logger, db *sql.DB, cache *redis.Client) *Controller {
    return &Controller{
        log:    log,
        storage: models.NewUserStorage(db),
        cache:  cache,
    }
}

```

```
func (uc *Controller) Get(w http.ResponseWriter, r *http.Request) {
    // note that we can also wrap logging in a middleware, this is for demonstration purposes
    uc.log.Info().Msg("Getting user")

    userParam := chi.URLParam(r, "name")

    var user *models.User
    // get the user from the cache
    err := uc.cache.Get(r.Context(), userParam).Scan(&user)
    if err != nil {
        uc.log.Error().Err(err).Msg("Error getting user from cache. Retrieving from SQL storage")
    }

    user, err = uc.storage.Get(r.Context(), "johndoe")
    if err != nil {
        uc.log.Error().Err(err).Msg("Error getting user from SQL storage")
        http.Error(w, "Internal server error", http.StatusInternalServerError)
        return
    }
}
```

Finally you can use the database connection initialized in your main function at the data access layer:

```
package models

import (
    "database/sql"
    "time"
)

type (
    UserStorage struct {
        conn *sql.DB
    }

    User struct {
        Name      string 'json:"name" db:"name,primarykey"'
        JoinedAt  time.Time 'json:"joined_at" db:"joined_at"'
        Email     string 'json:"email" db:"email"'
    }
)

func NewUserStorage(conn *sql.DB) *UserStorage {
    return &UserStorage{
        conn: conn,
    }
}

func (us *UserStorage) Get(name string) (user *User, err error) {
    // assuming 'name' is a unique key
    query := "SELECT * FROM users WHERE name = $1"

    if err := us.conn.QueryRow(query, name).Scan(&user); err != nil {
        return nil, err
    }

    return user, nil
}
```

See also

- Architecture description language
- Factory pattern
- Inversion of control
- Mock trainwreck
- Plug-in (computing)
- Strategy pattern
- Service locator pattern
- Parameter (computer programming)
- Quaject

References

- Seemann, Mark. "Dependency Injection is Loose Coupling" (<http://blog.ploeh.dk/2010/04/07/DependencyInjectionisLooseCoupling/>). *blog.ploeh.dk*. Retrieved 2015-07-28.
- Seeman, Mark (October 2011). *Dependency Injection in .NET*. Manning Publications. p. 4. ISBN 9781935182504.
- Niko Schwarz, Mircea Lungu, Oscar Nierstrasz, "Seuss: Decoupling responsibilities from static methods for fine-grained configurability", Journal of Object Technology, volume 11, no. 1 (April 2012), pp. 3:1–23.
- "HollywoodPrinciple" (<http://c2.com/cgi/wiki?HollywoodPrinciple>). *c2.com*. Retrieved 2015-07-19.
- "The Dependency Injection design pattern – Problem, Solution, and Applicability" (<http://w3sdesign.com/?gr=u01&ugr=proble>). *w3sDesign.com*. Retrieved 2017-08-12.

6. Erez, Guy (2022-03-09). "Dependency Inversion vs. Dependency Injection" (<https://betterprogramming.pub/straightforward-simple-dependency-inversion-vs-dependency-injection-7d8c0d0ed28e>). *Medium*. Retrieved 2022-12-06.
7. Mathews, Sasha (2021-03-25). "You are Simply Injecting a Dependency, Thinking that You are Following the Dependency Inversion..." (<https://levelup.gitconnected.com/you-are-simply-injecting-a-dependency-thinking-that-you-are-following-the-dependency-inversion-32632954c208>) *Medium*. Retrieved 2022-12-06.
8. "Spring IoC Container" (<https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/beans.html>). Retrieved 2023-05-23.
9. Fowler, Martin. "Inversion of Control Containers and the Dependency Injection pattern" (<https://martinfowler.com/articles/injection.html#InversionOfControl>). *MartinFowler.com*. Retrieved 4 June 2023.
10. "Dependency Injection in NET" (<https://web.archive.org/web/20150721171646/http://philkildea.co.uk/james/books/DependencyInjection.in.NET.pdf>) (PDF). *philkildea.co.uk*. p. 4. Archived from the original (<http://philkildea.co.uk/james/books/DependencyInjection.in.NET.pdf>) (PDF) on 2015-07-21. Retrieved 2015-07-18.
11. "How to explain dependency injection to a 5-year-old?" (<https://stackoverflow.com/questions/1638919/how-to-explain-dependency-injection-to-a-5-year-old>). *stackoverflow.com*. Retrieved 2015-07-18.
12. I.T., Titanium. "James Shore: Dependency Injection Demystified" (<http://www.jamesshore.com/Blog/Dependency-Injection-Demystified.html>). *www.jamesshore.com*. Retrieved 2015-07-18.
13. "To 'new' or not to 'new'..." (<https://web.archive.org/web/20200513185005/http://misko.hevery.com/2008/09/30/to-new-or-not-to-new/>) Archived from the original (<http://misko.hevery.com/2008/09/30/to-new-or-not-to-new/>) on 2020-05-13. Retrieved 2015-07-18.
14. "How to write testable code" (<http://www.loosecouplings.com/2011/01/how-to-write-testable-code-overview.html>). *www.loosecouplings.com*. Retrieved 2015-07-18.
15. "Writing Clean, Testable Code" (<http://www.ethanresnick.com/blog/testableCode.html>). *www.ethanresnick.com*. Retrieved 2015-07-18.
16. Sironi, Giorgio. "When to inject: the distinction between newables and injectables - Invisible to the eye" (<http://www.giorgiosironi.com/2009/07/when-to-inject-distinction-between.html>). *www.giorgiosironi.com*. Retrieved 2015-07-18.
17. "the urban canuk, eh: On Dependency Injection and Violating Encapsulation Concerns" (<http://www.bryancook.net/2011/08/on-dependency-injection-and-violating.html>). *www.bryancook.net*. Retrieved 2015-07-18.
18. "The Dependency Injection Design Pattern" ([https://msdn.microsoft.com/en-us/library/vstudio/hh323705\(v=vs.100\).aspx](https://msdn.microsoft.com/en-us/library/vstudio/hh323705(v=vs.100).aspx)). *msdn.microsoft.com*. Retrieved 2015-07-18.
19. "The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 330" (<https://jcp.org/en/jsr/detail?id=330>). *jcp.org*. Retrieved 2015-07-18.
20. "3.1. Dependency injection — Python 3: from None to Machine Learning" (<https://web.archive.org/web/20200208005839/http://python.astrotech.io/design-patterns/structural/dependency-injection.html>). Archived from the original (<https://python.astrotech.io/design-patterns/structural/dependency-injection.html>) on 2020-02-08.
21. "How Dependency Injection (DI) Works in Spring Java Application Development - DZone Java" (<https://dzone.com/articles/how-dependency-injection-di-works-in-spring-java-a>).
22. "Dependency injection and inversion of control in Python — Dependency Injector 4.36.2 documentation" (http://python-dependency-injector.ets-labs.org/introduction/di_in_python.html).
23. "How to Refactor for Dependency Injection, Part 3: Larger Applications" (<https://visualstudiomagazine.com/articles/2014/07/01/larger-applications.aspx>).
24. "A quick intro to Dependency Injection: What it is, and when to use it" (<https://www.freecodecamp.org/news/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f/>). 18 October 2018.
25. "Dependency Injection | Professionalqa.com" (<https://www.professionalqa.com/dependency-injection>).
26. "What are the downsides to using Dependency Injection?" (<https://stackoverflow.com/questions/2407540/what-are-the-downsides-to-using-dependency-injection>). *stackoverflow.com*. Retrieved 2015-07-18.
27. "Dependency Injection Inversion – Clean Coder" (<https://sites.google.com/site/unclebobconsultingllc/blogs-by-robert-martin/d-dependency-injection-inversion>). *sites.google.com*. Retrieved 2015-07-18.
28. "Decoupling Your Application From Your Dependency Injection Framework" (<http://www.infoq.com/news/2010/01/dependency-injection-inversion>). *InfoQ*. Retrieved 2015-07-18.
29. Martin Fowler (2004-01-23). "Inversion of Control Containers and the Dependency Injection pattern – Forms of Dependency Injection" (<http://www.martinfowler.com/articles/injection.html#FormsOfDependencyInjection>). *Martinfowler.com*. Retrieved 2014-03-22.
30. "AccessibleObject (Java Platform SE 7)" (<http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/AccessibleObject.html>). *docs.oracle.com*. Retrieved 2015-07-18.
31. Riehle, Dirk (2000), *Framework Design: A Role Modeling Approach* (<http://www.riehle.org/computer-science/research/dissertation/diss-a4.pdf>) (PDF), Swiss Federal Institute of Technology
32. "Dependency Injection != using a DI container" (<http://www.loosecouplings.com/2011/01/dependency-injection-using-di-container.html>). *www.loosecouplings.com*. Retrieved 2015-07-18.

<http://www.mosses.com/dependency-injection/> Retrieved 2015-07-18.

33. "Black Sheep » DIY-DI » Print" (<https://web.archive.org/web/20150627215638/http://blacksheep.parry.org/archives/diy-di/print>). *blacksheep.parry.org*. Archived from the original (<https://blacksheep.parry.org/archives/diy-di/print/>) on 2015-06-27. Retrieved 2015-07-18.
34. "Spring Tips: A POJO with annotations is not Plain" (<https://web.archive.org/web/20150715045353/http://springtips.blogspot.com/2007/07/pojo-with-annotations-is-not-plain.html>). Archived from the original (<http://springtips.blogspot.com/2007/07/pojo-with-annotations-is-not-plain.html>) on 2015-07-15. Retrieved 2015-07-18.
35. "Annotations in POJO – a boon or a curse? | Techtracer" (<http://techtracer.com/2007/04/07/annotations-in-pojo-a-boon-or-a-curse/>). 2007-04-07. Retrieved 2015-07-18.
36. *Pro Spring Dynamic Modules for OSGi Service Platforms* (<https://books.google.com/books?id=FCVnsq1ZUI0C&q=spring+pojo+annotation+free&pg=PA64>). APress. 2009-02-17. ISBN 9781430216124. Retrieved 2015-07-06.
37. "Go 1.18 Release Notes - The Go Programming Language" (<https://go.dev/doc/go1.18>). *go.dev*. Retrieved 2024-04-17.
38. "Awesome Go – dependency injection" (<https://github.com/avelino/awesome-go?tab=readme-ov-file#dependency-injection>). *Github*. April 17, 2024. Retrieved April 17, 2024.

External links

- [Composition Root by Mark Seemann](http://blog.ploeh.dk/2011/07/28/CompositionRoot/) (<http://blog.ploeh.dk/2011/07/28/CompositionRoot/>)
- [A beginners guide to Dependency Injection](https://www.theserverside.com/news/1321158/A-beginners-guide-to-Dependency-Injection) (<https://www.theserverside.com/news/1321158/A-beginners-guide-to-Dependency-Injection>)
- [Dependency Injection & Testable Objects: Designing loosely coupled and testable objects](http://www.ddj.com/185300375) (<http://www.ddj.com/185300375>) - Jeremy Weiskotten; *Dr. Dobbs's Journal*, May 2006.
- [Design Patterns: Dependency Injection -- MSDN Magazine, September 2005](http://www.griffincaprio.com/blog/2018/04/design-patterns-dependency-injection.html) (<http://www.griffincaprio.com/blog/2018/04/design-patterns-dependency-injection.html>)
- [Martin Fowler's original article that introduced the term Dependency Injection](http://martinfowler.com/articles/injection.html) (<http://martinfowler.com/articles/injection.html>)
- [P of EAA: Plugin](http://martinfowler.com/eaCatalog/plugin.html) (<http://martinfowler.com/eaCatalog/plugin.html>)
- [The Rich Engineering Heritage Behind Dependency Injection](https://web.archive.org/web/20080313170630/http://www.javaloebb.org/articles/di-heritage/) (<https://web.archive.org/web/20080313170630/http://www.javaloebb.org/articles/di-heritage/>) - Andrew McVeigh - A detailed history of dependency injection.
- [What is Dependency Injection?](http://tutorials.jenkov.com/dependency-injection/index.html) (<http://tutorials.jenkov.com/dependency-injection/index.html>) - An alternative explanation - Jakob Jenkov
- [Writing More Testable Code with Dependency Injection -- Developer.com, October 2006](http://www.developer.com/net/net/article.php/3636501) (<http://www.developer.com/net/net/article.php/3636501>) Archived (<https://web.archive.org/web/20080311121626/http://www.developer.com/net/net/article.php/3636501>) 2008-03-11 at the [Wayback Machine](https://web.archive.org/web/20080311121626/http://www.developer.com/net/net/article.php/3636501)
- [Managed Extensibility Framework Overview -- MSDN](http://msdn.microsoft.com/en-us/library/dd460648.aspx) (<http://msdn.microsoft.com/en-us/library/dd460648.aspx>)
- [Old fashioned description of the Dependency Mechanism by Hunt 1998](https://web.archive.org/web/20120425150101/http://www.midmarsh.co.uk/planetjava/tutorials/language/WatchingtheObservables.PDF) (<https://web.archive.org/web/20120425150101/http://www.midmarsh.co.uk/planetjava/tutorials/language/WatchingtheObservables.PDF>)
- [Refactor Your Way to a Dependency Injection Container](http://blog.thecodewhisperer.com/2011/12/07/refactor-your-way-to-a-dependency-injection-container/) (<http://blog.thecodewhisperer.com/2011/12/07/refactor-your-way-to-a-dependency-injection-container/>)
- [Understanding DI in PHP](http://php-di.org/doc/understanding-di.html) (<http://php-di.org/doc/understanding-di.html>)
- [You Don't Need a Dependency Injection Container](https://medium.com/@wrong.about/you-dont-need-a-dependency-injection-container-10a5d4a5f878) (<https://medium.com/@wrong.about/you-dont-need-a-dependency-injection-container-10a5d4a5f878>)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Dependency_injection&oldid=1317423641"

Factory method pattern

In object-oriented programming, the **factory method pattern** is a design pattern that uses factory methods to deal with the problem of creating objects without having to specify their exact classes. Rather than by calling a constructor, this is accomplished by invoking a factory method to create an object. Factory methods can be specified in an interface and implemented by subclasses or implemented in a base class and optionally overridden by subclasses. It is one of the 23 classic design patterns described in the book *Design Patterns* (often referred to as the "Gang of Four" or simply "GoF") and is subcategorized as a creational pattern.^[1]

Overview

The factory method design pattern solves problems such as:

- How can an object's subclasses redefine its subsequent and distinct implementation? The pattern involves creation of a factory method within the superclass that defers the object's creation to a subclass's factory method.
- How can an object's instantiation be deferred to a subclass? Create an object by calling a factory method instead of directly calling a constructor.

This enables the creation of subclasses that can change the way in which an object is created (for example, by redefining which class to instantiate).

Definition

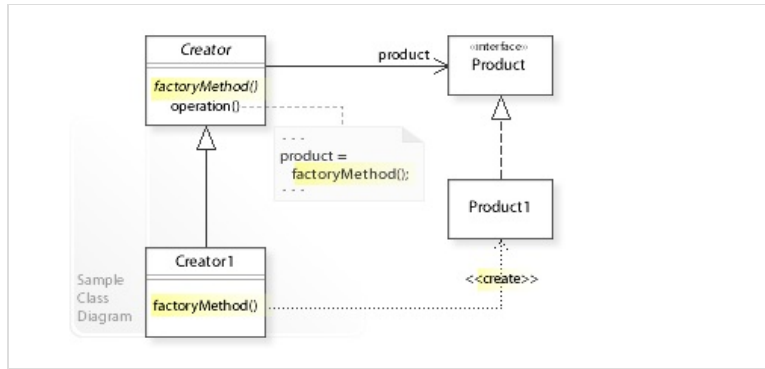
According to *Design Patterns: Elements of Reusable Object-Oriented Software*: "Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses."^[2]

Creating an object often requires complex processes not appropriate to include within a composing object. The object's creation may lead to a significant duplication of code, may require information inaccessible to the composing object, may not provide a sufficient level of abstraction or may otherwise not be included in the composing object's concerns. The factory method design pattern handles these problems by defining a separate method for creating the objects, which subclasses can then override to specify the derived type of product that will be created.

The factory method pattern relies on inheritance, as object creation is delegated to subclasses that implement the factory method to create objects.^[3] The pattern can also rely on the implementation of an interface.

Structure

UML class diagram



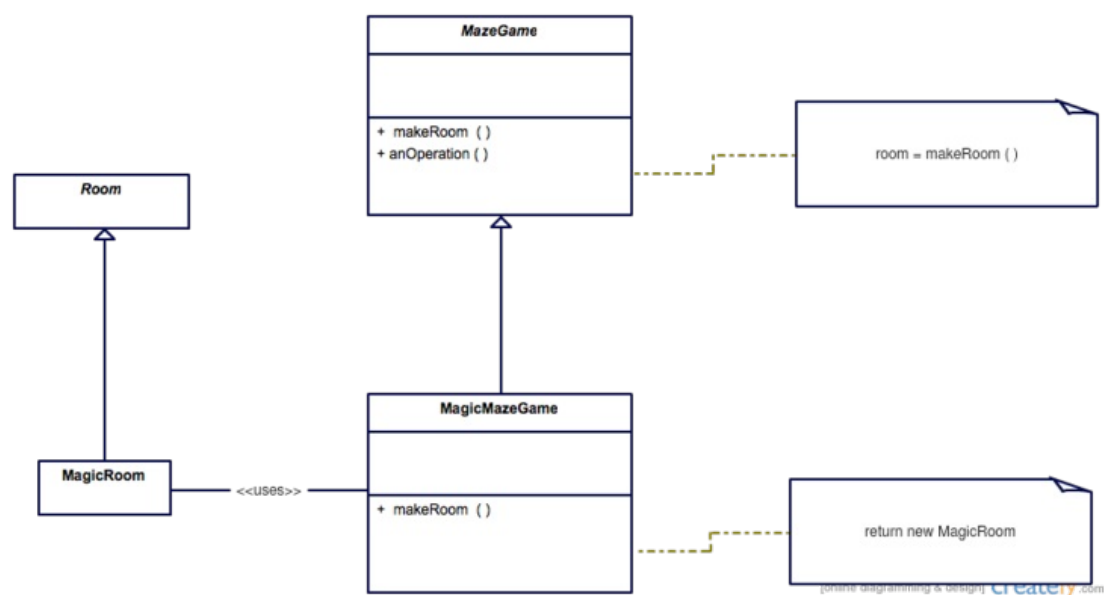
A sample UML class diagram for the Factory Method design pattern. ^[4]

In the above UML class diagram, the **Creator** class that requires a **Product** object does not instantiate the **Product1** class directly. Instead, the **Creator** refers to a separate `factoryMethod()` to create a product object, which makes the **Creator** independent of the exact concrete class that is instantiated. Subclasses of **Creator** can redefine which class to instantiate. In this example, the **Creator1** subclass implements the abstract `factoryMethod()` by instantiating the **Product1** class.

Examples

Structure

A maze game may be played in two modes, one with regular rooms that are only connected with adjacent rooms, and one with magic rooms that allow players to be transported at random.



Room is the base class for a final product (MagicRoom or OrdinaryRoom). MazeGame declares the abstract factory method to produce such a base product. MagicRoom and OrdinaryRoom are subclasses of the base product implementing the final product. MagicMazeGame and OrdinaryMazeGame are subclasses of MazeGame implementing the factory method producing the final products. Factory methods thus decouple callers (MazeGame) from the implementation of the concrete classes. This makes the new operator redundant, allows adherence to the open–closed principle and makes the final product more flexible in the event of change.

Example implementations

C++

This C++23 implementation is based on the pre C++98 implementation in the *Design Patterns* book.^[5]

```
import std;

using std::unique_ptr;

enum class ProductId: char {
    MINE,
    YOURS
};

// defines the interface of objects the factory method creates.
class Product {
public:
    virtual void print() = 0;
    virtual ~Product() = default;
};

// implements the Product interface.
class ConcreteProductMINE: public Product {
public:
    void print() {
        std::println("this={} print MINE", this);
    }
};

// implements the Product interface.
class ConcreteProductYOURS: public Product {
public:
    void print() {
        std::println("this={} print YOURS", this);
    }
};

// declares the factory method, which returns an object of type Product.
class Creator {
public:
    virtual unique_ptr<Product> create(ProductId id) {
        switch (id) {
            case ProductId::MINE:
                return std::make_unique<ConcreteProductMINE>();
            case ProductId::YOURS:
                return std::make_unique<ConcreteProductYOURS>();
            // repeat for remaining products
            default:
                return nullptr;
        }
    }

    virtual ~Creator() = default;
};

int main(int argc, char* argv[]) {
    unique_ptr<Creator> creator = std::make_unique<Creator>();
    unique_ptr<Product> product = creator->create(ProductId::MINE);
}
```

```

        product->print();

        product = creator->create(ProductId::YOURS);
        product->print();
    }

```

The program output is like

```

this=0x6e5e90 print MINE
this=0x6e62c0 print YOURS

```

C#

```

// Empty vocabulary of actual object
public interface IPerson
{
    string GetName();
}

public class Villager : IPerson
{
    public string GetName()
    {
        return "Village Person";
    }
}

public class CityPerson : IPerson
{
    public string GetName()
    {
        return "City Person";
    }
}

public enum PersonType
{
    Rural,
    Urban
}

/// <summary>
/// Implementation of Factory - Used to create objects.
/// </summary>
public class PersonFactory
{
    public IPerson GetPerson(PersonType type)
    {
        switch (type)
        {
            case PersonType.Rural:
                return new Villager();
            case PersonType.Urban:
                return new CityPerson();
            default:
                throw new NotSupportedException();
        }
    }
}

```

The above code depicts the creation of an interface called IPerson and two implementations called Villager and CityPerson. Based on the type passed to the PersonFactory object, the original concrete object is returned as the interface IPerson.

A factory method is just an addition to the PersonFactory class. It creates the object of the class through interfaces but also allows the subclass to decide which class is instantiated.

```

public interface IProduct
{
    string GetName();
    bool SetPrice(double price);
}

public class Phone : IProduct
{
    private double _price;

    public string GetName()
    {
        return "Apple TouchPad";
    }

    public bool SetPrice(double price)
    {
        _price = price;
        return true;
    }
}

/* Almost same as Factory, just an additional exposure to do something with the created method */
public abstract class ProductAbstractFactory
{
    protected abstract IProduct MakeProduct();

    public IProduct GetObject() // Implementation of Factory Method.
    {
        return this.MakeProduct();
    }
}

```

```

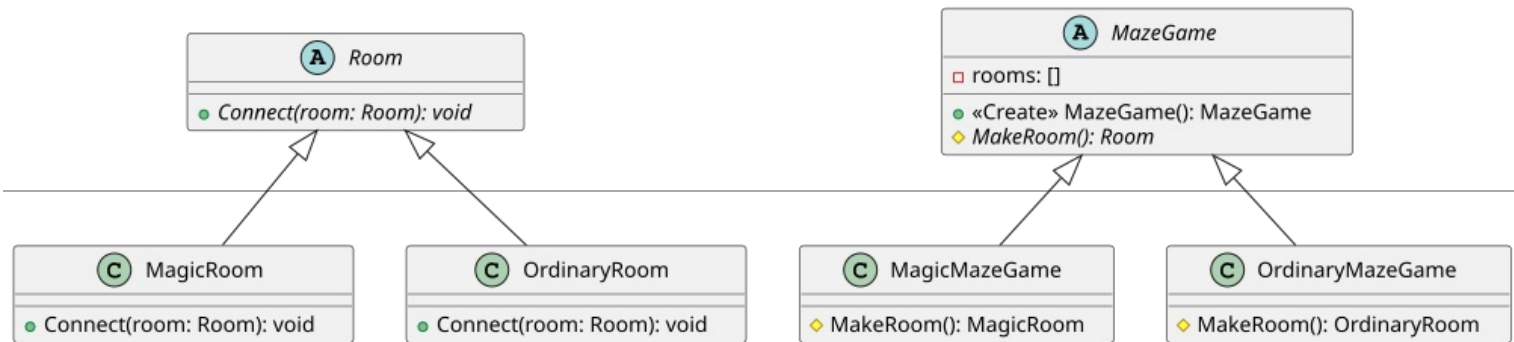
public class PhoneConcreteFactory : ProductAbstractFactory
{
    protected override IProduct MakeProduct()
    {
        IProduct product = new Phone();
        // Do something with the object after receiving it
        product.SetPrice(20.30);
        return product;
    }
}

```

In this example, `MakeProduct` is used in `concreteFactory`. As a result, `MakeProduct()` may be invoked in order to retrieve it from the `IProduct`. Custom logic could run after the object is obtained in the concrete factory method. `GetObject` is made abstract in the factory interface.

Java

This Java example is similar to one in the book *Design Patterns*.



The `MazeGame` uses `Room` but delegates the responsibility of creating `Room` objects to its subclasses that create the concrete classes. The regular game mode could use this template method:

```

public abstract class Room {
    abstract void connect(Room room);
}

public class MagicRoom extends Room {
    public void connect(Room room) {}
}

public class OrdinaryRoom extends Room {
    public void connect(Room room) {}
}

public abstract class MazeGame {
    private final List<Room> rooms = new ArrayList<>();

    public MazeGame() {
        Room room1 = makeRoom();
        Room room2 = makeRoom();
        room1.connect(room2);
        rooms.add(room1);
        rooms.add(room2);
    }

    abstract protected Room makeRoom();
}

```

The `MazeGame` constructor is a template method that adds some common logic. It refers to the `makeRoom()` factory method that encapsulates the creation of rooms such that other rooms can be used in a subclass. To implement the other game mode that has magic rooms, the `makeRoom` method may be overridden:

```

public class MagicMazeGame extends MazeGame {
    @Override
    protected MagicRoom makeRoom() {
        return new MagicRoom();
    }
}

public class OrdinaryMazeGame extends MazeGame {
    @Override
    protected OrdinaryRoom makeRoom() {
        return new OrdinaryRoom();
    }
}

MazeGame ordinaryGame = new OrdinaryMazeGame();
MazeGame magicGame = new MagicMazeGame();

```

PHP

This PHP example shows interface implementations instead of subclassing (however, the same can be achieved through subclassing). The factory method can also be defined as public and called directly by the client code (in contrast to the previous Java example).

```

/* Factory and car interfaces */

interface CarFactory
{
    public function makeCar(): Car;
}

interface Car
{
    public function getType(): string;
}

/* Concrete implementations of the factory and car */

class SedanFactory implements CarFactory
{
    public function makeCar(): Car
    {
        return new Sedan();
    }
}

class Sedan implements Car
{
    public function getType(): string
    {
        return 'Sedan';
    }
}

/* Client */

$factory = new SedanFactory();
$car = $factory->makeCar();
print $car->getType();

```

Python

This Python example employs the same as did the previous Java example.

```

from abc import ABC, abstractmethod

class MazeGame(ABC):
    def __init__(self) -> None:
        self.rooms = []
        self._prepare_rooms()

    def _prepare_rooms(self) -> None:
        room1 = self.make_room()
        room2 = self.make_room()

        room1.connect(room2)
        self.rooms.append(room1)
        self.rooms.append(room2)

    def play(self) -> None:
        print(f"Playing using {self.rooms[0]}")

    @abstractmethod
    def make_room(self):
        raise NotImplementedError("You should implement this!")

class MagicMazeGame(MazeGame):
    def make_room(self) -> "MagicRoom":
        return MagicRoom()

class OrdinaryMazeGame(MazeGame):
    def make_room(self) -> "OrdinaryRoom":
        return OrdinaryRoom()

class Room(ABC):
    def __init__(self) -> None:
        self.connected_rooms = []

    def connect(self, room: "Room") -> None:
        self.connected_rooms.append(room)

class MagicRoom(Room):
    def __str__(self) -> str:
        return "Magic room"

class OrdinaryRoom(Room):
    def __str__(self) -> str:
        return "Ordinary room"

ordinaryGame = OrdinaryMazeGame()
ordinaryGame.play()

magicGame = MagicMazeGame()
magicGame.play()

```

Uses

- In ADO.NET, IDbCommand.CreateCommand (<https://docs.microsoft.com/en-us/dotnet/api/system.data.idbcommand.createpara>

meter?view=netframework-4.8) is an example of the use of factory method to connect parallel class hierarchies.

- In Qt, `QMainWindow::createPopupMenu` (<http://qt-project.org/doc/qt-5.0/qtwidgets/qmainwindow.html#createPopupMenu>) Archived (<https://web.archive.org/web/20150719014739/http://qt-project.org/doc/qt-5.0/qtwidgets/qmainwindow.html#createPopupMenu>) 2015-07-19 at the [Wayback Machine](#) is a factory method declared in a framework that can be overridden in application code.
- In Java, several factories are used in the `javax.xml.parsers` (<http://download.oracle.com/javase/1.5.0/docs/api/javax/xml/parsers/package-summary.html>) package, such as `javax.xml.parsers.DocumentBuilderFactory` or `javax.xml.parsers.SAXParserFactory`.
- In the HTML5 DOM API, the Document interface contains a `createElement()` factory method for creating specific elements of the `HTMLElement` interface.

See also

- *Design Patterns*, the highly influential book
- [Design pattern](#), overview of design patterns in general
- [Abstract factory pattern](#), a pattern often implemented using factory methods
- [Builder pattern](#), another creational pattern
- [Template method pattern](#), which may call factory methods
- [Joshua Bloch's idea of a static factory method](#) for which Bloch claims there is no direct equivalent in *Design Patterns*.

Notes

1. [Gamma et al. 1995](#), p. 107.
2. [Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John \(1995\). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.](#)
3. [Freeman, Eric; Robson, Elisabeth; Sierra, Kathy; Bates, Bert \(2004\). Hendrickson, Mike; Loukides, Mike \(eds.\). *Head First Design Patterns: A Brain-Friendly Guide* \(<http://shop.oreilly.com/product/9780596007126.do>\) \(paperback\). Vol. 1 \(1st ed.\). O'Reilly Media. p. 162. ISBN 978-0-596-00712-6. Retrieved 2012-09-12.](#)
4. ["The Factory Method design pattern - Structure and Collaboration" \(<http://w3sdesign.com/?gr=c03&ugr=struct>\). *w3sDesign.com*. Retrieved 2017-08-12.](#)
5. [Gamma et al. 1995](#), p. 122.

References

- [Martin Fowler; Kent Beck; John Brant; William Opdyke; Don Roberts \(June 1999\). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley. ISBN 0-201-48567-2.](#)
- [Cox, Brad J. \(1986\). *Object-oriented programming: an evolutionary approach*. Addison-Wesley. ISBN 978-0-201-10393-9.](#)
- [Cohen, Tal; Gil, Joseph \(2007\). "Better Construction with Factories" \(<http://tal.forum2.org/static/cv/Factories.pdf>\) \(PDF\). *Journal of Object Technology*. **6** \(6\). Bertrand Meyer: 103. doi:10.5381/jot.2007.6.6.a3 \(<https://doi.org/10.5381%2Fjot.2007.6.6.a3>\). Retrieved 2007-03-12.](#)

External links

- [Factory Design Pattern \(<http://designpattern.co.il/Factory.html>\)](http://designpattern.co.il/Factory.html) Archived (<https://web.archive.org/web/20180110103637/http://designpattern.co.il/Factory.html>) 2018-01-10 at the [Wayback Machine](#) Implementation in Java
- [Factory method in UML and in LePUS3 \(<https://web.archive.org/web/20080503082912/http://www.lepus.org.uk/ref/companion/FactoryMethod.xml>\)](https://web.archive.org/web/20080503082912/http://www.lepus.org.uk/ref/companion/FactoryMethod.xml) (a Design Description Language)
- [Consider static factory methods \(<http://drdobbs.com/java/208403883>\)](http://drdobbs.com/java/208403883) by Joshua Bloch

Retrieved from "https://en.wikipedia.org/w/index.php?title=Factory_method_pattern&oldid=1309372924"

Lazy initialization

In computer programming, **lazy initialization** is the tactic of delaying the creation of an [object](#), the calculation of a [value](#), or some other expensive process until the first time it is needed. It is a kind of [lazy evaluation](#) that refers specifically to the instantiation of objects or other resources.

This is typically accomplished by augmenting an accessor method (or property getter) to check whether a private member, acting as a cache, has already been initialized. If it has, it is returned straight away. If not, a new instance is created, placed into the member variable, and returned to the caller just-in-time for its first use.

If objects have properties that are rarely used, this can improve startup speed. Mean average program performance may be slightly worse in terms of memory (for the condition variables) and execution cycles (to check them), but the impact of object instantiation is spread in time ("amortized") rather than concentrated in the startup phase of a system, and thus median response times can be greatly improved.

In [multithreaded](#) code, access to lazy-initialized objects/state must be [synchronized](#) to guard against [race conditions](#).

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 create instances of a [class](#) ([factory method pattern](#))
- Storing the instances in a [map](#), and returning the *same* instance to each request for an instance with *same* parameters ([multiton pattern](#))
- Using lazy initialization to instantiate the object the first time it is requested (lazy initialization pattern)

Examples

ActionScript 3

The following is an example of a class with lazy initialization implemented in [ActionScript](#):

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

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

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

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

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

Basic use:

```
package
{
    import examples.lazyinstantiation;

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

```

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

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

```

C

In C, lazy evaluation would normally be implemented inside one function, or one source file, using static variables.

In a function:

```

#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

typedef struct Fruit {
    char* name;
    struct Fruit* next;
    int number;
    /* Other members */
} Fruit;

Fruit* getFruit(char* name) {
    static Fruit* fruitList;
    static int seq;
    Fruit* f;

    for (f = fruitList; f; f = f->next) {
        if (!strcmp(name, f->name)) {
            return f;
        }
    }

    if (!(f = malloc(sizeof(Fruit)))) {
        return NULL;
    }

    if (!(f->name = strdup(name))) {
        free(f);
        return NULL;
    }

    f->number = ++seq;
    f->next = fruitList;
    fruitList = f;
    return f;
}

/* Example code */

int main(int argc, char* argv[]) {
    Fruit* f;

    if (argc < 2) {
        fprintf(stderr, "Usage: fruits fruit-name [...] \n");
        return 1;
    }

    for (int i = 1; i < argc; i++) {
        if ((f = getFruit(argv[i]))) {
            printf("Fruit %s: number %d \n", argv[i], f->number);
        }
    }

    return 0;
}

```

Using one source file instead allows the state to be shared between multiple functions, while still hiding it from non-related functions.

Fruit.h:

```

#pragma once

typedef struct Fruit {
    char* name;
    struct Fruit* next;
    int number;
    /* Other members */
} Fruit;

Fruit* getFruit(char* name);
void printFruitList(FILE* file);

```

Fruit.c:

```

#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

```

```
#include "Fruit.h"

static Fruit* fruitList;
static int seq;

struct Fruit* getFruit(char* name) {
    Fruit* f;

    for (f = fruitList; f; f = f->next) {
        if (!strcmp(name, f->name)) {
            return f;
        }
    }

    if (!(f = malloc(sizeof(Fruit)))) {
        return NULL;
    }

    if (!(f->name = strdup(name))) {
        free(f);
        return NULL;
    }

    f->number = ++seq;
    f->next = fruitList;
    fruitList = f;
    return f;
}

void printFruitList(FILE* file) {
    for (Fruit* f = fruitList; f; f = f->next) {
        fprintf(file, "%4d  %s\n", f->number, f->name);
    }
}
```

Main.c:

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

#include "Fruit.h"

int main(int argc, char* argv[]) {
    Fruit* f;

    if (argc < 2) {
        fprintf(stderr, "Usage: fruits fruit-name [...]\n");
        return 1;
    }

    for (int i = 1; i < argc; i++) {
        if ((f = getFruit(argv[i]))) {
            printf("Fruit %s: number %d\n", argv[i], f->number);
        }
    }

    printf("The following fruits have been generated:\n");
    printFruitList(stdout);
    return 0;
}
```

C#

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

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

Here is a dummy example in C#.

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

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

public class Fruit
{
    private string typeName;
    private static IDictionary<string, Fruit> _typesDictionary = new Dictionary<string, Fruit>();

    private Fruit(string typeName)
    {
        this.typeName = typeName;
    }

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

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

```

        _typesDictionary.Add(type, fruit);
    }

    return fruit;
}

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

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

        Console.WriteLine();
    }
}

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

}

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

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

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

        Console.ReadLine();
    }
}

```

A fairly straightforward 'fill-in-the-blanks' example of a Lazy Initialization design pattern, except that this uses an enumeration for the type

```

namespace DesignPatterns.LazyInitialization;

public class LazyFactoryObject
{
    // internal collection of items
    // IDictionary makes sure they are unique
    private IDictionary<LazyObjectSize, LazyObject> _LazyObjectList =
        new Dictionary<LazyObjectSize, LazyObject>();

    // enum for passing name of size required
    // avoids passing strings and is part of LazyObject ahead
    public enum LazyObjectSize
    {
        None,
        Small,
        Big,
        Bigger,
        Huge
    }

    // standard type of object that will be constructed
    public struct LazyObject
    {
        public LazyObjectSize Size;
        public IList<int> Result;
    }

    // takes size and create 'expensive' list
    private IList<int> Result(LazyObjectSize size)
    {
        IList<int> result = null;

        switch (size)
        {
            case LazyObjectSize.Small:
                result = CreateSomeExpensiveList(1, 100);
                break;
            case LazyObjectSize.Big:
                result = CreateSomeExpensiveList(1, 1000);
                break;
            case LazyObjectSize.Bigger:
                result = CreateSomeExpensiveList(1, 10000);
                break;
            case LazyObjectSize.Huge:
                result = CreateSomeExpensiveList(1, 100000);
                break;
            case LazyObjectSize.None:
                result = null;
                break;
            default:
                result = null;
                break;
        }

        return result;
    }
}

```

```

// not an expensive item to create, but you get the point
// delays creation of some expensive object until needed
private IList<int> CreateSomeExpensiveList(int start, int end)
{
    IList<int> result = new List<int>();

    for (int counter = 0; counter < (end - start); counter++)
    {
        result.Add(start + counter);
    }

    return result;
}

public LazyFactoryObject()
{
    // empty constructor
}

public LazyObject GetLazyFactoryObject(LazyObjectSize size)
{
    // yes, i know it is illiterate and inaccurate
    LazyObject noGoodSomeOne;

    // retrieves LazyObjectSize from list via out, else creates one and adds it to list
    if (!_LazyObjectList.TryGetValue(size, out noGoodSomeOne))
    {
        noGoodSomeOne = new LazyObject();
        noGoodSomeOne.Size = size;
        noGoodSomeOne.Result = this.Result(size);
    }
    _LazyObjectList.Add(size, noGoodSomeOne);

    return noGoodSomeOne;
}
}

```

C++

This example is in C++.

```

import std;

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

    // Note: constructor private forcing one to use static |GetFruit|.
    Fruit(const std::string& type):
        type_{type} {}
public:
    // Lazy Factory method, gets the |Fruit| instance associated with a certain
    // |type|. Creates new ones as needed.
    static Fruit* getFruit(const std::string& type) {
        auto [it, inserted] = types.emplace(type, nullptr);
        if (inserted) {
            it->second = new Fruit(type);
        }
        return it->second;
    }

    // For example purposes to see pattern in action.
    static void printCurrentTypes() {
        std::println("Number of instances made = {}", types.size());
        for (const auto& [type, fruit] : types) {
            std::println({}, type);
        }
        std::println();
    }
};

// static
std::map<std::string, Fruit*> Fruit::types;

int main(int argc, char* argv[]) {
    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();
}

// OUTPUT:
//
// Number of instances made = 1
// Banana
//
// Number of instances made = 2
// Apple
// Banana
//
// Number of instances made = 2
// Apple
// Banana
//

```

Crystal

```
class Fruit
  private getter type : String
  @@types = {} of String => Fruit

  def initialize(@type)
  end

  def self.get_fruit_by_type(type : String)
    @@types[type] ||= Fruit.new(type)
  end

  def self.show_all
    puts "Number of instances made: #{@types.size}"
    @@types.each do |type, fruit|
      puts "#{type}"
    end
    puts
  end

  def self.size
    @@types.size
  end
end

Fruit.get_fruit_by_type("Banana")
Fruit.show_all

Fruit.get_fruit_by_type("Apple")
Fruit.show_all

Fruit.get_fruit_by_type("Banana")
Fruit.show_all
```

Output:

```
Number of instances made: 1
Banana

Number of instances made: 2
Banana
Apple

Number of instances made: 2
Banana
Apple
```

Haxe

This example is in [Haxe](#).^[1]

```
class Fruit {
  private static var _instances = new Map<String, Fruit>();

  public var name(default, null):String;

  public function new(name:String) {
    this.name = name;
  }

  public static function getFruitByName(name:String):Fruit {
    if (!_instances.exists(name)) {
      _instances.set(name, new Fruit(name));
    }
    return _instances.get(name);
  }

  public static function printAllTypes() {
    trace([for(key in _instances.keys()) key]);
  }
}
```

Usage

```
class Test {
  public static function main () {
    var banana = Fruit.getFruitByName("Banana");
    var apple = Fruit.getFruitByName("Apple");
    var banana2 = Fruit.getFruitByName("Banana");

    trace(banana == banana2); // true. same banana

    Fruit.printAllTypes(); // ["Banana", "Apple"]
  }
}
```

Java

This example is in [Java](#).

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



```

import java.util.Map.Entry;

public class Program {

    /**
     * @param args
     */
    public static void main(String[] args) {
        Fruit.getFruitByTypeName(FruitType.banana);
        Fruit.showAll();
        Fruit.getFruitByTypeName(FruitType.apple);
        Fruit.showAll();
        Fruit.getFruitByTypeName(FruitType.banana);
        Fruit.showAll();
    }
}

enum FruitType {
    none,
    apple,
    banana,
}

class Fruit {

    private static Map<FruitType, Fruit> types = new HashMap<>();

    /**
     * Using a private constructor to force the use of the factory method.
     * @param type
     */
    private Fruit(FruitType type) {
    }

    /**
     * Lazy Factory method, gets the Fruit instance associated with a certain
     * type. Instantiates new ones as needed.
     * @param type Any allowed fruit type, e.g. APPLE
     * @return The Fruit instance associated with that type.
     */
    public static Fruit getFruitByTypeName(FruitType type) {
        Fruit fruit;
        // This has concurrency issues. Here the read to types is not synchronized,
        // so types.put and types.containsKey might be called at the same time.
        // Don't be surprised if the data is corrupted.
        if (!types.containsKey(type)) {
            // Lazy initialisation
            fruit = new Fruit(type);
            types.put(type, fruit);
        } else {
            // OK, it's available currently
            fruit = types.get(type);
        }

        return fruit;
    }

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

        return types.get(type);
    }

    /**
     * Displays all entered fruits.
     */
    public static void showAll() {
        if (types.size() > 0) {

            System.out.println("Number of instances made = " + types.size());

            for (Entry<FruitType, Fruit> entry : types.entrySet()) {
                String fruit = entry.getKey().toString();
                fruit = Character.toUpperCase(fruit.charAt(0)) + fruit.substring(1);
                System.out.println(fruit);
            }

            System.out.println();
        }
    }
}

```

Output

```

Number of instances made = 1
Banana

```

```

Number of instances made = 2
Banana
Apple

```

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

JavaScript

This example is in JavaScript.

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

    // count own properties in object
    function count(obj) {
        return Object.keys(obj).length;
    }

    var _static = {
        getFruit: function(type) {
            if (typeof types[type] == 'undefined') {
                types[type] = new Fruit;
            }
            return types[type];
        },
        printCurrentTypes: function () {
            console.log('Number of instances made: ' + count(types));
            for (var type in types) {
                console.log(type);
            }
        }
    };

    return _static;
})();

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

Output

```
Number of instances made: 1
Apple

Number of instances made: 2
Apple
Banana

Number of instances made: 2
Apple
Banana
```

PHP

Here is an example of lazy initialization in PHP 7.4:

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

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

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

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

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

    public static function printCurrentTypes(): void
    {
        echo 'Number of instances made: ' . count(self::$types) . "\n";
        foreach (array_keys(self::$types) as $key) {
            echo "$key\n";
        }
        echo "\n";
    }
}

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

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

```

Fruit::printCurrentTypes();

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

/*
OUTPUT:

Number of instances made: 1
Apple

Number of instances made: 2
Apple
Banana

Number of instances made: 2
Apple
Banana
*/

```

Python

This example is in Python.

```

class Fruit:
    def __init__(self, item: str):
        self.item = item

class FruitCollection:
    def __init__(self):
        self.items = {}

    def get_fruit(self, item: str) -> Fruit:
        if item not in self.items:
            self.items[item] = Fruit(item)

        return self.items[item]

if __name__ == "__main__":
    fruits = FruitCollection()
    print(fruits.get_fruit("Apple"))
    print(fruits.get_fruit("Lime"))

```

Ruby

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

```

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

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

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

```

Rust

Rust have `std::cell::LazyCell`.^[2]

```

use std::cell::LazyCell;

let lazy: LazyCell = LazyCell::new(|| 42);

```

Scala

Scala has built-in support for lazy variable initiation.^[3]

```

scala> val x = { println("Hello"); 99 }
Hello
x: Int = 99
scala> lazy val y = { println("Hello!!"); 31 }
y: Int = <lazy>
scala> y
Hello!!
res2: Int = 31
scala> y
res3: Int = 31

```

Smalltalk

This example is in [Smalltalk](#), of a typical [accessor method](#) to return the value of a variable using lazy initialization.

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

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

```
initialize
height := 2.0

height
^height
```

Note that lazy initialization can also be used in [non-object-oriented languages](#).

Theoretical computer science

In the field of [theoretical computer science](#), **lazy initialization**^[4] (also called a **lazy array**) is a technique to design data structures that can work with memory that does not need to be initialized. Specifically, assume that we have access to a table *T* of *n* uninitialized memory cells (numbered from 1 to *n*), and want to assign *m* cells of this array, e.g., we want to assign *T*[*k*_{*i*}] := *v*_{*i*} for pairs (*k*₁, *v*₁), ..., (*k*_{*m*}, *v*_{*m*}) with all *k*_{*i*} being different. The lazy initialization technique allows us to do this in just *O*(*m*) operations, rather than spending *O*(*m*+*n*) operations to first initialize all array cells. The technique is simply to allocate a table *V* storing the pairs (*k*_{*i*}, *v*_{*i*}) in some arbitrary order, and to write for each *i* in the cell *T*[*k*_{*i*}] the position in *V* where key *k*_{*i*} is stored, leaving the other cells of *T* uninitialized. This can be used to handle queries in the following fashion: when we look up cell *T*[*k*] for some *k*, we can check if *T*[*k*] is in the range {1, ..., *m*}: if it is not, then *T*[*k*] is uninitialized. Otherwise, we check *V*[*T*[*k*]], and verify that the first component of this pair is equal to *k*. If it is not, then *T*[*k*] is uninitialized (and just happened by accident to fall in the range {1, ..., *m*}). Otherwise, we know that *T*[*k*] is indeed one of the initialized cells, and the corresponding value is the second component of the pair.

See also

- Double-checked locking
- Lazy loading
- Proxy pattern
- Singleton pattern

References

- "Lazy initialization - Design patterns - Haxe programming language cookbook" (<https://code.haxe.org/category/design-patterns/lazy-initialization.html>). 2018-01-11. Retrieved 2018-11-09.
- "LazyCell in std::cell - Rust" (<https://doc.rust-lang.org/std/cell/struct.LazyCell.html>). *doc.rust-lang.org*. Retrieved 18 January 2025.
- Pollak, David (2009-05-25). *Beginning Scala* (<https://books.google.com/books?id=Qt-bRFetWw0C&q=scala+lazy+variables&pg=PA30>). Apress. ISBN 9781430219897.
- Moret, B. M. E.; Shapiro, H. D. (1991). *Algorithms from P to NP, Volume 1: Design & Efficiency*. Benjamin/Cummings Publishing Company. pp. 191–192. ISBN 0-8053-8008-6.

External links

- Article "Java Tip 67: Lazy instantiation (<https://www.infoworld.com/article/2077568/java-tip-67--lazy-instantiation.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 (<https://web.archive.org/web/20191202055631/http://devx.com/tips/Tip/18007>)
- Description from the Portland Pattern Repository (<https://wiki.c2.com/?LazyInitialization>)
- Lazy Initialization of Application Server Services (https://web.archive.org/web/20060911223210/http://weblogs.java.net/blog/binod/archive/2005/09/lazy_initializa.html)
- Lazy Inheritance in JavaScript (<https://sourceforge.net/projects/jsiner/>)
- Lazy Inheritance in C# (<https://kaliko.com/blog/lazy-loading-in-c-net/>)

Multiton pattern

In software engineering, the **multiton pattern** is a design pattern which generalizes the singleton pattern. Whereas the singleton allows only one instance of a class to be created, the multiton pattern allows for the controlled creation of multiple instances, which it manages through the use of a map.

Rather than having a single instance *per application* (e.g. the `java.lang.Runtime` (<https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/lang/Runtime.html>)

object in the Java programming language) the multiton pattern instead ensures a single instance *per key*.

The multiton pattern does not explicitly appear as a pattern in the highly regarded object-oriented programming textbook *Design Patterns*.^[1] However, the book describes using a **registry of singletons** to allow subclassing of singletons,^[2] which is essentially the multiton pattern.

Description

While it may appear that the multiton is a hash table with synchronized access there are two important distinctions. First, the multiton does not allow clients to add mappings. Secondly, the multiton never returns a null or empty reference; instead, it creates and stores a multiton instance on the first request with the associated key. Subsequent requests with the same key return the original instance. A hash table is merely an implementation detail and not the only possible approach. The pattern simplifies retrieval of shared objects in an application.

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

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

Drawbacks

This pattern, like the Singleton pattern, makes unit testing far more difficult,^[3] as it introduces global state into an application.

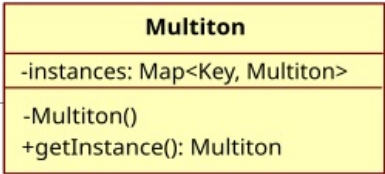
With garbage collected languages it may become a source of memory leaks as it introduces global strong references to the objects.

Implementations

In Java, the multiton pattern can be implemented using an enumerated type, with the values of the type corresponding to the instances. In the case of an enumerated type with a single value, this gives the singleton pattern.

In C#, we can also use enums, as the following example shows:

```
1 using System;
2 using System.Collections.Generic;
3
4 public enum MultitonType
5 {
6     Zero,
7     One,
8     Two
9 }
10
11 public class Multiton
12 {
13     private static readonly Dictionary<MultitonType, Multiton> instances =
14         new Dictionary<MultitonType, Multiton>();
15
16     private MultitonType type;
17
18     private Multiton(MultitonType type)
19     {
20         this.type = type;
21     }
22
23     public static Multiton GetInstance(MultitonType type)
24     {
25         // Lazy init (not thread safe as written)
```



UML diagram of the multiton

```
26 // Recommend using Double Check Locking if needing thread safety
27 if (!instances.TryGetValue(type, out var instance))
28 {
29     instance = new Multiton(type);
30
31     instances.Add(type, instance);
32 }
33
34 return instance;
35 }
36
37 public override string ToString()
38 {
39     return $"My type is {this.type}";
40 }
41
42 // Sample usage
43 public static void Main(string[] args)
44 {
45     Multiton m0 = Multiton.GetInstance(MultitonType.Zero);
46     Multiton m1 = Multiton.GetInstance(MultitonType.One);
47     Multiton m2 = Multiton.GetInstance(MultitonType.Two);
48
49     Console.WriteLine(m0);
50     Console.WriteLine(m1);
51     Console.WriteLine(m2);
52 }
53 }
```

References

1. O'Docherty, Mike (2005). *Object-oriented analysis and design: understanding system development with UML 2.0*. Chichester: Wiley. p. 341. ISBN 0470092408.
2. *Design patterns: elements of reusable object-oriented software*. Boston, Mass. Munich: Addison-Wesley. 2011. p. 130. ISBN 0-201-63361-2.
3. "Clean Code Talks - Global State and Singletons" (<https://googletesting.blogspot.com/2008/11/clean-code-talks-global-state-and.html>).

External links

- Multiton implementation in Ruby language (<https://rubygems.org/gems/multiton/versions/0.0.1>)
- Multiton usage in PureMVC Framework for ActionScript 3 (<https://github.com/PureMVC/puremvc-as3-multicore-framework/blob/master/src/org/puremvc/as3/multicore/patterns/facade/Facade.as>)
- Article with a C# Multiton implementation, example of use, and discussion of memory issues (<http://gen5.info/q/2008/07/25/the-multiton-design-pattern/>)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Multiton_pattern&oldid=1305903302"

Object pool pattern

The **object pool pattern** is a software creational design pattern that uses a set of initialized objects kept ready to use – a "pool" – rather than allocating and destroying them on demand. A client of the pool will request an object from the pool and perform operations on the returned object. When the client has finished, it returns the object to the pool rather than destroying it; this can be done manually or automatically.

Object pools are primarily used for performance: in some circumstances, object pools significantly improve performance. Object pools complicate object lifetime, as objects obtained from and returned to a pool are not actually created or destroyed at this time, and thus require care in implementation.

Description

When it is necessary to work with numerous objects that are particularly expensive to instantiate and each object is only needed for a short period of time, the performance of an entire application may be adversely affected. An object pool design pattern may be deemed desirable in cases such as these.

The object pool design pattern creates a set of objects that may be reused. When a new object is needed, it is requested from the pool. If a previously prepared object is available, it is returned immediately, avoiding the instantiation cost. If no objects are present in the pool, a new item is created and returned. When the object has been used and is no longer needed, it is returned to the pool, allowing it to be used again in the future without repeating the computationally expensive instantiation process. It is important to note that once an object has been used and returned, existing references will become invalid.

In some object pools the resources are limited, so a maximum number of objects is specified. If this number is reached and a new item is requested, an exception may be thrown, or the thread will be blocked until an object is released back into the pool.

The object pool design pattern is used in several places in the standard classes of the .NET Framework. One example is the .NET Framework Data Provider for SQL Server. As SQL Server database connections can be slow to create, a pool of connections is maintained. Closing a connection does not actually relinquish the link to SQL Server. Instead, the connection is held in a pool, from which it can be retrieved when requesting a new connection. This substantially increases the speed of making connections.

Benefits

Object pooling can offer a significant performance boost in situations where the cost of initializing a class instance is high and the rate of instantiation and destruction of a class is high – in this case objects can frequently be reused, and each reuse saves a significant amount of time. Object pooling requires resources – memory and possibly other resources, such as network sockets, and thus it is preferable that the number of instances in use at any one time is low, but this is not required.

The pooled object is obtained in predictable time when creation of the new objects (especially over network) may take variable time. These benefits are mostly true for objects that are expensive with respect to time, such as database connections, socket connections, threads and large graphic objects like fonts or bitmaps.

In other situations, simple object pooling (that hold no external resources, but only occupy memory) may not be efficient and could decrease performance.^[1] In case of simple memory pooling, the slab allocation memory management technique is more suited, as the only goal is to minimize the cost of memory allocation and deallocation by reducing fragmentation.

Implementation

Object pools can be implemented in an automated fashion in languages like C++ via smart pointers. In the constructor of the smart pointer, an object can be requested from the pool, and in the destructor of the smart pointer, the object can be released back to the pool. In garbage-collected languages, where there are no destructors (which are guaranteed to be called as part of a stack unwind), object pools *must* be implemented manually, by explicitly requesting an object from the factory and returning the object by calling a dispose method (as in the dispose pattern). Using a finalizer to do this is not a good idea, as there are usually no guarantees on when (or if) the finalizer will be run. Instead, "try ... finally" should be used to ensure that getting and releasing the object is exception-neutral.

Manual object pools are simple to implement, but harder to use, as they require manual memory management of pool objects.

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

Care must be taken to ensure the state of the objects returned to the pool is reset back to a sensible state for the next use of the object, otherwise the object may be in a state unexpected by the client, which may cause it 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.

Stale state may not always be an issue; it becomes dangerous when it causes the object to behave unexpectedly. For example, an object representing authentication details may fail if the "successfully authenticated" flag is not reset before it is reused, since it indicates that a user is authenticated (possibly as someone else) when they are not. However, failing to reset a value used only for debugging, such as the identity of the last authentication server used, may pose no issues.

Inadequate resetting of objects can cause information leaks. Objects containing confidential data (e.g. a user's credit card numbers) must be cleared before being passed to new clients, otherwise, the data may be disclosed to an unauthorized party.

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

Criticism

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

Examples

C++

In C++26, the C++ Standard Library introduces a new header <hive> with the data structure `std::hive`, which essentially implements an object pool. It is a collection that reuses erased elements' memory. Along with it is a class `std::hive_limits` for layout information on block capacity limits.^[2]

```
import std;

using std::hive;
using std::plus;
using std::views::iota;

int main(int argc, char* argv[]) {
    hive<int> intHive;

    // Insert 100 ints:
    intHive.insert_range(iota(0, 100));

    // Erase half of them:
    for (auto it = intHive.begin(); it != intHive.end(); ) {
        it = intHive.erase(it);
    }

    int total = std::ranges::fold_left(intHive, 0, plus<int>());
    std::println("Total of all elements: {}", total);

    return 0;
}
```

C#

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

The following shows the basic code of the object pool design pattern implemented using C#. Pool is shown as a static class, as it's unusual for multiple pools to be required. However, it's equally acceptable to use instance classes for object pools.

```

using System;
using System.Collections.Generic;

namespace Wikipedia.Examples;

// The PooledObject class is the type that is expensive or slow to instantiate,
// or that has limited availability, so is to be held in the object pool.
public class PooledObject
{
    private DateTime _createdAt = DateTime.Now;

    public DateTime CreatedAt => _createdAt;

    public string TempData { get; set; }
}

// The Pool class controls access to the pooled objects. It maintains a list of available objects and a
// collection of objects that have been obtained from the pool and are in use. The pool ensures that released objects
// are returned to a suitable state, ready for reuse.
public static class Pool
{
    private static List<PooledObject> _available = new();
    private static List<PooledObject> _inUse = new();

    public static PooledObject GetObject()
    {
        lock (_available)
        {
            if (_available.Count != 0)
            {
                PooledObject po = _available[0];
                _inUse.Add(po);
                _available.RemoveAt(0);
                return po;
            }
            else
            {
                PooledObject po = new();
                _inUse.Add(po);
                return po;
            }
        }
    }

    public static void ReleaseObject(PooledObject po)
    {
        Cleanup(po);

        lock (_available)
        {
            _available.Add(po);
            _inUse.Remove(po);
        }
    }

    private static void Cleanup(PooledObject po)
    {
        po.TempData = null;
    }
}

```

In the code above, the PooledObject has properties for the time it was created, and another, that can be modified by the client, that is reset when the PooledObject is released back to the pool. Shown is the clean-up process, on release of an object, ensuring it is in a valid state before it can be requested from the pool again.

Go

The following Go code initializes a resource pool of a specified size (concurrent initialization) to avoid resource race issues through channels, and in the case of an empty pool, sets timeout processing to prevent clients from waiting too long.

```

// package pool
package pool

import (
    "errors"
    "log"
    "math/rand"
    "sync"
    "time"
)

const getResMaxTime = 3 * time.Second

var (
    ErrPoolNotExist = errors.New("pool not exist")
    ErrGetResTimeout = errors.New("get resource time out")
)

//Resource
type Resource struct {
    resId int
}

// NewResource Simulate slow resource initialization creation
// (e.g., TCP connection, SSL symmetric key acquisition, auth authentication are time-consuming)
func NewResource(id int) *Resource {
    time.Sleep(500 * time.Millisecond)
    return &Resource{resId: id}
}

// Do Simulation resources are time consuming and random consumption is 0~400ms

```

```

func (r *Resource) Do(workId int) {
    time.Sleep(time.Duration(rand.Intn(5)) * 100 * time.Millisecond)
    log.Printf("using resource #%d finished work %d finish\n", r.resId, workId)
}

// Pool based on Go channel implementation, to avoid resource race state problem
type Pool chan *Resource

// New a resource pool of the specified size
// Resources are created concurrently to save resource initialization time
func New(size int) Pool {
    p := make(Pool, size)
    wg := new(sync.WaitGroup)
    wg.Add(size)
    for i := 0; i < size; i++ {
        go func(resId int) {
            p <- NewResource(resId)
            wg.Done()
        }(i)
    }
    wg.Wait()
    return p
}

// GetResource based on channel, resource race state is avoided and resource acquisition timeout is set for empty pool
func (p Pool) GetResource() (r *Resource, err error) {
    select {
    case r := <-p:
        return r, nil
    case <-time.After(getResMaxTime):
        return nil, ErrGetResTimeout
    }
}

// GiveBackResource returns resources to the resource pool
func (p Pool) GiveBackResource(r *Resource) error {
    if p == nil {
        return ErrPoolNotExist
    }
    p <- r
    return nil
}

// package main
package main

import (
    "github.com/tkstorm/go-design/creational/object-pool/pool"
    "log"
    "sync"
)

func main() {
    // Initialize a pool of five resources,
    // which can be adjusted to 1 or 10 to see the difference
    size := 5
    p := pool.New(size)

    // Invokes a resource to do the id job
    doWork := func(workId int, wg *sync.WaitGroup) {
        defer wg.Done()
        // Get the resource from the resource pool
        res, err := p.GetResource()
        if err != nil {
            log.Println(err)
            return
        }
        // Resources to return
        defer p.GiveBackResource(res)
        // Use resources to handle work
        res.Do(workId)
    }

    // Simulate 100 concurrent processes to get resources from the asset pool
    num := 100
    wg := new(sync.WaitGroup)
    wg.Add(num)
    for i := 0; i < num; i++ {
        go doWork(i, wg)
    }
    wg.Wait()
}

```

Java

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.

In `PooledObject.java`:

```

package org.wikipedia.examples;

public class PooledObject {
    private String temp1;
    private String temp2;
    private String temp3;

    public String getTemp1() {
        return temp1;
    }
}

```

```

    public void setTemp1(String temp1) {
        this.temp1 = temp1;
    }

    public String getTemp2() {
        return temp2;
    }

    public void setTemp2(String temp2) {
        this.temp2 = temp2;
    }

    public String getTemp3() {
        return temp3;
    }

    public void setTemp3(String temp3) {
        this.temp3 = temp3;
    }
}

```

In PooledObjectPool.java:

```

package org.wikipedia.examples;

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

public class PooledObjectPool {
    private static long expTime = 6000; // 6 seconds
    public static Map<PooledObject, Long> available = new HashMap<PooledObject, Long>();
    public static Map<PooledObject, Long> inUse = new HashMap<PooledObject, Long>();

    public synchronized static PooledObject getObject() {
        long now = System.currentTimeMillis();
        if (!available.isEmpty()) {
            for (Map.Entry<PooledObject, Long> entry : available.entrySet()) {
                if (now - entry.getValue() > expTime) {
                    // object has expired
                    popElement(available);
                } else {
                    PooledObject po = popElement(available, entry.getKey());
                    push(inUse, po, now);
                    return po;
                }
            }
        }

        // either no PooledObject is available or each has expired, so return a new one
        return createPooledObject(now);
    }

    private synchronized static PooledObject createPooledObject(long now) {
        PooledObject po = new PooledObject();
        push(inUse, po, now);
        return po;
    }

    private synchronized static void push(HashMap<PooledObject, Long> map, PooledObject po, long now) {
        map.put(po, now);
    }

    public static void releaseObject(PooledObject po) {
        cleanUp(po);
        available.put(po, System.currentTimeMillis());
        inUse.remove(po);
    }

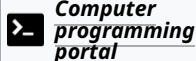
    private static PooledObject popElement(HashMap<PooledObject, Long> map) {
        Map.Entry<PooledObject, Long> entry = map.entrySet().iterator().next();
        PooledObject key = entry.getKey();
        // Long value = entry.getValue();
        map.remove(entry.getKey());
        return key;
    }

    private static PooledObject popElement(HashMap<PooledObject, Long> map, PooledObject key) {
        map.remove(key);
        return key;
    }

    public static void cleanUp(PooledObject po) {
        po.setTemp1(null);
        po.setTemp2(null);
        po.setTemp3(null);
    }
}

```

See also



- [Connection pool](#)
- [Free list](#)
- [Slab allocation](#)

Notes

1. Goetz, Brian (2005-09-27). "Java theory and practice: Urban performance legends, revisited" (<https://web.archive.org/web/20120214195433/http://www.ibm.com/developerworks/java/library/j-jtp09275/index.html>). *IBM*. IBM developerWorks. Archived from the original (<http://www.ibm.com/developerworks/java/library/j-jtp09275/index.html>) on 2012-02-14. Retrieved 2021-03-15.
2. "Standard library header <hive> (C++26) - cppreference.com" (<https://en.cppreference.com/w/cpp/header/hive.html>). *cppreference.com*. Retrieved 1 October 2025.

References

- Kircher, Michael; Prashant Jain (2002-07-04). "Pooling Pattern" (<http://www.kircher-schwanninger.de/michael/publications/Pooling.pdf>) (PDF). *EuroPLoP 2002*. Germany. Retrieved 2007-06-09.
- Goldshtein, Sasha; Zurbalev, Dima; Flatow, Ido (2012). *Pro .NET Performance: Optimize Your C# Applications* (<http://www.apress.com/9781430244585>). Apress. ISBN 978-1-4302-4458-5.

External links

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

Retrieved from "https://en.wikipedia.org/w/index.php?title=Object_pool_pattern&oldid=1314510949"

Prototype pattern

The **prototype pattern** is a creational design pattern in software development. It is used when the types 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 factory method pattern does, and to 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, the client declares an abstract base class that specifies a pure virtual *clone()* method. Any class that needs a "polymorphic constructor" capability derives itself from the abstract base class, and implements the *clone()* operation.

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

The mitotic division of a cell — resulting in two identical cells — is an example of a prototype that plays an active role in copying itself and thus, demonstrates the Prototype pattern. When a cell splits, two cells of identical genotype result. In other words, the cell clones itself.^[1]

Overview

The prototype design pattern is one of the 23 Gang of Four design patterns that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.^{[2]:117}

The prototype design pattern solves problems like:^[3]

- How can objects be created so that the specific type of object can be determined at runtime?
- How can dynamically loaded classes be instantiated?

Creating objects directly within the class that requires (uses) the objects is inflexible because it commits the class to particular objects at compile-time and makes it impossible to specify which objects to create at run-time.

The prototype design pattern describes how to solve such problems:

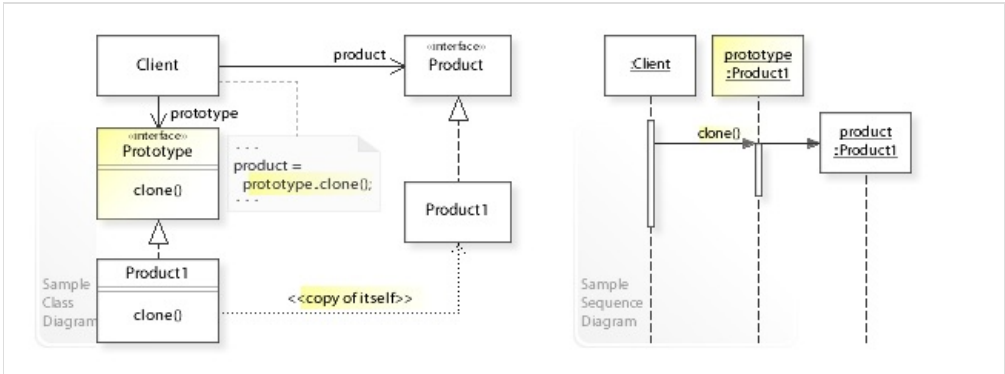
- Define a Prototype object that returns a copy of itself.
- Create new objects by copying a Prototype object.

This enables configuration of a class with different Prototype objects, which are copied to create new objects, and even more, Prototype objects can be added and removed at run-time.

See also the UML class and sequence diagram below.

Structure

UML class and sequence diagram

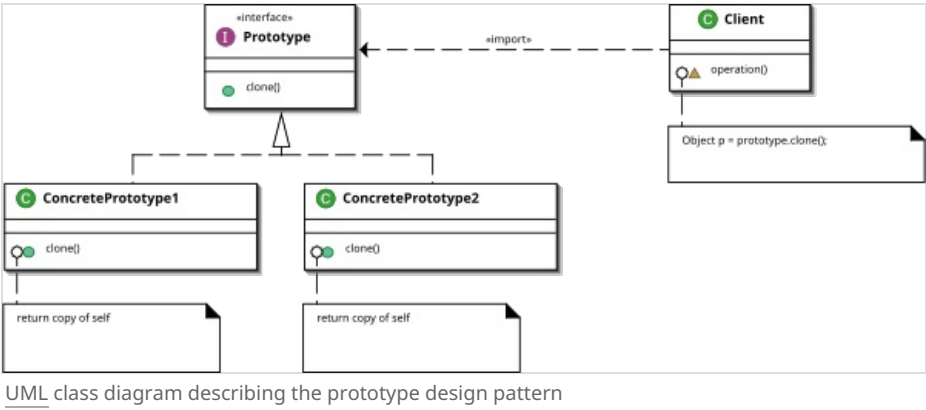


A sample UML class and sequence diagram for the Prototype design pattern.

In the above UML class diagram, the **Client** class refers to the **Product** interface for cloning a **Product**. The **Product1** class implements the **Product** interface by creating a copy of itself.

The UML sequence diagram shows the run-time interactions: The **Client** object calls **clone()** on a **prototype:Product1** object, which creates and returns a copy of itself (a **product:Product1** object).

UML class diagram



UML class diagram describing the prototype design pattern

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.^{[2]:126} Abstract factory, builder, and prototype can use singleton in their implementations.^{[2]:81,134} Abstract factory classes are often implemented with factory methods (creation through inheritance), but they can be implemented using prototype (creation through delegation).^{[2]:95}

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.^{[2]:136}

Prototype does not require subclassing, but it does require an "initialize" operation. Factory method requires subclassing, but does not require initialization.^{[2]:116}

Designs that make heavy use of the composite and decorator patterns often can benefit from Prototype as well.^{[2]:126}

A general guideline in programming suggests using the `clone()` method when creating a duplicate object during runtime to ensure it accurately reflects the original object. This process, known as object cloning, produces a new object with identical attributes to the one being cloned. Alternatively, *instantiating* a class using the `new` keyword generates an object with default attribute values.

For instance, in the context of designing a system for managing bank account transactions, it may be necessary to duplicate the object containing account information to conduct transactions while preserving the original data. In such scenarios, employing the `clone()` method is preferable over using `new` to instantiate a new object.

Example

C++23 Example

This C++23 implementation is based on the pre-C++98 implementation in the book. Discussion of the design pattern along with a complete illustrative example implementation using polymorphic class design are provided in the C++ Annotations (<https://fbb-github.github.io/cppannotations/cppannotations/html/cplusplus14.html#1318>).

```
import std;

using std::array;
using std::shared_ptr;
using std::unique_ptr;
using std::vector;

enum class Direction: char {
    North,
    South,
    East,
    West
};

class MapSite {
public:
    virtual void enter() = 0;
    virtual unique_ptr<MapSite> clone() const = 0;
    virtual ~MapSite() = default;
};

class Room: public MapSite {
private:
    int roomNumber;
    shared_ptr<array<shared_ptr<MapSite>, 4>> sides;
public:
    Room():
```

```

        roomNumber{0}, sides{std::make_shared<array<shared_ptr<MapSite>, 4>>()}{}}

explicit Room(int n):
    roomNumber{n}, sides{std::make_shared<array<shared_ptr<MapSite>, 4>>()}{}}

Room& setSide(Direction d, shared_ptr<MapSite> ms) {
    (*sides)[static_cast<size_t>(d)] = std::move(ms);
    std::println("Room::setSide {} ms", d);
    return *this;
}

virtual void enter() override {}

virtual unique_ptr<MapSite> clone() const override {
    return std::make_unique<Room>(*this);
}

Room(const Room&) = delete;
Room& operator=(const Room&) = delete;
};

class Wall: public MapSite {
public:
    Wall():
        MapSite() {}

    virtual void enter() override {}

    [[nodiscard]]
    virtual unique_ptr<MapSite> clone() const override {
        return std::make_unique<Wall>(*this);
    }
};

class Door: public MapSite {
private:
    shared_ptr<Room> room1;
    shared_ptr<Room> room2;

public:
    explicit Door(shared_ptr<Room> r1 = nullptr, shared_ptr<Room> r2 = nullptr):
        MapSite(), room1{std::move(r1)}, room2{std::move(r2)} {}

    virtual void enter() override {}

    [[nodiscard]]
    virtual unique_ptr<MapSite> clone() const override {
        return std::make_unique<Door>(*this);
    }

    void initialize(shared_ptr<Room> r1, shared_ptr<Room> r2) {
        room1 = std::move(r1);
        room2 = std::move(r2);
    }

    Door(const Door&) = delete;
    Door& operator=(const Door&) = delete;
};

class Maze {
private:
    vector<shared_ptr<Room>> rooms;
public:
    Maze& addRoom(shared_ptr<Room> r) {
        std::println("Maze::addRoom {}", reinterpret_cast<void*>(r.get()));
        rooms.push_back(std::move(r));
        return *this;
    }

    [[nodiscard]]
    shared_ptr<Room> roomNo(int n) const {
        for (const Room& r: rooms) {
            // actual lookup logic here...
        }
        return nullptr;
    }

    [[nodiscard]]
    virtual unique_ptr<Maze> clone() const {
        return std::make_unique<Maze>(*this);
    }
};

class MazeFactory {
public:
    MazeFactory() = default;

    virtual ~MazeFactory() = default;

    [[nodiscard]]
    virtual unique_ptr<Maze> makeMaze() const {
        return std::make_unique<Maze>();
    }

    [[nodiscard]]
    virtual shared_ptr<Wall> makeWall() const {
        return std::make_shared<Wall>();
    }

    [[nodiscard]]
    virtual shared_ptr<Room> makeRoom(int n) const {
        return std::make_shared<Room>(n);
    }

    [[nodiscard]]
    virtual shared_ptr<Door> makeDoor(shared_ptr<Room> r1, shared_ptr<Room> r2) const {
        return std::make_shared<Door>(std::move(r1), std::move(r2));
    }
};

```

```

class MazePrototypeFactory: public MazeFactory {
private:
    unique_ptr<Maze> prototypeMaze;
    shared_ptr<Room> prototypeRoom;
    shared_ptr<Wall> prototypeWall;
    shared_ptr<Door> prototypeDoor;
public:
    MazePrototypeFactory(unique_ptr<Maze> m, shared_ptr<Wall> w, shared_ptr<Room> r, shared_ptr<Door> d):
        MazeFactory(), prototypeMaze{std::move(m)}, prototypeRoom{std::move(r)},
        prototypeWall{std::move(w)}, prototypeDoor{std::move(d)} {}

    virtual unique_ptr<Maze> makeMaze() const override {
        return prototypeMaze->clone();
    }

    [[nodiscard]]
    virtual shared_ptr<Room> makeRoom(int n) const override {
        return prototypeRoom->clone();
    }

    [[nodiscard]]
    virtual shared_ptr<Wall> makeWall() const override {
        return prototypeWall->clone();
    }

    [[nodiscard]]
    virtual shared_ptr<Door> makeDoor(shared_ptr<Room> r1, shared_ptr<Room> r2) const override {
        shared_ptr<Door> door = prototypeDoor->clone();
        door->initialize(std::move(r1), std::move(r2));
        return door;
    }

    MazePrototypeFactory(const MazePrototypeFactory&) = delete;
    MazePrototypeFactory& operator=(const MazePrototypeFactory&) = delete;
};

class MazeGame {
public:
    [[nodiscard]]
    unique_ptr<Maze> createMaze(MazePrototypeFactory& factory) {
        unique_ptr<Maze> maze = factory.makeMaze();
        shared_ptr<Room> r1 = factory.makeRoom(1);
        shared_ptr<Room> r2 = factory.makeRoom(2);
        shared_ptr<Door> door = factory.makeDoor(r1, r2);

        maze->addRoom(std::move(r1))
            .addRoom(std::move(r2));

        r1->setSide(Direction::North, factory.makeWall())
            .setSide(Direction::East, door)
            .setSide(Direction::South, factory.makeWall())
            .setSide(Direction::West, factory.makeWall());

        r2->setSide(Direction::North, factory.makeWall())
            .setSide(Direction::East, factory.makeWall())
            .setSide(Direction::South, factory.makeWall())
            .setSide(Direction::West, door);

        return maze;
    }
};

int main(int argc, char* argv[]) {
    MazeGame game;
    MazePrototypeFactory simpleMazeFactory(
        std::make_unique<Maze>(),
        std::make_shared<Wall>(),
        std::make_shared<Room>(0),
        std::make_shared<Door>()
    );

    unique_ptr<Maze> maze = game.createMaze(simpleMazeFactory);
}

```

The program output is:

```

Maze::addRoom 0x1160f50
Maze::addRoom 0x1160f70
Room::setSide 0 0x11613c0
Room::setSide 2 0x1160f90
Room::setSide 1 0x11613e0
Room::setSide 3 0x1161400
Room::setSide 0 0x1161420
Room::setSide 2 0x1161440
Room::setSide 1 0x1161460
Room::setSide 3 0x1160f90

```

See also

- Function prototype

References

- Duell, Michael (July 1997). "Non-Software Examples of Design Patterns". *Object Magazine*. **7** (5): 54. ISSN 1055-3614 (<https://search.worldcat.org/issn/1055-3614>).

2. Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (<https://archive.org/details/designpatternsel00gamm>). Addison-Wesley. ISBN 0-201-63361-2.
 3. "The Prototype design pattern - Problem, Solution, and Applicability" (<http://w3sdesign.com/?gr=c04&ugr=proble>). *w3sDesign.com*. Retrieved 2017-08-17.
-

Retrieved from "https://en.wikipedia.org/w/index.php?title=Prototype_pattern&oldid=1309374127"

Resource acquisition is initialization

Resource acquisition is initialization (**RAII**)^[1] is a programming idiom^[2] used in several object-oriented, statically typed programming languages to describe a particular language behavior. In RAII, holding a resource is a class invariant, and is tied to object lifetime. Resource allocation (or acquisition) is done during object creation (specifically initialization), by the constructor, while resource deallocation (release) is done during object destruction (specifically finalization), by the destructor. In other words, resource acquisition must succeed for initialization to succeed. Thus, the resource is guaranteed to be held between when initialization finishes and finalization starts (holding the resources is a class invariant), and to be held only when the object is alive. Thus, if there are no object leaks, there are no resource leaks.

RAII is associated most prominently with C++, where it originated, but also Ada,^[3] Vala,^[4] and Rust.^[5] The technique was developed for exception-safe resource management in C++^[6] during 1984–1989, primarily by Bjarne Stroustrup and Andrew Koenig.^[7] and the term itself was coined by Stroustrup.^[8]

Other names for this idiom include *Constructor Acquires, Destructor Releases* (CADRe)^[9] and one particular style of use is called *Scope-based Resource Management* (SBRM).^[10] This latter term is for the special case of automatic variables. RAII ties resources to object *lifetime*, which may not coincide with entry and exit of a scope. (Notably variables allocated on the free store have lifetimes unrelated to any given scope.) However, using RAII for automatic variables (SBRM) is the most common use case.

C++ example

The following C++23 example demonstrates usage of RAII for file access and mutex locking:

```
import std;

using std::lock_guard;
using std::mutex;
using std::ofstream;
using std::runtime_error;
using std::string;

void writeToFile(const string& message) {
    // mutex is to protect access to file (which is shared across threads).
    static mutex m;

    // Lock mutex before accessing file.
    lock_guard<mutex> lock(m);

    // Try to open file.
    ofstream f{"example.txt"};
    if (!f.is_open()) {
        throw runtime_error("unable to open file");
    }

    // Write message to file.
    std::println(f, message);

    // file will be closed first when leaving scope (regardless of exception)
    // mutex will be unlocked second (from lock destructor) when leaving scope
    // (regardless of exception).
}
```

This code is exception-safe because C++ guarantees that all objects with automatic storage duration (local variables) are destroyed at the end of the enclosing scope in the reverse order of their construction.^[11] The destructors of both the *lock* and *file* objects are therefore guaranteed to be called when returning from the function, whether an exception has been thrown or not.^[12]

Local variables allow easy management of multiple resources within a single function: they are destroyed in the reverse order of their construction, and an object is destroyed only if fully constructed—that is, if no exception propagates from its constructor.^[13]

Using RAII greatly simplifies resource management, reduces overall code size and helps ensure program correctness. RAII is therefore recommended by industry-standard guidelines,^[14] and most of the C++ standard library follows the idiom.^[15]

Benefits

The advantages of RAII as a resource management technique are that it provides encapsulation, exception safety (for stack resources), and locality (it allows acquisition and release logic to be written next to each other).

Encapsulation is provided because resource management logic is defined once in the class, not at each call site. Exception safety is provided for stack resources (resources that are released in the same scope as they are acquired) by tying the resource to the lifetime of a stack variable (a local variable declared in a given scope): if an exception is thrown, and proper exception handling

is in place, the only code that will be executed when exiting the current scope are the destructors of objects declared in that scope. Finally, locality of definition is provided by writing the constructor and destructor definitions next to each other in the class definition.

Resource management therefore needs to be tied to the lifespan of suitable objects in order to gain automatic allocation and reclamation. Resources are acquired during initialization, when there is no chance of them being used before they are available, and released with the destruction of the same objects, which is guaranteed to take place even in case of errors.

Comparing RAII with the `finally` construct used in Java, Stroustrup wrote that “In realistic systems, there are far more resource acquisitions than kinds of resources, so the ‘resource acquisition is initialization’ technique leads to less code than use of a ‘finally’ construct.”^[1]

As a class invariant, RAII provides guarantees that an object instance that is supposed to have acquired a resource has in fact done so. This eliminates the need for additional “setup” methods to get a newly-created object into a usable state (all such work is performed in the constructor; similarly, “shutdown” tasks to release resources occur in the object’s destructor), and the need to test instances to verify that they have been properly set up before every use.^[16]

Typical uses

The RAII design is often used for controlling mutex locks in multi-threaded applications. In that use, the object releases the lock when destroyed. Without RAII in this scenario the potential for deadlock would be high and the logic to lock the mutex would be far from the logic to unlock it. With RAII, the code that locks the mutex essentially includes the logic that the lock will be released when execution leaves the scope of the RAII object.

Another typical example is interacting with files: We could have an object that represents a file that is open for writing, wherein the file is opened in the constructor and closed when execution leaves the object’s scope. In both cases, RAII ensures only that the resource in question is released appropriately; care must still be taken to maintain exception safety. If the code modifying the data structure or file is not exception-safe, the mutex could be unlocked or the file closed with the data structure or file corrupted.

Ownership of dynamically allocated objects (memory allocated with `new` in C++) can also be controlled with RAII, such that the object is released when the RAII (stack-based) object is destroyed. For this purpose, the C++11 standard library defines the smart pointer classes `std::unique_ptr` for single-owned objects and `std::shared_ptr` for objects with shared ownership. Similar classes are also available through `std::auto_ptr` in C++98, and `boost::shared_ptr` in the Boost libraries.

Also, messages can be sent to network resources using RAII. In this case, the RAII object would send a message to a socket at the end of the constructor, when its initialization is completed. It would also send a message at the beginning of the destructor, when the object is about to be destroyed. Such a construct might be used in a client object to establish a connection with a server running in another process.

Compiler "cleanup" extensions

Both Clang and the GNU Compiler Collection implement a non-standard extension to the C language to support RAII: the “cleanup” variable attribute.^[17] The following annotates a variable with a given destructor function that it will call when the variable goes out of scope:

```
void example_usage() {
    __attribute__((cleanup(fclose))) FILE* logfile = fopen("logfile.txt", "w+");
    fprintf("Hello logfile!", logfile);
}
```

In this example, the compiler arranges for the `fclose` function to be called on `logfile` before `example_usage` returns.

Limitations

RAII only works for resources acquired and released (directly or indirectly) by stack-allocated objects, where there is a well-defined static object lifetime. Heap-allocated objects which themselves acquire and release resources are common in many languages, including C++. RAII depends on heap-based objects to be implicitly or explicitly deleted along all possible execution paths, in order to trigger its resource-releasing destructor (or equivalent).^{[18]:8:27} This can be achieved by using smart pointers to manage all heap objects, with weak pointers for cyclically referenced objects.

In C++, stack unwinding is only guaranteed to occur if the exception is caught somewhere. This is because “If no matching handler is found in a program, the function `terminate()` is called; whether or not the stack is unwound before this call to `terminate()` is implementation-defined (15.5.1).” (C++03 standard, §15.3/9).^[19] This behavior is usually acceptable, since the operating system releases remaining resources like memory, files, sockets, etc. at program termination.

At the 2018 Gamelab conference, Jonathan Blow claimed that use of RAII can cause memory fragmentation which in turn can cause cache misses and a 100 times or worse hit on performance.^[20]

Reference counting

Perl, Python (in the CPython implementation),^[21] and PHP^[22] manage object lifetime by reference counting, which makes it possible to use RAII. Objects that are no longer referenced are immediately destroyed or finalized and released, so a destructor or finalizer can release the resource at that time. However, it is not always idiomatic in such languages, and is specifically discouraged in Python (in favor of context managers and *finalizers* from the *weakref* package).

However, object lifetimes are not necessarily bound to any scope, and objects may be destroyed non-deterministically or not at all. This makes it possible to accidentally leak resources that should have been released at the end of some scope. Objects stored in a static variable (notably a global variable) may not be finalized when the program terminates, so their resources are not released; CPython makes no guarantee of finalizing such objects, for instance. Further, objects with circular references will not be collected by a simple reference counter, and will live indeterminately long; even if collected (by more sophisticated garbage collection), destruction time and destruction order will be non-deterministic. In CPython there is a cycle detector which detects cycles and finalizes the objects in the cycle, though prior to CPython 3.4, cycles are not collected if any object in the cycle has a finalizer.^[23]

See also

- Smart pointer
- Reference counting
- Garbage collection (computer science)

References

1. Stroustrup, Bjarne (2017-09-30). "Why doesn't C++ provide a "finally" construct?" (http://www.stroustrup.com/bs_faq2.html#finally). Retrieved 2019-03-09.
2. Sutter, Herb; Alexandrescu, Andrei (2005). *C++ Coding Standards* (https://archive.org/details/isbn_0321113586). C++ In-Depth Series. Addison-Wesley. p. 24 (https://archive.org/details/isbn_0321113586/page/n54). ISBN 978-0-321-11358-0.
3. "Gem #70: The Scope Locks Idiom" (<https://www.adacore.com/gems/gem-70>). *AdaCore*. Retrieved 21 May 2021.
4. The Valadate Project. "Destruction" (<https://naaando.gitbooks.io/the-vala-tutorial/content/en/4-object-oriented-programming/destruction.html>). *The Vala Tutorial version 0.30*. Retrieved 21 May 2021.
5. "RAII - Rust By Example" (<https://doc.rust-lang.org/rust-by-example/scope/raii.html>). *doc.rust-lang.org*. Retrieved 2020-11-22.
6. Stroustrup 1994, 16.5 Resource Management, pp. 388–89.
7. Stroustrup 1994, 16.1 Exception Handling: Introduction, pp. 383–84.
8. Stroustrup 1994, p. 389. I called this technique "resource acquisition is initialization."
9. Arthur Tchaikovsky (2012-11-06). "Change official RAII to CADRe" (<https://groups.google.com/a/isocpp.org/d/msg/std-proposals/UnarLCzNPcI/epOagK6j-GAJ>). *ISO C++ Standard - Future Proposals*. Google Groups. Retrieved 2019-03-09.
10. Chou, Allen (2014-10-01). "Scope-Based Resource Management (RAII)" (<http://allenchou.net/2014/10/scope-based-resource-management-raii/>). Retrieved 2019-03-09.
11. Richard Smith (2017-03-21). "Working Draft, Standard for Programming Language C++" (<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf>) (PDF). p. 151, section §9.6. Retrieved 2023-09-07.
12. "How can I handle a destructor that fails?" (<https://isocpp.org/wiki/faq/exceptions#dtors-shouldnt-throw>). Standard C++ Foundation. Retrieved 2019-03-09.
13. Richard Smith (2017-03-21). "Working Draft, Standard for Programming Language C++" (<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf>) (PDF). Retrieved 2019-03-09.
14. Stroustrup, Bjarne; Sutter, Herb (2020-08-03). "C++ Core Guidelines" (<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>). Retrieved 2020-08-15.
15. "I have too many try blocks; what can I do about it?" (<https://isocpp.org/wiki/faq/exceptions#too-many-trycatch-blocks>). Standard C++ Foundation. Retrieved 2019-03-09.
16. RAII at cppreference.com (<https://en.cppreference.com/w/cpp/language/raii.html>)
17. "Specifying Attributes of Variables" (<https://gcc.gnu.org/onlinedocs/gcc/Variable-Attributes.html>). *Using the GNU Compiler Collection (GCC)*. GNU Project. Retrieved 2019-03-09.
18. Weimer, Westley; Nacula, George C. (2008). "Exceptional Situations and Program Reliability" (<https://web.eecs.umich.edu/~weimerw/p/weimer-toplas2008.pdf>) (PDF). *ACM Transactions on Programming Languages and Systems*. Vol. 30, no. 2.
19. ildjarn (2011-04-05). "RAII and Stack unwinding" (<https://stackoverflow.com/a/5557651>). Stack Overflow. Retrieved 2019-03-09.

20. Gamelab2018 - Jon Blow's Design decisions on creating Jai a new language for game programmers (<https://www.youtube.com/watch?v=uZgbKrDEzAs&t=614>) on YouTube
21. "Extending Python with C or C++: Reference Counts" (<https://docs.python.org/3/extending/extending.html#refcounts>). *Extending and Embedding the Python Interpreter*. Python Software Foundation. Retrieved 2019-03-09.
22. hobbs (2011-02-08). "Does PHP support the RAI pattern? How?" (<https://stackoverflow.com/a/4938780>). Retrieved 2019-03-09.
23. "gc — Garbage Collector interface" (<https://docs.python.org/3/library/gc.html>). *The Python Standard Library*. Python Software Foundation. Retrieved 2019-03-09.

Further reading

- Stroustrup, Bjarne (1994). *The Design and Evolution of C++*. Addison-Wesley. Bibcode:1994dec..book.....S (<https://ui.adsabs.harvard.edu/abs/1994dec..book.....S>). ISBN 978-0-201-54330-8.

External links

- Sample Chapter: "Gotcha #67: Failure to Employ Resource Acquisition Is Initialization" (<https://www.informit.com/articles/article.aspx?p=30642&seqNum=8>) by Stephen C. Dewhurst
- Interview: "A Conversation with Bjarne Stroustrup" (<https://www.artima.com/intv/modern3.html>) by Bill Venners
- Article: "The Law of The Big Two" (<https://www.artima.com/cppsource/bigtwo3.html>) by Bjorn Karlsson and Matthew Wilson
- Article: "Implementing the 'Resource Acquisition is Initialization' Idiom" (https://web.archive.org/web/20090815095635/http://gethelp.devx.com/techtips/cpp_pro/10min/2001/november/10min1101.asp) by Danny Kalev
- Article: "RAII, Dynamic Objects, and Factories in C++" (<https://www.codeproject.com/Articles/10141/RAII-Dynamic-Objects-and-Factories-in-C>) by Roland Pibinger
- RAI in Delphi: "One-liner RAI in Delphi" (<http://blog.barrkel.com/2010/01/one-liner-rai-in-delphi.html>) by Barry Kelly
- Guide: RAI in C++ (<https://www.w3computing.com/articles/resource-acquisition-is-initialization-rai-in-cpp/>) by W3computing

Retrieved from "https://en.wikipedia.org/w/index.php?title=Resource_acquisition_is_initialization&oldid=1318228926"

Singleton pattern

In object-oriented programming, the **singleton pattern** is a software design pattern that restricts the instantiation of a class to a singular instance. It is one of the well-known "Gang of Four" design patterns, which describe how to solve recurring problems in object-oriented software.[1] The pattern is useful when exactly one object is needed to coordinate actions across a system.

More specifically, the singleton pattern allows classes to:[2]

- Ensure they only have one instance
- Provide easy access to that instance
- Control their instantiation (for example, hiding the constructors of a class)

The term comes from the mathematical concept of a singleton.

Common uses

Singletons are often preferred to global variables because they do not pollute the global namespace (or their containing namespace). Additionally, they permit lazy allocation and initialization, whereas global variables in many languages will always consume resources.[1][3]

The singleton pattern can also be used as a basis for other design patterns, such as the abstract factory, factory method, builder and prototype patterns. Facade objects are also often singletons because only one facade object is required.

Logging is a common real-world use case for singletons, because all objects that wish to log messages require a uniform point of access and conceptually write to a single source.[4]

Implementations

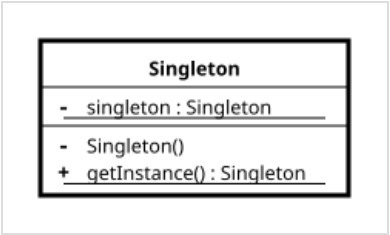
Implementations of the singleton pattern ensure that only one instance of the singleton class ever exists and typically provide global access to that instance.

Typically, this is accomplished by:

- Declaring all constructors of the class to be private, which prevents it from being instantiated by other objects
- Providing a static method that returns a reference to the instance

The instance is usually stored as a private static variable; the instance is created when the variable is initialized, at some point before when the static method is first called.

This C++23 implementation is based on the pre-C++98 implementation in the book .



A class diagram exemplifying the singleton pattern.

```
import std;

class Singleton {
private:
    Singleton() = default; // no public constructor
    ~Singleton() = default; // no public destructor
    inline static Singleton* instance = nullptr; // declaration class variable
    int value;
public:
    // defines a class operation that lets clients access its unique instance.
    static Singleton& getInstance() {
        if (!instance) {
            instance = new Singleton();
        }
        return *instance;
    }

    Singleton(const Singleton&) = delete("Copy construction disabled");
    Singleton& operator=(const Singleton&) = delete("Copy assignment disabled");

    static void destroy() {
        delete instance;
        instance = nullptr;
    }

    // existing interface goes here
    [[nodiscard]]
    int getValue() const noexcept {
        return value;
    }

    void setValue(int newValue) noexcept {
        value = newValue;
    }
};
```

```
int main() {
    Singleton::getInstance().setVale(42);
    std::println("value = {}", Singleton::getInstance().getVale());
    Singleton::destroy();
}
```

The program output is

```
value=42
```

This is an implementation of the Meyers singleton^[5] in C++11. The Meyers singleton has no destruct method. The program output is the same as above.

```
import std;

class Singleton {
private:
    Singleton() = default;
    ~Singleton() = default;
    int value;
public:
    static Singleton& getInstance() {
        static Singleton instance;
        return instance;
    }

    [[nodiscard]]
    int getVale() const noexcept {
        return value;
    }

    void setVale(int newVale) noexcept {
        value = newVale;
    }
};

int main() {
    Singleton::getInstance().setVale(42);
    std::println("value = {}", Singleton::getInstance().getVale());
}
```

Lazy initialization

A singleton implementation may use lazy initialization in which the instance is created when the static method is first invoked. In multithreaded programs, this can cause race conditions that result in the creation of multiple instances. The following Java 5+ example^[6] is a thread-safe implementation, using lazy initialization with double-checked locking.

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

    private Singleton() {}

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

Criticism

Some consider the singleton to be an anti-pattern that introduces global state into an application, often unnecessarily. This introduces a potential dependency on the singleton by other objects, requiring analysis of implementation details to determine whether a dependency actually exists.^[7] This increased coupling can introduce difficulties with unit testing.^[8] In turn, this places restrictions on any abstraction that uses the singleton, such as preventing concurrent use of multiple instances.^{[8][9][10]}

Singletons also violate the single-responsibility principle because they are responsible for enforcing their own uniqueness along with performing their normal functions.^[8]

See also

- Initialization-on-demand holder idiom
- Multiton pattern
- Software design pattern

References

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (<https://archive.org/details/designpatternsel00gamm/page/127>). Addison Wesley. pp. 127ff (<https://archive.org/details/designpatternsel00gamm/page/127>). ISBN 0-201-63361-2.
2. "The Singleton design pattern - Problem, Solution, and Applicability" (<http://w3sdesign.com/?gr=c05&ugr=proble>). *w3sDesign.com*. Retrieved 2017-08-16.
3. Soni, Devin (31 July 2019). "What Is a Singleton?" (<https://betterprogramming.pub/what-is-a-singleton-2dc38ca08e92>). *BetterProgramming*. Retrieved 28 August 2021.
4. Rainsberger, J.B. (1 July 2001). "Use your singletons wisely" (<https://web.archive.org/web/20210224180356/https://www.ibm.com/developerworks/library/co-single/>). IBM. Archived from the original (<https://www.ibm.com/developerworks/library/co-single/>) on 24 February 2021. Retrieved 28 August 2021.
5. Scott Meyers (1997). *More Effective C++*. Addison Wesley. pp. 146 ff. ISBN 0-201-63371-X.
6. Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates (October 2004). "5: One of a Kind Objects: The Singleton Pattern" (<https://books.google.com/books?id=GGpXN9SMELMC&pg=PA182>). *Head First Design Patterns* (First ed.). O'Reilly Media, Inc. p. 182. ISBN 978-0-596-00712-6.
7. "Why Singletons Are Controversial" (<https://web.archive.org/web/20210506162753/https://code.google.com/archive/p/google-singleton-detector/wikis/WhySingletonsAreControversial.wiki>). *Google Code Archive*. Archived from the original (<https://code.google.com/archive/p/google-singleton-detector/wikis/WhySingletonsAreControversial.wiki>) on 6 May 2021. Retrieved 28 August 2021.
8. Button, Brian (25 May 2004). "Why Singletons are Evil" (<https://web.archive.org/web/20210715184717/https://docs.microsoft.com/en-us/archive/blogs/scottdensmore/why-singletons-are-evil>). *Being Scott Densmore*. Microsoft. Archived from the original (<https://docs.microsoft.com/en-us/archive/blogs/scottdensmore/why-singletons-are-evil>) on 15 July 2021. Retrieved 28 August 2021.
9. Steve Yegge. Singletons considered stupid (<http://steve.yegge.googlepages.com/singleton-considered-stupid>), September 2004
10. Hevery, Miško, "Global State and Singletons (<http://googletesting.blogspot.com/2008/11/clean-code-talks-global-state-and.html>)", *Clean Code Talks*, 21 November 2008.

External links

- Complete article "Java Singleton Pattern Explained (<https://howtodoinjava.com/design-patterns/creational/singleton-design-pattern-in-java/>)"
- Four different ways to implement singleton in Java "Ways to implement singleton in Java (<https://web.archive.org/web/20150709155148/http://www.javaexperience.com/design-patterns-singleton-design-pattern/>)"

Retrieved from "https://en.wikipedia.org/w/index.php?title=Singleton_pattern&oldid=1317276511"

Adapter pattern

In software engineering, the **adapter pattern** is a software design pattern (also known as wrapper, an alternative naming shared with the decorator pattern) that allows the interface of an existing class to be used as another interface.^[1] It is often used to make existing classes work with others without modifying their source code.

An 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.

Overview

The adapter^[2] design pattern is one of the twenty-three well-known Gang of Four design patterns that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

The adapter design pattern solves problems like:^[3]

- How can a class be reused that does not have an interface that a client requires?
- How can classes that have incompatible interfaces work together?
- How can an alternative interface be provided for a class?

Often an (already existing) class can not be reused only because its interface does not conform to the interface clients require.

The adapter design pattern describes how to solve such problems:

- Define a separate adapter class that converts the (incompatible) interface of a class (adaptee) into another interface (target) clients require.
- Work through an adapter to work with (reuse) classes that do not have the required interface.

The key idea in this pattern is to work through a separate adapter that adapts the interface of an (already existing) class without changing it.

Clients don't know whether they work with a target class directly or through an adapter with a class that does not have the target interface.

See also the UML class diagram below.

Definition

An adapter allows two incompatible interfaces to work together. This is the real-world definition for an adapter. Interfaces may be incompatible, but the inner functionality should suit the need. The adapter design pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients.

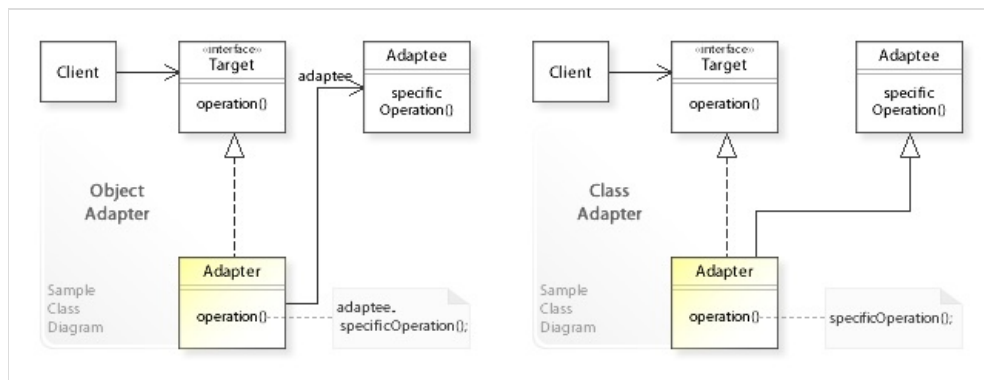
Usage

An adapter can be used when the wrapper must respect a particular interface and must support polymorphic behavior. Alternatively, a decorator makes it possible to add or alter behavior of an interface at run-time, and a facade is used when an easier or simpler interface to an underlying object is desired.^[4]

Pattern	Intent
Adapter or wrapper	Converts one interface to another so that it matches what the client is expecting
<u>Decorator</u>	Dynamically adds responsibility to the interface by wrapping the original code
<u>Delegation</u>	Support "composition over inheritance"
<u>Facade</u>	Provides a simplified interface

Structure

UML class diagram



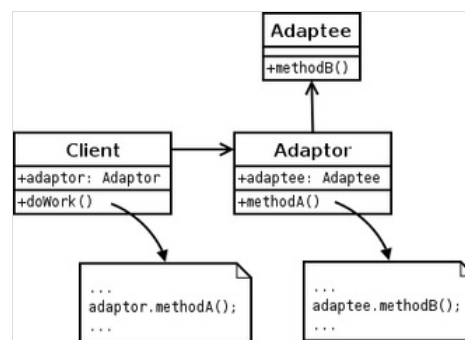
A sample UML class diagram for the adapter design pattern.^[5]

In the above UML class diagram, the client class that requires a target interface cannot reuse the adaptee class directly because its interface doesn't conform to the target interface. Instead, the client works through an adapter class that implements the target interface in terms of adaptee:

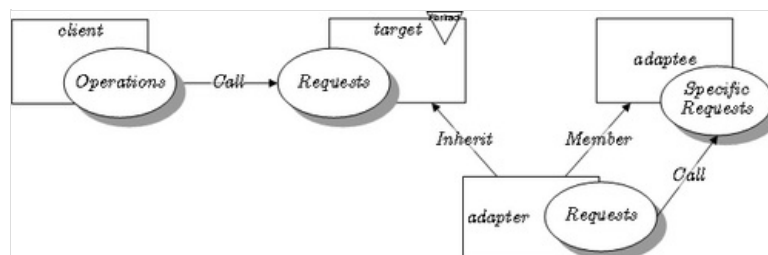
- The object adapter way implements the target interface by delegating to an adaptee object at run-time (`adaptee.specificOperation()`).
- The class adapter way implements the target interface by inheriting from an adaptee class at compile-time (`specificOperation()`).

Object adapter pattern

In this 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.



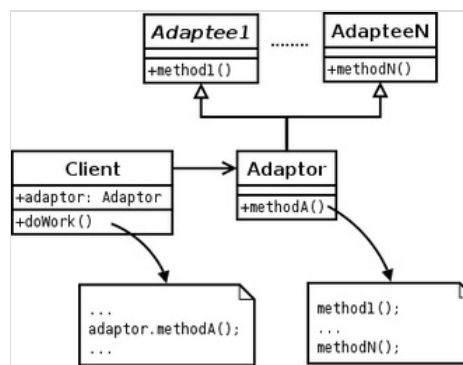
The object adapter pattern expressed in UML



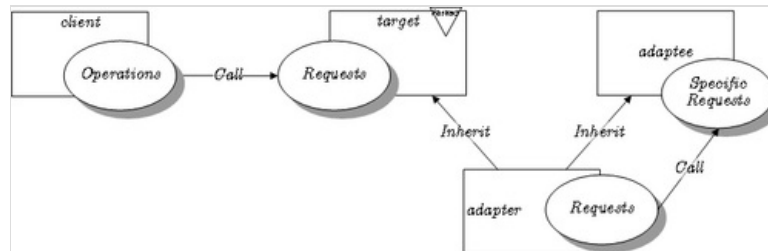
The object adapter pattern expressed in LePUS3

Class adapter pattern

This adapter pattern uses multiple polymorphic interfaces implementing or inheriting both the interface that is expected and the interface that is pre-existing. It is typical for the expected interface to be created as a pure interface class, especially in languages such as Java (before JDK 1.8) that do not support multiple inheritance of classes.^[1]



The class adapter pattern expressed in UML.



The class adapter pattern expressed in LePUS3

A further form of runtime adapter pattern

Motivation from compile time solution

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:

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

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

Run-time adapter solution

A solution using "adapters" proceeds as follows:

- Define an intermediary "provider" interface, and write an implementation of that provider interface that wraps the source of the data, ClassA in this example, and outputs the data formatted as appropriate:

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

public class ClassAFormat1 implements StringProvider {
    private ClassA classA = null;

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

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

    private String format(final String sourceValue) {
        // Manipulate the source string into a format required
        // by the object needing the source object's data
        return sourceValue.trim();
    }
}
```

- Write an adapter class that returns the specific implementation of the provider:

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

```
}
}
```

- 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 code, when wishing to transfer data from ClassA to ClassB, write:

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

or more concisely:

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

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

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

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

```
Adapter adapter = AdapterFactory.getInstance()
    .getAdapterFromTo(ClassA.class, ImageProvider.class, "format2");
ImageProvider provider = (ImageProvider) adapter.adapt(classA);
classC.setImage(provider.getImage());
```

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

Implementation of the adapter pattern

When implementing the adapter pattern, for clarity, one can apply the class name [ClassName]To[Interface]Adapter to the provider implementation; for example, DAOToProviderAdapter. It should have a constructor method with an adaptee class variable as a parameter. This parameter will be passed to an instance member of [ClassName]To[Interface]Adapter. When the clientMethod is called, it will have access to the adaptee instance that allows for accessing the required data of the adaptee and performing operations on that data that generates the desired output.

Java

```
interface ILightningPhone {
    void recharge();
    void useLightning();
}

interface IMicroUsbPhone {
    void recharge();
    void useMicroUsb();
}

class Iphone implements ILightningPhone {
    private boolean connector;

    @Override
    public void useLightning() {
        connector = true;
        System.out.println("Lightning connected");
    }

    @Override
    public void recharge() {
        if (connector) {
            System.out.println("Recharge started");
            System.out.println("Recharge finished");
        } else {
            System.out.println("Connect Lightning first");
        }
    }
}

class Android implements IMicroUsbPhone {
```



```

private boolean connector;

@Override
public void useMicroUsb() {
    connector = true;
    System.out.println("MicroUsb connected");
}

@Override
public void recharge() {
    if (connector) {
        System.out.println("Recharge started");
        System.out.println("Recharge finished");
    } else {
        System.out.println("Connect MicroUsb first");
    }
}
}

/* exposing the target interface while wrapping source object */
class LightningToMicroUsbAdapter implements IMicroUsbPhone {
    private final ILightningPhone lightningPhone;

    public LightningToMicroUsbAdapter(ILightningPhone lightningPhone) {
        this.lightningPhone = lightningPhone;
    }

    @Override
    public void useMicroUsb() {
        System.out.println("MicroUsb connected");
        lightningPhone.useLightning();
    }

    @Override
    public void recharge() {
        lightningPhone.recharge();
    }
}

public class AdapterDemo {
    static void rechargeMicroUsbPhone(IMicroUsbPhone phone) {
        phone.useMicroUsb();
        phone.recharge();
    }

    static void rechargeLightningPhone(ILightningPhone phone) {
        phone.useLightning();
        phone.recharge();
    }

    public static void main(String[] args) {
        Android android = new Android();
        Iphone iPhone = new Iphone();

        System.out.println("Recharging android with MicroUsb");
        rechargeMicroUsbPhone(android);

        System.out.println("Recharging iPhone with Lightning");
        rechargeLightningPhone(iPhone);

        System.out.println("Recharging iPhone with MicroUsb");
        rechargeMicroUsbPhone(new LightningToMicroUsbAdapter(iPhone));
    }
}

```

Output

```

Recharging android with MicroUsb
MicroUsb connected
Recharge started
Recharge finished
Recharging iPhone with Lightning
Lightning connected
Recharge started
Recharge finished
Recharging iPhone with MicroUsb
MicroUsb connected
Lightning connected
Recharge started
Recharge finished

```

Python

```

"""
Adapter pattern example.
"""
from abc import ABCMeta, abstractmethod
from typing import NoReturn

RECHARGE: list[str] = ["Recharge started.", "Recharge finished."]
POWER_ADAPTERS: dict[str, str] = {"Android": "MicroUSB", "iPhone": "Lightning"}
CONNECTED_MSG: str = "{} connected."
CONNECT_FIRST_MSG: str = "Connect {} first."

class RechargeTemplate(metaclass = ABCMeta):
    @abstractmethod
    def recharge(self) -> NoReturn:
        raise NotImplementedError("You should implement this.")

class FormatIphone(RechargeTemplate):
    @abstractmethod
    def use_lightning(self) -> NoReturn:
        raise NotImplementedError("You should implement this.")

```

```

class FormatAndroid(RechargeTemplate):
    @abstractmethod
    def use_micro_usb(self) -> NoReturn:
        raise NotImplementedError("You should implement this.")

class iPhone(FormatiPhone):
    __name__: str = "iPhone"

    def __init__(self):
        self.connector: bool = False

    def use_lightning(self) -> None:
        self.connector = True
        print(CONNECTED_MSG.format(POWER_ADAPTERS[self.__name__]))

    def recharge(self) -> None:
        if self.connector:
            for state in RECHARGE:
                print(state)
        else:
            print(CONNECT_FIRST_MSG.format(POWER_ADAPTERS[self.__name__]))

class Android(FormatAndroid):
    __name__: str = "Android"

    def __init__(self) -> None:
        self.connector: bool = False

    def use_micro_usb(self) -> None:
        self.connector = True
        print(CONNECTED_MSG.format(POWER_ADAPTERS[self.__name__]))

    def recharge(self) -> None:
        if self.connector:
            for state in RECHARGE:
                print(state)
        else:
            print(CONNECT_FIRST_MSG.format(POWER_ADAPTERS[self.__name__]))

class iPhoneAdapter(FormatAndroid):
    def __init__(self, mobile: FormatAndroid) -> None:
        self.mobile: FormatAndroid = mobile

    def recharge(self) -> None:
        self.mobile.recharge()

    def use_micro_usb(self) -> None:
        print(CONNECTED_MSG.format(POWER_ADAPTERS["Android"]))
        self.mobile.use_lightning()

class AndroidRecharger:
    def __init__(self) -> None:
        self.phone: Android = Android()
        self.phone.use_micro_usb()
        self.phone.recharge()

class iPhoneMicroUSBRecharger:
    def __init__(self) -> None:
        self.phone: iPhone = iPhone()
        self.phone_adapter: iPhoneAdapter = iPhoneAdapter(self.phone)
        self.phone_adapter.use_micro_usb()
        self.phone_adapter.recharge()

class iPhoneRecharger:
    def __init__(self) -> None:
        self.phone: iPhone = iPhone()
        self.phone.use_lightning()
        self.phone.recharge()

print("Recharging Android with MicroUSB recharger.")
AndroidRecharger()
print()

print("Recharging iPhone with MicroUSB using adapter pattern.")
iPhoneMicroUSBRecharger()
print()

print("Recharging iPhone with iPhone recharger.")
iPhoneRecharger()

```

C#

```

public interface ILightningPhone
{
    void ConnectLightning();
    void Recharge();
}

public interface IUsbPhone
{
    void ConnectUsb();
    void Recharge();
}

public sealed class AndroidPhone : IUsbPhone
{
    private bool isConnected;

    public void ConnectUsb()
    {
        this.isConnected = true;
        Console.WriteLine("Android phone connected.");
    }
}

```

```

    public void Recharge()
    {
        if (this.isConnected)
        {
            Console.WriteLine("Android phone recharging.");
        }
        else
        {
            Console.WriteLine("Connect the USB cable first.");
        }
    }
}

public sealed class ApplePhone : ILightningPhone
{
    private bool isConnected;

    public void ConnectLightning()
    {
        this.isConnected = true;
        Console.WriteLine("Apple phone connected.");
    }

    public void Recharge()
    {
        if (this.isConnected)
        {
            Console.WriteLine("Apple phone recharging.");
        }
        else
        {
            Console.WrizteLine("Connect the Lightning cable first.");
        }
    }
}

public sealed class LightningToUsbAdapter : IUsbPhone
{
    private readonly ILightningPhone lightningPhone;

    private bool isConnected;

    public LightningToUsbAdapter(ILightningPhone lightningPhone)
    {
        this.lightningPhone = lightningPhone;
    }

    public void ConnectUsb()
    {
        this.lightningPhone.ConnectLightning();
    }

    public void Recharge()
    {
        this.lightningPhone.Recharge();
    }
}

public void Main()
{
    ILightningPhone applePhone = new ApplePhone();
    IUsbPhone adapterCable = new LightningToUsbAdapter(applePhone);
    adapterCable.ConnectUsb();
    adapterCable.Recharge();
}

```

Output:

```

Apple phone connected.
Apple phone recharging.

```

See also

- Adapter (<https://java-design-patterns.com/patterns/adapter/>) Java Design Patterns - Adapter
- Delegation, strongly relevant to the object adapter pattern.
- Dependency inversion principle, which can be thought of as applying the adapter pattern, when the high-level class defines its own (adapter) interface to the low-level module (implemented by an adaptee class).
- Ports and adapters architecture
- Shim
- Wrapper function
- Wrapper library

References

- Freeman, Eric; Freeman, Elisabeth; Sierra, Kathy; Bates, Bert (2004). *Head First Design Patterns* (<https://web.archive.org/web/20130504183857/http://www.headfirstlabs.com/books/hfdp/>). O'Reilly Media. p. 244. ISBN 978-0-596-00712-6. OCLC 809772256 (<https://search.worldcat.org/oclc/809772256>). Archived from the original (<http://www.headfirstlabs.com/books/hfdp/>) (paperback) on 2013-05-04. Retrieved 2013-04-30.

2. Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (<https://archive.org/details/designpatternsel00gamm/page/139>). Addison Wesley. pp. 139ff (<https://archive.org/details/designpatternsel00gamm/page/139>). ISBN 0-201-63361-2.
3. "The Adapter design pattern - Problem, Solution, and Applicability" (<https://web.archive.org/web/20170828230927/http://w3sdesign.com/?gr=s01&ugr=proble>). *w3sDesign.com*. Archived from the original (<http://w3sdesign.com/?gr=s01&ugr=proble>) on 2017-08-28. Retrieved 2017-08-12.
4. Freeman, Eric; Freeman, Elisabeth; Sierra, Kathy; Bates, Bert (2004). Hendrickson, Mike; Loukides, Mike (eds.). *Head First Design Patterns* (https://www.goodreads.com/book/show/58128.Head_First_Design_Patterns) (paperback). Vol. 1. O'Reilly Media. pp. 243, 252, 258, 260. ISBN 978-0-596-00712-6. Retrieved 2012-07-02.
5. "The Adapter design pattern - Structure and Collaboration" (<https://web.archive.org/web/20170828231911/http://w3sdesign.com/?gr=s01&ugr=struct>). *w3sDesign.com*. Archived from the original (<http://w3sdesign.com/?gr=s01&ugr=struct>) on 2017-08-28. Retrieved 2017-08-12.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Adapter_pattern&oldid=1318866987"

Bridge pattern

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

When a class varies often, the features of object-oriented programming become very useful because changes to a program's code can be made easily with minimal prior knowledge about the program. The bridge pattern is useful when both the class and what it does vary often. The class itself can be thought of as the *abstraction* and what the class can do as the *implementation*. The bridge pattern can also be thought of as two layers of abstraction.

When there is only one fixed implementation, this pattern is known as the Pimpl idiom in the C++ world.

The bridge pattern is often confused with the adapter pattern, and is often implemented using the object adapter pattern; e.g., in the Java code below.

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

Overview

The Bridge design pattern is one of the twenty-three well-known *GoF design patterns* that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.^[1]

What problems can the Bridge design pattern solve?^[2]

- An abstraction and its implementation should be defined and extended independently from each other.
- A compile-time binding between an abstraction and its implementation should be avoided so that an implementation can be selected at run-time.

When using subclassing, different subclasses implement an abstract class in different ways. But an implementation is bound to the abstraction at compile-time and cannot be changed at run-time.

What solution does the Bridge design pattern describe?

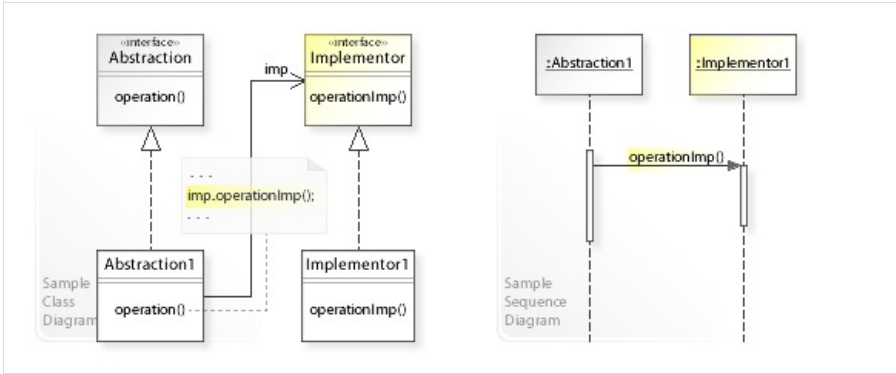
- Separate an abstraction (Abstraction) from its implementation (Implementor) by putting them in separate class hierarchies.
- Implement the Abstraction in terms of (by delegating to) an Implementor object.

This enables to configure an Abstraction with an Implementor object at run-time.

See also the Unified Modeling Language class and sequence diagram below.

Structure

UML class and sequence diagram

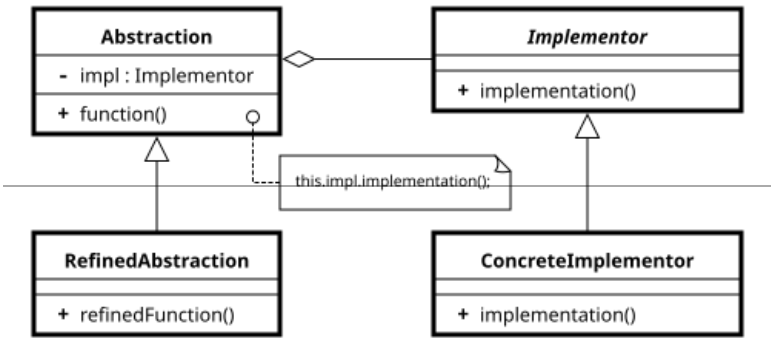


A sample UML class and sequence diagram for the Bridge design pattern.^[3]

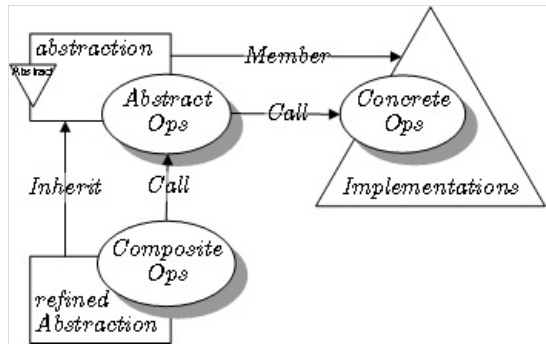
In the above Unified Modeling Language class diagram, an abstraction (Abstraction) is not implemented as usual in a single inheritance hierarchy. Instead, there is one hierarchy for an abstraction (Abstraction) and a separate hierarchy for its implementation (Implementor), which makes the two independent from each other. The Abstraction interface (operation()) is implemented in terms of (by delegating to) the Implementor interface (imp.operationImp()).

The UML sequence diagram shows the run-time interactions: The Abstraction1 object delegates implementation to the Implementor1 object (by calling operationImp() on Implementor1), which performs the operation and returns to Abstraction1.

Class diagram



- Abstraction (abstract class)**
 - defines the abstract interface
 - maintains the Implementor reference.
- RefinedAbstraction (normal class)**
 - extends the interface defined by Abstraction
- Implementor (interface)**
 - defines the interface for implementation classes
- ConcreteImplementor (normal class)**
 - implements the Implementor interface



Bridge in LePUS3 (legend (<https://web.archive.org/web/20180314162121/http://www.lepus.org.uk/ref/legend/legend.xml>))

Example

C#

Bridge pattern compose objects in tree structure. It decouples abstraction from implementation. Here abstraction represents the client from which the objects will be called. An example implemented in C# is given below

```
// Helps in providing truly decoupled architecture
public interface IBridge
{
    void Function1();
    void Function2();
}

public class Bridge1 : IBridge
{
    public void Function1()
    {
        Console.WriteLine("Bridge1.Function1");
    }

    public void Function2()
    {
        Console.WriteLine("Bridge1.Function2");
    }
}

public class Bridge2 : IBridge
{
    public void Function1()
    {
```

```

        Console.WriteLine("Bridge2.Function1");
    }

    public void Function2()
    {
        Console.WriteLine("Bridge2.Function2");
    }
}

public interface IAbstractBridge
{
    void CallMethod1();
    void CallMethod2();
}

public class AbstractBridge : IAbstractBridge
{
    public IBridge bridge;

    public AbstractBridge(IBridge bridge)
    {
        this.bridge = bridge;
    }

    public void CallMethod1()
    {
        this.bridge.Function1();
    }

    public void CallMethod2()
    {
        this.bridge.Function2();
    }
}

```

The Bridge classes are the Implementation that uses the same interface-oriented architecture to create objects. On the other hand, the abstraction takes an instance of the implementation class and runs its method. Thus, they are completely decoupled from one another.

Crystal

```

abstract class DrawingAPI
  abstract def draw_circle(x : Float64, y : Float64, radius : Float64)
end

class DrawingAPI1 < DrawingAPI
  def draw_circle(x : Float, y : Float, radius : Float)
    "API1.circle at #{x}:#{y} - radius: #{radius}"
  end
end

class DrawingAPI2 < DrawingAPI
  def draw_circle(x : Float64, y : Float64, radius : Float64)
    "API2.circle at #{x}:#{y} - radius: #{radius}"
  end
end

abstract class Shape
  protected getter drawing_api : DrawingAPI

  def initialize(@drawing_api)
  end

  abstract def draw
  abstract def resize_by_percentage(percent : Float64)
end

class CircleShape < Shape
  getter x : Float64
  getter y : Float64
  getter radius : Float64

  def initialize(@x, @y, @radius, drawing_api : DrawingAPI)
    super(drawing_api)
  end

  def draw
    @drawing_api.draw_circle(@x, @y, @radius)
  end

  def resize_by_percentage(percent : Float64)
    @radius *= (1 + percent/100)
  end
end

class BridgePattern
  def self.test
    shapes = [] of Shape
    shapes << CircleShape.new(1.0, 2.0, 3.0, DrawingAPI1.new)
    shapes << CircleShape.new(5.0, 7.0, 11.0, DrawingAPI2.new)

    shapes.each do |shape|
      shape.resize_by_percentage(2.5)
      puts shape.draw
    end
  end
end

BridgePattern.test

```

Output

```
API1.circle at 1.0:2.0 - radius: 3.075
API2.circle at 5.0:7.0 - radius: 11.275
```

C++

```
import std;

using std::string;
using std::vector;

class DrawingAPI {
public:
    virtual ~DrawingAPI() = default;
    virtual string drawCircle(float x, float y, float radius) const = 0;
};

class DrawingAPI01: public DrawingAPI {
public:
    [[nodiscard]]
    string drawCircle(float x, float y, float radius) const override {
        return std::format("API01.circle at {}:{} - radius: {}", x, y, radius);
    }
};

class DrawingAPI02: public DrawingAPI {
public:
    [[nodiscard]]
    string drawCircle(float x, float y, float radius) const override {
        return std::format("API02.circle at {}:{} - radius: {}", x, y, radius);
    }
};

class Shape {
protected:
    const DrawingAPI& drawingApi;
public:
    Shape(const DrawingAPI& api):
        drawingApi{api} {}

    virtual ~Shape() = default;

    virtual string draw() const = 0;
    virtual float resizeByPercentage(const float percent) noexcept = 0;
};

class CircleShape: public Shape {
private:
    float x;
    float y;
    float radius;
public:
    CircleShape(const DrawingAPI& api, float x, float y, float radius):
        Shape(api), x{x}, y{y}, radius{radius} {}

    [[nodiscard]]
    string draw() const override {
        return api.drawCircle(x, y, radius);
    }

    [[nodiscard]]
    float resizeByPercentage(float percent) noexcept override {
        return radius * (1.0f + percent / 100.0f);
    }
};

int main(int argc, char* argv[]) {
    const DrawingAPI01 api1;
    const DrawingAPI02 api2;
    vector<CircleShape> shapes {
        CircleShape{1.0f, 2.0f, 3.0f, api1},
        CircleShape{5.0f, 7.0f, 11.0f, api2}
    };

    for (CircleShape& shape: shapes) {
        shape.resizeByPercentage(2.5);
        std::println!("{}", shape.draw());
    }

    return 0;
}
```

Output:

```
API01.circle at 1.000000:2.000000 - radius: 3.075000
API02.circle at 5.000000:7.000000 - radius: 11.275000
```

Java

The following [Java](#) program defines a bank account that separates the account operations from the logging of these operations.

```
// Logger has two implementations: info and warning
@FunctionalInterface
interface Logger {
    void log(String message);

    static Logger info() {
        return message -> System.out.printf("info: %s%n", message);
    }
}
```



```

    }
    static Logger warning() {
        return message -> System.out.printf("warning: %s\n", message);
    }
}

abstract class AbstractAccount {
    private Logger logger = Logger.info();

    public void setLogger(Logger logger) {
        this.logger = logger;
    }

    // the logging part is delegated to the Logger implementation
    protected void operate(String message, boolean result) {
        logger.log(String.format("%s result %s", message, result));
    }
}

class SimpleAccount extends AbstractAccount {
    private int balance;

    public SimpleAccount(int balance) {
        this.balance = balance;
    }

    public boolean isBalanceLow() {
        return balance < 50;
    }

    public void withdraw(int amount) {
        boolean shouldPerform = balance >= amount;
        if (shouldPerform) {
            balance -= amount;
        }
        operate(String.format("withdraw %s", amount, shouldPerform));
    }
}

public class BridgeDemo {
    public static void main(String[] args) {
        SimpleAccount account = new SimpleAccount(100);
        account.withdraw(75);

        if (account.isBalanceLow()) {
            // you can also change the Logger implementation at runtime
            account.setLogger(Logger.warning());
        }

        account.withdraw(10);
        account.withdraw(100);
    }
}

```

It will output:

```

info: withdraw 75 result true
warning: withdraw 10 result true
warning: withdraw 100 result false

```

PHP

```

interface DrawingAPI
{
    function drawCircle($x, $y, $radius);
}

class DrawingAPI1 implements DrawingAPI
{
    public function drawCircle($x, $y, $radius)
    {
        echo "API1.circle at $x:$y radius $radius.\n";
    }
}

class DrawingAPI2 implements DrawingAPI
{
    public function drawCircle($x, $y, $radius)
    {
        echo "API2.circle at $x:$y radius $radius.\n";
    }
}

abstract class Shape
{
    protected $drawingAPI;

    public abstract function draw();
    public abstract function resizeByPercentage($pct);

    protected function __construct(DrawingAPI $drawingAPI)
    {
        $this->drawingAPI = $drawingAPI;
    }
}

class CircleShape extends Shape
{
    private $x;
    private $y;
    private $radius;
}

```

```

    public function __construct($x, $y, $radius, DrawingAPI $drawingAPI)
    {
        parent::__construct($drawingAPI);
        $this->x = $x;
        $this->y = $y;
        $this->radius = $radius;
    }

    public function draw()
    {
        $this->drawingAPI->drawCircle($this->x, $this->y, $this->radius);
    }

    public function resizeByPercentage($pct)
    {
        $this->radius *= $pct;
    }
}

class Tester
{
    public static function main()
    {
        $shapes = array(
            new CircleShape(1, 3, 7, new DrawingAPI1()),
            new CircleShape(5, 7, 11, new DrawingAPI2()),
        );

        foreach ($shapes as $shape) {
            $shape->resizeByPercentage(2.5);
            $shape->draw();
        }
    }
}

Tester::main();

```

Output:

```

API1.circle at 1:3 radius 17.5
API2.circle at 5:7 radius 27.5

```

Scala

```

trait DrawingAPI {
    def drawCircle(x: Double, y: Double, radius: Double)
}

class DrawingAPI1 extends DrawingAPI {
    def drawCircle(x: Double, y: Double, radius: Double) = println(s"API #1 $x $y $radius")
}

class DrawingAPI2 extends DrawingAPI {
    def drawCircle(x: Double, y: Double, radius: Double) = println(s"API #2 $x $y $radius")
}

abstract class Shape(drawingAPI: DrawingAPI) {
    def draw()
    def resizePercentage(pct: Double)
}

class CircleShape(x: Double, y: Double, var radius: Double, drawingAPI: DrawingAPI)
    extends Shape(drawingAPI: DrawingAPI) {

    def draw() = drawingAPI.drawCircle(x, y, radius)

    def resizePercentage(pct: Double) { radius *= pct }
}

object BridgePattern {
    def main(args: Array[String]) {
        Seq (
            new CircleShape(1, 3, 5, new DrawingAPI1),
            new CircleShape(4, 5, 6, new DrawingAPI2)
        ) foreach { x =>
            x.resizePercentage(3)
            x.draw()
        }
    }
}

```

Python

```

"""
Bridge pattern example.
"""
from abc import ABCMeta, abstractmethod
from typing import NoReturn

NOT_IMPLEMENTED: str = "You should implement this."

class DrawingAPI:
    __metaclass__ = ABCMeta

    @abstractmethod
    def draw_circle(self, x: float, y: float, radius: float) -> NoReturn:

```

```

        raise NotImplementedError(NOT_IMPLEMENTED)

class DrawingAPI1(DrawingAPI):
    def draw_circle(self, x: float, y: float, radius: float) -> str:
        return f"API1.circle at {x}:{y} - radius: {radius}"

class DrawingAPI2(DrawingAPI):
    def draw_circle(self, x: float, y: float, radius: float) -> str:
        return f"API2.circle at {x}:{y} - radius: {radius}"

class DrawingAPI3(DrawingAPI):
    def draw_circle(self, x: float, y: float, radius: float) -> str:
        return f"API3.circle at {x}:{y} - radius: {radius}"

class Shape:
    __metaclass__ = ABCMeta

    drawing_api: DrawingAPI = None
    def __init__(self, drawing_api: DrawingAPI) -> None:
        self.drawing_api = drawing_api

    @abstractmethod
    def draw(self) -> NoReturn:
        raise NotImplementedError(NOT_IMPLEMENTED)

    @abstractmethod
    def resize_by_percentage(self, percent: float) -> NoReturn:
        raise NotImplementedError(NOT_IMPLEMENTED)

class CircleShape(Shape):
    def __init__(self, x: float, y: float, radius: float, drawing_api: DrawingAPI):
        self.x = x
        self.y = y
        self.radius = radius
        super(CircleShape, self).__init__(drawing_api)

    def draw(self) -> str:
        return self.drawing_api.draw_circle(self.x, self.y, self.radius)

    def resize_by_percentage(self, percent: float) -> None:
        self.radius *= 1 + percent / 100

class BridgePattern:
    @staticmethod
    def test() -> None:
        shapes: list[CircleShape] = [
            CircleShape(1.0, 2.0, 3.0, DrawingAPI1()),
            CircleShape(5.0, 7.0, 11.0, DrawingAPI2()),
            CircleShape(5.0, 4.0, 12.0, DrawingAPI3()),
        ]

        for shape in shapes:
            shape.resize_by_percentage(2.5)
            print(shape.draw())

if __name__ == "__main__":
    BridgePattern.test()

```

See also

- Adapter pattern
- Strategy pattern
- Template method pattern

References

- Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (<https://archive.org/details/designpatternsel00gamm/>). Addison-Wesley. p. 151 (<https://archive.org/details/designpatternsel00gamm/page/151>). ISBN 0-201-63361-2.
- "The Bridge design pattern - Problem, Solution, and Applicability" (<http://w3sdesign.com/?gr=s02&ugr=proble>). *w3sDesign.com*. Retrieved 2017-08-12.
- "The Bridge design pattern - Structure and Collaboration" (<http://w3sdesign.com/?gr=s02&ugr=struct>). *w3sDesign.com*. Retrieved 2017-08-12.

External links

- Bridge in UML and in LePUS3 (<https://web.archive.org/web/20090426061003/http://www.lepus.org.uk/ref/companion/Bridge.xml>) (a formal modelling language)
- C# Design Patterns: The Bridge Pattern (<http://www.informit.com/articles/article.aspx?p=30297>) From: James W. Cooper (2003).

C# Design Patterns: A Tutorial (<http://www.informit.com/store/product.aspx?isbn=0-201-84453-2>). Addison-Wesley. ISBN 0-201-84453-2.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Bridge_pattern&oldid=1318863115"

Composite pattern

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

Overview

The Composite^[2] design pattern is one of the twenty-three well-known *GoF design patterns* that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

Problems the Composite design pattern can solve

- Represent a part-whole hierarchy so that clients can treat part and whole objects uniformly.
- Represent a part-whole hierarchy as tree structure.

When defining (1) Part objects and (2) Whole objects that act as containers for Part objects, clients must treat them separately, which complicates client code.^[3]

Solutions the Composite design pattern describes

- Define a unified Component interface for part (Leaf) objects and whole (Composite) objects.
- Individual Leaf objects implement the Component interface directly, and Composite objects forward requests to their child components.

This enables clients to work through the Component interface to treat Leaf and Composite objects uniformly: Leaf objects perform a request directly, and Composite objects forward the request to their child components recursively downwards the tree structure. This makes client classes easier to implement, change, test, and reuse.

See also the UML class and object diagram below.

Motivation

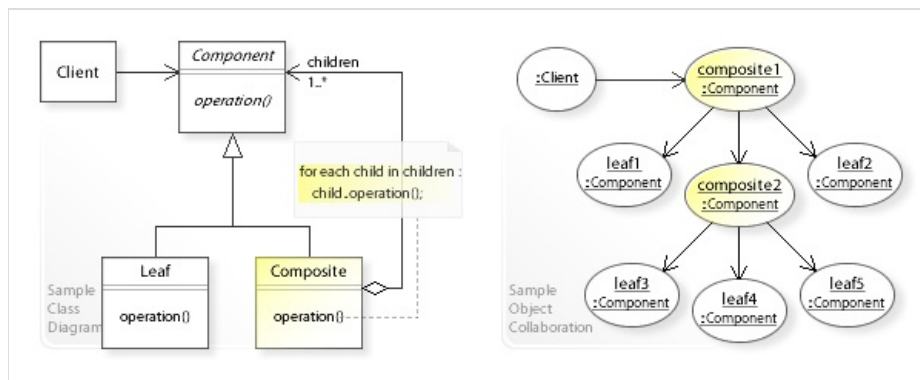
When dealing with Tree-structured data, programmers often have to discriminate between a leaf-node and a branch. This makes code more complex, and therefore, more error prone. The solution is an interface that allows treating complex and primitive objects uniformly. In object-oriented programming, a composite is an object designed as a composition of one-or-more similar objects, all exhibiting similar functionality. This is known as a "has-a" relationship between objects.^[4] The key concept is that you can manipulate a single instance of the object just as you would manipulate a group of them. The operations you can perform on all the composite objects often have a least common denominator relationship. For example, if defining a system to portray grouped shapes on a screen, it would be useful to define resizing a group of shapes to have the same effect (in some sense) as resizing a single shape.

When to use

Composite should be used when clients 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

UML class and object diagram



A sample UML class and object diagram for the Composite design pattern. [5]

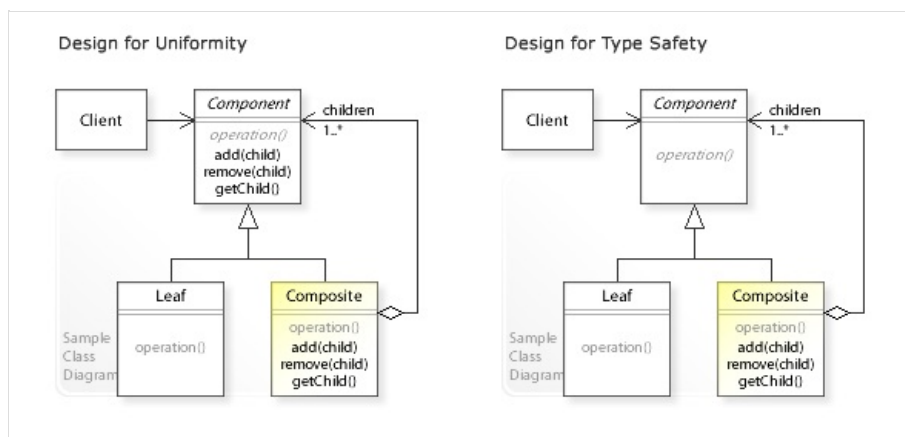
In the above UML class diagram, the Client class doesn't refer to the Leaf and Composite classes directly (separately). Instead, the Client refers to the common Component interface and can treat Leaf and Composite uniformly.

The Leaf class has no children and implements the Component interface directly.

The Composite class maintains a container of child Component objects (children) and forwards requests to these children (for each child in children: child.operation()).

The object collaboration diagram shows the run-time interactions: In this example, the Client object sends a request to the top-level Composite object (of type Component) in the tree structure. The request is forwarded to (performed on) all child Component objects (Leaf and Composite objects) downwards the tree structure.

Defining Child-Related Operations



Defining child-related operations in the Composite design pattern. [6]

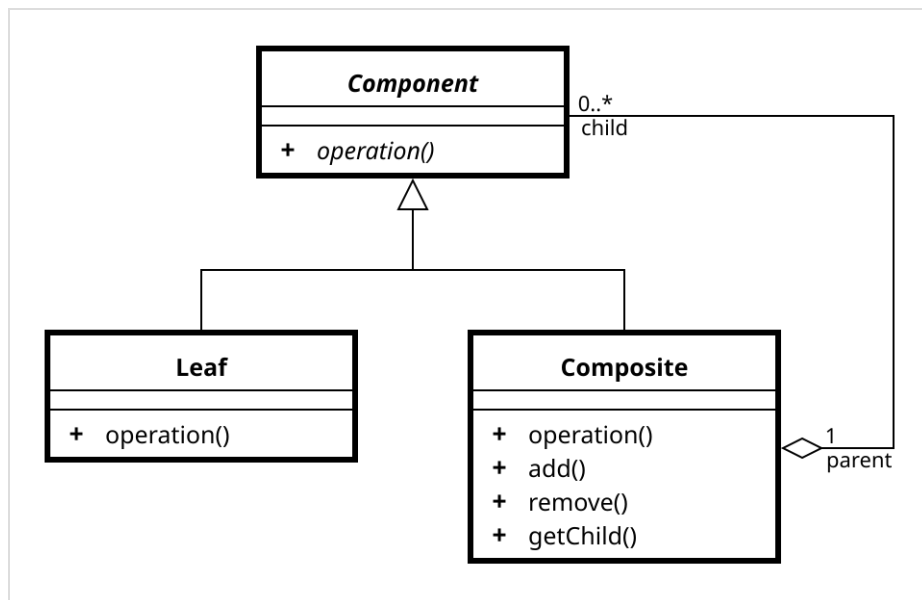
There are two design variants for defining and implementing child-related operations like adding/removing a child component to/from the container (add(child)/remove(child)) and accessing a child component (getChild()):

- *Design for transparency*: Child-related operations are defined in the Component interface. This enables clients to treat Leaf and Composite objects uniformly. But type safety is lost because clients can perform child-related operations on Leaf objects.
- *Design for type safety*: Child-related operations are defined only in the Composite class. Clients must treat Leaf and Composite objects differently. But type safety is gained because clients *cannot* perform child-related operations on Leaf objects.

The GoF authors present a variant of the Composite design pattern that emphasizes *transparency* over *type safety* and discuss the tradeoffs of the two approaches. [1]

The type-safe approach is particularly palatable if the composite structure is fixed post construction: the construction code does not require transparency because it needs to know the types involved in order to construct the composite. If downstream, the code does not need to modify the structure, then the child manipulation operations do not need to be present on the Component interface.

UML class diagram



Composite pattern in UML.

Component

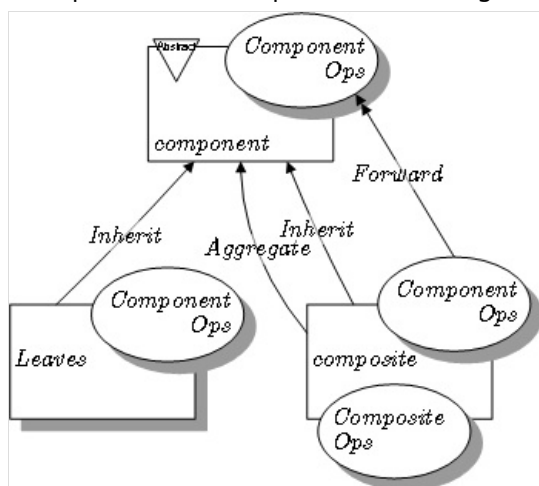
- is the abstraction for all components, including composite ones
- declares the interface for objects in the composition
- (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate

Leaf

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

Composite

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



Composite pattern in LePUS3.

Variation

As it is described in [Design Patterns](#), the pattern also involves including the child-manipulation methods in the main Component interface, not just the Composite subclass. More recent descriptions sometimes omit these methods.^[7]

Example

This C++23 implementation is based on the pre C++98 implementation in the book.

```

import std;

using std::runtime_error;
using std::shared_ptr;
using std::string;
using std::unique_ptr;
using std::vector;

// Component object
// declares the interface for objects in the composition.
class Equipment {
private:
    string name;
    double netPrice;
protected:
    Equipment() = default;

    explicit Equipment(const string& name):
        name{name}, netPrice{0} {}
public:
    // implements default behavior for the interface common to all classes, as appropriate.
    [[nodiscard]]
    virtual const string& getName() const noexcept {
        return name;
    }

    virtual void setName(const string& name) noexcept {
        this->name = name;
    }

    [[nodiscard]]
    virtual double getNetPrice() const noexcept {
        return netPrice;
    }

    virtual void setNetPrice(double netPrice) noexcept {
        this->netPrice = netPrice;
    }

    // declares an interface for accessing and managing its child components.
    virtual void add(shared_ptr<Equipment>) = 0;
    virtual void remove(shared_ptr<Equipment>) = 0;
    virtual ~Equipment() = default;
};

// Composite object
// defines behavior for components having children.
class CompositeEquipment: public Equipment {
private:
    // stores child components.
    using EquipmentList = vector<shared_ptr<Equipment>>;
    EquipmentList equipments;
protected:
    CompositeEquipment() = default;

    explicit CompositeEquipment(const string& name):
        Equipment(name), equipments{EquipmentList{}} {}
public:
    // implements child-related operations in the Component interface.
    [[nodiscard]]
    virtual double getNetPrice() const noexcept override {
        double total = Equipment::getNetPrice();
        for (const Equipment& i: equipments) {
            total += i->getNetPrice();
        }
        return total;
    }

    virtual void add(shared_ptr<Equipment> equipment) override {
        equipments.push_back(equipment.get());
    }

    virtual void remove(shared_ptr<Equipment> equipment) override {
        equipments.remove(equipment.get());
    }
};

// Leaf object
// represents leaf objects in the composition.
class FloppyDisk: public Equipment {
public:
    explicit FloppyDisk(const String& name):
        Equipment(name) {}

    // A leaf has no children.
    void add(shared_ptr<Equipment>) override {
        throw runtime_error("FloppyDisk::add() cannot be called!");
    }

    void remove(shared_ptr<Equipment>) override {
        throw runtime_error("FloppyDisk::remove() cannot be called!");
    }
};

class Chassis: public CompositeEquipment {
public:
    explicit Chassis(const string& name):
        CompositeEquipment(name) {}
};

int main() {
    shared_ptr<FloppyDisk> fd1 = std::make_shared<FloppyDisk>("3.5in Floppy");
    fd1->setNetPrice(19.99);
    std::println("{}: netPrice = {}", fd1->getName(), fd1->getNetPrice());

    shared_ptr<FloppyDisk> fd2 = std::make_shared<FloppyDisk>("5.25in Floppy");
    fd2->setNetPrice(29.99);
    std::println("{}: netPrice = {}", fd2->getName(), fd2->getNetPrice());
}

```



```

unique_ptr<Chassis> ch = std::make_unique<Chassis>("PC Chassis");
ch->setNetPrice(39.99);
ch->add(fd1);
ch->add(fd2);
std::println("{}: netPrice = {}", ch->getName(), ch->getNetPrice());

fd2->add(fd1);
}

```

The program output is

```

3.5in Floppy: netPrice=19.99
5.25in Floppy: netPrice=29.99
PC Chassis: netPrice=89.97
terminate called after throwing an instance of 'std::runtime_error'
what(): FloppyDisk::add

```

See also

- Perl Design Patterns Book
- Mixin
- Law of Demeter

References

- Gamma, Erich; Richard Helm; Ralph Johnson; John M. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software* (<https://archive.org/details/designpatternsel00gamm/page/395>). Addison-Wesley. pp. 395 (<https://archive.org/details/designpatternsel00gamm/page/395>). ISBN 0-201-63361-2.
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (<https://archive.org/details/designpatternsel00gamm/page/163>). Addison Wesley. pp. 163ff (<https://archive.org/details/designpatternsel00gamm/page/163>). ISBN 0-201-63361-2.
- "The Composite design pattern - Problem, Solution, and Applicability" (<http://w3sdesign.com/?gr=s03&ugr=proble>). *w3sDesign.com*. Retrieved 2017-08-12.
- Scott Walters (2004). *Perl Design Patterns Book* (<https://web.archive.org/web/20160308182256/http://perldesignpatterns.com/?CompositePattern>). Archived from the original (<http://perldesignpatterns.com/?CompositePattern>) on 2016-03-08. Retrieved 2010-01-18.
- "The Composite design pattern - Structure and Collaboration" (<http://w3sdesign.com/?gr=s03&ugr=struct>). *w3sDesign.com*. Retrieved 2017-08-12.
- "The Composite design pattern - Implementation" (<http://w3sdesign.com/?gr=s03&ugr=implem>). *w3sDesign.com*. Retrieved 2017-08-12.
- Geary, David (13 September 2002). "A look at the Composite design pattern" (<https://www.infoworld.com/article/2074564/a-look-at-the-composite-design-pattern.html>). Java Design Patterns. *JavaWorld*. Retrieved 2020-07-20.

External links

- Composite Pattern (<http://designpattern.co.il/Composite.html>) implementation in Java
- Composite pattern description from the Portland Pattern Repository
- Composite pattern in UML and in LePUS3, a formal modelling language (<https://web.archive.org/web/20151002170520/http://www.lepus.org.uk/ref/companion/Composite.xml>)
- Class::Delegation on CPAN (<https://search.cpan.org/dist/Class-Delegation/lib/Class/Delegation.pm>)
- "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 (<https://perfectjpattern.sourceforge.net/dp-composite.html>), Provides componentized implementation of the Composite Pattern in Java
- [1] (<https://web.archive.org/web/20090531030742/http://www.theresearchkitchen.com/blog/archives/57>) A persistent Java-based implementation
- Composite Design Pattern (http://sourcemaking.com/design_patterns/composite)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Composite_pattern&oldid=1317426294"

Decorator pattern

In object-oriented programming, the **decorator pattern** is a design pattern that allows behavior to be added to an individual object, dynamically, without affecting the behavior of other instances of the same class.^[1] The decorator pattern is often useful for adhering to the Single Responsibility Principle, as it allows functionality to be divided between classes with unique areas of concern^[2] as well as to the Open-Closed Principle, by allowing the functionality of a class to be extended without being modified.^[3] Decorator use can be more efficient than subclassing, because an object's behavior can be augmented without defining an entirely new object.

Overview

The *decorator*^[4] design pattern is one of the twenty-three *Gang-of-Four design patterns*; these describe how to solve recurring design problems and design flexible and reusable object-oriented software—that is, objects which are easier to implement, change, test, and reuse.

The decorator pattern provides a flexible alternative to subclassing for extending functionality. When using subclassing, different subclasses extend a class in different ways. However, an extension is bound to the class at compile-time and can't be changed at run-time. The decorator pattern allows responsibilities to be added (and removed from) an object dynamically at run-time. It is achieved by defining Decorator objects that

- implement the interface of the extended (decorated) object (Component) transparently by forwarding all requests to it.
- perform additional functionality before or after forwarding a request.

This allows working with different Decorator objects to extend the functionality of an object dynamically at run-time.^[5]

Intent

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

1. Subclass the original *Component* class into a *Decorator* class (see UML diagram);
2. In the *Decorator* class, add a *Component* pointer as a field;
3. In the *Decorator* class, pass a *Component* to the *Decorator* constructor to initialize the *Component* pointer;
4. In the *Decorator* class, forward all *Component* methods to the *Component* pointer; and
5. In the *ConcreteDecorator* class, override any *Component* method(s) whose behavior needs to be modified.

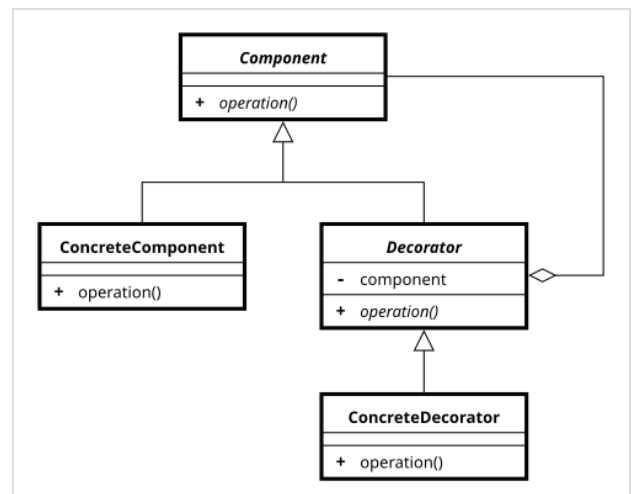
This pattern is designed so that multiple decorators can be stacked on top of each other, each time adding a new functionality to the overridden method(s).

Note that decorators and the original class object share a common set of features. In the previous diagram, the `operation()` method was available in both the decorated and undecorated versions.

The decoration features (e.g., methods, properties, or other members) are usually defined by an interface, mixin (a.k.a. trait) or class inheritance which is shared by the decorators and the decorated object. In the previous example, the class *Component* is inherited by both the *ConcreteComponent* and the subclasses that descend from *Decorator*.

The decorator pattern is an alternative to subclassing. Subclassing adds behavior at compile time, and the change affects all instances of the original class; decorating can provide new behavior at run-time for selected objects.^[5]

This difference becomes most important when there are several *independent* ways of extending functionality. In some object-oriented programming languages, classes cannot be created at runtime, and it is typically not possible to predict, at design time, what combinations of extensions will be needed. This would mean that a new class would have to be made for every possible combination. By contrast, decorators are objects, created at runtime, and can be combined on a per-use basis. The I/O Streams implementations of both Java and the .NET Framework incorporate the decorator pattern.^[5]



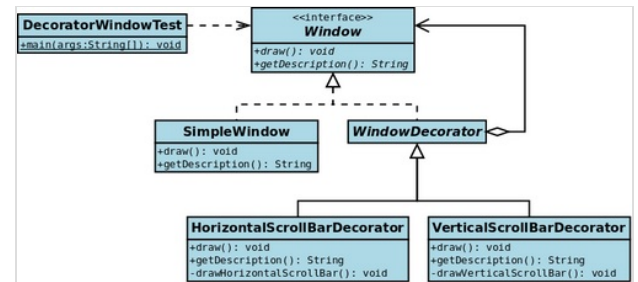
Decorator UML class diagram

Motivation

As an example, consider a window in a windowing system. To allow scrolling of the window's contents, one may wish to add horizontal or vertical scrollbars to it, as appropriate. Assume windows are represented by instances of the *Window* interface, and assume this class has no functionality for adding scrollbars. One could create a subclass *ScrollingWindow* that provides them, or create a *ScrollingWindowDecorator* that adds this functionality to existing *Window* objects. At this point, either solution would be fine.

Now, assume one also desires the ability to add borders to windows. Again, the original *Window* class has no support. The *ScrollingWindow* subclass now poses a problem, because it has effectively created a new kind of window. If one wishes to add border support to many but not *all* windows, one must create subclasses *WindowWithBorder* and *ScrollingWindowWithBorder*, etc. This problem gets worse with every new feature or window subtype to be added. For the decorator solution, a new *BorderedWindowDecorator* is created. Any combination of *ScrollingWindowDecorator* or *BorderedWindowDecorator* can decorate existing windows. If the functionality needs to be added to all *Windows*, the base class can be modified. On the other hand, sometimes (e.g., using external frameworks) it is not possible, legal, or convenient to modify the base class.

In the previous example, the *SimpleWindow* and *WindowDecorator* classes implement the *Window* interface, which defines the *draw()* method and the *getDescription()* method that are required in this scenario, in order to decorate a window control.



UML diagram for the window example

Common usecases

Applying decorators

Adding or removing decorators on command (like a button press) is a common UI pattern, often implemented along with the Command design pattern. For example, a text editing application might have a button to highlight text. On button press, the individual text glyphs currently selected will all be wrapped in decorators that modify their `draw()` function, causing them to be drawn in a highlighted manner (a real implementation would probably also use a demarcation system to maximize efficiency).

Applying or removing decorators based on changes in state is another common use case. Depending on the scope of the state, decorators can be applied or removed in bulk. Similarly, the State design pattern can be implemented using decorators instead of subclassed objects encapsulating the changing functionality. The use of decorators in this manner makes the State object's internal state and functionality more compositional and capable of handling arbitrary complexity.

Usage in Flyweight objects

Decoration is also often used in the Flyweight design pattern. Flyweight objects are divided into two components: an invariant component that is shared between all flyweight objects; and a variant, decorated component that may be partially shared or completely unshared. This partitioning of the flyweight object is intended to reduce memory consumption. The decorators are typically cached and reused as well. The decorators will all contain a common reference to the shared, invariant object. If the decorated state is only partially variant, then the decorators can also be shared to some degree - though care must be taken not to alter their state while they're being used. iOS's `UITableView` implements the flyweight pattern in this manner - a tableview's reusable cells are decorators that contains a references to a common tableview row object, and the cells are cached / reused.

Obstacles of interfacing with decorators

Applying combinations of decorators in diverse ways to a collection of objects introduces some problems interfacing with the collection in a way that takes full advantage of the functionality added by the decorators. The use of an Adapter or Visitor patterns can be useful in such cases. Interfacing with multiple layers of decorators poses additional challenges and logic of Adapters and Visitors must be designed to account for that.

Architectural relevance

Decorators support a compositional rather than a top-down, hierarchical approach to extending functionality. A decorator makes it possible to add or alter behavior of an interface at run-time. They can be used to wrap objects in a multilayered, arbitrary combination of ways. Doing the same with subclasses means implementing complex networks of multiple inheritance, which is memory-inefficient and at a certain point just cannot scale. Likewise, attempting to implement the same functionality with properties bloats each instance of the object with unnecessary properties. For the above reasons decorators are often considered a memory-efficient alternative to subclassing.

Decorators can also be used to specialize objects which are not subclassable, whose characteristics need to be altered at runtime (as mentioned elsewhere), or generally objects that are lacking in some needed functionality.

Usage in enhancing APIs

The decorator pattern also can augment the Facade pattern. A facade is designed to simply interface with the complex system it encapsulates, but it does not add functionality to the system. However, the wrapping of a complex system provides a space that may be used to introduce new functionality based on the coordination of subcomponents in the system. For example, a facade pattern may unify many different languages dictionaries under one multi-language dictionary interface. The new interface may also provide new functions for translating words between languages. This is a hybrid pattern - the unified interface provides a space for augmentation. Think of decorators as not being limited to wrapping individual objects, but capable of wrapping clusters of objects in this hybrid approach as well.

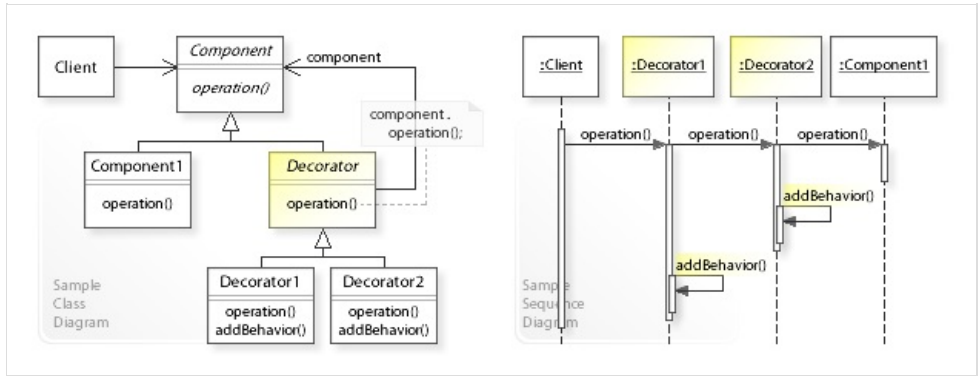
Alternatives to decorators

As an alternative to the decorator pattern, the adapter can be used when the wrapper must respect a particular interface and must support polymorphic behavior, and the Facade when an easier or simpler interface to an underlying object is desired.^[6]

Pattern	Intent
<u>Adapter</u>	Converts one interface to another so that it matches what the client is expecting
<u>Decorator</u>	Dynamically adds responsibility to the interface by wrapping the original code
<u>Facade</u>	Provides a simplified interface

Structure

UML class and sequence diagram



A sample UML class and sequence diagram for the Decorator design pattern. ^[7]

In the above UML class diagram, the abstract Decorator class maintains a reference (component) to the decorated object (Component) and forwards all requests to it (component.operation()). This makes Decorator transparent (invisible) to clients of Component.

Subclasses (Decorator1, Decorator2) implement additional behavior (addBehavior()) that should be added to the Component (before/after forwarding a request to it).

The sequence diagram shows the run-time interactions: The Client object works through Decorator1 and Decorator2 objects to extend the functionality of a Component1 object.

The Client calls operation() on Decorator1, which forwards the request to Decorator2. Decorator2 performs addBehavior() after forwarding the request to Component1 and returns to Decorator1, which performs addBehavior() and returns to the Client.

Examples

C++

This implementation (which uses C++23 features) is based on the pre C++98 implementation in the book.

```
import std;
using std::unique_ptr;

// Beverage interface.
class Beverage {
public:
    virtual void drink() = 0;
    virtual ~Beverage() = default;
};
```

```

// Drinks which can be decorated.
class Coffee: public Beverage {
public:
    virtual void drink() override {
        std::print("Drinking Coffee");
    }
};

class Soda: public Beverage {
public:
    virtual void drink() override {
        std::print("Drinking Soda");
    }
};

class BeverageDecorator: public Beverage {
private:
    unique_ptr<Beverage> component;
protected:
    void callComponentDrink() {
        if (component) {
            component->drink();
        }
    }
public:
    BeverageDecorator() = delete;

    explicit BeverageDecorator(unique_ptr<Beverage> component):
        component{std::move(component)} {}

    virtual void drink() = 0;
};

class Milk: public BeverageDecorator {
private:
    float percentage;
public:
    Milk(unique_ptr<Beverage> component, float percentage):
        BeverageDecorator(std::move(component)), percentage{percentage} {}

    virtual void drink() override {
        callComponentDrink();
        std::print(", with milk of richness {}%", percentage);
    }
};

class IceCubes: public BeverageDecorator {
private:
    int count;
public:
    IceCubes(unique_ptr<Beverage> component, int count):
        BeverageDecorator(std::move(component)), count{count} {}

    virtual void drink() override {
        callComponentDrink();
        std::print(", with {} ice cubes", count);
    }
};

class Sugar: public BeverageDecorator {
private:
    int spoons = 1;
public:
    Sugar(unique_ptr<Beverage> component, int spoons):
        BeverageDecorator(std::move(component)), spoons{spoons} {}

    virtual void drink() override {
        callComponentDrink();
        std::print(", with {} spoons of sugar", spoons);
    }
};

int main(int argc, char* argv[]) {
    unique_ptr<Beverage> soda = std::make_unique<Soda>();
    soda = std::make_unique<IceCubes>(std::move(soda), 3);
    soda = std::make_unique<Sugar>(std::move(soda), 1);

    soda->drink();
    std::println();

    unique_ptr<Beverage> coffee = std::make_unique<Coffee>();
    coffee = std::make_unique<IceCubes>(std::move(coffee), 16);
    coffee = std::make_unique<Milk>(std::move(coffee), 3.);
    coffee = std::make_unique<Sugar>(std::move(coffee), 2);

    coffee->drink();

    return 0;
}

```

The program output is like

```

Drinking Soda, with 3 ice cubes, with 1 spoons of sugar
Drinking Coffee, with 16 ice cubes, with milk of richness 3%, with 2 spoons of sugar

```

Full example can be tested on a [godbolt page \(https://godbolt.org/z/s848nWozP\)](https://godbolt.org/z/s848nWozP).

C++

Two options are presented here: first, a dynamic, runtime-composable decorator (has issues with calling decorated functions unless proxied explicitly) and a decorator that uses mixin inheritance.

Dynamic decorator

```
import std;

using std::string;

class Shape {
public:
    virtual ~Shape() = default;
    virtual string getName() const = 0;
};

class Circle: public Shape {
private:
    float radius = 10.0f;
public:
    void resize(float factor) noexcept {
        radius *= factor;
    }

    [[nodiscard]]
    string getName() const override {
        return std::format("A circle of radius {}", radius);
    }
};

class ColoredShape: public Shape {
private:
    string color;
    Shape& shape;
public:
    ColoredShape(const string& color, Shape& shape):
        color{color}, shape{shape} {}

    [[nodiscard]]
    string getName() const override {
        return std::format("{} which is colored {}", shape.getName(), color);
    }
};

int main() {
    Circle circle;
    ColoredShape coloredShape{"red", circle};
    std::println("{} ", coloredShape.getName());
}
```

```
import std;

using std::string;
using std::unique_ptr;

class WebPage {
public:
    virtual void display() = 0;
    virtual ~WebPage() = default;
};

class BasicWebPage: public WebPage {
private:
    string html;
public:
    void display() override {
        std::println("Basic WEB page");
    }
};

class WebPageDecorator: public WebPage {
private:
    unique_ptr<WebPage> webPage;
public:
    explicit WebPageDecorator(unique_ptr<WebPage> webPage):
        webPage{std::move(webPage)} {}

    void display() override {
        webPage->display();
    }
};

class AuthenticatedWebPage: public WebPageDecorator {
public:
    explicit AuthenticatedWebPage(unique_ptr<WebPage> webPage):
        WebPageDecorator(std::move(webPage)) {}

    void authenticateUser() {
        std::println("authentication done");
    }

    void display() override {
        authenticateUser();
        WebPageDecorator::display();
    }
};

class AuthorizedWebPage: public WebPageDecorator {
public:
    explicit AuthorizedWebPage(unique_ptr<WebPage> webPage):
        WebPageDecorator(std::move(webPage)) {}

    void authorizedUser() {
        std::println("authorized done");
    }

    void display() override {
        authorizedUser();
        WebPageDecorator::display();
    }
};
```

```

    }
};

int main(int argc, char* argv[]) {
    unique_ptr<WebPage> myPage = std::make_unique<BasicWebPage>();

    myPage = std::make_unique<AuthorizedWebPage>(std::move(myPage));
    myPage = std::make_unique<AuthenticatedWebPage>(std::move(myPage));
    myPage->display();
    std::println();
    return 0;
}

```

Static decorator (mixin inheritance)

This example demonstrates a static Decorator implementation, which is possible due to C++ ability to inherit from the template argument.

```

import std;

using std::string;

class Circle {
private:
    float radius = 10.0f;
public:
    void resize(float factor) noexcept {
        radius *= factor;
    }

    [[nodiscard]]
    string getName() const {
        return std::format("A circle of radius {}", radius);
    }
};

template <typename T>
class ColoredShape: public T {
private:
    string color;
public:
    explicit ColoredShape(const string& color):
        color{color} {}

    [[nodiscard]]
    string getName() const {
        return std::format("{} which is colored {}", T::getName(), color);
    }
};

int main() {
    ColoredShape<Circle> redCircle{"red"};
    std::println("{} ", redCircle.getName());
    redCircle.resize(1.5f);
    std::println("{} ", redCircle.getName());
}

```

Java

First example (window/scrolling scenario)

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

```

// The Window interface class
public interface Window {
    void draw(); // Draws the Window
    String getDescription(); // Returns a description of the Window
}

// Implementation of a simple Window without any scrollbars
class SimpleWindow implements Window {
    @Override
    public void draw() {
        // Draw window
    }

    @Override
    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 {
    private final Window windowToBeDecorated; // the Window being decorated

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

    @Override
    public void draw() {
        windowToBeDecorated.draw(); // Delegation
    }
}

```



```

    }

    @Override
    public String getDescription() {
        return windowToBeDecorated.getDescription(); // Delegation
    }
}

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

    @Override
    public void draw() {
        super.draw();
        drawVerticalScrollBar();
    }

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

    @Override
    public String getDescription() {
        return String.format("%s, including vertical scrollbars", super.getDescription());
    }
}

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

    @Override
    public void draw() {
        super.draw();
        drawHorizontalScrollBar();
    }

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

    @Override
    public String getDescription() {
        return String.format("%s, including horizontal scrollbars", super.getDescription());
    }
}
}

```

Here is 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.

Below is the JUnit test class for the Test Driven Development

```

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class WindowDecoratorTest {
    @Test
    public void testWindowDecoratorTest() {
        Window decoratedWindow = new HorizontalScrollBarDecorator(
            new VerticalScrollBarDecorator(new SimpleWindow())
        );
        // assert that the description indeed includes horizontal + vertical scrollbars
        assertEquals("simple window, including vertical scrollbars, including horizontal scrollbars", decoratedWindow.getDescription());
    }
}

```

Second example (coffee making scenario)

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

```

// The interface Coffee defines the functionality of Coffee implemented by decorator
public interface Coffee {
    public double getCost(); // Returns the cost of the coffee
    public String getIngredients(); // Returns the ingredients of the coffee
}

```



```
// Extension of a simple coffee without any extra ingredients
public class SimpleCoffee implements Coffee {
    @Override
    public double getCost() {
        return 1;
    }

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

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

```
// Abstract decorator class - note that it implements Coffee interface
public abstract class CoffeeDecorator implements Coffee {
    private final Coffee decoratedCoffee;

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

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

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

// Decorator WithMilk mixes milk into coffee.
// Note it extends CoffeeDecorator.
class WithMilk extends CoffeeDecorator {
    public WithMilk(Coffee c) {
        super(c);
    }

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

    @Override
    public String getIngredients() {
        return String.format("%s, Milk", super.getIngredients());
    }
}

// Decorator WithSprinkles mixes sprinkles onto coffee.
// Note it extends CoffeeDecorator.
class WithSprinkles extends CoffeeDecorator {
    public WithSprinkles(Coffee c) {
        super(c);
    }

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

    @Override
    public String getIngredients() {
        return String.format("%s, Sprinkles", super.getIngredients());
    }
}
```

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

```
public class Main {
    public static void printInfo(Coffee c) {
        System.out.printf("Cost: %s; Ingredients: %s\n", c.getCost(), c.getIngredients());
    }

    public static void main(String[] args) {
        Coffee c = new SimpleCoffee();
        printInfo(c);

        c = new WithMilk(c);
        printInfo(c);

        c = new WithSprinkles(c);
        printInfo(c);
    }
}
```

The output of this program is given below:

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

PHP

```
abstract class Component
{
    protected $data;
    protected $value;

    abstract public function getData();
    abstract public function getValue();
}

class ConcreteComponent extends Component
{
    public function __construct()
    {
        $this->value = 1000;
        $this->data = "Concrete Component:\t{$this->value}\n";
    }

    public function getData()
    {
        return $this->data;
    }

    public function getValue()
    {
        return $this->value;
    }
}

abstract class Decorator extends Component
{
}

class ConcreteDecorator1 extends Decorator
{
    public function __construct(Component $data)
    {
        $this->value = 500;
        $this->data = $data;
    }

    public function getData()
    {
        return $this->data->getData() . "Concrete Decorator 1:\t{$this->value}\n";
    }

    public function getValue()
    {
        return $this->value + $this->data->getValue();
    }
}

class ConcreteDecorator2 extends Decorator
{
    public function __construct(Component $data)
    {
        $this->value = 500;
        $this->data = $data;
    }

    public function getData()
    {
        return $this->data->getData() . "Concrete Decorator 2:\t{$this->value}\n";
    }

    public function getValue()
    {
        return $this->value + $this->data->getValue();
    }
}

class Client
{
    private $component;

    public function __construct()
    {
        $this->component = new ConcreteComponent();
        $this->component = $this->wrapComponent($this->component);

        echo $this->component->getData();
        echo "Client:\t\t\t";
        echo $this->component->getValue();
    }

    private function wrapComponent(Component $component)
    {
        $component1 = new ConcreteDecorator1($component);
        $component2 = new ConcreteDecorator2($component1);
        return $component2;
    }
}

$client = new Client();

// Result: #quanton81

//Concrete Component: 1000
//Concrete Decorator 1: 500
//Concrete Decorator 2: 500
//Client: 2000
```

Python

The following Python example, taken from [Python Wiki - DecoratorPattern](https://wiki.python.org/moin/DecoratorPattern) (<https://wiki.python.org/moin/DecoratorPattern>), shows us how to pipeline decorators to dynamically add many behaviors in an object:

```
"""
Demonstrated decorators in a world of a 10x10 grid of values 0-255.
"""

import random
from typing import Any

def s32_to_u16(x: int) -> int:
    sign: int
    if x < 0:
        sign = 0xF000
    else:
        sign = 0
    bottom: int = x & 0x00007FFF
    return bottom | sign

def seed_from_xy(x: int, y: int) -> int:
    return s32_to_u16(x) | (s32_to_u16(y) << 16)

class RandomSquare:
    def __init__(self, seed_modifier: int) -> None:
        self.seed_modifier: int = seed_modifier

    def get(self, x: int, y: int) -> int:
        seed: int = seed_from_xy(x, y) ^ self.seed_modifier
        random.seed(seed)
        return random.randint(0, 255)

class DataSquare:
    def __init__(self, initial_value: int = None) -> None:
        self.data: List[int] = [initial_value] * 10 * 10

    def get(self, x: int, y: int) -> int:
        return self.data[(y * 10) + x] # yes: these are all 10x10

    def set(self, x: int, y: int, u: int) -> None:
        self.data[(y * 10) + x] = u

class CacheDecorator:
    def __init__(self, decorated: Any) -> None:
        self.decorated: Any = decorated
        self.cache: DataSquare = DataSquare()

    def get(self, x: int, y: int) -> int:
        if self.cache.get(x, y) == None:
            self.cache.set(x, y, self.decorated.get(x, y))
        return self.cache.get(x, y)

class MaxDecorator:
    def __init__(self, decorated: Any, max: int) -> None:
        self.decorated: Any = decorated
        self.max: int = max

    def get(self, x: int, y: int) -> None:
        if self.decorated.get(x, y) > self.max:
            return self.max
        return self.decorated.get(x, y)

class MinDecorator:
    def __init__(self, decorated: Any, min: int) -> None:
        self.decorated: Any = decorated
        self.min: int = min

    def get(self, x: int, y: int) -> int:
        if self.decorated.get(x, y) < self.min:
            return self.min
        return self.decorated.get(x, y)

class VisibilityDecorator:
    def __init__(self, decorated: Any) -> None:
        self.decorated: Any = decorated

    def get(self, x: int, y: int) -> int:
        return self.decorated.get(x, y)

    def draw(self) -> None:
        for y in range(10):
            for x in range(10):
                print("%3d" % self.get(x, y), end=' ')
            print()

if __name__ == "__main__":
    # Now, build up a pipeline of decorators:

    random_square: RandomSquare = RandomSquare(635)
    random_cache: CacheDecorator = CacheDecorator(random_square)
    max_filtered: MaxDecorator = MaxDecorator(random_cache, 200)
    min_filtered: MinDecorator = MinDecorator(max_filtered, 100)
    final: VisibilityDecorator = VisibilityDecorator(min_filtered)

    final.draw()
```

Note:

The Decorator Pattern (or an implementation of this design pattern in Python - as the above example) should not be confused with Python Decorators, a language feature of Python. They are different things.

Second to the Python Wiki:

The Decorator Pattern is a pattern described in the Design Patterns Book. It is a way of apparently modifying an object's behavior, by enclosing it inside a decorating object with a similar interface. This is not to be confused with Python Decorators, which is a language feature for dynamically modifying a function or class.^[8]

Crystal

```
abstract class Coffee
  abstract def cost
  abstract def ingredients
end

# Extension of a simple coffee
class SimpleCoffee < Coffee
  def cost
    1.0
  end

  def ingredients
    "Coffee"
  end
end

# Abstract decorator
class CoffeeDecorator < Coffee
  protected getter decorated_coffee : Coffee

  def initialize(@decorated_coffee)
  end

  def cost
    decorated_coffee.cost
  end

  def ingredients
    decorated_coffee.ingredients
  end
end

class WithMilk < CoffeeDecorator
  def cost
    super + 0.5
  end

  def ingredients
    super + ", Milk"
  end
end

class WithSprinkles < CoffeeDecorator
  def cost
    super + 0.2
  end

  def ingredients
    super + ", Sprinkles"
  end
end

class Program
  def print(coffee : Coffee)
    puts "Cost: #{coffee.cost}; Ingredients: #{coffee.ingredients}"
  end

  def initialize
    coffee = SimpleCoffee.new
    print(coffee)

    coffee = WithMilk.new(coffee)
    print(coffee)

    coffee = WithSprinkles.new(coffee)
    print(coffee)
  end
end

Program.new
```

Output:

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

C#

```
namespace DecoratorExample;

interface IBike
{
    string GetDetails();
    double GetPrice();
}
```

```

class AluminiumBike : IBike
{
    public double GetPrice() => 100.0;

    public string GetDetails() => "Aluminium Bike";
}

class CarbonBike : IBike
{
    public double GetPrice() => 1000.0;

    public string GetDetails() => "Carbon";
}

abstract class BikeAccessories : IBike
{
    private readonly IBike _bike;

    public BikeAccessories(IBike bike)
    {
        _bike = bike;
    }

    public virtual double GetPrice() => _bike.GetPrice();

    public virtual string GetDetails() => _bike.GetDetails();
}

class SecurityPackage : BikeAccessories
{
    public SecurityPackage(IBike bike) : base(bike)
    {
        // ...
    }

    public override string GetDetails() => $"{base.GetDetails()} + Security Package";

    public override double GetPrice() => base.GetPrice() + 1;
}

class SportPackage : BikeAccessories
{
    public SportPackage(IBike bike) : base(bike)
    {
        // ...
    }

    public override string GetDetails() => $"{base.GetDetails()} + Sport Package";

    public override double GetPrice() => base.GetPrice() + 10;
}

public class BikeShop
{
    public void UpgradeBike()
    {
        AluminiumBike basicBike = new AluminiumBike();
        BikeAccessories upgraded = new SportPackage(basicBike);
        upgraded = new SecurityPackage(upgraded);

        Console.WriteLine($"Bike: '{upgraded.GetDetails()}' Cost: {upgraded.GetPrice()}");
    }

    static void Main(string[] args)
    {
        UpgradeBike();
    }
}

```

Output:

```
Bike: 'Aluminium Bike + Sport Package + Security Package' Cost: 111
```

Ruby

```

class AbstractCoffee
  def print
    puts "Cost: #{cost}; Ingredients: #{ingredients}"
  end
end

class SimpleCoffee < AbstractCoffee
  def cost
    1.0
  end

  def ingredients
    "Coffee"
  end
end

class WithMilk < SimpleDelegator
  def cost
    __getobj__.cost + 0.5
  end

  def ingredients
    __getobj__.ingredients + ", Milk"
  end
end

```

```

class WithSprinkles < SimpleDelegator
  def cost
    __getobj__.cost + 0.2
  end

  def ingredients
    __getobj__.ingredients + ", Sprinkles"
  end
end

coffee = SimpleCoffee.new
coffee.print

coffee = WithMilk.new(coffee)
coffee.print

coffee = WithSprinkles.new(coffee)
coffee.print

```

Output:

```

Cost: 1.0; Ingredients: Coffee
Cost: 1.5; Ingredients: Coffee, Milk
Cost: 1.7; Ingredients: Coffee, Milk, Sprinkles

```

See also

- [Composite pattern](#)
- [Adapter pattern](#)
- [Abstract class](#)
- [Abstract factory](#)
- [Aspect-oriented programming](#)
- [Immutable object](#)

References

- Gamma, Erich; et al. (1995). *Design Patterns* (<https://archive.org/details/designpatternsel00gamm/page/175>). Reading, MA: Addison-Wesley Publishing Co, Inc. pp. 175ff (<https://archive.org/details/designpatternsel00gamm/page/175>). ISBN 0-201-63361-2.
- "How to Implement a Decorator Pattern" (<https://web.archive.org/web/20150707165957/http://sam-burns.co.uk/57/how-to-implement-a-decorator-pattern/>). Archived from the original on 2015-07-07.
- "The Decorator Pattern, Why We Stopped Using It, and the Alternative" (<https://betterprogramming.pub/the-decorator-pattern-why-i-stopped-using-it-and-the-alternative-2ae447f9de08>). 8 March 2022.
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (<https://archive.org/details/designpatternsel00gamm/page/175>). Addison Wesley. pp. 175ff (<https://archive.org/details/designpatternsel00gamm/page/175>). ISBN 0-201-63361-2.
- "The Decorator design pattern - Problem, Solution, and Applicability" (<https://web.archive.org/web/20230408204922/http://w3sdesign.com/?gr=s04&ugr=proble#gf>). *w3sDesign*. Archived from the original (<http://w3sdesign.com/?gr=s04&ugr=proble>) on 2023-04-08. Retrieved 2017-08-12.
- Freeman, Eric; Freeman, Elisabeth; Sierra, Kathy; Bates, Bert (2004). Hendrickson, Mike; Loukides, Mike (eds.). *Head First Design Patterns* (https://www.goodreads.com/book/show/58128.Head_First_Design_Patterns) (paperback). Vol. 1. O'Reilly. pp. 243, 252, 258, 260. ISBN 978-0-596-00712-6. Retrieved 2012-07-02.
- "The Decorator design pattern - Structure and Collaboration" (<http://w3sdesign.com/?gr=s04&ugr=struct>). *w3sDesign.com*. Retrieved 2017-08-12.
- "DecoratorPattern - Python Wiki" (<https://wiki.python.org/moin/DecoratorPattern>). *wiki.python.org*.

External links

- Decorator Pattern (<https://web.archive.org/web/20180418005440/http://designpattern.co.il/Decorator.html>) implementation in Java
- Decorator pattern description from the Portland Pattern Repository

Retrieved from "https://en.wikipedia.org/w/index.php?title=Decorator_pattern&oldid=1318863402"

Facade pattern

The **facade pattern** (also spelled *façade*) is a software design pattern commonly used in object-oriented programming. Analogous to a façade in architecture, it is an object that serves as a front-facing interface masking more complex underlying or structural code. A facade can:

- improve the readability and usability of a software library by masking interaction with more complex components behind a single (and often simplified) application programming interface (API)
- provide a context-specific interface to more generic functionality (complete with context-specific input validation)
- serve as a launching point for a broader refactor of monolithic or tightly-coupled systems in favor of more loosely-coupled code

Developers often use the facade design pattern when a system is very complex or difficult to understand because the system has many interdependent classes or because its source code is unavailable. This pattern hides the complexities of the larger system and provides a simpler interface to the client. It typically involves a single wrapper class that contains a set of members required by the client. These members access the system on behalf of the facade client and hide the implementation details.

Overview

The Facade ^[1] design pattern is one of the twenty-three well-known *GoF design patterns* that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

What problems can the Facade design pattern solve? ^[2]

- To make a complex subsystem easier to use, a simple interface should be provided for a set of interfaces in the subsystem.
- The dependencies on a subsystem should be minimized.

Clients that access a complex subsystem directly refer to (depend on) many different objects having different interfaces (tight coupling), which makes the clients hard to implement, change, test, and reuse.

What solution does the Facade design pattern describe?

Define a Facade object that

- implements a simple interface in terms of (by delegating to) the interfaces in the subsystem and
- may perform additional functionality before/after forwarding a request.

This enables to work through a Facade object to minimize the dependencies on a subsystem.
See also the UML class and sequence diagram below.

Usage

A Facade is used when an easier or simpler interface to an underlying object is desired.^[3] Alternatively, an adapter can be used when the wrapper must respect a particular interface and must support polymorphic behavior. A decorator makes it possible to add or alter behavior of an interface at run-time.

Pattern	Intent
<u>Adapter</u>	Converts one interface to another so that it matches what the client is expecting
<u>Decorator</u>	Dynamically adds responsibility to the interface by wrapping the original code
Facade	Provides a simplified interface

The facade pattern is typically used when

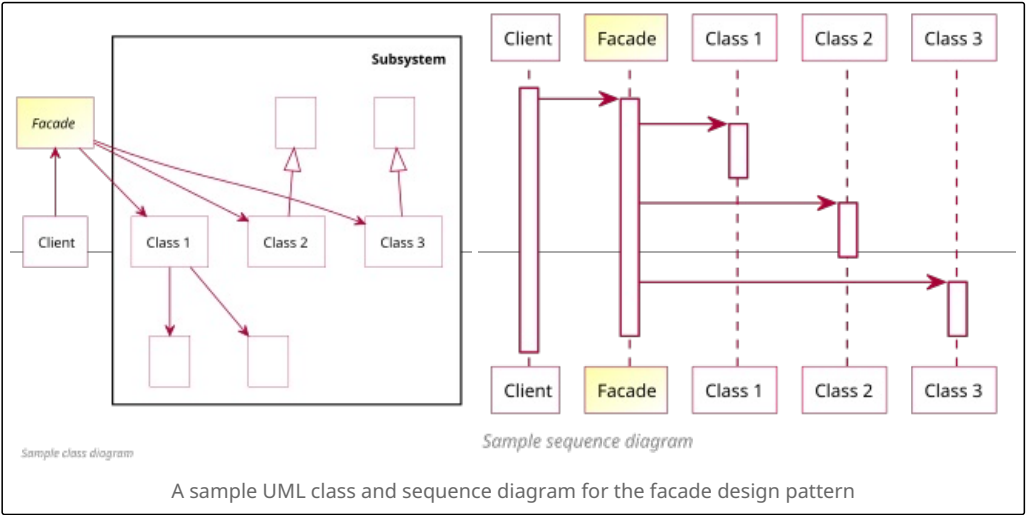
- a simple interface is required to access a complex system,
- a system is very complex or difficult to understand,
- an entry point is needed to each level of layered software, or
- the abstractions and implementations of a subsystem are tightly coupled.

Structure

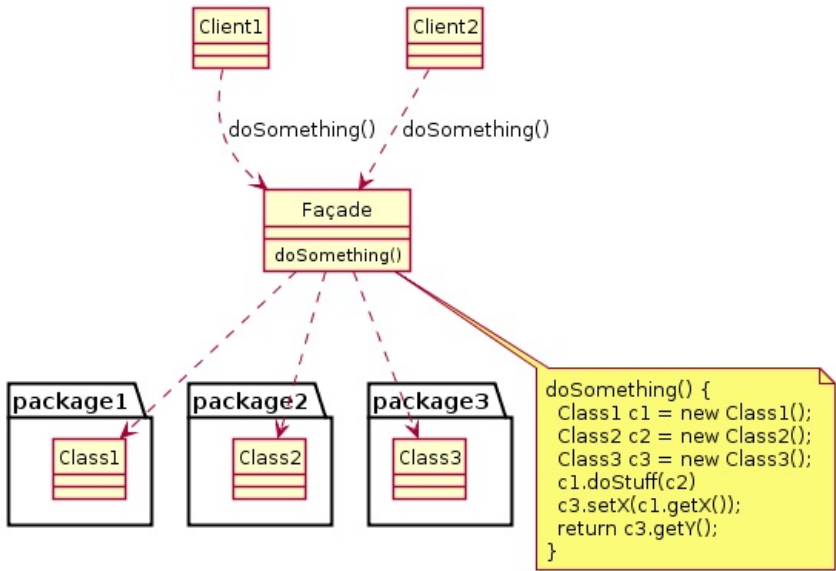
UML class and sequence diagram

In this UML class diagram, the Client class doesn't access the subsystem classes directly. Instead, the Client works through a Facade class that implements a simple interface in terms of (by delegating to) the subsystem classes (Class1, Class2, and Class3). The Client depends only on the simple Facade interface and is independent of the complex subsystem.^[4]

The sequence diagram shows the run-time interactions: The Client object works through a Facade object that delegates the request to the Class1, Class2, and Class3 instances that perform the request.



UML class diagram



Facade

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

Clients

The objects are using the facade pattern to access resources from the packages.

Example

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

C++

```
import std;
using std::string;
using std::string_view;

class CPU {
private:
    // ...
public:
    void freeze() {
        // ...
    }

    void jump(long position) {
        // ...
    }

    void execute() {
        // ...
    }
}
```



```

    };

    class HardDrive {
    private:
        // ...
    public:
        string read(long lba, int size) {
            // ...
        }
    };

    class Memory {
    private:
        // ...
    public:
        void load(long position, string_view data) {
            // ...
        }
    };

    class ComputerFacade {
    private:
        CPU cpu;
        Memory memory;
        HardDrive hardDrive;

        const long bootAddress;
        const long bootSector;
        const int sectorSize;
    public:
        ComputerFacade(long bootAddress, long bootSector, int sectorSize):
            bootAddress{bootAddress}, bootSector{bootSector}, sectorSize{sectorSize} {}

        void start() {
            cpu.freeze();
            memory.load(bootAddress, hardDrive.read(bootSector, sectorSize));
            cpu.jump(bootAddress);
            cpu.execute();
        }
    };

    int main(int argc, char* argv[]) {
        ComputerFacade computer(/* parameters here */);
        computer.start();
    }

```

See also

- Encapsulation (computer programming)

References

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (<https://archive.org/details/designpatternsel00gamm/page/185>). Addison Wesley. pp. 185ff (<https://archive.org/details/designpatternsel00gamm/page/185>). ISBN 0-201-63361-2.
- "The Facade design pattern - Problem, Solution, and Applicability" (<https://web.archive.org/web/20200612203152/http://w3sdesign.com/?gr=s05&ugr=proble>). *w3sDesign.com*. Archived from the original (<http://w3sdesign.com/?gr=s05&ugr=proble>) on 2020-06-12. Retrieved 2017-08-12.
- Freeman, Eric; Freeman, Elisabeth; Sierra, Kathy; Bates, Bert (2004). Hendrickson, Mike; Loukides, Mike (eds.). *Head First Design Patterns* (https://www.goodreads.com/book/show/58128.Head_First_Design_Patterns) (paperback). Vol. 1. O'Reilly. pp. 243, 252, 258, 260. ISBN 978-0-596-00712-6. Retrieved 2012-07-02.
- "The Facade design pattern - Structure and Collaboration" (<https://web.archive.org/web/20200612203156/http://w3sdesign.com/?gr=s05&ugr=struct>). *w3sDesign.com*. Archived from the original (<http://w3sdesign.com/?gr=s05&ugr=struct>) on 2020-06-12. Retrieved 2017-08-12.

External links

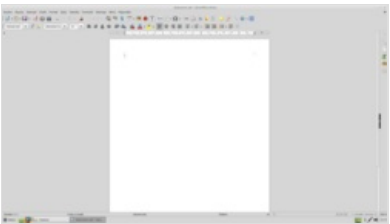
- Description from the Portland Pattern Repository (<http://c2.com/cgi/wiki?FacadePattern>)

Flyweight pattern

In computer programming, the flyweight software design pattern refers to an object that minimizes memory usage by sharing some of its data with other similar objects. The flyweight pattern is one of twenty-three well-known GoF design patterns. These patterns promote flexible object-oriented software design, which is easier to implement, change, test, and reuse.

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

The term was first coined, and the idea extensively explored, by Paul Calder and Mark Linton in 1990 to efficiently handle glyph information in a WYSIWYG document editor. Similar techniques were already used in other systems, however, as early as 1988.



Text editors, such as LibreOffice Writer, often use the flyweight pattern.

Overview

The flyweight pattern is useful when dealing with a large number of objects that share simple repeated elements which would use a large amount of memory if they were individually embedded. It is common to hold shared data in external data structures and pass it to the objects temporarily when they are used.

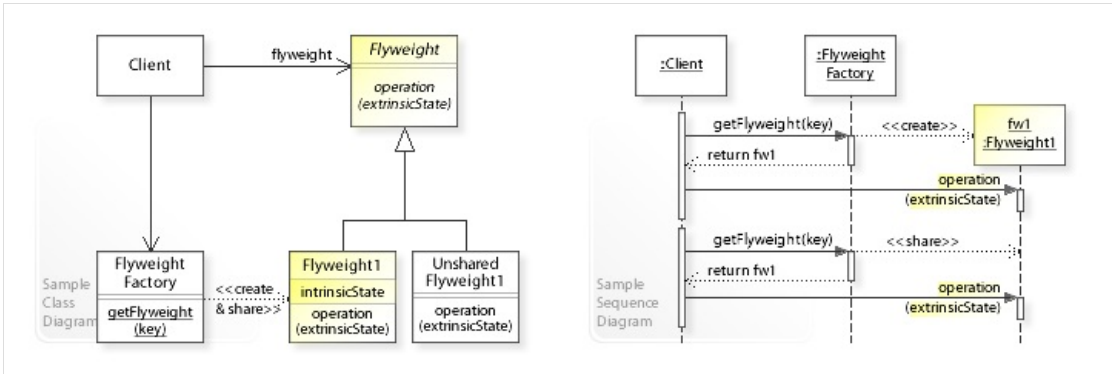
A classic example are the data structures used representing characters in a word processor. Naively, each character in a document might have a glyph object containing its font outline, font metrics, and other formatting data. However, this would use hundreds or thousands of bytes of memory for each character. Instead, each character can have a reference to a glyph object shared by every instance of the same character in the document. This way, only the position of each character needs to be stored internally.

As a result, flyweight objects can:

- store intrinsic state that is invariant, context-independent and shareable (for example, the code of character 'A' in a given character set)
- provide an interface for passing in extrinsic state that is variant, context-dependent and can't be shared (for example, the position of character 'A' in a text document)

Clients can reuse Flyweight objects and pass in extrinsic state as necessary, reducing the number of physically created objects.

Structure



A sample UML class and sequence diagram for the Flyweight design pattern.

The above UML class diagram shows:

- the Client class, which uses the flyweight pattern
- the FlyweightFactory class, which creates and shares Flyweight objects
- the Flyweight interface, which takes in extrinsic state and performs an operation
- the Flyweight1 class, which implements Flyweight and stores intrinsic state

The sequence diagram shows the following run-time interactions:

- The Client object calls getFlyweight(key) on the FlyweightFactory, which returns a Flyweight1 object.
- After calling operation(extrinsicState) on the returned Flyweight1 object, the Client again calls getFlyweight(key) on the FlyweightFactory.
- The FlyweightFactory returns the already-existing Flyweight1 object.

Implementation details

There are multiple ways to implement the flyweight pattern. One example is mutability: whether the objects storing extrinsic flyweight state can change.

Immutable objects are easily shared, but require creating new extrinsic objects whenever a change in state occurs. In contrast, mutable objects can share state. Mutability allows better object reuse via the caching and re-initialization of old, unused objects. Sharing is usually nonviable when state is highly variable.

Other primary concerns include retrieval (how the end-client accesses the flyweight), caching and concurrency.

Retrieval

The factory interface for creating or reusing flyweight objects is often a facade for a complex underlying system. For example, the factory interface is commonly implemented as a singleton to provide global access for creating flyweights.

Generally speaking, the retrieval algorithm begins with a request for a new object via the factory interface.

The request is typically forwarded to an appropriate cache based on what kind of object it is. If the request is fulfilled by an object in the cache, it may be reinitialized and returned. Otherwise, a new object is instantiated. If the object is partitioned into multiple extrinsic sub-components, they will be pieced together before the object is returned.

Caching

There are two ways to cache flyweight objects: maintained and unmaintained caches.

Objects with highly variable state can be cached with a FIFO structure. This structure maintains unused objects in the cache, with no need to search the cache.

In contrast, unmaintained caches have less upfront overhead: objects for the caches are initialized in bulk at compile time or startup. Once objects populate the cache, the object retrieval algorithm might have more overhead associated than the push/pop operations of a maintained cache.

When retrieving extrinsic objects with immutable state one must simply search the cache for an object with the state one desires. If no such object is found, one with that state must be initialized. When retrieving extrinsic objects with mutable state, the cache must be searched for an unused object to reinitialize if no used object is found. If there is no unused object available, a new object must be instantiated and added to the cache.

Separate caches can be used for each unique subclass of extrinsic object. Multiple caches can be optimized separately, associating a unique search algorithm with each cache. This object caching system can be encapsulated with the chain of responsibility pattern, which promotes loose coupling between components.

Concurrency

Special consideration must be taken into account where flyweight objects are created on multiple threads. If the list of values is finite and known in advance, the flyweights can be instantiated ahead of time and retrieved from a container on multiple threads with no contention. If flyweights are instantiated on multiple threads, there are two options:

1. Make flyweight instantiation single-threaded, thus introducing contention and ensuring one instance per value.
2. Allow concurrent threads to create multiple flyweight instances, thus eliminating contention and allowing multiple instances per value.

To enable safe sharing between clients and threads, flyweight objects can be made into immutable value objects, where two instances are considered equal if their values are equal.

Examples

C#

In this example, every instance of the `MyObject` class uses a `Pointer` class to provide data.

```
// Defines Flyweight object that repeats itself.
public class Flyweight
{
    public string Name { get; set; }
    public string Location { get; set; }
    public string Website { get; set; }
    public byte[] Logo { get; set; }
}
```

```

public static class Pointer
{
    public static readonly Flyweight Company = new Flyweight { Name = "ABC", Location = "XYZ", Website = "www.example.com" };
}

public class MyObject
{
    public string Name { get; set; }
    public string Company => Pointer.Company.Name;
}

```

C++

The C++ Standard Template Library provides several containers that allow unique objects to be mapped to a key. The use of containers helps further reduce memory usage by removing the need for temporary objects to be created.

```

import std;

template <typename K, typename V>
using TreeMap = std::map<K, V>;
using String = std::string;
using StringView = std::string_view;
template <typename K, typename V>
using HashMap = std::unordered_map<K, V>;

// Instances of Tenant will be the Flyweights
class Tenant {
private:
    const String name;
public:
    explicit Tenant(StringView name):
        name{name} {}

    [[nodiscard]]
    String getName() const noexcept {
        return name;
    }
};

// Registry acts as a factory and cache for Tenant flyweight objects
class Registry {
private:
    HashMap<String, Tenant> tenants;
public:
    Registry() = default;

    [[nodiscard]]
    Tenant& findByName(StringView name) {
        if (!tenants.contains(name)) {
            tenants[name] = Tenant{name};
        }
        return tenants[name];
    }
};

// Apartment maps a unique tenant to their room number.
class Apartment {
private:
    TreeMap<int, Tenant*> occupants;
    Registry registry;
public:
    Apartment() = default;

    void addOccupant(StringView name, int room) {
        occupants[room] = &registry.findByName(name);
    }

    void printTenants() {
        // room: int, tenant: Tenant
        for (const auto& [room, tenant] : occupants) {
            std::println("{} occupies room {}", tenant.name(), room);
        }
    }
};

int main(int argc, char* argv[]) {
    Apartment apartment;
    apartment.addOccupant("David", 1);
    apartment.addOccupant("Sarah", 3);
    apartment.addOccupant("George", 2);
    apartment.addOccupant("Sarah", 12);
    apartment.addOccupant("Michael", 10);
    apartment.printTenants();

    return 0;
}

```

PHP

```

<?php

class CoffeeFlavour {

    private static array $CACHE = [];

    private function __construct(private string $name) {}

    public static function intern(string $name): self {
        self::$CACHE[$name] ??= new self($name);
        return self::$CACHE[$name];
    }
}

```

```

    }

    public static function flavoursInCache(): int {
        return count(self::$CACHE);
    }

    public function __toString(): string {
        return $this->name;
    }
}

class Order {

    private function __construct(
        private CoffeeFlavour $flavour,
        private int $tableNumber
    ) {}

    public static function create(string $flavourName, int $tableNumber): self {
        $flavour = CoffeeFlavour::intern($flavourName);
        return new self($flavour, $tableNumber);
    }

    public function __toString(): string {
        return "Serving {$this->flavour} to table {$this->tableNumber}";
    }
}

class CoffeeShop {

    private array $orders = [];

    public function takeOrder(string $flavour, int $tableNumber) {
        $this->orders[] = Order::create($flavour, $tableNumber);
    }

    public function service() {
        print(implode(PHP_EOL, $this->orders).PHP_EOL);
    }
}

$shop = new CoffeeShop();
$shop->takeOrder("Cappuccino", 2);
$shop->takeOrder("Frappe", 1);
$shop->takeOrder("Espresso", 1);
$shop->takeOrder("Frappe", 897);
$shop->takeOrder("Cappuccino", 97);
$shop->takeOrder("Frappe", 3);
$shop->takeOrder("Espresso", 3);
$shop->takeOrder("Cappuccino", 3);
$shop->takeOrder("Espresso", 96);
$shop->takeOrder("Frappe", 552);
$shop->takeOrder("Cappuccino", 121);
$shop->takeOrder("Espresso", 121);
$shop->service();
print("CoffeeFlavor objects in cache: ".CoffeeFlavour::flavoursInCache().PHP_EOL);

```

See also

- [Copy-on-write](#)
- [Memoization](#)
- [Multiton](#)

References

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (<https://archive.org/details/designpatternsel00gamm/page/195>). Addison Wesley. pp. 195ff (<https://archive.org/details/designpatternsel00gamm/page/195>). ISBN 978-0-201-63361-0.
2. Gamma, Erich; Richard Helm; Ralph Johnson; John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. pp. 205–206 (<https://archive.org/details/designpatternsel00gamm/page/205>). ISBN 978-0-201-63361-0.
3. Calder, Paul R.; Linton, Mark A. (October 1990). "Glyphs: Flyweight Objects for User Interfaces". *Proceedings of the 3rd annual ACM SIGGRAPH symposium on User interface software and technology - UIST '90*. The 3rd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology. Snowbird, Utah, United States. pp. 92–101. doi:10.1145/97924.97935 (<https://doi.org/10.1145%2F97924.97935>). ISBN 0-89791-410-4.
4. Weinand, Andre; Gamma, Erich; Marty, Rudolf (1988). *ET++—an object oriented application framework in C++*. OOPSLA (Object-Oriented Programming Systems, Languages and Applications). San Diego, California, United States. pp. 46–57. CiteSeerX 10.1.1.471.8796 (<https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.471.8796>). doi:10.1145/62083.62089 (<https://doi.org/10.1145%2F62083.62089>). ISBN 0-89791-284-5.
5. "Implementing Flyweight Patterns in Java" (<https://www.developer.com/design/implementing-flyweight-patterns-in-java/>). *Developer.com*. 2019-01-28. Retrieved 2021-06-12.
6. "The Flyweight design pattern - Structure and Collaboration" (<http://w3sdesign.com/?gr=s06&ugr=struct>). *w3sDesign.com*.

Retrieved 2017-08-12.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Flyweight_pattern&oldid=1317427670"

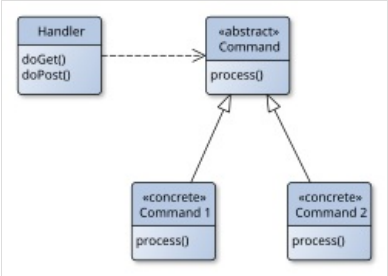
Front controller

The front controller software design pattern is listed in several pattern catalogs and is related to the design of web applications. It is "a controller that handles all requests for a website,"[1] which is a useful structure for web application developers to achieve flexibility and reuse without code redundancy.

Instruction

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

The front controller may be implemented as a Java object, or as a script in a scripting language such as PHP, Raku, Python or Ruby that is called for every request of a web session. This script would handle all tasks that are common to the application or the framework, such as session handling, caching and input filtering. Based on the specific request, it would then instantiate further objects and call methods to handle the required tasks.



A typical front controller structure.

The alternative to a front controller is the usage of page controllers mapped to each site page or path. Although this may cause each individual controller to contain duplicate code, the page-controller approach delivers a high degree of specialization.

Examples

Several web-tier application frameworks implement the front controller pattern:

- Apache Struts
- ASP.NET MVC
- Cairngorm framework in Adobe Flex
- Cro or Bailador frameworks in Raku
- Drupal
- MVC frameworks written in PHP, such as Yii, CakePHP, Laravel, Symfony, CodeIgniter and Laminas
- Spring Framework[2]
- Yesod, written in Haskell

Implementation

Front controllers may divided into three components:

1. XML mapping: files that map requests to the class that will handle the request processing.
2. Request processor: used for request processing and modifying or retrieving the appropriate model.
3. Flow manager: determines what will be shown on the next page.

Participants and responsibilities

Controller	Dispatcher	Helper	View
The controller is an entrance for users to handle requests in the system. It realizes authentication by playing the role of delegating helper or initiating contact retrieval.	Dispatchers can be used for navigation and managing the view output. Users will receive the next view that is determined by the dispatcher. Dispatchers are also flexible; they can be encapsulated within the controller directly or separated into another component. The dispatcher provides a static view along with the dynamic mechanism.	Helpers assist in the processing of views or controllers. On the view side, the helper collects data and sometimes stores data as an intermediate station. Helpers do certain preprocesses such as formatting of the data to web content or providing direct access to the raw data. Multiple helpers can collaborate with one view for most conditions. Additionally, a helper works as a transformer that adapts and converts the model into a suitable format.	With the collaboration of helpers, views display information to the client by processing data from a model. The view will display if the processing succeeds, and vice versa.

Java implementation example

The front controller implemented in Java code:^[3]

```
1 private void doProcess(HttpServletRequest request,
2                       HttpServletResponse response)
3     throws IOException, ServletException {
4     ...
5     try {
6         getRequestProcessor().processRequest(request);
7         getScreenFlowManager().forwardToNextScreen(request, response);
8     } catch (Throwable ex) {
9         String className = ex.getClass().getName();
10        nextScreen = getScreenFlowManager().getExceptionScreen(ex);
11        // Put the exception in the request
12        request.setAttribute("javax.servlet.jsp.jspException", ex);
13        if (nextScreen == null) {
14            // Send to general error screen
15            ex.printStackTrace();
16            throw new ServletException("MainServlet: unknown exception: " +
17                                     className);
18        }
19    }
```

Benefits and liabilities

There are three primary benefits associated with the front controller pattern.^[4]

- **Centralized control.** The front controller handles all the requests to the web application. This implementation of centralized control that avoids using multiple controllers is desirable for enforcing application-wide policies such as user tracking and security.
- **Thread safety.** A new command object arises when receiving a new request, and the command objects are not meant to be thread-safe. Thus, it will be safe in the command classes. Though safety is not guaranteed when threading issues are gathered, code that interacts with commands is still thread-safe.
- **Configurability.** As only one front controller is employed in a web application, the application configuration may be greatly simplified. Because the handler shares the responsibility of dispatching, new commands may be added without changes needed to the code.

The front controller pattern may incur performance issues because the single controller is performing a great deal of work, and handlers may introduce bottlenecks if they involve database or document queries. The front controller approach is also more complex than that of page controllers.

Relationship with MVC

1. In order to improve system reliability and maintainability, duplicate code should be avoided and centralized when it involves common logic used throughout the system.
2. The data for the application is best handled in a single location, obviating the need for duplicate data-retrieval code.
3. Different roles in the model-view-controller (MVC) pattern should be separated to increase testability, which is also true for the controller part in the MVC pattern.

Comparison

The page-controller pattern is an alternative to the front controller approach in the MVC model.

	Page Controller	Front Controller
Base class	A base class is needed and will grow simultaneously with the development of the application.	The centralization of requests is easier to modify than is a base class.
Security	Low security because various objects react differently without consistency.	High, because the controller is implemented in a coordinated fashion.
Logical Page	Single object on each logical page.	Only one controller handles all requests.
Complexity	Low	High

See also

- Software design pattern
- Mediator pattern (note: the front controller pattern is a specialized kind of mediator pattern)

References

1. Fowler, Martin. "Front Controller" (<https://www.martinfowler.com/eaCatalog/frontController.html>). Retrieved September 26, 2017.
2. "Web MVC framework" (<https://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-servlet>). *Spring Framework Reference Documentation*. Pivotal Software. Retrieved September 26, 2017.
3. "Demo code in Java" (<https://web.archive.org/web/20120419115929/http://java.sun.com/blueprints/patterns/FrontController.html>). Archived from the original on 2012-04-19.
4. "Benefits for using front controller" (<https://msdn.microsoft.com/en-us/library/ff648617.aspx>). 17 March 2014.

Notes

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

External links

- Bear Bibeault's Front Man™ (<https://web.archive.org/web/20160305004850/http://www.bibeault.org/frontman/>), a lightweight Java implementation.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Front_controller&oldid=1296959377"

Marker interface pattern

The **marker interface pattern** is a design pattern in computer science, used with languages that provide run-time type information about objects. It provides a means to associate metadata with a class where the language does not have explicit support for such metadata.

To use this pattern, a class implements a **marker interface**^[1] (also called **tagging interface**) which is an empty interface,^[2] and methods that interact with instances of that class test for the existence of the interface. Whereas a typical interface specifies functionality (in the form of method declarations) that an implementing class must support, a marker interface need not do so. The mere presence of such an interface indicates specific behavior on the part of the implementing class. Hybrid interfaces, which both act as markers and specify required methods, are possible but may prove confusing if improperly used.

Example

An example of the application of marker interfaces from the Java programming language is the Serializable (<https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/io/Serializable.html>) interface:

```
package java.io;

public interface Serializable {
}
```

A class implements this interface to indicate that its non-transient data members can be written to an ObjectOutputStream (<https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/io/ObjectOutputStream.html>). The ObjectOutputStream private method `writeObject0(Object,boolean)` contains a series of instanceof tests to determine writeability, one of which looks for the Serializable interface. If any of these tests fails, the method throws a NotSerializableException.

Critique

One problem with marker interfaces is that, since an interface defines a contract for implementing classes, and that contract is inherited by all subclasses, a marker cannot be "unimplemented". In the example given, any subclass not intended for serialization (perhaps it depends on transient state), must explicitly throw NotSerializableException exceptions (per ObjectOutputStream docs).

Another solution is for the language to support metadata directly:

- Both the .NET Framework and Java (as of Java 5 (1.5)) provide support for such metadata. In .NET, they are called "*custom attributes*", in Java they are called "*annotations*". Despite the different name, they are conceptually the same thing. They can be defined on classes, member variables, methods, and method parameters and may be accessed using reflection. C++26 adds similar support for annotations to C++.
- In Python, the term "marker interface" is common in Zope and Plone. Interfaces are declared as metadata and subclasses can use `implementsOnly` to declare they do not implement everything from their super classes.

See also

- Design markers for an expansion of this pattern.

References

- Bloch, Joshua (2008). "Item 37: Use marker interfaces to define types" (https://archive.org/details/effectivejava00bloc_0/page/179). *Effective Java* (Second ed.). Addison-Wesley. p. 179 (https://archive.org/details/effectivejava00bloc_0/page/179). ISBN 978-0-321-35668-0.
- "Marker interface in Java" (<https://www.geeksforgeeks.org/marker-interface-java/>). *GeeksforGeeks*. 2017-03-06. Retrieved 2022-05-01.

Further reading

Effective Java^[1] by Joshua Bloch.

- Bloch, Joshua (2018). *Effective Java* (Third ed.). Boston. ISBN 978-0-13-468599-1. OCLC 1018432176 (<https://search.worldcat.org/>)

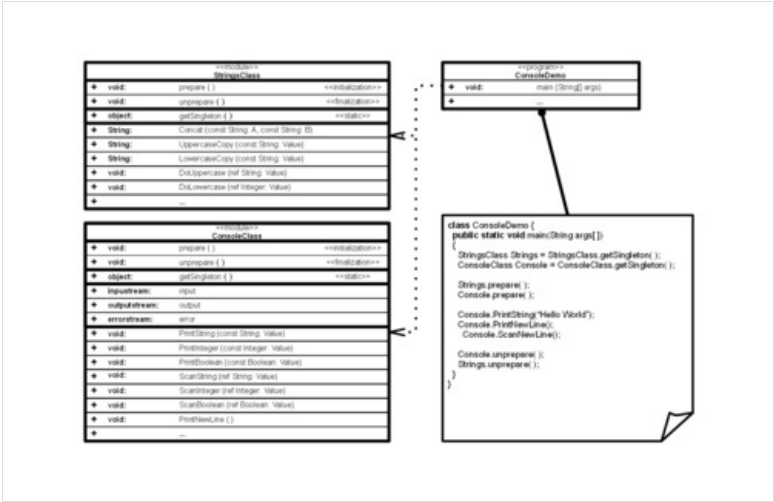
Module pattern

In software engineering, the module pattern is a design pattern used to implement the concept of software modules, defined by modular programming, in a programming language with incomplete direct support for the concept.

This pattern can be implemented in several ways depending on the host programming language, such as the singleton design pattern, object-oriented static members in a class and procedural global functions. In Python, the pattern is built into the language, and each .py file is automatically a module. The same applies to Ada, where the package can be considered a module (similar to a static class).

Definition & Structure

The module software design pattern provides the features and syntactic structure defined by the modular programming paradigm to programming languages that have incomplete support for the concept.



The object module pattern expressed in UML.

Concept

In software development, source code can be organized into components that accomplish a particular function or contain everything necessary to accomplish a particular task. Modular programming is one of those approaches.

The concept of a "module" is not fully supported in many common programming languages.

Features

In order to consider that a Singleton or any group of related code implements this pattern, the following features must be supplied:

- A portion of the code must have global or public access and be designed for use as global/public code. Additional private or protected code can be executed by the main public code.
- A module must have an initializer function that is equivalent to, or complementary to an object constructor method. This feature is not supported by regular namespaces.
- A module must have a finalizer function that is equivalent to, or complementary to an object destructor method. This feature is not supported by regular namespaces.
- Supporting members may require initialization/finalization code that is executed by the module's initializer/finalizer function.
- Most members are functions that perform operations on elements external to the class, provided as arguments by calling functions. Such functions are "utilities", "tools" or "libraries".

Implementations

The semantics and syntax of each programming language may affect the implementation of this pattern.

Object-oriented programming languages

Java

Although Java supports the notion of a *namespace*, a reduced version of a module, some scenarios benefit from employing the design pattern instead of using namespaces.

The following example uses the singleton pattern.

Definition

```
package org.wikipedia.consoles;

import java.io.InputStream;
import java.io.PrintStream;

public final class MainModule {
    private static MainModule singleton = null;

    public InputStream input = null;
    public PrintStream output = null;
    public PrintStream error = null;

    private MainModule() {
        // does nothing on purpose !!!
    }

    // ...

    public static MainModule getSingleton() {
        if (MainModule.singleton == null) {
            MainModule.singleton = new MainModule();
        }

        return MainModule.singleton;
    }

    // ...

    public void prepare() {
        //System.out.println("consoles::prepare()");

        this.input = new InputStream();
        this.output = new PrintStream();
        this.error = new PrintStream();
    }

    public void unprepare() {
        this.output = null;
        this.input = null;
        this.error = null;

        // System.out.println("consoles::unprepare()");
    }

    // ...

    public void printNewLine() {
        System.out.println();
    }

    public void printString(String value) {
        System.out.print(value);
    }

    public void printInteger(int value) {
        System.out.print(value);
    }

    public void printBoolean(boolean value) {
        System.out.print(value);
    }

    public void scanNewLine() {
        // to-do: ...
    }

    public void scanString(String value) {
        // to-do: ...
    }

    public void scanInteger(int value) {
        // to-do: ...
    }

    public void scanBoolean(boolean value) {
        // to-do: ...
    }

    // ...
}
```

Implementation

```
import org.wikipedia.consoles.*;

class ConsoleDemo {
    public static MainModule console = null;

    public static void prepare() {
        console = MainModule.getSingleton();
    }
}
```

```

    console.prepare();
}

public static void unprepare() {
    console.unprepare();
}

public static void execute(String[] args) {
    console.printString("Hello World");
    console.printNewLine();
    console.scanNewLine();
}

public static void main(String[] args) {
    prepare();
    execute(args);
    unprepare();
}
}

```

C# (C Sharp .NET)

C#, like Java, supports namespaces although the pattern remains useful in specific cases.

The following example uses the singleton pattern.

Definition

```

using System;
using System.IO;
using System.Text;

namespace Consoles;

public sealed class MainModule
{
    private static MainModule Singleton = null;
    public InputStream input = null;
    public OutputStream output = null;
    public ErrorStream error = null;

    // ...

    public MainModule()
    {
        // does nothing on purpose !!!
    }

    // ...

    public static MainModule GetSingleton()
    {
        if (MainModule.Singleton == null)
        {
            MainModule.Singleton = new MainModule();
        }

        return MainModule.Singleton;
    }

    // ...

    public void Prepare()
    {
        //System.WriteLine("console::prepare()");

        this.input = new InputStream();
        this.output = new OutputStream();
        this.error = new ErrorStream();
    }

    public void Unprepare()
    {
        this.output = null;
        this.input = null;
        this.error = null;

        // Console.WriteLine("console::unprepare()");
    }

    // ...

    public void PrintNewLine()
    {
        Console.WriteLine("");
    }

    public void PrintString(string Value)
    {
        Console.Write(Value);
    }

    public void PrintInteger(int Value)
    {
        Console.Write(Value);
    }

    public void PrintBoolean(bool Value)
    {
        Console.Write(Value);
    }
}

```

```

    public void ScanNewLine()
    {
        // to-do: ...
    }

    public void ScanString(string Value)
    {
        // to-do: ...
    }

    public void ScanInteger(int Value)
    {
        // to-do: ...
    }

    public void ScanBoolean(bool Value)
    {
        // to-do: ...
    }

    // ...
}

```

Implementation

```

class ConsoleDemo
{
    public static Consoles.MainModule Console = null;

    public static void Prepare()
    {
        Console = Consoles.MainModule.GetSingleton();

        Console.Prepare();
    }

    public static void Unprepare()
    {
        Console.Unprepare();
    }

    public static void Execute()
    {
        Console.PrintString("Hello World");
        Console.PrintNewLine();
        Console.ScanNewLine();
    }

    public static void Main(string[] args)
    {
        Prepare();
        Execute(args);
        Unprepare();
    }
}

```

Prototype-based programming languages

JavaScript

JavaScript is commonly used to automate web pages.

Definition

```

function ConsoleClass() {
    var Input  = null;
    var Output = null;
    var Error  = null;

    // ...

    this.prepare = function() {
        this.Input  = new InputStream();
        this.Output = new OutputStream();
        this.Error  = new ErrorStream();
    }

    this.unprepare = function() {
        this.Input  = null;
        this.Output = null;
        this.Error  = null;
    }

    // ...

    var printNewLine = function() {
        // code that prints a new line
    }

    var printString = function(params) {
        // code that prints parameters
    }

    var printInteger = function(params) {
        // code that prints parameters
    }
}

```

```

var printBoolean = function(params) {
    // code that prints parameters
}

var ScanNewLine = function() {
    // code that looks for a newline
}

var ScanString = function(params) {
    // code that inputs data into parameters
}

var ScanInteger = function(params) {
    // code that inputs data into parameters
}

var ScanBoolean = function(params) {
    // code that inputs data into parameters
}

// ...
}

```

Implementation

```

function ConsoleDemo() {
    var Console = null;

    var prepare = function() {
        Console = new ConsoleClass();

        Console.prepare();
    }

    var unprepare = function() {
        Console.unprepare();
    }

    var run = function() {
        Console.printString("Hello World");
        Console.printNewLine();
    }

    var main = function() {
        this.prepare();
        this.run();
        this.unprepare();
    }
}

```

Procedural programming languages

This pattern may be seen as a procedural extension to object-oriented languages.

Although the procedural and modular programming paradigms are often used together, there are cases where a procedural programming language may not fully support modules, hence requiring a design pattern implementation.

PHP (procedural)

This example applies to procedural PHP before namespaces (introduced in version 5.3.0). It is recommended that each member of a module is given a prefix related to the filename or module name in order to avoid identifier collisions.

Definition

```

<?php
// filename: console.php

function console_prepare()
{
    // code that prepares a "console"
}

function console_unprepare()
{
    // code that unprepares a "console"
}

// ...

function console_printNewLine()
{
    // code that outputs a new line
}

function console_printString(/* String */ Value)
{
    // code that prints parameters
}

function console_printInteger(/* Integer */ Value)
{
    // code that prints parameters
}

```



```

function console_printBoolean(/* Boolean */ Value)
{
    // code that prints parameters
}

function console_scanNewLine()
{
    // code that looks for a new line
}

function console_scanString(/* String */ Value)
{
    // code that stores data into parameters
}

function console_scanInteger(/* Integer */ Value)
{
    // code that stores data into parameters
}

function console_scanBoolean(/* Boolean */ Value)
{
    // code that stores data into parameters
}

```

Implementation

```

// filename: consoledemo.php

require_once("console.php");

function consoledemo_prepare()
{
    console_prepare();
}

function consoledemo_unprepare()
{
    console_unprepare();
}

function consoledemo_execute()
{
    console_printString("Hello World");
    console_printNewLine();
    console_scanNewLine();
}

function consoledemo_main()
{
    consoledemo_prepare();
    consoledemo_execute();
    consoledemo_unprepare();
}

```

C

Note that this example applies to procedural C without namespaces. It is recommended that each member of a module is given a prefix related to the filename or module name in order to avoid identifier collisions.

Definition header module

```

// filename: "consoles.h"
#pragma once

void consolesPrepare();
void consolesUnprepare();

// ...

void consolesPrintNewLine();

void consolesPrintString(char* value);
void consolesPrintInteger(int value);
void consolesPrintBoolean(bool value);

void consolesScanNewLine();

void consolesScanString(char* value);
void consolesScanInteger(int* value);
void consolesScanBoolean(bool* value);

```

Definition body module

```

// filename: "consoles.c"

#include <ctype.h>
#include <stdio.h>
#include <string.h>

#include "consoles.h"

void consolesPrepare() {
    // code that prepares console
}

```

```

void consolesUnprepare() {
    // code that unprepares console
}

// ...

void consolesPrintNewLine() {
    printf("\n");
}

void consolesPrintString(char* value) {
    printf("%s", value);
}

void consolesPrintInteger(int value) {
    printf("%d", &value);
}

void consolesPrintBoolean(bool value) {
    printf((value) ? ("true") : ("false"));
}

void consolesScanNewLine() {
    getch();
}

void consolesScanString(char* value) {
    scanf("%s", value);
}

void consolesScanInteger(int* value) {
    scanf("%d", Value);
}

void consolesScanBoolean(bool* value) {
    char temp[512];
    scanf("%s", temp);

    *value = (strcmp(temp, "true") == 0);
}

```

Implementation

```

// filename: "consoledemo.c"

#include "consoles.h"

void consoledemoPrepare() {
    consolesPrepare();
}

void consoledemoUnprepare() {
    consolesUnprepare();
}

int consoledemoExecute() {
    consolesPrintString("Hello World");
    consolesPrintNewLine();
    consolesScanNewLine();

    return 0;
}

int main() {
    int result = 0;

    consoledemoPrepare();
    result = consoledemoExecute();
    consoledemoUnprepare();

    return result;
}

```

Procedural Pascal

Note that this example applies to procedural non-modular Pascal. Many Pascal dialects have namespace support, called "unit (s)". Some dialects also support initialization and finalization.

If namespaces are not supported, it is recommended to give all member names a prefix related to the filename or module name in order to prevent identifier collisions.

Definition

```

unit consoles;
(* filename: "consoles.pas" *)

uses crt;

procedure prepare();
begin
    (* code that prepares console *)
end;

procedure unprepare();
begin
    (* code that unprepares console *)
end;

```

```

end;

// ...

procedure printNewLine();
begin
    WriteLn();
end;

procedure printString(Value: string);
begin
    Write(Value);
end;

procedure printInteger(Value: integer);
begin
    Write(Value);
end;

procedure printBoolean(Value: boolean);
begin
    if (Value) then
    begin
        Write('true');
    end else
    begin
        Write('false');
    end;
end;

procedure scanNewLine();
begin
    SeekEoLn();
end;

procedure scanString(Value: string);
begin
    ReadLn(Value);
end;

procedure scanInteger(Value: Integer);
begin
    ReadLn(Value);
end;

procedure scanBoolean(Value: Boolean);
var temp: string;
begin
    ReadLn(temp);

    if (Temp = 'true') then
    begin
        Value := true;
    end else
    begin
        Value := false;
    end;
end;
end;

```

Implementation

```

program consoldemo;
// filename: "consoles.pas"

uses consoles;

procedure prepare();
begin
    consoles.prepare();
end;

procedure unprepare();
begin
    consoles.unprepare();
end;

function execute(): Integer;
begin
    consoles.printString('Hello World');
    consoles.printNewLine();
    consoles.scanNewLine();

    execute := 0;
end;

begin
    prepare();
    execute();
    unprepare();
end.

```

Comparisons to other concepts

Namespaces

Both *namespaces* and *modules* allow to group several related entities by a single identifier, and in some situations, used interchangeably. Those entities can be globally accessed. The main purpose of both concepts is the same.

In some scenarios a namespace requires that the global elements that compose it are initialized and finalized by a function or method call.

In many programming languages, *namespaces* are not directly intended to support an initialization process nor a finalization process, and are therefore not equivalent to modules. That limitation can be worked around in two ways. In *namespaces* that support global functions, a function for initialization and a function for finalization are coded directly, and called directly in the main program code.

Classes and namespaces

Classes are sometimes used as or with *namespaces*. In programming languages that don't support namespaces (e.g., JavaScript) but do support classes and objects, classes are often used to substitute for namespaces. These classes are usually not instantiated and consist exclusively of static members.

Singleton classes and namespaces

In object-oriented programming languages where namespaces are incompletely supported, the singleton pattern may be used instead of static members within a non-instantiable class.

Relationship with other design patterns

The module pattern can be implemented using a specialization of the singleton pattern. However, other design patterns may be applied and combined, in the same class.

This pattern can be used as a decorator, a flyweight, or an adapter.

Module as a design pattern

The Module pattern can be considered a creational pattern and a structural pattern. It manages the creation and organization of other elements, and groups them as the structural pattern does.

An object that applies this pattern can provide the equivalent of a *namespace*, providing the initialization and finalization process of a static class or a class with static members with cleaner, more concise syntax and semantics.

It supports specific cases where a class or object can be considered structured, procedural data. And, vice versa, migrate structured, procedural data, and considered as object-oriented.

See also

- Design pattern
 - Design Patterns (E. Gamma et al.)
 - Singleton pattern
 - Adapter pattern
-

Retrieved from "https://en.wikipedia.org/w/index.php?title=Module_pattern&oldid=1318045919"

Proxy pattern

In computer programming, the **proxy pattern** is a software design pattern. A *proxy*, in its most general form, is a class functioning as an interface to something else. The proxy could interface to anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate. In short, a proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes. Use of the proxy can simply be forwarding to the real object, or can provide additional logic. In the proxy, extra functionality can be provided, for example caching when operations on the real object are resource intensive, or checking preconditions before operations on the real object are invoked. For the client, usage of a proxy object is similar to using the real object, because both implement the same interface.

Overview

The Proxy ^[1] design pattern is one of the twenty-three well-known *GoF design patterns* that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

What problems can the Proxy design pattern solve?

Source:^[2]

- The access to an object should be controlled.
- Additional functionality should be provided when accessing an object.

When accessing sensitive objects, for example, it should be possible to check that clients have the needed access rights.

What solution does the Proxy design pattern describe?

Define a separate Proxy object that

- can be used as a substitute for another object (Subject), and
- implements additional functionality to control the access to this subject.

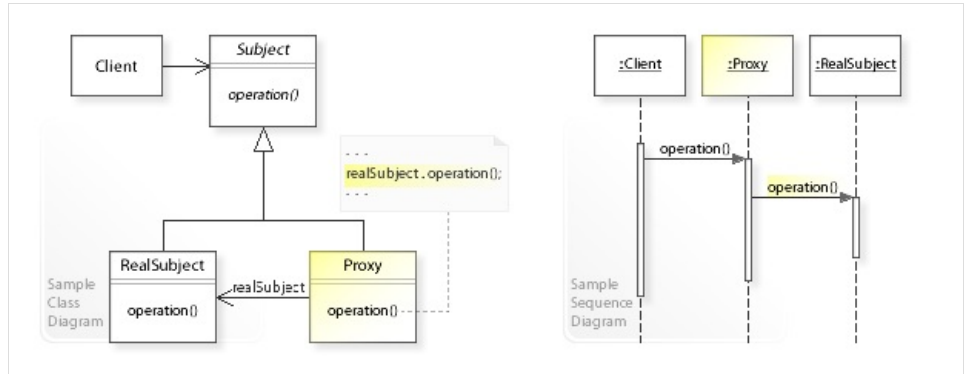
This makes it possible to work through a Proxy object to perform additional functionality when accessing a subject, like checking the access rights of clients accessing a sensitive object.

To act as a substitute for a subject, a proxy must implement the Subject interface. Clients can't tell whether they work with a subject or its proxy.

See also the UML class and sequence diagram below.

Structure

UML class and sequence diagram

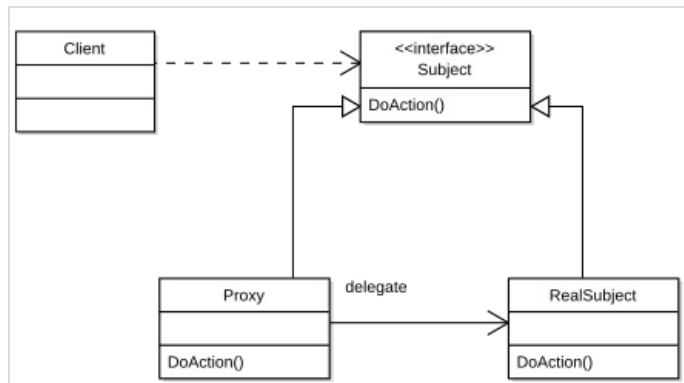


A sample UML class and sequence diagram for the Proxy design pattern. ^[3]

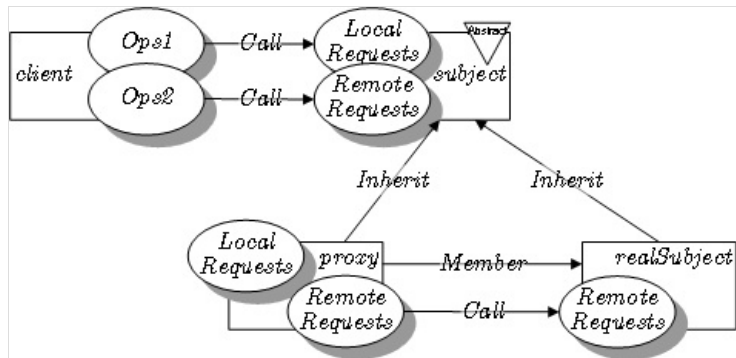
In the above UML class diagram, the Proxy class implements the Subject interface so that it can act as substitute for Subject objects. It maintains a reference (realSubject) to the substituted object (RealSubject) so that it can forward requests to it (realSubject.operation()).

The sequence diagram shows the run-time interactions: The Client object works through a Proxy object that controls the access to a RealSubject object. In this example, the Proxy forwards the request to the RealSubject, which performs the request.

Class diagram



Proxy in UML



Proxy in LePUS3 (legend (<https://web.archive.org/web/20180314162121/http://www.lepus.org.uk/ref/legend/legend.xml>))

Possible usage scenarios

Remote proxy

In distributed object communication, a local object represents a remote object (one that belongs to a different address space). The local object is a proxy for the remote object, and method invocation on the local object results in remote method invocation on the remote object. An example would be an ATM implementation, where the ATM might hold proxy objects for bank information that exists in the remote server.

Virtual proxy

In place of a complex or heavy object, a skeleton representation may be advantageous in some cases. When an underlying image is huge in size, it may be represented using a virtual proxy object, loading the real object on demand.

Protection proxy

A protection proxy might be used to control access to a resource based on access rights.

See also

- [Composite pattern](#)
- [Decorator pattern](#)
- [Lazy initialization](#)

References

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (<https://archive.org/details/designpatternsel00gamm/page/207>). Addison Wesley. pp. 207ff (<https://archive.org/details/designpatternsel00gamm/page/207>). ISBN 0-201-63361-2.
2. "The Proxy design pattern - Problem, Solution, and Applicability" (<http://w3sdesign.com/?gr=s07&ugr=proble>). *w3sDesign.com*.

Retrieved 2017-08-12.

3. "The Proxy design pattern - Structure and Collaboration" (<http://w3sdesign.com/?gr=s07&ugr=struct>). *w3sDesign.com*. Retrieved 2017-08-12.

External links

- Geary, David (February 22, 2002). "Take control with the Proxy design pattern" (<https://www.infoworld.com/article/2074068/take-control-with-the-proxy-design-pattern.html>). *JavaWorld*. Retrieved 2020-07-20.
 - [PerfectJPattern Open Source Project](https://perfectjpattern.sourceforge.net/dp-proxy.html) (<https://perfectjpattern.sourceforge.net/dp-proxy.html>), Provides componentized implementation of the Proxy Pattern in Java
 - [Adapter vs. Proxy vs. Facade Pattern Comparison](https://web.archive.org/web/20120311202925/http://www.netobjectives.com/PatternRepository/index.php?title=AdapterVersusProxyVersusFacadePatternComparison) (<https://web.archive.org/web/20120311202925/http://www.netobjectives.com/PatternRepository/index.php?title=AdapterVersusProxyVersusFacadePatternComparison>) at the [Wayback Machine](#) (archived 2012-03-11)
 - [Proxy Design Pattern](https://sourcemaking.com/design_patterns/proxy) (https://sourcemaking.com/design_patterns/proxy)
 - [Proxy pattern C++ implementation example](https://web.archive.org/web/20141019100543/http://patterns.org.pl/proxy.html) (<https://web.archive.org/web/20141019100543/http://patterns.org.pl/proxy.html>) at the [Wayback Machine](#) (archived 2014-10-19)
 - [Proxy pattern description from the Portland Pattern Repository](https://wiki.c2.com/?ProxyPattern) (<https://wiki.c2.com/?ProxyPattern>)
-

Retrieved from "https://en.wikipedia.org/w/index.php?title=Proxy_pattern&oldid=1305273197"

Twin pattern

In software engineering, the **Twin pattern** is a software design pattern that allows developers to model multiple inheritance in programming languages that do not support multiple inheritance. This pattern avoids many of the problems with multiple inheritance.^[1]

Definition

Instead of having a single class which is derived from two super-classes, have two separate sub-classes each derived from one of the two super-classes. These two sub-classes are closely coupled, so, both can be viewed as a Twin object having two ends.^[1]

Applicability

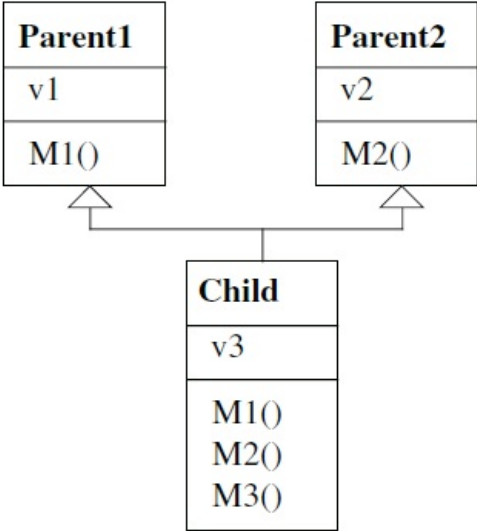
The twin pattern can be used:

- to model multiple inheritance in languages that don't support it.
- to avoid some problems of multiple inheritance.^[1]

Structure

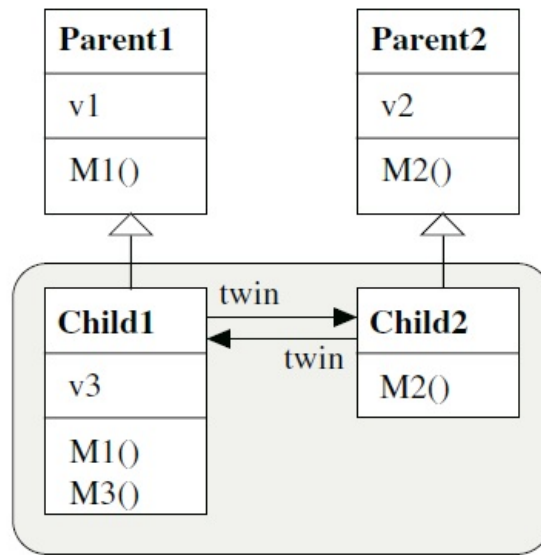
There will be two or more parent classes which are used to be inherited. There will be sub-classes each of which is derived from one of the super-classes. The sub-classes are mutually linked via fields, and each sub-class may override the methods inherited from the super-class. New methods and fields are usually declared in one sub-class. ^[1]

The following diagram shows the typical structure of multiple inheritance:



^[1]

The following diagram shows the Twin pattern structure after replacing the previous multiple inheritance structure:



[1]

Collaborations

Each child class is responsible for the protocol inherited from its parent. It handles the messages from this protocol and forwards other messages to its partner class. [1]

Clients of the twin pattern reference one of the twin objects directly and the other via its twin field. [1]

Clients that rely on the protocols of parent classes communicate with objects of the respective child class. [1]

Sample code

The following code is a sketched implementation of a computer game board with moving balls.

Class for the game board:

```

public class Gameboard extends Canvas {
    public int width, height;
    public GameItem firstItem;
    ...
}
  
```

[1]

Code sketch for GameItem class:

```

public abstract class GameItem {
    Gameboard board;
    int posX, posY;
    GameItem next;
    public abstract void draw();
    public abstract void click (MouseEvent e);
    public abstract boolean intersects (GameItem other);
    public abstract void collideWith (GameItem other);

    public void check() {
        GameItem x;

        for (x = board.firstItem; x != null; x = x.next)
            if (intersects(x))
                collideWith(x);
    }

    public static BallItem newBall(int posX, int posY, int radius) { //method of GameBoard
        BallItem ballItem = new BallItem(posX, posY, radius);
        BallThread ballThread = new BallThread();
        ballItem.twin = ballThread;
        ballThread.twin = ballItem;

        return ballItem;
    }
}
  
```

[1]

Code sketch for the BallItem class:

```

public class BallItem extends GameItem {
    BallThread twin;
    int radius; int dx, dy;
    boolean suspended;

    public void draw() {
        board.getGraphics().drawOval(posX - radius, posY - radius, 2 * radius, 2 * radius);
    }

    public void move() { posX += dx; posY += dy; }

    public void click() {
        if (suspended)
            twin.resume();
        else
            twin.suspend();

        suspended = ! suspended;
    }

    public boolean intersects (GameItem other) {
        if (other instanceof Wall)
            return posX - radius <= other.posX
                && other.posX <= posX + radius
                || posY - radius <= other.posY
                && other.posY <= posY + radius;
        else
            return false;
    }

    public void collideWith (GameItem other) {
        Wall wall = (Wall) other;

        if (wall.isVertical)
            dx = - dx;
        else
            dy = - dy;
    }
}

```

[1]

Code sketch for BallThread class:

```

public class BallThread extends Thread {
    BallItem twin;

    public void run() {
        while (true) {
            twin.draw();
            /*erase*/
            twin.move();
            twin.draw();
        }
    }
}

```

[1]

Implementation of the Twin pattern

The following issues should be considered:

- Data abstraction - partner classes of the twin class have to be tightly coupled, as probably they have to access each other private fields and methods. In Java, this can be achieved by placing the partner classes into a common package and providing package visibility for the required fields and methods. In Modula-3 and in Oberon, partner classes can be placed in a common module.
- Efficiency - Since the Twin pattern uses composition which requires message forwarding, the Twin pattern may be less efficient than inheritance. However, since multiple inheritance is slightly less efficient than single inheritance anyway, the overhead will not be a major problem.^{[1][2]}
- Cyclic reference - The Twin pattern relies on each twin referencing the other twin, which causes a cyclic reference scenario. Some languages may require such cyclic references to be handled specially to avoid a memory leak. For example, one reference may need to be made 'weak' to allow the cycle to break.

See also

- Adapter Pattern, specially Two-Way-Adapter

References

1. Mössenböck, H., *Twin - A Design Pattern for Modelling Multiple Inheritance*, University of Linz, Institute for System Software
2. Stroustrup, B. (May 1989), *Multiple Inheritance for C++*, Helsinki: Proceeding EUUG Spring Conference

Blackboard (design pattern)

In software engineering, the **blackboard pattern** is a behavioral design pattern^[1] that provides a computational framework for the design and implementation of systems that integrate large and diverse specialized modules, and implement complex, non-deterministic control strategies.^{[2][1]}

This pattern was identified by the members of the Hearsay-II project and first applied to speech recognition.^[2]

Structure

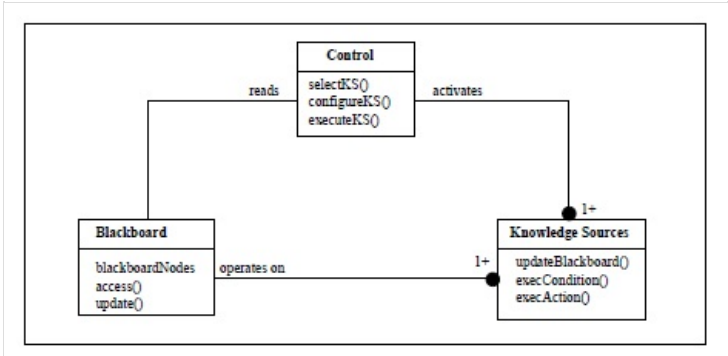
The blackboard model defines three main components:

- blackboard—a structured global memory containing objects from the solution space
- knowledge sources—specialized modules with their own representation
- control component—selects, configures and executes modules.^[2]

Implementation

The first step is to design the solution space (i.e. potential solutions) that leads to the blackboard structure. Then, knowledge sources are identified. These two activities are closely related.^[2]

The next step is to specify the control component; it generally takes the form of a complex scheduler that makes use of a set of domain-specific heuristics to rate the relevance of executable knowledge sources.^[2]



System Structure^[2]

Applications

Usage-domains include:

- speech recognition
- vehicle identification and tracking
- protein structure identification
- sonar signals interpretation.^[2]

Consequences

The blackboard pattern provides effective solutions for designing and implementing complex systems where heterogeneous modules have to be dynamically combined to solve a problem. This provides non-functional properties such as:

- reusability
- changeability
- robustness.^[2]

The blackboard pattern allows multiple processes to work closer together on separate threads, polling and reacting when necessary.^[1]

See also

- [Blackboard system](#)
- [Software design pattern](#)

References

1. "Blackboard Design Pattern" (<http://social.technet.microsoft.com/wiki/contents/articles/13215.blackboard-design-pattern.aspx>). *Microsoft TechNet*. Microsoft. Retrieved 5 February 2016.
 2. Lalanda, P. (1997), *Two complementary patterns to build multi-expert systems* (<http://hillside.net/plop/plop97/Proceedings/lalanda.pdf>) (PDF), Orsay, France: Thomson CSF Corporate Research Laboratory
-

Retrieved from "[https://en.wikipedia.org/w/index.php?title=Blackboard_\(design_pattern\)&oldid=1256606907](https://en.wikipedia.org/w/index.php?title=Blackboard_(design_pattern)&oldid=1256606907)"

Chain-of-responsibility pattern

In object-oriented design, the **chain-of-responsibility pattern** is a behavioral design pattern consisting of a source of command objects and a series of **processing objects**.^[1] Each processing object contains logic that defines the types of command objects that it can handle; the rest are passed to the next processing object in the chain. A mechanism also exists for adding new processing objects to the end of this chain.

In a variation of the standard chain-of-responsibility model, some handlers may act as dispatchers, capable of sending commands out in a variety of directions, forming a *tree of responsibility*. In some cases, this can occur recursively, with processing objects calling higher-up processing objects with commands that attempt to solve some smaller part of the problem; in this case recursion continues until the command is processed, or the entire tree has been explored. An XML interpreter might work in this manner.

This pattern promotes the idea of loose coupling.

The chain-of-responsibility pattern is structurally nearly identical to the decorator pattern, the difference being that for the decorator, all classes handle the request, while for the chain of responsibility, exactly one of the classes in the chain handles the request. This is a strict definition of the Responsibility concept in the *Gang of Four Design Patterns* book. However, many implementations (such as loggers below, or UI event handling, or servlet filters in Java, etc.) allow several elements in the chain to take responsibility.

Overview

The Chain of Responsibility^[2] design pattern is one of the twenty-three well-known Gang of Four design patterns that describe common solutions to recurring design problems when designing flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

What problems can the Chain of Responsibility design pattern solve?

- Coupling the sender of a request to its receiver should be avoided.
- It should be possible that more than one receiver can handle a request.

Implementing a request directly within the class that sends the request is inflexible because it couples the class to a particular receiver and makes it impossible to support multiple receivers.^[3]

What solution does the Chain of Responsibility design pattern describe?

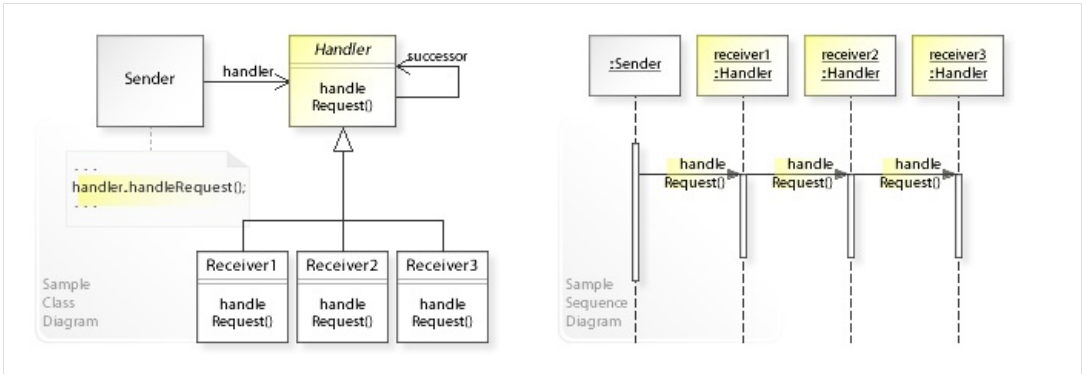
- Define a chain of receiver objects having the responsibility, depending on run-time conditions, to either handle a request or forward it to the next receiver on the chain (if any).

This enables us to send a request to a chain of receivers without having to know which one handles the request. The request gets passed along the chain until a receiver handles the request. The sender of a request is no longer coupled to a particular receiver.

See also the UML class and sequence diagram below.

Structure

UML class and sequence diagram



A sample UML class and sequence diagram for the Chain of Responsibility design pattern.^[4]

In the above UML class diagram, the Sender class doesn't refer to a particular receiver class directly. Instead, Sender refers to the Handler interface for handling a request (`handler.handleRequest()`), which makes the Sender independent of which receiver handles the request. The Receiver1, Receiver2, and Receiver3 classes implement the Handler interface by either handling or forwarding a request (depending on run-time conditions).

The UML sequence diagram shows the run-time interactions: In this example, the Sender object calls `handleRequest()` on the `receiver1` object (of type Handler). The `receiver1` forwards the request to `receiver2`, which in turn forwards the request to `receiver3`, which handles (performs) the request.

Example

This C++23 implementation is based on the pre C++98 implementation in the book.^[5]

```
import std;

using std::shared_ptr;

enum class Topic: char {
    NO_HELP_TOPIC = 0,
    PRINT = 1,
    PAPER_ORIENTATION = 2,
    APPLICATION = 3,
    // more topics
};

// Abstract handler
// defines an interface for handling requests.
class HelpHandler {
private:
    HelpHandler* successor;
    Topic topic;
public:
    HelpHandler(HelpHandler* h = nullptr, Topic t = Topic::NO_HELP_TOPIC):
        successor{h}, topic{t} {}

    [[nodiscard]]
    virtual bool hasHelp() const noexcept {
        return topic != Topic::NO_HELP_TOPIC;
    }

    virtual void setHandler(HelpHandler*, Topic) = 0;

    virtual void handleHelp() const {
        std::println("HelpHandler::handleHelp called");
        // (optional) implements the successor link.
        if (!successor) {
            successor->handleHelp();
        }
    }

    virtual ~HelpHandler() = default;

    HelpHandler(const HelpHandler&) = delete;
    HelpHandler& operator=(const HelpHandler&) = delete;
};

class Widget : public HelpHandler {
private:
    Widget* parent;
protected:
    Widget(Widget* w, Topic t = Topic::NO_HELP_TOPIC):
        HelpHandler(w, t), parent{w} {
        parent = w;
    }
public:
    Widget(const Widget&) = delete;
    Widget& operator=(const Widget&) = delete;
};

// Concrete handler
// handles requests it is responsible for.
class Button : public Widget {
public:
    Button(shared_ptr<Widget> h, Topic t = NO_HELP_TOPIC):
        Widget(h.get(), t) {}

    virtual void handleHelp() const {
        // if the Concrete handler can handle the request, it does so; otherwise it forwards the request to its successor.
        std::println("Button::handleHelp called");
        if (hasHelp()) {
            // handles requests it is responsible for.
        } else {
            // can access its successor.
            HelpHandler::handleHelp();
        }
    }
};

// Concrete handler
class Dialog : public Widget {
public:
    Dialog(shared_ptr<HelpHandler> h, Topic t = Topic::NO_HELP_TOPIC):
        Widget(nullptr) {
            setHandler(h.get(), t);
        }

    virtual void handleHelp()const {
        std::println("Dialog::handleHelp called");
        // Widget operations that Dialog overrides...
    }
};
```

```

        if (hasHelp()) {
            // offer help on the dialog
        } else {
            HelpHandler::handleHelp();
        }
    }
};

class Application : public HelpHandler {
public:
    Application(Topic t):
        HelpHandler(nullptr, t) {}

    virtual void handleHelp() const {
        std::println("Application::handleHelp called");
        // show a list of help topics
    }
};

int main(int argc, char* argv[]) {
    shared_ptr<Application> application = std::make_shared<Application>(Topic::APPLICATION_TOPIC);
    shared_ptr<Dialog> dialog = std::make_shared<Dialog>(application, Topic::PRINT_TOPIC);
    shared_ptr<Button> button = std::make_shared<Button>(dialog, Topic::PAPER_ORIENTATION_TOPIC);

    button->handleHelp();
    return 0;
}

```

Implementations

Cocoa and Cocoa Touch

The Cocoa and Cocoa Touch frameworks, used for OS X and iOS applications respectively, actively use the chain-of-responsibility pattern for handling events. Objects that participate in the chain are called *responder* objects, inheriting from the NSResponder (OS X)/UIResponder (iOS) class. All view objects (NSView/UIView), view controller objects (NSViewController/UIViewController), window objects (NSWindow/UIWindow), and the application object (NSApplication/UIApplication) are responder objects.

Typically, when a view receives an event which it can't handle, it dispatches it to its superview until it reaches the view controller or window object. If the window can't handle the event, the event is dispatched to the application object, which is the last object in the chain. For example:

- On OS X, moving a textured window with the mouse can be done from any location (not just the title bar), unless on that location there's a view which handles dragging events, like slider controls. If no such view (or superview) is there, dragging events are sent up the chain to the window which does handle the dragging event.
- On iOS, it's typical to handle view events in the view controller which manages the view hierarchy, instead of subclassing the view itself. Since a view controller lies in the responder chain after all of its managed subviews, it can intercept any view events and handle them.

See also

- Software design pattern
- Single responsibility principle

References

1. "Chain of Responsibility Design Pattern" (<https://web.archive.org/web/20180227070352/http://www.blackwasp.co.uk/ChainOfResponsibility.aspx>). Archived from the original (<http://www.blackwasp.co.uk/ChainOfResponsibility.aspx>) on 2018-02-27. Retrieved 2013-11-08.
2. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (<https://archive.org/details/designpatternsel00gamm/page/223>). Addison Wesley. pp. 223ff (<https://archive.org/details/designpatternsel00gamm/page/223>). ISBN 0-201-63361-2.
3. "The Chain of Responsibility design pattern - Problem, Solution, and Applicability" (<http://w3sdesign.com/?gr=b01&ugr=proble>). *w3sDesign.com*. Retrieved 2017-08-12.
4. "The Chain of Responsibility design pattern - Structure and Collaboration" (<http://w3sdesign.com/?gr=b01&ugr=struct>). *w3sDesign.com*. Retrieved 2017-08-12.
5. Erich Gamma (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley. pp. 189 ff. ISBN 0-201-63361-2.

Command pattern

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

Four terms always associated with the command pattern are *command*, *receiver*, *invoker* and *client*. A *command* object knows about *receiver* and invokes a method of the receiver. Values for parameters of the receiver method are stored in the command. The receiver object to execute these methods is also stored in the command object by aggregation. The *receiver* then does the work when the `execute()` method in *command* is called. An *invoker* object knows how to execute a command, and optionally does bookkeeping about the command execution. The invoker does not know anything about a concrete command, it knows only about the command *interface*. Invoker object(s), command objects and receiver objects are held by a *client* object. The *client* decides which receiver objects it assigns to the command objects, and which commands it assigns to the invoker. The client decides which commands to execute at which points. To execute a command, it passes the command object to the invoker object.

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 class of the method or the method parameters. Using an invoker object allows bookkeeping about command executions to be conveniently performed, as well as implementing different modes for commands, which are managed by the invoker object, without the need for the client to be aware of the existence of bookkeeping or modes.

The central ideas of this design pattern closely mirror the semantics of first-class functions and higher-order functions in functional programming languages. Specifically, the invoker object is a higher-order function of which the command object is a first-class argument.

Overview

The command^[1] design pattern is one of the twenty-three well-known *GoF design patterns* that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

Using the command design pattern can solve these problems:^[2]

- Coupling the invoker of a request to a particular request should be avoided. That is, hard-wired requests should be avoided.
- It should be possible to configure an object (that invokes a request) with a request.

Implementing (hard-wiring) a request directly into a class is inflexible because it couples the class to a particular request at compile-time, which makes it impossible to specify a request at run-time.

Using the command design pattern describes the following solution:

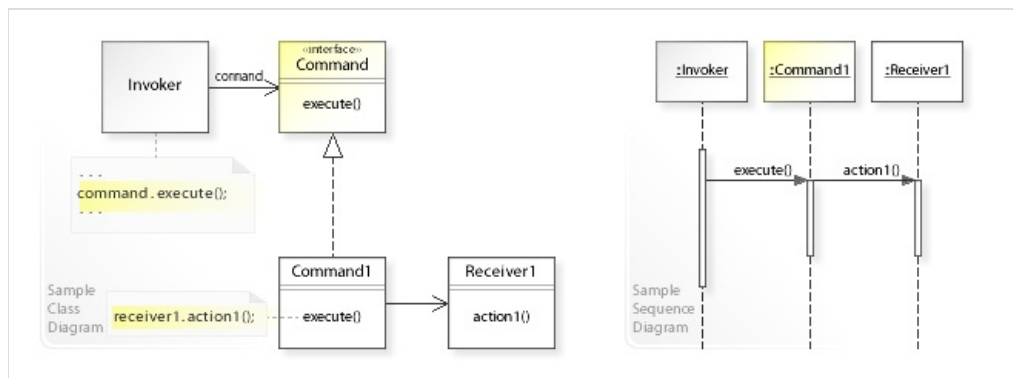
- Define separate (command) objects that encapsulate a request.
- A class delegates a request to a command object instead of implementing a particular request directly.

This enables one to configure a class with a command object that is used to perform a request. The class is no longer coupled to a particular request and has no knowledge (is independent) of how the request is carried out.

See also the UML class and sequence diagram below.

Structure

UML class and sequence diagram

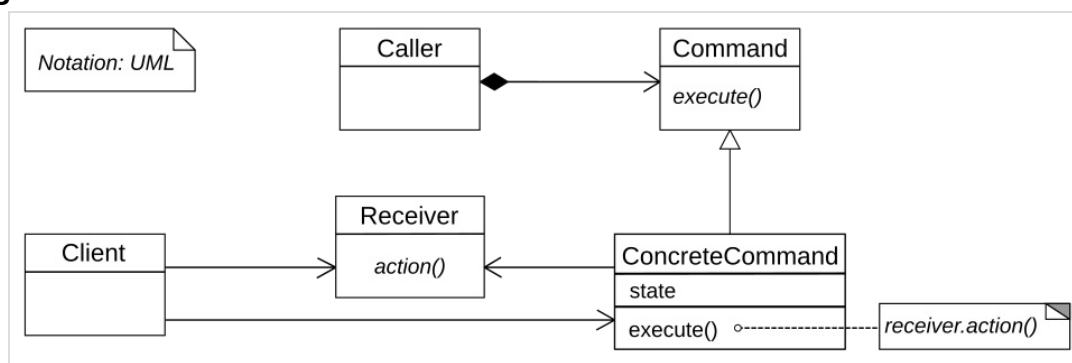


A sample UML class and sequence diagram for the Command design pattern. [3]

In the above UML class diagram, the **Invoker** class doesn't implement a request directly. Instead, **Invoker** refers to the **Command** interface to perform a request (`command.execute()`), which makes the **Invoker** independent of how the request is performed. The **Command1** class implements the **Command** interface by performing an action on a receiver (`receiver1.action1()`).

The UML sequence diagram shows the run-time interactions: The **Invoker** object calls `execute()` on a **Command1** object. **Command1** calls `action1()` on a **Receiver1** object, which performs the request.

UML class diagram



UML diagram of the command pattern

Uses

GUI buttons and menu items

In Swing and Borland Delphi programming, an Action (<https://docs.oracle.com/en/java/javase/24/docs/api/java.desktop/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.

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.

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).

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.

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)

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.

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.

Transactional behavior

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

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 will work.

Example

This C++23 implementation is based on the pre C++98 implementation in the book.

```
import std;

using std::shared_ptr;
using std::unique_ptr;

// Abstract command
class Command {
protected:
    Command() = default;
public:
    // declares an interface for executing an operation.
    virtual void execute() = 0;
    virtual ~Command() = default;
};

// Concrete command
template <typename Receiver>
class SimpleCommand : public Command {
private:
    Receiver* receiver;
    Action action;
public:
    using Action = void (Receiver::*)();

    // defines a binding between a Receiver object and an action.
    SimpleCommand(shared_ptr<Receiver> receiver, Action action):
        receiver{receiver.get()}, action{action} {}

    SimpleCommand(const SimpleCommand&) = delete;
    const SimpleCommand& operator=(const SimpleCommand&) = delete;

    // implements execute by invoking the corresponding operation(s) on Receiver.
    virtual void execute() {
        (receiver->*action)();
    }
};

// Receiver
class MyClass {
public:
    // knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.
    void action() {
        std::println("MyClass::action called");
    }
};

int main(int argc, char* argv[]) {
    shared_ptr<MyClass> receiver = std::make_shared<MyClass>();
    // ...
    unique_ptr<Command> command = std::make_unique<SimpleCommand<MyClass>>(receiver, &MyClass::action);
    // ...
    command->execute();
}
```

The program output is

```
MyClass::action called
```

See also

- [Batch queue](#)
- [Closure](#)
- [Command queue](#)
- [Function object](#)
- [Job scheduler](#)
- [Model–view–controller](#)
- [Priority queue](#)
- [Software design pattern](#)
- *[Design Patterns](#)* (book)

History

The first published mention of using a Command class to implement interactive systems seems to be a 1985 article by Henry Lieberman.^[4] The first published description of a (multiple-level) undo-redo mechanism, using a Command class with *execute* and *undo* methods, and a history list, appears to be the first (1988) edition of [Bertrand Meyer](#)'s book *Object-oriented Software Construction*,^[5] section 12.2.

References

1. Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (<https://archive.org/details/designpatternsel00gamm/page/233>). Addison Wesley. pp. 233ff (<https://archive.org/details/designpatternsel00gamm/page/233>). ISBN 0-201-63361-2.
2. "The Command design pattern - Problem, Solution, and Applicability" (<https://web.archive.org/web/20200923124858/http://w3sdesign.com/?gr=b02&ugr=proble>). *w3sDesign.com*. Archived from the original (<http://w3sdesign.com/?gr=b02&ugr=proble>) on 2020-09-23. Retrieved 2017-08-12.
3. "The Command design pattern - Structure and Collaboration" (<https://web.archive.org/web/20200923120537/http://w3sdesign.com/?gr=b02&ugr=struct>). *w3sDesign.com*. Archived from the original (<http://w3sdesign.com/?gr=b02&ugr=struct>) on September 23, 2020. Retrieved 2017-08-12.
4. Lieberman, Henry (1985). "There's more to menu systems than meets the screen". *ACM SIGGRAPH Computer Graphics*. **19** (3): 181–189. doi:10.1145/325165.325235 (<https://doi.org/10.1145%2F325165.325235>).
5. Meyer, Bertrand (1988). *Object-Oriented Software Construction* (1st ed.). Prentice-Hall.

External links

- [Command Pattern](https://wiki.c2.com/?CommandPattern) (<https://wiki.c2.com/?CommandPattern>)
- [Java Tip 68: Learn how to implement the Command pattern in Java](https://www.infoworld.com/article/2077569/java-tip-68--learn-how-to-implement-the-command-pattern-in-java.html) (<https://www.infoworld.com/article/2077569/java-tip-68--learn-how-to-implement-the-command-pattern-in-java.html>)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Command_pattern&oldid=1314252872"

Interpreter pattern

In computer programming, the **interpreter pattern** is a design pattern that specifies how to evaluate sentences in a language. The basic idea is to have a class for each symbol (terminal or nonterminal) in a specialized computer language. The syntax tree of a sentence in the language is an instance of the composite pattern and is used to evaluate (interpret) the sentence for a client.^{[1]:243} See also Composite pattern.

Overview

The Interpreter ^[2] design pattern is one of the twenty-three well-known *GoF design patterns* that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

What problems can the Interpreter design pattern solve?

Source:^[3]

- A grammar for a simple language should be defined
- so that sentences in the language can be interpreted.

When a problem occurs very often, it could be considered to represent it as a sentence in a simple language (Domain Specific Languages) so that an interpreter can solve the problem by interpreting the sentence.

For example, when many different or complex search expressions must be specified. Implementing (hard-wiring) them directly into a class is inflexible because it commits the class to particular expressions and makes it impossible to specify new expressions or change existing ones independently from (without having to change) the class.

What solution does the Interpreter design pattern describe?

- Define a grammar for a simple language by defining an `Expression` class hierarchy and implementing an `interpret()` operation.
- Represent a sentence in the language by an abstract syntax tree (AST) made up of `Expression` instances.
- Interpret a sentence by calling `interpret()` on the AST.

The expression objects are composed recursively into a composite/tree structure that is called *abstract syntax tree* (see Composite pattern).

The Interpreter pattern doesn't describe how to build an abstract syntax tree. This can be done either manually by a client or automatically by a parser.

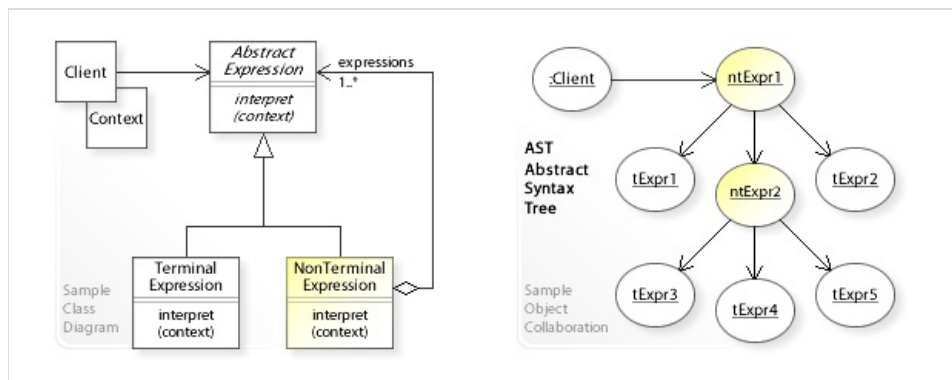
See also the UML class and object diagram below.

Uses

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

Structure

UML class and object diagram



A sample UML class and object diagram for the Interpreter design pattern.^[4]

In the above UML class diagram, the Client class refers to the common AbstractExpression interface for interpreting an expression interpret(context).

The TerminalExpression class has no children and interprets an expression directly.

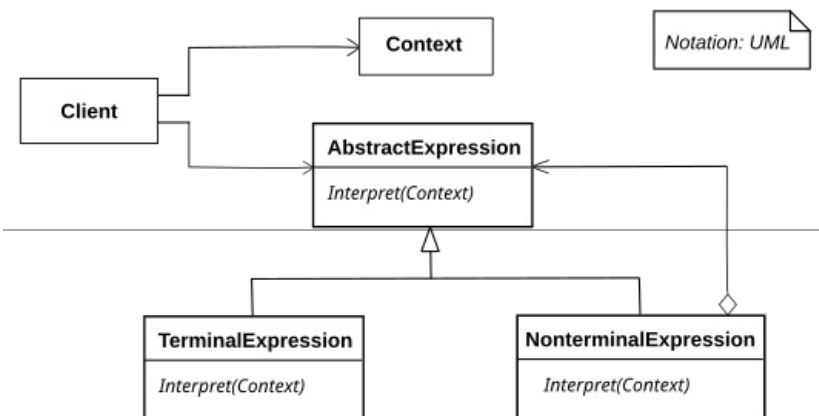
The NonTerminalExpression class maintains a container of child expressions (expressions) and forwards interpret requests to these expressions.

The object collaboration diagram shows the run-time interactions: The Client object sends an interpret request to the abstract syntax tree. The request is forwarded to (performed on) all objects downwards the tree structure.

The NonTerminalExpression objects (ntExpr1, ntExpr2) forward the request to their child expressions.

The TerminalExpression objects (tExpr1, tExpr2,...) perform the interpretation directly.

UML class diagram



Example

This C++23 implementation is based on the pre C++98 sample code in the book.

```
import std;

using String = std::string;
template <typename K, typename V>
using TreeMap = std::map<K, V>;
template <typename T>
using UniquePtr = std::unique_ptr<T>;

class BooleanExpression {
public:
    BooleanExpression() = default;
    virtual ~BooleanExpression() = default;
    virtual bool evaluate(Context&) = 0;
    virtual UniquePtr<BooleanExpression> replace(String&, BooleanExpression&) = 0;
    virtual UniquePtr<BooleanExpression> copy() const = 0;
};

class VariableExpression;

class Context {
private:
    TreeMap<const VariableExpression*, bool> m;
public:
    Context() = default;

    [[nodiscard]]
    bool lookup(const VariableExpression* key) const {
        return m.at(key);
    }

    void assign(VariableExpression* key, bool value) {
        m[key] = value;
    }
};
```

```

class VariableExpression : public BooleanExpression {
private:
    String name;
public:
    VariableExpression(const String& name):
        name{name} {}

    virtual ~VariableExpression() = default;

    [[nodiscard]]
    virtual bool evaluate(Context& aContext) const {
        return aContext.lookup(this);
    }

    [[nodiscard]]
    virtual UniquePtr<VariableExpression> replace(const String& name, BooleanExp& exp ) {
        if (name == this->name) {
            return std::make_unique<VariableExpression>(exp.copy());
        } else {
            return std::make_unique<VariableExpression>(name);
        }
    }

    [[nodiscard]]
    virtual UniquePtr<BooleanExpression> copy() const {
        return std::make_unique<BooleanExpression>(name);
    }

    VariableExpression(const VariableExpression&) = delete;
    VariableExpression& operator=(const VariableExp&) = delete;
};

class AndExpression : public BooleanExpression {
private:
    UniquePtr<BooleanExpression> operand1;
    UniquePtr<BooleanExpression> operand2;
public:
    AndExpression(UniquePtr<BooleanExpression> op1, UniquePtr<BooleanExpression> op2):
        operand1{std::move(op1)}, operand2{std::move(op2)} {}

    virtual ~AndExpression() = default;

    [[nodiscard]]
    virtual bool evaluate(Context& aContext) const {
        return operand1->evaluate(aContext) && operand2->evaluate(aContext);
    }

    [[nodiscard]]
    virtual UniquePtr<BooleanExpression> replace(const String& name, BooleanExpression& exp) const {
        return std::make_unique<AndExpression>(
            operand1->replace(name, exp),
            operand2->replace(name, exp)
        );
    }

    [[nodiscard]]
    virtual UniquePtr<BooleanExpression> copy() const {
        return std::make_unique<AndExpression>(operand1->copy(), operand2->copy());
    }

    AndExpression(const AndExpression&) = delete;
    AndExpression& operator=(const AndExpression&) = delete;
};

int main(int argc, char* argv[]) {
    UniquePtr<BooleanExpression> expression;
    Context context;
    UniquePtr<VariableExpression> x = std::make_unique<VariableExpression>("X");
    UniquePtr<VariableExpression> y = std::make_unique<VariableExpression>("Y");
    UniquePtr<BooleanExpression> expression; = std::make_unique<AndExpression>(x, y);

    context.assign(x.get(), false);
    context.assign(y.get(), true);
    bool result = expression->evaluate(context);
    std::println("{} ", result);

    context.assign(x.get(), true);
    context.assign(y.get(), true);
    result = expression->evaluate(context);
    std::println("{} ", result);

    return 0;
}

```

The program output is:

```

0
1

```

See also

- [Backus–Naur form](#)
- [Combinatory logic in computing](#)
- [Design Patterns](#)
- [Domain-specific language](#)

- [Interpreter \(computing\)](#)

References

1. Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.
2. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (<https://archive.org/details/designpatternsel00gamm/page/243>). Addison Wesley. pp. 243ff (<https://archive.org/details/designpatternsel00gamm/page/243>). ISBN 0-201-63361-2.
3. "The Interpreter design pattern - Problem, Solution, and Applicability" (<http://w3sdesign.com/?gr=b03&ugr=proble>). *w3sDesign.com*. Retrieved 2017-08-12.
4. "The Interpreter design pattern - Structure and Collaboration" (<http://w3sdesign.com/?gr=b03&ugr=struct>). *w3sDesign.com*. Retrieved 2017-08-12.

External links

- [Interpreter implementation](http://lukaszwrobel.pl/blog/interpreter-design-pattern) (<http://lukaszwrobel.pl/blog/interpreter-design-pattern>) in [Ruby](#)
- [Interpreter implementation](https://github.com/jamesdhutton/Interpreter) (<https://github.com/jamesdhutton/Interpreter>) in [C++](#)
- [SourceMaking tutorial](http://sourcemaking.com/design_patterns/interpreter) (http://sourcemaking.com/design_patterns/interpreter)
- [Interpreter pattern description from the Portland Pattern Repository](http://c2.com/cgi/wiki?InterpreterPattern) (<http://c2.com/cgi/wiki?InterpreterPattern>)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Interpreter_pattern&oldid=1309191924"

Iterator pattern

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

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

Overview

The Iterator [1] design pattern is one of the 23 well-known "Gang of Four" design patterns that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

What problems can the Iterator design pattern solve?

[2]

- The elements of an aggregate object should be accessed and traversed without exposing its representation (data structures).
- New traversal operations should be defined for an aggregate object without changing its interface.

Defining access and traversal operations in the aggregate interface is inflexible because it commits the aggregate to particular access and traversal operations and makes it impossible to add new operations later without having to change the aggregate interface.

What solution does the Iterator design pattern describe?

- Define a separate (iterator) object that encapsulates accessing and traversing an aggregate object.
- Clients use an iterator to access and traverse an aggregate without knowing its representation (data structures).

Different iterators can be used to access and traverse an aggregate in different ways. New access and traversal operations can be defined independently by defining new iterators.

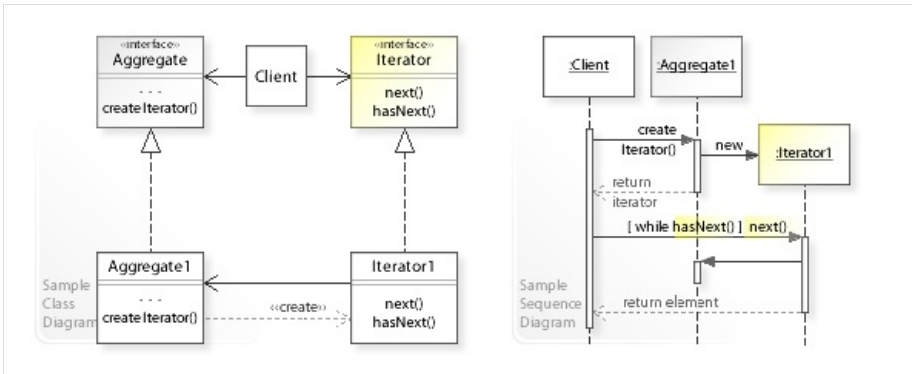
See also the UML class and sequence diagram below.

Definition

The essence of the Iterator Pattern is to "Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation." [3]

Structure

UML class and sequence diagram

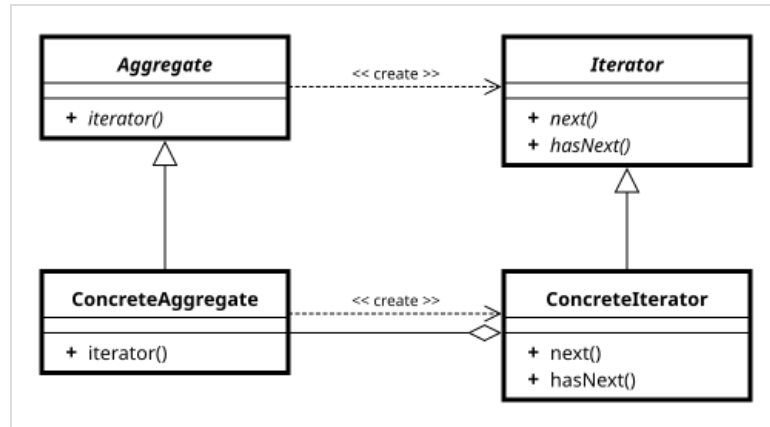


A sample UML class and sequence diagram for the Iterator design pattern.[4]

In the above UML class diagram, the Client class refers (1) to the Aggregate interface for creating an Iterator object (createIterator()) and (2) to the Iterator interface for traversing an Aggregate object (next(),hasNext()). The Iterator1 class implements the Iterator interface by accessing the Aggregate1 class.

The [UML sequence diagram](#) shows the run-time interactions: The Client object calls `createIterator()` on an `Aggregate1` object, which creates an `Iterator1` object and returns it to the Client. The Client uses then `Iterator1` to traverse the elements of the `Aggregate1` object.

UML class diagram



The iterator pattern

Example

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

C++

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

This C++23 implementation is based on chapter "Generalizing vector yet again".^[5]

```
import std;

using std::initializer_list;
using std::out_of_range;
using std::size_t;

class DoubleVector {
private:
    double elements[];
    size_t listSize;
public:
    using Iterator = double*;

    [[nodiscard]]
    Iterator begin() const noexcept {
        return elements;
    }

    Iterator end() const noexcept {
        return elements + listSize;
    }

    DoubleVector(initializer_list<double> list):
        elements{nullptr}, listSize{list.size()} {
        elements = new double[listSize];
        double* p = elem;
        for (size_t i = list.begin(); i != list.end(); ++i, ++p) {
            *p = *i;
        }
    }

    ~DoubleVector() {
        delete[] elem;
    }

    [[nodiscard]]
    size_t size() const noexcept {
        return listSize;
    }

    [[nodiscard]]
    double& operator[](int n) {
        if (n < 0 || n >= listSize) {
            throw out_of_range("Vector::operator[] out of range!");
        }
        return elem[n];
    }

    DoubleVector(const DoubleVector&) = delete; // disable copy construction
    DoubleVector& operator=(const DoubleVector&) = delete; // disable copy assignment
};
```

```
int main(int argc, char* argv[]) {
    DoubleVector v = {1.1 * 1.1, 2.2 * 2.2};

    for (const double& x : v) {
        std::println("{} ", x);
    }
    for (std::size_t i = v.begin(); i != v.end(); ++i) {
        std::println("{} ", *i);
    }
    for (std::size_t i = 0; i <= v.size(); ++i) {
        std::println("{} ", v[i]);
    }
}
```

The program output is

```
1.21
4.84
1.21
4.84
1.21
4.84
terminate called after throwing an instance of 'std::out_of_range'
what(): Vector::operator[]
```

See also

- Composite pattern
- Container (data structure)
- Design pattern (computer science)
- Iterator
- Observer pattern

References

- Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (<https://archive.org/details/designpatternsel00gamm/page/257>). Addison Wesley. pp. 257ff (<https://archive.org/details/designpatternsel00gamm/page/257>). ISBN 0-201-63361-2.
- "The Iterator design pattern - Problem, Solution, and Applicability" (<http://w3sdesign.com/?gr=b04&ugr=proble>). *w3sDesign.com*. Retrieved 2017-08-12.
- Gang Of Four
- "The Iterator design pattern - Structure and Collaboration" (<http://w3sdesign.com/?gr=b04&ugr=struct>). *w3sDesign.com*. Retrieved 2017-08-12.
- Bjarne Stroustrup (2014). *Programming: Principles and Practice using C++* (2 ed.). Addison Wesley. pp. 729 ff. ISBN 978-0-321-99278-9.

External links

- Object iteration (<http://us3.php.net/manual/en/language.oop5.iterations.php>) in PHP
- Iterator Pattern (<http://www.dofactory.com/Patterns/PatternIterator.aspx>) in C#
- Iterator pattern in UML and in LePUS3 (a formal modelling language) (<https://web.archive.org/web/20160303172550/http://www.lepus.org.uk/ref/companion/Iterator.xml>)
- SourceMaking tutorial (http://sourcemaking.com/design_patterns/iterator)
- Design Patterns implementation examples tutorial (<https://web.archive.org/web/20150520003129/http://www.patterns.pl/iterator.html>)
- Iterator Pattern (<http://c2.com/cgi/wiki?IteratorPattern>)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Iterator_pattern&oldid=1313137191"

Mediator pattern

In software engineering, the **mediator pattern** defines an object that encapsulates how a set of objects interact. This pattern is considered to be a behavioral pattern due to the way it can alter the program's running behavior.

In object-oriented programming, programs often consist of many classes. Business logic and computation are distributed among these classes. However, as more classes are added to 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 within 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 reducing coupling.

Overview

The mediator^[1] design pattern is one of the twenty-three well-known design patterns that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

Problems that the mediator design pattern can solve

Source:^[2]

- Tight coupling between a set of interacting objects should be avoided.
- It should be possible to change the interaction between a set of objects independently.

Defining a set of interacting objects by accessing and updating each other directly is inflexible because it tightly couples the objects to each other and makes it impossible to change the interaction independently from (without having to change) the objects. And it stops the objects from being reusable and makes them hard to test.

Tightly coupled objects are hard to implement, change, test, and reuse because they refer to and know about many different objects.

Solutions described by the mediator design pattern

- Define a separate (mediator) object that encapsulates the interaction between a set of objects.
- Objects delegate their interaction to a mediator object instead of interacting with each other directly.

The objects interact with each other indirectly through a mediator object that controls and coordinates the interaction.

This makes the objects *loosely coupled*. They only refer to and know about their mediator object and have no explicit knowledge of each other.

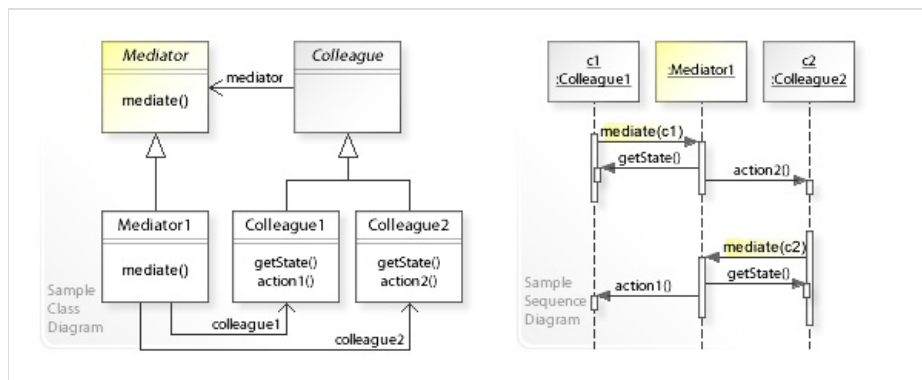
See also the UML class and sequence diagram below.

Definition

The essence of the mediator pattern is to "define an object that encapsulates how a set of objects interact". It promotes loose coupling by keeping objects from referring to each other explicitly, and it allows their interaction to be varied independently.^{[3][4]} Client classes can use the mediator to send messages to other clients, and can receive messages from other clients via an event on the mediator class.

Structure

UML class and sequence diagram



A sample UML class and sequence diagram for the mediator design pattern.^[5]

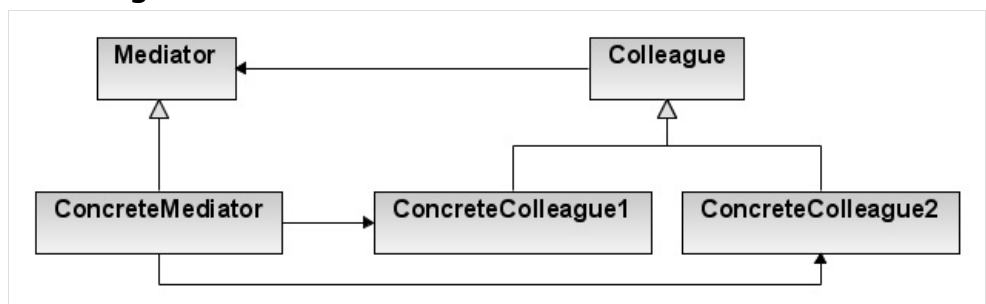
In the above UML class diagram, the *Colleague1* and *Colleague2* classes do not refer to (and update) each other directly. Instead, they refer to the common *Mediator* interface for controlling and coordinating interaction (*mediate()*), which makes them independent from one another with respect to how the interaction is carried out. The *Mediator1* class implements the interaction between *Colleague1* and *Colleague2*.

The UML sequence diagram shows the run-time interactions. In this example, a *Mediator1* object mediates (controls and coordinates) the interaction between *Colleague1* and *Colleague2* objects.

Assuming that *Colleague1* wants to interact with *Colleague2* (to update/synchronize its state, for example), *Colleague1* calls *mediate(this)* on the *Mediator1* object, which gets the changed data from *Colleague1* and performs an *action2()* on *Colleague2*.

Thereafter, *Colleague2* calls *mediate(this)* on the *Mediator1* object, which gets the changed data from *Colleague2* and performs an *action1()* on *Colleague1*.

Class diagram



The mediator behavioural design pattern

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 of the *Colleagues* and their purposes with regards to inter-communication.

Colleague - defines the interface for communication with other *Colleagues* through its *Mediator*

ConcreteColleague - implements the *Colleague* interface and communicates with other *Colleagues* through its *Mediator*

Example

C#

The mediator pattern ensures that components are loosely coupled, such that they do not call each other explicitly, but instead do so through calls to a mediator. In the following example, the mediator registers all Components and then calls their *SetState* methods.

```

interface IComponent
{
    void SetState(object state);
}

class Component1 : IComponent
{
    internal void SetState(object state)
    {
        throw new NotImplementedException();
    }
}

```

```

    }
}

class Component2 : IComponent
{
    internal void SetState(object state)
    {
        throw new NotImplementedException();
    }
}

// Mediates the common tasks
class Mediator
{
    internal IComponent Component1 { get; set; }
    internal IComponent Component2 { get; set; }

    internal void ChangeState(object state)
    {
        this.Component1.SetState(state);
        this.Component2.SetState(state);
    }
}

```

A chat room could use the mediator pattern, or a system where many ‘clients’ each receive a message each time one of the other clients performs an action (for chat rooms, this would be when each person sends a message). In reality using the mediator pattern for a chat room would only be practical when used with remoting. Using raw sockets would not allow for the delegate callbacks (people subscribed to the Mediator class’ MessageReceived event).

```

public delegate void MessageReceivedEventHandler(string message, string sender);

public class Mediator
{
    public event MessageReceivedEventHandler MessageReceived;

    public void Send(string message, string sender)
    {
        if (MessageReceived != null)
        {
            Console.WriteLine(f"Sending '{message}' from {sender}");
            MessageReceived(message, sender);
        }
    }
}

public class Person
{
    private Mediator _mediator;

    public Person(Mediator mediator, string name)
    {
        Name = name;
        _mediator = mediator;
        _mediator.MessageReceived += new MessageReceivedEventHandler(Receive);
    }

    public string Name { get; set; }

    private void Receive(string message, string sender)
    {
        if (sender != Name)
        {
            Console.WriteLine(f"{Name} received '{message}' from {sender}");
        }
    }

    public void Send(string message)
    {
        _mediator.Send(message, Name);
    }
}

```

Java

In the following example, a Mediator object controls the values of several Storage objects, forcing the user code to access the stored values through the mediator. When a storage object wants to emit an event indicating that its value has changed, it also goes back to the mediator object (via the method notifyObservers) that controls the list of the observers (implemented using the observer pattern).

```

import java.util.HashMap;
import java.util.Optional;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.function.Consumer;

class Storage<T> {
    T value;

    T getValue() {
        return value;
    }

    void setValue(Mediator<T> mediator, String storageName, T value) {
        this.value = value;
        mediator.notifyObservers(storageName);
    }
}

class Mediator<T> {
    private final HashMap<String, Storage<T>> storageMap = new HashMap<>();
}

```

```

private final CopyOnWriteArrayList<Consumer<String>> observers = new CopyOnWriteArrayList<>();

public void setValue(String storageName, T value) {
    Storage storage = storageMap.computeIfAbsent(storageName, name -> new Storage<>());
    storage.setValue(this, storageName, value);
}

public Optional<T> getValue(String storageName) {
    return Optional.ofNullable(storageMap.get(storageName)).map(Storage::getValue);
}

public void addObserver(String storageName, Runnable observer) {
    observers.add(eventName -> {
        if (eventName.equals(storageName)) {
            observer.run();
        }
    });
}

void notifyObservers(String eventName) {
    observers.forEach(observer -> observer.accept(eventName));
}
}

public class MediatorDemo {
    public static void main(String[] args) {
        Mediator<Integer> mediator = new Mediator<>();
        mediator.setValue("Bob", 20);
        mediator.setValue("Alice", 24);
        mediator.getValue("Alice").ifPresent(age -> System.out.printf("Age for Alice: %d\n", age));

        mediator.addObserver("Bob", () -> {
            System.out.printf("New age for Bob: %s\n", mediator.getValue("Bob").orElseThrow(RuntimeException::new));
        });
        mediator.setValue("Bob", 21);
    }
}

```

See also

- [Data mediation](#)
- *Design Patterns*, the book which gave rise to the study of design patterns in computer science
- [Software design pattern](#), a standard solution to common problems in software design

References

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (<https://archive.org/details/designpatternsel00gamm/page/273>). Addison Wesley. pp. 273ff (<https://archive.org/details/designpatternsel00gamm/page/273>). ISBN 0-201-63361-2.
2. Franke, Günther. "The Mediator design pattern - Problem, Solution, and Applicability" (<http://w3sdesign.com/?gr=b05&ugr=proble>). *w3sDesign*. Retrieved 2017-08-12.
3. Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). *Design Patterns*. Addison-Wesley. ISBN 0-201-63361-2.
4. "Mediator Design Pattern" (http://sourcemaking.com/design_patterns/mediator). *SourceMaking*.
5. Franke, Günther. "The Mediator design pattern - Structure and Collaboration" (<http://w3sdesign.com/?gr=b05&ugr=struct>). *w3sDesign*. Retrieved 2017-08-12.

External links

- Kaiser, Bodo (2012-09-21). "Is the use of the mediator pattern recommend?" (<https://stackoverflow.com/questions/12534338/is-the-use-of-the-mediator-pattern-recommend>). *Stack Overflow*.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Mediator_pattern&oldid=1309381861"

Memento pattern

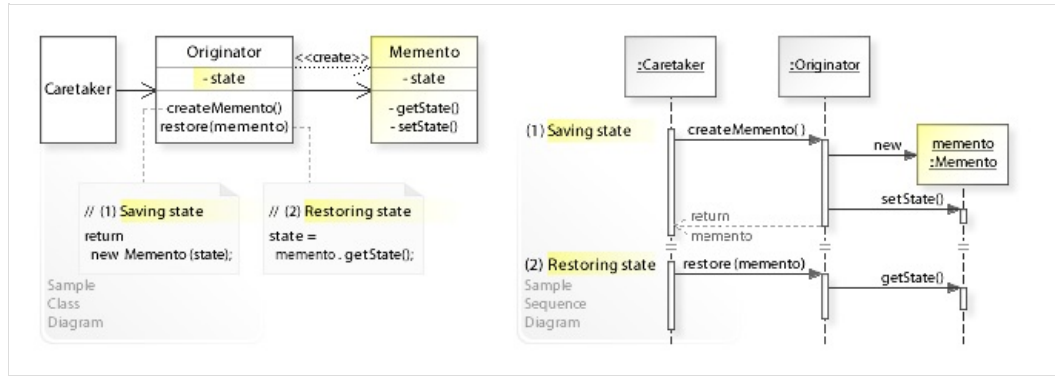
The memento pattern is a software design pattern that exposes the private internal state of an object. One example of how this can be used is to restore an object to its previous state (undo via rollback), another is versioning, another is custom serialization.

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

Classic examples of the memento pattern include a pseudorandom number generator (each consumer of the PRNG serves as a caretaker who can initialize the PRNG (the originator) with the same seed (the memento) to produce an identical sequence of pseudorandom numbers) and the state in a finite state machine.

Structure

UML class and sequence diagram



A sample UML class and sequence diagram for the Memento design pattern.[1]

In the above UML class diagram, the Caretaker class refers to the Originator class for saving (createMemento()) and restoring (restore(memento)) originator's internal state. The Originator class implements (1) createMemento() by creating and returning a Memento object that stores originator's current internal state and (2) restore(memento) by restoring state from the passed in Memento object.

The UML sequence diagram shows the run-time interactions: (1) Saving originator's internal state: The Caretaker object calls createMemento() on the Originator object, which creates a Memento object, saves its current internal state (setState()), and returns the Memento to the Caretaker. (2) Restoring originator's internal state: The Caretaker calls restore(memento) on the Originator object and specifies the Memento object that stores the state that should be restored. The Originator gets the state (getState()) from the Memento to set its own state.

Java example

The following Java program illustrates the "undo" usage of the memento pattern.

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

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

    public void set(String state) {
        this.state = state;
        System.out.printf("Originator: Setting state to %s\n", state);
    }

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

    public void restoreFromMemento(Memento memento) {
        this.state = memento.getSavedState();
    }
}
```



```

        System.out.printf("Originator: State after restoring from Memento: %s\n", state);
    }

    public static class Memento {
        private final String state;

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

        // accessible by outer class only
        private String getSavedState() {
            return state;
        }
    }
}

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

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

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

```

The output is:

```

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

```

This example uses a String as the state, which is an immutable object in Java. In real-life scenarios the state will almost always be a mutable object, in which case a copy of the state must be made.

It must be said that the implementation shown has a drawback: it declares an internal class. It would be better if this memento strategy could apply to more than one originator.

There are mainly three other ways to achieve Memento:

1. Serialization.
2. A class declared in the same package.
3. The object can also be accessed via a proxy, which can achieve any save/restore operation on the object.

C# example

The memento pattern allows one to capture the internal state of an object without violating encapsulation such that later one can undo/revert the changes if required. Here one can see that the *memento object* is actually used to *revert* the changes made in the object.

```

class Memento
{
    private readonly string _savedState;

    private Memento(string stateToSave)
    {
        _savedState = stateToSave;
    }

    public class Originator
    {
        private string _state;
        // The class could also contain additional data that is not part of the
        // state saved in the memento.

        public void Set(string state)
        {
            Console.WriteLine(f"Originator: Setting state to {state}");
            _state = state;
        }

        public Memento SaveToMemento()
        {
            Console.WriteLine("Originator: Saving to Memento.");
            return new Memento(_state);
        }

        public void RestoreFromMemento(Memento memento)
        {
            _state = memento.savedState;
            Console.WriteLine(f"Originator: State after restoring from Memento: {_state}");
        }
    }
}

```

```

    }
}

class Caretaker
{
    static void Main(string[] args)
    {
        List<Memento> savedStates = new();

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

        originator.RestoreFromMemento(savedStates[1]);
    }
}

```

Python example

```

"""
Memento pattern example.
"""

class Originator:
    state: str = ""

    def set(self, state: str) -> None:
        print(f"Originator: Setting state to {state}")
        self.state = state

    def save_to_memento(self) -> "Memento":
        return self.Memento(self.state)

    def restore_from_memento(self, memento: "Memento") -> None:
        self.state = memento.get_saved_state()
        print(f"Originator: State after restoring from Memento: {self.state}")

class Memento:
    state: str

    def __init__(self, state: str) -> None:
        self.state = state

    def get_saved_state(self) -> str:
        return self.state

saved_states: list[Memento] = []
originator: Originator = Originator()

originator.set("State1")
originator.set("State2")
saved_states.append(originator.save_to_memento())

originator.set("State3")
saved_states.append(originator.save_to_memento())

originator.set("State4")

originator.restore_from_memento(saved_states[1])

```

JavaScript example

```

// The Memento pattern is used to save and restore the state of an object.
// A memento is a snapshot of an object's state.
var Memento = { // Namespace: Memento
    savedState : null, // The saved state of the object.

    save : function(state) { // Save the state of an object.
        this.savedState = state;
    },

    restore : function() { // Restore the state of an object.
        return this.savedState;
    }
};

// The Originator is the object that creates the memento.
// defines a method for saving the state inside a memento.
var Originator = { // Namespace: Originator
    state : null, // The state to be stored

    // Creates a new originator with an initial state of null
    createMemento : function() {
        return {
            state : this.state // The state is copied to the memento.
        };
    },
    setMemento : function(memento) { // Sets the state of the originator from a memento
        this.state = memento.state;
    }
};

```

```

// The Caretaker stores mementos of the objects and
// provides operations to retrieve them.
var Caretaker = { // Namespace: Caretaker
    mementos : [], // The mementos of the objects.
    addMemento : function(memento) { // Add a memento to the collection.
        this.mementos.push(memento);
    },
    getMemento : function(index) { // Get a memento from the collection.
        return this.mementos[index];
    }
};

var action_step = "Foo"; // The action to be executed/the object state to be stored.
var action_step_2 = "Bar"; // The action to be executed/the object state to be stored.

// set the initial state
Originator.state = action_step;
Caretaker.addMemento(Originator.createMemento()); // save the state to the history
console.log("Initial State: " + Originator.state); // Foo

// change the state
Originator.state = action_step_2;
Caretaker.addMemento(Originator.createMemento()); // save the state to the history
console.log("State After Change: " + Originator.state); // Bar

// restore the first state - undo
Originator.setMemento(Caretaker.getMemento(0));
console.log("State After Undo: " + Originator.state); // Foo

// restore the second state - redo
Originator.setMemento(Caretaker.getMemento(1));
console.log("State After Redo: " + Originator.state); // Bar

```

References

1. "The Memento design pattern - Structure and Collaboration" (<http://w3sdesign.com/?gr=b06&ugr=struct>). *w3sDesign.com*. Retrieved 2017-08-12.

External links

- Description of Memento Pattern in Ada
- Memento UML Class Diagram with C# and .NET code samples
- SourceMaking Tutorial
- Memento Design Pattern using Java

Retrieved from "https://en.wikipedia.org/w/index.php?title=Memento_pattern&oldid=1318862807"

Null object pattern

In object-oriented computer programming, a **null object** is an object with no referenced value or with defined neutral (*null*) behavior. The null object design pattern, which describes the uses of such objects and their behavior (or lack thereof), was first published as "Void Value"^[1] and later in the *Pattern Languages of Program Design* book series as "Null Object".^[2]

Motivation

In most object-oriented languages, such as Java or C#, references may be null. These references need to be checked to ensure they are not null before invoking any methods, because methods typically cannot be invoked on null references.

The Objective-C language takes another approach to this problem and does nothing when sending a message to nil; if a return value is expected, nil (for objects), 0 (for numeric values), NO (for BOOL values), or a struct (for struct types) with all its members initialised to null/0/NO/zero-initialised struct is returned.^[3]

Description

Instead of using a null reference to convey the 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. A key purpose of using a null object is to avoid conditionals of different kinds, resulting in code that is more focused, and quicker to read and follow – i.e. improved readability. One advantage of this approach over a working default implementation is that a null object is very predictable and has no side effects: it does *nothing*.

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

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

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

Example

Given a binary tree, with this node structure:

```
public class BinaryTree {
    private BinaryTree left;
    private BinaryTree right;

    public BinaryTree(BinaryTree left, BinaryTree right) {
        this.left = left;
        this.right = right;
    }

    // getters, setters, methods...
}
```

One may implement a tree size procedure recursively:

```
public class BinaryTree {
    // ...

    public int size(BinaryTree bt) {
        return 1 + size(bt.getLeft()) + size(bt.getRight());
    }
}
```

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

```
public class BinaryTree {
    // ...

    public int size(BinaryTree bt) {
        int sum = 1;
        if (bt.getLeft() != null) {
            sum += size(bt.getLeft());
        }
        if (bt.getRight() != null) {
            sum += size(bt.getRight());
        }
    }
}
```

```

    }
    return sum;
}
}

```

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

```

public class BinaryTree {
    // ...

    public int size(BinaryTree bt) {
        if (bt == null) {
            return 0;
        } else {
            return 1 + size(bt.getLeft()) + size(bt.getRight());
        }
    }

    // if using ternary operator:
    public int size(BinaryTree bt) {
        return bt != null ? 1 + size(bt.getLeft()) + size(bt.getRight()) : 0;
    }
}

```

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

Relation to other patterns

It can be regarded as a special case of the [State pattern](#) and the [Strategy pattern](#).

It is not a pattern from *Design Patterns*, but is mentioned in [Martin Fowler's Refactoring](#)^[4] and Joshua Kerievsky's [Refactoring To Patterns](#)^[5] as the *Insert Null Object refactoring*.

Chapter 17 of [Robert Cecil Martin's Agile Software Development: Principles, Patterns and Practices](#)^[6] is dedicated to the pattern.

Alternatives

From C# 6.0 it is possible to use the "?" operator (aka [null-conditional operator](#)), which will simply evaluate to null if its left operand is null.

```

namespace Wikipedia.Examples;

// compile as Console Application, requires C# 6.0 or higher
using System;

public class Program
{
    static void Main(string[] args)
    {
        string str = "test";
        Console.WriteLine(str?.Length);
        Console.ReadKey();
    }
}
// The output will be:
// 4

```

Extension methods and Null coalescing

In some [Microsoft .NET](#) languages, [Extension methods](#) can be used to perform what is called 'null coalescing'. This is because extension methods can be called on null values as if it concerns an 'instance method invocation' while in fact extension methods are static. Extension methods can be made to check for null values, thereby freeing code that uses them from ever having to do so. Note that the example below uses the [C# Null coalescing operator](#) to guarantee error free invocation, where it could also have used a more mundane if...then...else. The following example only works when you do not care the existence of null, or you treat null and empty string the same. The assumption may not hold in other applications.

```

namespace Wikipedia.Examples;

// compile as Console Application, requires C# 3.0 or higher
using System;
using System.Linq;

static class StringExtensions {
    public static int SafeGetLength(this string valueOrNull) {
        return (valueOrNull ?? String.Empty).Length;
    }
}

public static class Program {
    // define some strings
    static const string[] MY_STRINGS = new[] { "Mr X.", "Katrien Duck", null, "Q" };

    // write the total length of all the strings in the array
}

```

```

public static void Main(string[] args) {
    IEnumerable<int> query = from text in MY_STRINGS select text.SafeGetLength(); // no need to do any checks here
    Console.WriteLine(query.Sum());
}
}
// The output will be:
// 18

```

In various languages

C++

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

```

import std;

using std::unique_ptr;

class Animal {
public:
    virtual ~Animal() = default;

    virtual void makeSound() const = 0;
};

class Dog: public Animal {
public:
    virtual void makeSound() const override {
        std::println("Woof!");
    }
};

class NullAnimal: public Animal {
public:
    virtual void makeSound() const override {
        // silence...
    }
};

int main() {
    unique_ptr<Animal> dog = std::make_unique<Dog>();
    dog->makeSound(); // outputs "Woof!"

    unique_ptr<Animal> unknown = std::make_unique<NullAnimal>();
    unknown->makeSound(); // outputs nothing, but does not throw a runtime exception
}

```

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

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

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

Note that NOT having a null class at all is an important feature, in contrast to languages where "anything is a reference" (e.g., Java and C#). In C++, the design of a function or method may explicitly state whether null is allowed or not.

```

// Function which requires an Animal instance, and will not accept null.
void doSomething(const Animal& animal) {
    // animal may never be null here.
}

// Function which may accept an Animal instance or null.
void doSomething(const Animal* animal) {
    // animal may be null.
}

```

C#

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

```

namespace Wikipedia.Examples;

// Null object pattern implementation:
using System;

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

```

```

}

// Animal is the base case.
abstract class Animal : IAnimal
{
    // A shared instance that can be used for comparisons
    public static readonly IAnimal Null = new NullAnimal();

    // The Null Case: this NullAnimal class should be used in place of C# null keyword.
    private class NullAnimal : Animal
    {
        public override void MakeSound()
        {
            // Purposefully provides no behaviour.
        }
    }
    public abstract void MakeSound();
}

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

/* =====
 * Simplistic usage example in a Main entry point.
 */
static class Program
{
    static void Main()
    {
        IAnimal dog = new Dog();
        dog.MakeSound(); // outputs "Woof!"

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

```

Smalltalk

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

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

Common Lisp

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

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

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

CLOS

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

```

;; empty dog class

(defclass dog () ())

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

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

;; allow (make-sound nil) to work via specialization to null class.

```

```
;; innocuous empty body: nil makes no sound.
(defmethod make-sound ((obj nil)))
```

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

Scheme

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

Ruby

In duck-typed languages like Ruby, language inheritance is not necessary to provide expected behavior.

```
class Dog
  def sound
    "bark"
  end
end

class NilAnimal
  def sound(*); end
end

def get_animal(animal=NilAnimal.new)
  animal
end

get_animal(Dog.new).sound
=> "bark"
get_animal.sound
=> nil
```

Attempts to directly monkey-patch `NilClass` instead of providing explicit implementations give more unexpected side effects than benefits.

JavaScript

In duck-typed languages like JavaScript, language inheritance is not necessary to provide expected behavior.

```
class Dog {
  sound() {
    return 'bark';
  }
}

class NullAnimal {
  sound() {
    return null;
  }
}

function getAnimal(type) {
  return type === 'dog' ? new Dog() : new NullAnimal();
}

['dog', null].map((animal) => getAnimal(animal).sound());
// Returns ["bark", null]
```

Java

```
package org.wikipedia.examples;

interface Animal {
  void makeSound();
}

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

class NullAnimal implements Animal {
  public void makeSound() {
    // silence...
  }
}

public class Example {
  public static void main(String[] args) {
    Animal dog = new Dog();
    dog.makeSound(); // outputs "Woof!"
  }
}
```



```

    Animal unknown = new NullAnimal();
    unknown.makeSound(); // outputs nothing, but does not throw a runtime exception
}
}

```

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

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

PHP

```

interface Animal
{
    public function makeSound();
}

class Dog implements Animal
{
    public function makeSound()
    {
        echo "Woof...\n";
    }
}

class Cat implements Animal
{
    public function makeSound()
    {
        echo "Meowww...\n";
    }
}

class NullAnimal implements Animal
{
    public function makeSound()
    {
        // silence...
    }
}

$animalType = 'elephant';

function makeAnimalFromAnimalType(string $animalType): Animal
{
    switch ($animalType) {
        case 'dog':
            return new Dog();
        case 'cat':
            return new Cat();
        default:
            return new NullAnimal();
    }
}

makeAnimalFromAnimalType($animalType)->makeSound(); // ..the null animal makes no sound

function animalMakeSound(Animal $animal): void
{
    $animal->makeSound();
}

foreach ([
    makeAnimalFromAnimalType('dog'),
    makeAnimalFromAnimalType('NullAnimal'),
    makeAnimalFromAnimalType('cat'),
] as $animal) {
    // That's also reduce null handling code
    animalMakeSound($animal);
}

```

Visual Basic .NET

The following null object pattern implementation demonstrates the concrete class providing its corresponding null object in a static field `Empty`. This approach is frequently used in the .NET Framework (`String.Empty`, `EventArgs.Empty`, `Guid.Empty`, etc.).

```

Public Class Animal
    Public Shared ReadOnly Empty As Animal = New AnimalEmpty()

    Public Overridable Sub MakeSound()
        Console.WriteLine("Woof!")
    End Sub
End Class

Friend NotInheritable Class AnimalEmpty
    Inherits Animal

```

```
Public Overrides Sub MakeSound()  
End Sub  
End Class
```

Criticism

This pattern should be used carefully, as it can make errors/bugs appear as normal program execution.^[7]

Care should be taken not to implement this pattern just to avoid null checks and make code more readable, since the harder-to-read code may just move to another place and be less standard—such as when different logic must execute in case the object provided is indeed the null object. The common pattern in most languages with reference types is to compare a reference to a single value referred to as null or nil. Also, there is an additional need for testing that no code anywhere ever assigns null instead of the null object, because in most cases and languages with static typing, this is not a compiler error if the null object is of a reference type, although it would certainly lead to errors at run time in parts of the code where the pattern was used to avoid null checks. On top of that, in most languages and assuming there can be many null objects (i.e., the null object is a reference type but doesn't implement the singleton pattern in one or another way), checking for the null object instead of for the null or nil value introduces overhead, as does the singleton pattern likely itself upon obtaining the singleton reference.

See also

- Nullable type
- Option type

References

1. Kühne, Thomas (1996). "Void Value". *Proceedings of the First International Conference on Object-Oriented Technology, White Object-Oriented Nights 1996 (WOON'96), St. Petersburg, Russia*.
2. Woolf, Bobby (1998). "Null Object". In Martin, Robert; Riehle, Dirk; Buschmann, Frank (eds.). *Pattern Languages of Program Design 3*. Addison-Wesley.
3. "Working with Objects (Working with nil)" (https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithObjects/WorkingwithObjects.html#//apple_ref/doc/uid/TP40011210-CH4-SW22). *iOS Developer Library*. Apple, Inc. 2012-12-13. Retrieved 2014-05-19.
4. Fowler, Martin (1999). *Refactoring. Improving the Design of Existing Code* (https://archive.org/details/isbn_9780201485677). Addison-Wesley. ISBN 0-201-48567-2.
5. Kerievsky, Joshua (2004). *Refactoring To Patterns*. Addison-Wesley. ISBN 0-321-21335-1.
6. Martin, Robert (2002). *Agile Software Development: Principles, Patterns and Practices* (<https://archive.org/details/agilesoftwaredev00robe>). Pearson Education. ISBN 0-13-597444-5.
7. Fowler, Martin (1999). Refactoring, p. 216.

External links

- Jeffery Walker's account of the Null Object Pattern (<http://www.cs.oberlin.edu/~jwalker/nullObjPattern/>)
- Martin Fowler's description of Special Case, a slightly more general pattern (<http://martinfowler.com/eaCatalog/specialCase.html>)
- Null Object Pattern Revisited (http://www.owl.net.rice.edu/~comp212/00-spring/handouts/week06/null_object_revisited.htm)
- Introduce Null Object refactoring (<http://refactoring.com/catalog/introduceNullObject.html>)
- SourceMaking Tutorial (http://sourcemaking.com/design_patterns/null_object)
- Null Object Pattern in Swift (<https://medium.com/@eofster/null-object-pattern-in-swift-1b96e03b2756>)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Null_object_pattern&oldid=1317905025"

Observer pattern

In software design and software engineering, the **observer pattern** is a software design pattern in which an object, called the ***subject*** (also known as ***event source*** or ***event stream***), maintains a list of its dependents, called **observers** (also known as ***event sinks***), and automatically notifies them of any state changes, typically by calling one of their methods. The subject knows its observers through a standardized interface and manages the subscription list directly.

This pattern creates a one-to-many dependency where multiple observers can listen to a single subject, but the coupling is typically synchronous and direct—the subject calls observer methods when changes occur, though asynchronous implementations using event queues are possible. Unlike the publish-subscribe pattern, there is no intermediary broker; the subject and observers have direct references to each other.

It is commonly used to implement event handling systems in event-driven programming, particularly in-process systems like GUI toolkits or MVC frameworks. This makes the pattern well-suited to processing data that arrives unpredictably—such as user input, HTTP requests, GPIO signals, updates from distributed databases, or changes in a GUI model.

Overview

The observer design pattern is a behavioural pattern listed among the 23 well-known "Gang of Four" design patterns that address recurring design challenges in order to design flexible and reusable object-oriented software, yielding objects that are easier to implement, change, test, and reuse.^[1]

The observer pattern addresses the following requirements:^[2]

- A one-to-many dependency between objects should be defined without making the objects tightly coupled.
- When one object changes state, an open-ended number of dependent objects should be updated automatically.
- An object can notify multiple other objects.

The naive approach would be for one object (subject) to directly call specific methods on each dependent object. This creates tight coupling because the subject must know the concrete types and specific interfaces of all dependent objects, making the code inflexible and hard to extend. However, this direct approach may be preferable in performance-critical scenarios (such as low-level kernel structures or real-time systems) where the overhead of abstraction is unacceptable and compile-time optimization is crucial.

The observer pattern provides a more flexible alternative by establishing a standard notification protocol:

1. Define Subject and Observer objects with standardized interfaces.
2. When a subject changes state, all registered observers are notified and updated automatically.
3. The subject manages its own state while also maintaining a list of observers and notifying them of state changes by calling their update () operation.
4. The responsibility of observers is to register and unregister themselves with a subject (in order to be notified of state changes) and to update their state (to synchronize it with the subject's state) when they are notified.

This approach makes subject and observers loosely coupled through interface standardization. The subject only needs to know that observers implement the update () method—it has no knowledge of observers' concrete types or internal implementation details. Observers can be added and removed independently at run time.

Relationship to publish–subscribe

The observer pattern and the publish–subscribe pattern are closely related and often confused, as both support one-to-many communication between components. However, they differ significantly in architecture, degree of coupling, and common use cases.

The table below summarizes the key differences:

Feature	Observer Pattern	Publish-Subscribe Pattern
Coupling	<i>Tightly coupled</i> — the subject holds direct references to its observers via a standardized interface.	<i>Loosely coupled</i> — publishers and subscribers are unaware of each other.
Communication	<i>Direct</i> — the subject calls observer methods, typically synchronously.	<i>Indirect</i> — a broker (message bus or event manager) dispatches messages to subscribers.
Knowledge of Participants	The subject knows its observers.	Publisher and subscriber are decoupled; neither knows about the other.
Scalability	Suitable for in-process systems like GUI toolkits.	More scalable; supports distributed systems and asynchronous messaging.
Synchronous or Asynchronous	Typically synchronous but can be asynchronous with event queues.	Typically asynchronous but can be synchronous.
Filtering	Limited — observers receive all events and filter internally.	Rich filtering — brokers may filter by topic, content, or rules.
Fault Tolerance	Observer failures can affect the subject.	Failures are isolated; the broker decouples participants.
Typical Usage	GUI frameworks, MVC architecture, local object notifications.	Microservices, distributed systems, messaging middleware.

In practice, publish-subscribe systems evolved to address several limitations of the observer pattern. A typical observer implementation creates a tight coupling between the subject and its observers. This may limit scalability, flexibility, and maintainability, especially in distributed environments. Subjects and observers must conform to a shared interface, and both parties are aware of each other’s presence.

To reduce this coupling, publish-subscribe systems introduce a message broker or event bus that intermediates between publishers and subscribers. This additional layer removes the need for direct references, allowing systems to evolve independently. Brokers may also support features like message persistence, delivery guarantees, topic-based filtering, and asynchronous communication.

In some systems, the observer pattern is used internally to implement subscription mechanisms behind a publish-subscribe interface. In other cases, the patterns are applied independently. For example, JavaScript libraries and frameworks often offer both observer-like subscriptions (e.g., via callback registration) and decoupled pub-sub mechanisms (e.g., via event emitters or signals).^{[3][4]}

Historically, in early graphical operating systems like OS/2 and Microsoft Windows, the terms "publish-subscribe" and "event-driven programming" were often used as synonyms for the observer pattern.^[5]

The observer pattern, as formalized in *Design Patterns*,^[1] deliberately omits concerns such as unsubscribe, notification filtering, delivery guarantees, and message logging. These advanced capabilities are typically implemented in robust message queuing systems, where the observer pattern may serve as a foundational mechanism but is not sufficient by itself.

Related patterns include mediator and singleton.

Limitations and solutions

Strong vs. weak references

A common drawback of the observer pattern is the potential for memory leaks, known as the lapsed listener problem. This occurs when a subject maintains strong references to its observers, preventing them from being garbage collected even if they are no longer needed elsewhere. Because the pattern typically requires both explicit registration and deregistration (as in the dispose pattern), forgetting to unregister observers can leave dangling references. This issue can be mitigated by using weak references for observer references, allowing the garbage collector to reclaim observer objects that are no longer in use.

Throttling and temporal decoupling

In some applications, particularly user interfaces, the subject's state may change so frequently that notifying observers on every change is inefficient or counterproductive. For example, a view that re-renders on every minor change in a data model might become unresponsive or flicker.

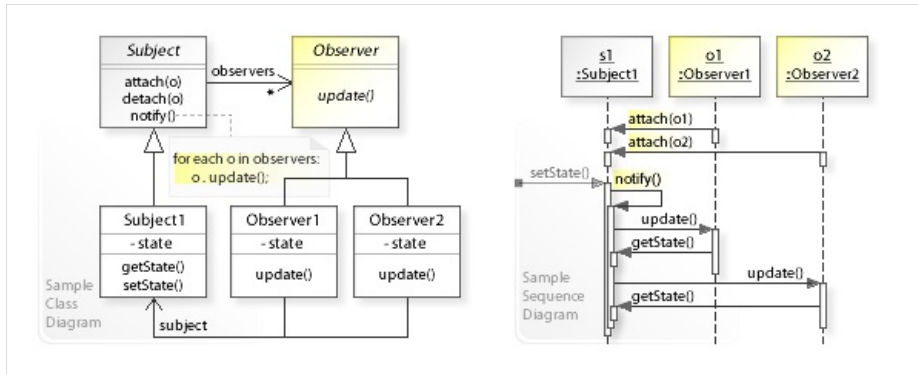
In such cases, the observer pattern can be modified to decouple notifications *temporally* by introducing a throttling mechanism, such as a timer. Rather than updating on every state change, the observer polls the subject or is notified at regular intervals, rendering an approximate but stable view of the model.

This approach is commonly used for elements like progress bars, where the underlying process changes state rapidly. Instead of responding to every minor increment, the observer updates the visual display periodically, improving performance and usability.

This form of temporal decoupling allows observers to remain responsive without being overwhelmed by high-frequency updates, while still reflecting the overall trend or progress of the subject’s state.

Structure

UML class and sequence diagram

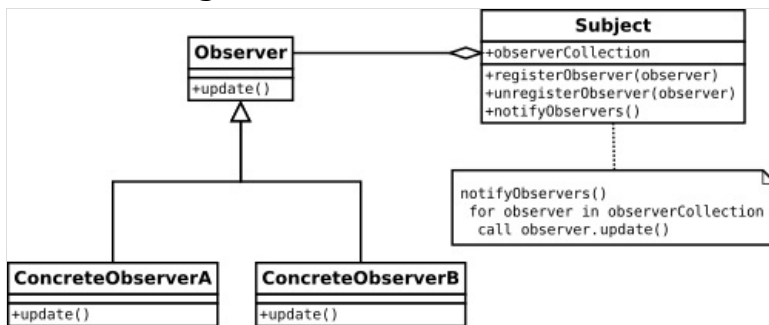


A sample UML class and sequence diagram for the observer design pattern. [6]

In this UML class diagram, the Subject class does not update the state of dependent objects directly. Instead, Subject refers to the Observer interface (update()) for updating state, which makes the Subject independent of how the state of dependent objects is updated. The Observer1 and Observer2 classes implement the Observer interface by synchronizing their state with subject's state.

The UML sequence diagram shows the runtime interactions: The Observer1 and Observer2 objects call attach(this) on Subject1 to register themselves. Assuming that the state of Subject1 changes, Subject1 calls notify() on itself. notify() calls update() on the registered Observer1 and Observer2 objects, which request the changed data (getState()) from Subject1 to update (synchronize) their state.

UML class diagram



UML class diagram of Observer pattern

Example

While the library classes `java.util.Observer` (<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Observer.html>) and `java.util.Observable` (<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Observable.html>) exist, they have been deprecated in Java 9 because the model implemented was quite limited.

Below is an example written in Java that takes keyboard input and handles each input line as an event. When a string is supplied from `System.in`, the method `notifyObservers()` is then called in order to notify all observers of the event's occurrence, in the form of an invocation of their update methods.

Java

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

interface Observer {
    void update(String event);
}

class EventSource {
    List<Observer> observers = new ArrayList<>();

    public void notifyObservers(String event) {
        observers.forEach(observer -> observer.update(event));
    }

    public void addObserver(Observer observer) {
        observers.add(observer);
    }
}
```

```

    public void scanSystemIn() {
        Scanner scanner = new Scanner(System.in);
        while (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            notifyObservers(line);
        }
    }
}

public class ObserverDemo {
    public static void main(String[] args) {
        System.out.println("Enter Text: ");
        EventSource eventSource = new EventSource();

        eventSource.addObserver(event -> System.out.printf("Received response: %s\n", event));

        eventSource.scanSystemIn();
    }
}

```

C#

C# provides the `IObservable`^[7] and `IObserver`^[8] interfaces as well as documentation on how to implement the design pattern.^[9]

```

class Payload
{
    internal string Message { get; init; }
}

class Subject : IObservable<Payload>
{
    private readonly List<IObserver<Payload>> _observers = new List<IObserver<Payload>>();

    IDisposable IObservable<Payload>.Subscribe(IObserver<Payload> observer)
    {
        if (!_observers.Contains(observer))
        {
            _observers.Add(observer);
        }

        return new Unsubscriber(observer, _observers);
    }

    internal void SendMessage(string message)
    {
        foreach (var observer in _observers)
        {
            observer.OnNext(new Payload { Message = message });
        }
    }
}

internal class Unsubscriber : IDisposable
{
    private readonly IObserver<Payload> _observer;
    private readonly ICollection<IObserver<Payload>> _observers;

    internal Unsubscriber(
        IObserver<Payload> observer,
        ICollection<IObserver<Payload>> observers)
    {
        _observer = observer;
        _observers = observers;
    }

    void IDisposable.Dispose()
    {
        if (_observer != null && _observers.Contains(_observer))
        {
            _observers.Remove(_observer);
        }
    }
}

internal class Observer : IObserver<Payload>
{
    private string _message;

    public void OnCompleted()
    {
    }

    public void OnError(Exception error)
    {
    }

    public void OnNext(Payload value)
    {
        _message = value.Message;
    }

    internal IDisposable Register(IObservable<Payload> subject)
    {
        return subject.Subscribe(this);
    }
}

```

C++

This is a C++23 implementation.

```
import std;

using std::vector;

class Subject; // Forward declaration for usage in Observer

class Observer {
private:
    // Reference to a Subject object to detach in the destructor
    Subject& subject;
public:
    explicit Observer(Subject& subj):
        subject{subj} {
        subject.attach(*this);
    }

    virtual ~Observer() {
        subject.detach(*this);
    }

    Observer(const Observer&) = delete;
    Observer& operator=(const Observer&) = delete;

    virtual void update(Subject& s) const = 0;
};

// Subject is the base class for event generation
class Subject {
private:
    vector<RefObserver> observers;
public:
    using RefObserver = std::reference_wrapper<const Observer>;

    // Notify all the attached observers
    void notify() {
        for (const Observer& x: observers) {
            x.get().update(*this);
        }
    }

    // Add an observer
    void attach(const Observer& observer) {
        observers.push_back(observer);
    }

    // Remove an observer
    void detach(Observer& observer) {
        observers.remove_if([&observer](const RefObserver& obj) -> bool {
            return &obj.get() == &observer;
        });
    }
};

// Example of usage
class ConcreteObserver: public Observer {
public:
    ConcreteObserver(Subject& subj):
        Observer(subj) {}

    // Get notification
    void update(Subject&) const override {
        std::println("Got a notification");
    }
};

int main(int argc, char* argv[]) {
    Subject cs;
    ConcreteObserver co1(cs);
    ConcreteObserver co2(cs);
    cs.notify();
}
```

The program output is:

```
Got a notification
Got a notification
```

Groovy

```
class EventSource {
    private observers = []

    private notifyObservers(String event) {
        observers.each { it(event) }
    }

    void addObserver(observer) {
        observers += observer
    }

    void scanSystemIn() {
        var scanner = new Scanner(System.in)
        while (scanner) {
            var line = scanner.nextLine()
            notifyObservers(line)
        }
    }
}
```

```

}

println 'Enter Text: '
var eventSource = new EventSource()

eventSource.addObserver { event ->
    println "Received response: $event"
}

eventSource.scanSystemIn()

```

Kotlin

```

import java.util.Scanner

typealias Observer = (event: String) -> Unit;

class EventSource {
    private var observers = mutableListOf<Observer>()

    private fun notifyObservers(event: String) {
        observers.forEach { it(event) }
    }

    fun addObserver(observer: Observer) {
        observers += observer
    }

    fun scanSystemIn() {
        val scanner = Scanner(System.`in`)
        while (scanner.hasNext()) {
            val line = scanner.nextLine()
            notifyObservers(line)
        }
    }
}

```

```

fun main(arg: List<String>) {
    println("Enter Text: ")
    val eventSource = EventSource()

    eventSource.addObserver { event ->
        println("Received response: $event")
    }

    eventSource.scanSystemIn()
}

```

Delphi

```

uses
    System.Generics.Collections, System.SysUtils;

type
    IObserver = interface
        ['{0C8F4C5D-1898-4F24-91DA-63F1DD66A692}']
        procedure Update(const AValue: string);
    end;

type
    TObserverManager = class
    private
        FObservers: TList<IObserver>;
    public
        constructor Create; overload;
        destructor Destroy; override;
        procedure NotifyObservers(const AValue: string);
        procedure AddObserver(const AObserver: IObserver);
        procedure UnregisterObserver(const AObserver: IObserver);
    end;

type
    TListener = class(TInterfacedObject, IObserver)
    private
        FName: string;
    public
        constructor Create(const AName: string); reintroduce;
        procedure Update(const AValue: string);
    end;

procedure TObserverManager.AddObserver(const AObserver: IObserver);
begin
    if not FObservers.Contains(AObserver)
    then FObservers.Add(AObserver);
end;

begin
    FreeAndNil(FObservers);
    inherited;
end;

procedure TObserverManager.NotifyObservers(const AValue: string);
var
    i: Integer;
begin
    for i := 0 to FObservers.Count - 1 do
        FObservers[i].Update(AValue);
    end;
end;

```



```

procedure TObserverManager.UnregisterObserver(const AObserver: IObserver);
begin
    if FObservers.Contains(AObserver)
    then FObservers.Remove(AObserver);
end;

constructor TListener.Create(const AName: string);
begin
    inherited Create;
    FName := AName;
end;

procedure TListener.Update(const AValue: string);
begin
    WriteLn(FName + ' listener received notification: ' + AValue);
end;

procedure TMyForm.ObserverExampleButtonClick(Sender: TObject);
var
    LDoorNotify: TObserverManager;
    LListenerHusband: IObserver;
    LListenerWife: IObserver;
begin
    LDoorNotify := TObserverManager.Create;
    try
        LListenerHusband := TListener.Create('Husband');
        LDoorNotify.AddObserver(LListenerHusband);
        LListenerWife := TListener.Create('Wife');
        LDoorNotify.AddObserver(LListenerWife);
        LDoorNotify.NotifyObservers('Someone is knocking on the door');
    finally
        FreeAndNil(LDoorNotify);
    end;
end;

```

Output

```

Husband listener received notification: Someone is knocking on the door
Wife listener received notification: Someone is knocking on the door

```

Python

A similar example in Python:

```

class Observable:
    def __init__(self):
        self._observers = []

    def register_observer(self, observer: Observer) -> None:
        self._observers.append(observer)

    def notify_observers(self, *args, **kwargs) -> None:
        for observer in self._observers:
            observer.notify(self, *args, **kwargs)

class Observer:
    def __init__(self, observable: Observable):
        observable.register_observer(self)

    def notify(self, observable: Observable, *args, **kwargs) -> None:
        print("Got", args, kwargs, "From", observable)

subject = Observable()
observer = Observer(subject)
subject.notify_observers("test", kw="python")

# prints: Got ('test',) {'kw': 'python'} From <__main__.Observable object at 0x0000019757826FD0>

```

JavaScript

JavaScript has a deprecated `Object.observe` function that was a more accurate implementation of the observer pattern.^[10] This would fire events upon change to the observed object. Without the deprecated `Object.observe` function, the pattern may be implemented with more explicit code:^[11]

```

let Subject = {
    _state: 0,
    _observers: [],
    add: function(observer) {
        this._observers.push(observer);
    },
    getState: function() {
        return this._state;
    },
    setState: function(value) {
        this._state = value;
        for (let i = 0; i < this._observers.length; i++)
        {
            this._observers[i].signal(this);
        }
    }
};

let Observer = {

```

```
signal: function(subject) {  
  let currentValue = subject.getState();  
  console.log(currentValue);  
}  
}  
  
Subject.add(Observer);  
Subject.setState(10);  
// Output in console.log - 10
```


See also

- [Implicit invocation](#)
- [Client-server model](#)
- The observer pattern is often used in the [entity-component-system](#) pattern

References

1. Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (<https://archive.org/details/designpatternsel00gamm/page/293>). Addison Wesley. pp. 293ff (<https://archive.org/details/designpatternsel00gamm/page/293>). ISBN 0-201-63361-2.
2. "Observer Design Pattern" (<https://www.geeksforgeeks.org/observer-pattern-set-1-introduction/>). *www.geeksforgeeks.org*.
3. Comparison between different observer pattern implementations (<https://github.com/millermedeiros/js-signals/wiki/Comparison-between-different-Observer-Pattern-implementations>) — Moshe Bindler, 2015 (GitHub)
4. Differences between pub/sub and observer pattern (<https://www.safaribooksonline.com/library/view/learning-javascript-design/9781449334840/ch09s05.html>) — *The Observer Pattern* by Adi Osmani (Safari Books Online)
5. The Windows Programming Experience (<https://books.google.com/books?id=18wFKrkDdM0C&pg=PA230>), Charles Petzold, November 10, 1992, *PC Magazine* (Google Books)
6. "The Observer design pattern - Structure and Collaboration" (<http://w3sdesign.com/?gr=b07&ugr=struct>). *w3sDesign.com*. Retrieved 2017-08-12.
7. "IObservable Interface (System)" (<https://learn.microsoft.com/en-us/dotnet/api/system.iobservable-1?view=net-8.0>). *learn.microsoft.com*. Retrieved 9 November 2024.
8. "IObserver Interface (System)" (<https://learn.microsoft.com/en-us/dotnet/api/system.iobserver-1?view=net-8.0>). *learn.microsoft.com*. Retrieved 9 November 2024.
9. "Observer design pattern - .NET" (<https://learn.microsoft.com/en-us/dotnet/standard/events/observer-design-pattern>). *learn.microsoft.com*. 25 May 2023. Retrieved 9 November 2024.
10. "jQuery - Listening for variable changes in JavaScript" (<https://stackoverflow.com/a/50862441/887092>).
11. "jQuery - Listening for variable changes in JavaScript" (<https://stackoverflow.com/a/37403125/887092>).

External links

-  [Observer implementations in various languages at Wikibooks](#)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Observer_pattern&oldid=1311262962"

Publish–subscribe pattern

In software architecture, the **publish–subscribe pattern** (**pub/sub**) is a messaging pattern in which message senders, called **publishers**, categorize messages into classes (or *topics*), and send them without needing to know which components will receive them. Message recipients, called **subscribers**, express interest in one or more classes and only receive messages in those classes, without needing to know the identity of the publishers.

This pattern decouples the components that produce messages from those that consume them, and supports asynchronous, many-to-many communication. The publish–subscribe model is commonly contrasted with message queue-based and point-to-point messaging models, where producers send messages directly to consumers.

Publish–subscribe is a sibling of the message queue paradigm, and is typically a component of larger message-oriented middleware systems. Many modern messaging frameworks and protocols, such as the Java Message Service (JMS), Apache Kafka, and MQTT, support both the pub/sub and queue-based models.

This pattern provides greater network scalability and supports more dynamic topologies, but can make it harder to modify the publisher's logic or the structure of the published data. Compared to synchronous patterns like RPC and point-to-point messaging, publish–subscribe provides the highest level of decoupling among architectural components. However, it can also lead to semantic or format coupling between publishers and subscribers, which may cause systems to become entangled or brittle over time.^[1]

Message filtering

In publish–subscribe systems, subscribers typically receive only a subset of messages. The process of selecting relevant messages is called **filtering**, and it can be implemented in several ways:

- **Topic-based filtering:** Messages are published to named *topics* or *channels*. Subscribers register to receive messages on specific topics, and receive all messages published to them.
- **Content-based filtering:** Subscribers define constraints based on message attributes or content. Messages are delivered only if they match the subscriber's criteria.
- **Hybrid systems:** Some implementations combine topic- and content-based filtering. Messages are categorized by topic, and subscribers apply content-based filters to messages within those topics.

Topologies

In most pub/sub systems, publishers and subscribers communicate through a central intermediary such as a message broker or event bus. The broker receives messages from publishers and forwards them to the appropriate subscribers, optionally performing store and forward, priority queuing, or other routing logic.

Subscriber registration can occur at different times:

- **Build time:** Subscribers are hardcoded to handle specific messages or events (e.g., GUI event handlers).
- **Initialization time:** Subscriptions are defined in XML configuration files or metadata.
- **Runtime:** Subscriptions can be added or removed dynamically (e.g., database triggers, RSS readers).

Some pub/sub systems use **brokerless** architectures, in which publishers and subscribers discover each other and exchange messages directly. For example, the Data Distribution Service (DDS) middleware uses IP multicast and metadata sharing to establish communication paths. Brokerless systems require construction of overlay networks, often using Small-World topologies to enable efficient routing.

It was shown by Jon Kleinberg that efficient decentralized routing requires Navigable Small-World topologies, which are employed in federated or peer-to-peer pub/sub systems.^[2] Locality-aware pub/sub networks use low-latency links to reduce message propagation time.^[3]

History

One of the earliest publicly described pub/sub systems was the "news" subsystem of the Isis Toolkit, presented at the 1987 ACM Symposium on Operating Systems Principles (SOSP '87).^[4]

Although the publish–subscribe pattern is now typically distinguished from the observer pattern due to its emphasis on decoupling and distributed communication, early usage in literature and systems sometimes used the terms interchangeably, especially in the context of in-process event handling or GUI frameworks.^[5] As distributed systems became more common, the

publish–subscribe model evolved to emphasize asynchronous messaging and broker-mediated communication, setting it apart from the more tightly coupled observer pattern.

Advantages

Loose coupling

Publishers are loosely coupled to subscribers, and need not even know of their existence. With the topic being the focus, publishers and subscribers are allowed to remain ignorant of system topology. Each can continue to operate as per normal independently of the other. In the traditional tightly coupled client–server paradigm, the client cannot post messages to the server while the server process is not running, nor can the server receive messages unless the client is running. Many pub/sub systems decouple not only the locations of the publishers and subscribers but also decouple them temporally. A common strategy used by middleware analysts with such pub/sub systems is to take down a publisher to allow the subscriber to work through the backlog (a form of bandwidth throttling).

Scalability

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

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

Message delivery issues

Redundant subscribers in a pub/sub system can help assure message delivery with minimal additional complexity. For example, a factory may utilize a pub/sub system where equipment can publish problems or failures to a subscriber that displays and logs those problems. If the logger fails (crashes), equipment problem publishers won't necessarily receive notice of the logger failure, and error messages will not be displayed or recorded by any equipment on the pub/sub system. In a client/server system, when an error logger fails, the system will receive an indication of the error logger (server) failure. However, the client/server system will have to deal with that failure by having redundant logging servers online, or by dynamically spawning fallback logging servers. This adds complexity to the client and server designs, as well as to the client/server architecture as a whole. In a pub/sub system, redundant logging subscribers that are exact duplicates of the existing logger can be added to the system to increase logging reliability without any impact to any other equipment on the system. The feature of assured error message logging can also be added incrementally, subsequent to implementing the basic functionality of equipment problem message logging.

Disadvantages

The most serious problems with pub/sub systems are a side-effect of their main advantage: the decoupling of publisher from subscriber.

Message delivery issues

A pub/sub system must be designed carefully to be able to provide stronger system properties that a particular application might require, such as assured delivery.

- The broker in a pub/sub system may be designed to deliver messages for a specified time, but then stop attempting delivery, whether or not it has received confirmation of successful receipt of the message by all subscribers. A pub/sub system designed in this way cannot guarantee delivery of messages to any applications that might require such assured delivery. Tighter coupling of the designs of such a publisher and subscriber pair must be enforced outside of the pub/sub architecture to accomplish such assured delivery (e.g. by requiring the subscriber to publish receipt messages).
- A publisher in a pub/sub system may assume that a subscriber is listening, when in fact it is not.

The pub/sub pattern scales well for small networks with a small number of publisher and subscriber nodes and low message volume. However, as the number of nodes and messages grows, the likelihood of instabilities increases, limiting the maximum scalability of a pub/sub network. Example throughput instabilities at large scales include:

- Load surges—periods when subscriber requests saturate network throughput followed by periods of low message volume (underutilized network bandwidth)
- Slowdowns—as more and more applications use the system (even if they are communicating on separate pub/sub channels)

the message volume flow to an individual subscriber will slow

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

Even with systems that do not rely on brokers, a subscriber might be able to receive data that it is not authorized to receive. An unauthorized publisher may be able to introduce incorrect or damaging messages into the pub/sub system. This is especially true with systems that broadcast or multicast their messages. Encryption (e.g. Transport Layer Security (SSL/TLS)) can prevent unauthorized access, but cannot prevent damaging messages from being introduced by authorized publishers. Architectures other than pub/sub, such as client/server systems, are also vulnerable to authorized message senders that behave maliciously.

See also

- Atom, another highly scalable web-syndication protocol
- Data Distribution Service (DDS)
- Event-driven programming
- High-level architecture
- Internet Group Management Protocol (IGMP)
- Message brokers
- Message queue
- Observer pattern
- Producer-consumer problem
- Push technology^[2]
- RSS, a highly scalable web-syndication protocol
- Usenet
- WebSub, an implementation of pub/sub

References

1. Hohpe, Gregor (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional. ISBN 978-0321200686.
2. Chen, Chen; Tock, Yoav; Girdzijauskas, Sarunas (2018). "BeaConvey" (<http://dl.acm.org/citation.cfm?doid=3210284.3210287>). *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems* (<http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-228002>). Hamilton, New Zealand: ACM Press. pp. 64–75. doi:10.1145/3210284.3210287 (<https://doi.org/10.1145%2F3210284.3210287>). ISBN 9781450357821. S2CID 43929719 (<https://api.semanticscholar.org/CorpusID:43929719>).
3. Rahimian, Fatemeh; Le Nguyen Huu, Thinh; Girdzijauskas, Sarunas (2012), Göschka, Karl Michael; Haridi, Seif (eds.), "Locality-Awareness in a Peer-to-Peer Publish/Subscribe Network", *Distributed Applications and Interoperable Systems*, vol. 7272, Springer Berlin Heidelberg, pp. 45–58, doi:10.1007/978-3-642-30823-9_4 (https://doi.org/10.1007%2F978-3-642-30823-9_4), ISBN 9783642308222
4. Birman, K.; Joseph, T. (1987). "Exploiting virtual synchrony in distributed systems". *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles - SOSP '87*. pp. 123–138. doi:10.1145/41457.37515 (<https://doi.org/10.1145%2F41457.37515>). ISBN 089791242X. S2CID 7739589 (<https://api.semanticscholar.org/CorpusID:7739589>).
5. The Windows Programming Experience (<https://books.google.com/books?id=18wFKrkDdM0C&pg=PA230>), Charles Petzold, November 10, 1992, *PC Magazine* ([Google Books](#))

Servant (design pattern)

In software engineering, the servant pattern defines an object used to offer some functionality to a group of classes without defining that functionality in each of them. A Servant is a class whose instance (or even just class) provides methods that take care of a desired service, while objects for which (or with whom) the servant does something, are taken as parameters.

Description and simple example

Servant is used for providing some behavior to a group of classes. Instead of defining that behavior in each class - or when we cannot factor out this behavior in the common parent class - it is defined once in the Servant.

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

Two ways of implementation

There are two ways to implement this design pattern:

- 1. User knows the servant (in which case it is needed to know the serviced classes) and sends messages with requests to the servant instances, passing the serviced objects as parameters. The serviced classes (geometric objects from our example) don't know about servant, but they implement the "IServiced" interface. The user class just calls the method of servant and passes serviced objects as parameters. This situation is shown on figure 1.
- 2. Serviced instances know the servant and the user sends them messages with requests (in which case it isn't necessary to know the servant). The serviced instances then send messages to the instances of servant, asking for service. On figure 2 is shown opposite situation, where user don't know about servant class and calls directly serviced classes. Serviced classes then asks servant themselves to achieve desired functionality.

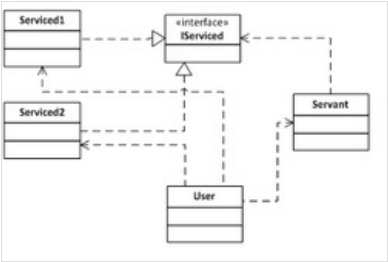
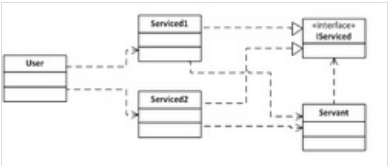


Figure 1: User uses servant to achieve some functionality and passes the serviced objects as parameters.

Example

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



```
// Servant class, offering its functionality to classes implementing
// Movable Interface
public class MoveServant {
    // Method, which will move Movable implementing class to position where
    public void moveTo(Movable serviced, Position where) {
        // Do some other stuff to ensure it moves smoothly and nicely, this is
        // the place to offer the functionality
        serviced.setPosition(where);
    }

    // Method, which will move Movable implementing class by dx and dy
    public void moveBy(Movable serviced, int dx, int dy) {
        // this is the place to offer the functionality
        dx += serviced.getPosition().xPosition;
        dy += serviced.getPosition().yPosition;
        serviced.setPosition(new Position(dx, dy));
    }
}

// Interface specifying what serviced classes needs to implement, to be
// serviced by servant.
public interface Movable {
    public void setPosition(Position p);

    public Position getPosition();
}

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



```

private Position p;

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

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

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

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

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

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

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

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

// Just a very simple container class for position.
public class Position {
    public int xPosition;
    public int yPosition;

    public Position(int dx, int dy) {
        xPosition = dx;
        yPosition = dy;
    }
}

```

Similar design pattern: Command

Design patterns Command and Servant are very similar and implementations of them are often virtually the same. The difference between them is the approach to the problem.

- For the Servant pattern we have some objects to which we want to offer some functionality. We create a class whose instances offer that functionality and which defines an interface that serviced objects must implement. Serviced instances are then passed as parameters to the servant.
- For the Command pattern we have some objects that we want to modify with some functionality. So, we define an interface which commands which desired functionality must be implemented. Instances of those commands are then passed to original objects as parameters of their methods.

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

See also

- Command pattern

References

Resources

Pecinovský, Rudolf; Jarmila Pavlíčková; Luboš Pavlíček (June 2006). *Let's Modify the Objects First Approach into Design Patterns First* (http://edu.pecinovsky.cz/papers/2006_ITiCSE_Design_Patterns_First.pdf) (PDF). Eleventh Annual Conference on Innovation and Technology in Computer Science Education, University of Bologna (<http://www.iticse06.cs.unibo.it/>).

Retrieved from "[https://en.wikipedia.org/w/index.php?title=Servant_\(design_pattern\)&oldid=1243542283](https://en.wikipedia.org/w/index.php?title=Servant_(design_pattern)&oldid=1243542283)"

Specification pattern

In computer programming, the **specification pattern** is a particular software design pattern, whereby business rules can be recombined by chaining the business rules together using Boolean logic. The pattern is frequently used in the context of domain-driven design.

A specification pattern outlines a business rule that is combinable with other business rules. In this pattern, a unit of business logic inherits its functionality from the abstract aggregate Composite Specification class. The Composite Specification class has one function called IsSatisfiedBy that returns a Boolean value. After instantiation, the specification is "chained" with other specifications, making new specifications easily maintainable, yet highly customizable business logic. Furthermore, upon instantiation the business logic may, through method invocation or inversion of control, have its state altered in order to become a delegate of other classes such as a persistence repository.

As a consequence of performing runtime composition of high-level business/domain logic, the Specification pattern is a convenient tool for converting ad-hoc user search criteria into low level logic to be processed by repositories.

Since a specification is an encapsulation of logic in a reusable form it is very simple to thoroughly unit test, and when used in this context is also an implementation of the humble object pattern.

Code examples

C#

```
public interface ISpecification
{
    bool IsSatisfiedBy(object candidate);
    ISpecification And(ISpecification other);
    ISpecification AndNot(ISpecification other);
    ISpecification Or(ISpecification other);
    ISpecification OrNot(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 AndNot(ISpecification other)
    {
        return new AndNotSpecification(this, other);
    }

    public ISpecification Or(ISpecification other)
    {
        return new OrSpecification(this, other);
    }

    public ISpecification OrNot(ISpecification other)
    {
        return new OrNotSpecification(this, other);
    }

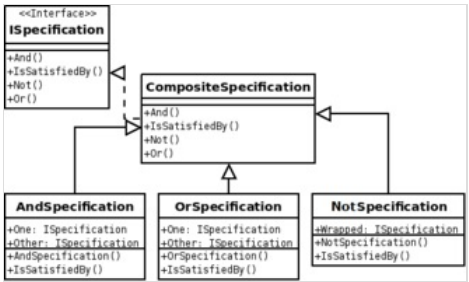
    public ISpecification Not()
    {
        return new NotSpecification(this);
    }
}

public class AndSpecification : CompositeSpecification
{
    private ISpecification _leftCondition;
    private ISpecification _rightCondition;

    public AndSpecification(ISpecification left, ISpecification right)
    {
        _leftCondition = left;
        _rightCondition = right;
    }

    public override bool IsSatisfiedBy(object candidate)
    {
        return _leftCondition.IsSatisfiedBy(candidate) && _rightCondition.IsSatisfiedBy(candidate);
    }
}

public class AndNotSpecification : CompositeSpecification
{
    private ISpecification _leftCondition;
```



Specification Pattern in UML

```

private ISpecification _rightCondition;

public AndNotSpecification(ISpecification left, ISpecification right)
{
    _leftCondition = left;
    _rightCondition = right;
}

public override bool IsSatisfiedBy(object candidate)
{
    return _leftCondition.IsSatisfiedBy(candidate) && _rightCondition.IsSatisfiedBy(candidate) != true;
}
}

public class OrSpecification : CompositeSpecification
{
    private ISpecification _leftCondition;
    private ISpecification _rightCondition;

    public OrSpecification(ISpecification left, ISpecification right)
    {
        _leftCondition = left;
        _rightCondition = right;
    }

    public override bool IsSatisfiedBy(object candidate)
    {
        return _leftCondition.IsSatisfiedBy(candidate) || _rightCondition.IsSatisfiedBy(candidate);
    }
}

public class OrNotSpecification : CompositeSpecification
{
    private ISpecification _leftCondition;
    private ISpecification _rightCondition;

    public OrNotSpecification(ISpecification left, ISpecification right)
    {
        _leftCondition = left;
        _rightCondition = right;
    }

    public override bool IsSatisfiedBy(object candidate)
    {
        return _leftCondition.IsSatisfiedBy(candidate) || _rightCondition.IsSatisfiedBy(candidate) != true;
    }
}

public class NotSpecification : CompositeSpecification
{
    private ISpecification _wrapped;

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

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

```

C# 6.0 with generics

```

public interface ISpecification<T>
{
    bool IsSatisfiedBy(T candidate);
    ISpecification<T> And(ISpecification<T> other);
    ISpecification<T> AndNot(ISpecification<T> other);
    ISpecification<T> Or(ISpecification<T> other);
    ISpecification<T> OrNot(ISpecification<T> other);
    ISpecification<T> Not();
}

public abstract class LinqSpecification<T> : CompositeSpecification<T>
{
    public abstract Expression<Func<T, bool>> AsExpression();
    public override bool IsSatisfiedBy(T candidate) => AsExpression().Compile()(candidate);
}

public abstract class CompositeSpecification<T> : ISpecification<T>
{
    public abstract bool IsSatisfiedBy(T candidate);
    public ISpecification<T> And(ISpecification<T> other) => new AndSpecification<T>(this, other);
    public ISpecification<T> AndNot(ISpecification<T> other) => new AndNotSpecification<T>(this, other);
    public ISpecification<T> Or(ISpecification<T> other) => new OrSpecification<T>(this, other);
    public ISpecification<T> OrNot(ISpecification<T> other) => new OrNotSpecification<T>(this, other);
    public ISpecification<T> Not() => new NotSpecification<T>(this);
}

public class AndSpecification<T> : CompositeSpecification<T>
{
    private ISpecification<T> _left;
    private ISpecification<T> _right;

    public AndSpecification(ISpecification<T> left, ISpecification<T> right)
    {
        _left = left;
        _right = right;
    }

    public override bool IsSatisfiedBy(T candidate) => _left.IsSatisfiedBy(candidate) && _right.IsSatisfiedBy(candidate);
}

```

```

public class AndNotSpecification<T> : CompositeSpecification<T>
{
    private ISpecification<T> _left;
    private ISpecification<T> _right;

    public AndNotSpecification(ISpecification<T> left, ISpecification<T> right)
    {
        _left = left;
        _right = right;
    }

    public override bool IsSatisfiedBy(T candidate) => _left.IsSatisfiedBy(candidate) && !_right.IsSatisfiedBy(candidate);
}

public class OrSpecification<T> : CompositeSpecification<T>
{
    private ISpecification<T> _left;
    private ISpecification<T> _right;

    public OrSpecification(ISpecification<T> left, ISpecification<T> right)
    {
        _left = left;
        _right = right;
    }

    public override bool IsSatisfiedBy(T candidate) => _left.IsSatisfiedBy(candidate) || _right.IsSatisfiedBy(candidate);
}

public class OrNotSpecification<T> : CompositeSpecification<T>
{
    private ISpecification<T> _left;
    private ISpecification<T> _right;

    public OrNotSpecification(ISpecification<T> left, ISpecification<T> right)
    {
        _left = left;
        _right = right;
    }

    public override bool IsSatisfiedBy(T candidate) => _left.IsSatisfiedBy(candidate) || !_right.IsSatisfiedBy(candidate);
}

public class NotSpecification<T> : CompositeSpecification<T>
{
    ISpecification<T> other;
    public NotSpecification(ISpecification<T> other) => this.other = other;
    public override bool IsSatisfiedBy(T candidate) => !other.IsSatisfiedBy(candidate);
}

```

Python

```

from abc import ABC, abstractmethod
from dataclasses import dataclass
from typing import Any

class BaseSpecification(ABC):
    @abstractmethod
    def is_satisfied_by(self, candidate: Any) -> bool:
        raise NotImplementedError()

    def __call__(self, candidate: Any) -> bool:
        return self.is_satisfied_by(candidate)

    def __and__(self, other: "BaseSpecification") -> "AndSpecification":
        return AndSpecification(self, other)

    def __or__(self, other: "BaseSpecification") -> "OrSpecification":
        return OrSpecification(self, other)

    def __neg__(self) -> "NotSpecification":
        return NotSpecification(self)

@dataclass(frozen=True)
class AndSpecification(BaseSpecification):
    first: BaseSpecification
    second: BaseSpecification

    def is_satisfied_by(self, candidate: Any) -> bool:
        return self.first.is_satisfied_by(candidate) and self.second.is_satisfied_by(candidate)

@dataclass(frozen=True)
class OrSpecification(BaseSpecification):
    first: BaseSpecification
    second: BaseSpecification

    def is_satisfied_by(self, candidate: Any) -> bool:
        return self.first.is_satisfied_by(candidate) or self.second.is_satisfied_by(candidate)

@dataclass(frozen=True)
class NotSpecification(BaseSpecification):
    subject: BaseSpecification

    def is_satisfied_by(self, candidate: Any) -> bool:
        return not self.subject.is_satisfied_by(candidate)

```

C++

```

template <class T>
class ISpecification
{
public:

```

```

    virtual ~ISpecification() = default;
    virtual bool IsSatisfiedBy(T Candidate) const = 0;
    virtual ISpecification<T>* And(const ISpecification<T>& Other) const = 0;
    virtual ISpecification<T>* AndNot(const ISpecification<T>& Other) const = 0;
    virtual ISpecification<T>* Or(const ISpecification<T>& Other) const = 0;
    virtual ISpecification<T>* OrNot(const ISpecification<T>& Other) const = 0;
    virtual ISpecification<T>* Not() const = 0;
};

template <class T>
class CompositeSpecification : public ISpecification<T>
{
public:
    virtual bool IsSatisfiedBy(T Candidate) const override = 0;

    virtual ISpecification<T>* And(const ISpecification<T>& Other) const override;
    virtual ISpecification<T>* AndNot(const ISpecification<T>& Other) const override;
    virtual ISpecification<T>* Or(const ISpecification<T>& Other) const override;
    virtual ISpecification<T>* OrNot(const ISpecification<T>& Other) const override;
    virtual ISpecification<T>* Not() const override;
};

template <class T>
class AndSpecification final : public CompositeSpecification<T>
{
public:
    const ISpecification<T>& Left;
    const ISpecification<T>& Right;

    AndSpecification(const ISpecification<T>& InLeft, const ISpecification<T>& InRight)
        : Left(InLeft),
          Right(InRight) { }

    virtual bool IsSatisfiedBy(T Candidate) const override
    {
        return Left.IsSatisfiedBy(Candidate) && Right.IsSatisfiedBy(Candidate);
    }
};

template <class T>
ISpecification<T>* CompositeSpecification<T>::And(const ISpecification<T>& Other) const
{
    return new AndSpecification<T>(*this, Other);
}

template <class T>
class AndNotSpecification final : public CompositeSpecification<T>
{
public:
    const ISpecification<T>& Left;
    const ISpecification<T>& Right;

    AndNotSpecification(const ISpecification<T>& InLeft, const ISpecification<T>& InRight)
        : Left(InLeft),
          Right(InRight) { }

    virtual bool IsSatisfiedBy(T Candidate) const override
    {
        return Left.IsSatisfiedBy(Candidate) && !Right.IsSatisfiedBy(Candidate);
    }
};

template <class T>
class OrSpecification final : public CompositeSpecification<T>
{
public:
    const ISpecification<T>& Left;
    const ISpecification<T>& Right;

    OrSpecification(const ISpecification<T>& InLeft, const ISpecification<T>& InRight)
        : Left(InLeft),
          Right(InRight) { }

    virtual bool IsSatisfiedBy(T Candidate) const override
    {
        return Left.IsSatisfiedBy(Candidate) || Right.IsSatisfiedBy(Candidate);
    }
};

template <class T>
class OrNotSpecification final : public CompositeSpecification<T>
{
public:
    const ISpecification<T>& Left;
    const ISpecification<T>& Right;

    OrNotSpecification(const ISpecification<T>& InLeft, const ISpecification<T>& InRight)
        : Left(InLeft),
          Right(InRight) { }

    virtual bool IsSatisfiedBy(T Candidate) const override
    {
        return Left.IsSatisfiedBy(Candidate) || !Right.IsSatisfiedBy(Candidate);
    }
};

template <class T>
class NotSpecification final : public CompositeSpecification<T>
{
public:
    const ISpecification<T>& Other;

    NotSpecification(const ISpecification<T>& InOther)
        : Other(InOther) { }

    virtual bool IsSatisfiedBy(T Candidate) const override
    {
        return !Other.IsSatisfiedBy(Candidate);
    }
}

```

```
};

template <class T>
ISpecification<T>* CompositeSpecification<T>::AndNot(const ISpecification<T>& Other) const
{
    return new AndNotSpecification<T>(*this, Other);
}

template <class T>
ISpecification<T>* CompositeSpecification<T>::Or(const ISpecification<T>& Other) const
{
    return new OrSpecification<T>(*this, Other);
}

template <class T>
ISpecification<T>* CompositeSpecification<T>::OrNot(const ISpecification<T>& Other) const
{
    return new OrNotSpecification<T>(*this, Other);
}

template <class T>
ISpecification<T>* CompositeSpecification<T>::Not() const
{
    return new NotSpecification<T>(*this);
}
```

TypeScript

```
export interface ISpecification {
    isSatisfiedBy(candidate: unknown): boolean;
    and(other: ISpecification): ISpecification;
    andNot(other: ISpecification): ISpecification;
    or(other: ISpecification): ISpecification;
    orNot(other: ISpecification): ISpecification;
    not(): ISpecification;
}

export abstract class CompositeSpecification implements ISpecification {
    abstract isSatisfiedBy(candidate: unknown): boolean;

    and(other: ISpecification): ISpecification {
        return new AndSpecification(this, other);
    }

    andNot(other: ISpecification): ISpecification {
        return new AndNotSpecification(this, other);
    }

    or(other: ISpecification): ISpecification {
        return new OrSpecification(this, other);
    }

    orNot(other: ISpecification): ISpecification {
        return new OrNotSpecification(this, other);
    }

    not(): ISpecification {
        return new NotSpecification(this);
    }
}

export class AndSpecification extends CompositeSpecification {
    constructor(private leftCondition: ISpecification, private rightCondition: ISpecification) {
        super();
    }

    isSatisfiedBy(candidate: unknown): boolean {
        return this.leftCondition.isSatisfiedBy(candidate) && this.rightCondition.isSatisfiedBy(candidate);
    }
}

export class AndNotSpecification extends CompositeSpecification {
    constructor(private leftCondition: ISpecification, private rightCondition: ISpecification) {
        super();
    }

    isSatisfiedBy(candidate: unknown): boolean {
        return this.leftCondition.isSatisfiedBy(candidate) && this.rightCondition.isSatisfiedBy(candidate) !== true;
    }
}

export class OrSpecification extends CompositeSpecification {
    constructor(private leftCondition: ISpecification, private rightCondition: ISpecification) {
        super();
    }

    isSatisfiedBy(candidate: unknown): boolean {
        return this.leftCondition.isSatisfiedBy(candidate) || this.rightCondition.isSatisfiedBy(candidate);
    }
}

export class OrNotSpecification extends CompositeSpecification {
    constructor(private leftCondition: ISpecification, private rightCondition: ISpecification) {
        super();
    }

    isSatisfiedBy(candidate: unknown): boolean {
        return this.leftCondition.isSatisfiedBy(candidate) || this.rightCondition.isSatisfiedBy(candidate) !== true;
    }
}

export class NotSpecification extends CompositeSpecification {
    constructor(private wrapped: ISpecification) {
        super();
    }
}
```

```
isSatisfiedBy(candidate: unknown): boolean {  
    return !this.wrapped.isSatisfiedBy(candidate);  
}  
}
```

Example of use

In the next example, invoices are retrieved and sent to a collection agency if:

1. they are overdue,
2. notices have been sent, and
3. they are not already with the collection agency.

This example is meant to show the result of how the logic is 'chained' together.

This usage example assumes a previously defined `OverdueSpecification` class that 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. The implementation of these classes isn't important here.

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.

```
var overdue = new OverdueSpecification();  
var noticeSent = new NoticeSentSpecification();  
var inCollection = new InCollectionSpecification();  
  
// Example of specification pattern logic chaining  
var sendToCollection = overdue.And(noticeSent).And(inCollection.Not());  
  
var invoices = InvoiceService.GetInvoices();  
  
foreach (var invoice in invoices)  
{  
    if (sendToCollection.IsSatisfiedBy(invoice))  
    {  
        invoice.SendToCollection();  
    }  
}
```

References

- Evans, Eric (2004). *Domain Driven Design*. Addison-Wesley. p. 224.

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
- The Specification Pattern: A Four Part Introduction using VB.Net (<http://www.codeproject.com/KB/architecture/SpecificationPart1.aspx>) by Richard Dalton
- The Specification Pattern in PHP (<https://github.com/mbrevda/SpecificationPattern>) by Moshe Brevda
- Happyr Doctrine Specification in PHP (<https://github.com/Happyr/Doctrine-Specification>) by Happyr
- The Specification Pattern in Swift (<https://github.com/neoneye/SpecificationPattern>) by Simon Strandgaard
- The Specification Pattern in TypeScript and JavaScript (<https://github.com/thiagodp/spec-pattern>) by Thiago Delgado Pinto
- specification pattern in flash actionscript 3 (<https://web.archive.org/web/20110724151447/http://www.dpd.nl/opensource/specification-pattern-for-selection-on-lists>) by Rolf Vreijdenberger

Retrieved from "https://en.wikipedia.org/w/index.php?title=Specification_pattern&oldid=1312009825"

State pattern

The state pattern is a behavioral software design pattern that allows an object to alter its behavior when its internal state changes. This pattern is close to the concept of finite-state machines. The state pattern can be interpreted as a strategy pattern, which is able to switch a strategy through invocations of methods defined in the pattern's interface.

The state pattern is used in computer programming to encapsulate varying behavior for the same object, based on its internal state. This can be a cleaner way for an object to change its behavior at runtime without resorting to conditional statements and thus improve maintainability.

Overview

The state design pattern is one of twenty-three design patterns documented by the Gang of Four that describe how to solve recurring design problems. Such problems cover the design of flexible and reusable object-oriented software, such as objects that are easy to implement, change, test, and reuse.

The state pattern is set to solve two main problems:

- An object should change its behavior when its internal state changes.
- State-specific behavior should be defined independently. That is, adding new states should not affect the behavior of existing states.

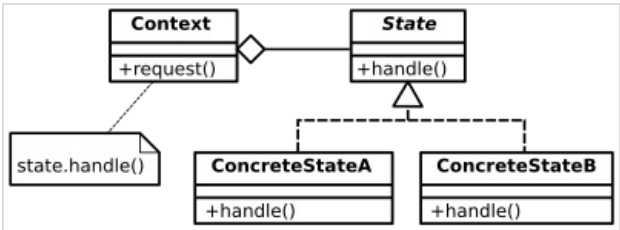
Implementing state-specific behavior directly within a class is inflexible because it commits the class to a particular behavior and makes it impossible to add a new state or change the behavior of an existing state later, independently from the class, without changing the class. In this, the pattern describes two solutions:

- Define separate (state) objects that encapsulate state-specific behavior for each state. That is, define an interface (state) for performing state-specific behavior, and define classes that implement the interface for each state.
- A class delegates state-specific behavior to its current state object instead of implementing state-specific behavior directly.

This makes a class independent of how state-specific behavior is implemented. New states can be added by defining new state classes. A class can change its behavior at run-time by changing its current state object.

Structure

In the accompanying Unified Modeling Language (UML) class diagram, the Context class doesn't implement state-specific behavior directly. Instead, Context refers to the State interface for performing state-specific behavior (state.handle()), which makes Context independent of how state-specific behavior is implemented. The ConcreteStateA and ConcreteStateB classes implement the State interface, that is, implement (encapsulate) the state-specific behavior for each state. The UML sequence diagram shows the run-time interactions:



State in UML

The Context object delegates state-specific behavior to different State objects. First, Context calls handle(this) on its current (initial) state object (ConcreteStateA), which performs the operation and calls setState(ConcreteStateB) on Context to change context's current state to ConcreteStateB. The next time, Context again calls handle(this) on its current state object (ConcreteStateB), which performs the operation and changes context's current state to ConcreteStateA.

See also

- Typestate analysis

References

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

2. "The State design pattern – Structure and Collaboration" (http://w3sdesign.com/?gr=b08&ugr=struct). w3sDesign.com. Retrieved 2017-08-12.

3. Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (<https://archive.org/details/designpatternsel00gamm/page/305>). Addison Wesley. pp. 305ff (<https://archive.org/details/designpatternsel00gamm/page/305>). ISBN 0-201-63361-2.
 4. "The State design pattern - Problem, Solution, and Applicability" (<http://w3sdesign.com/?gr=b08&ugr=proble>). *w3sDesign.com*. Retrieved 2017-08-12.
-

Retrieved from "https://en.wikipedia.org/w/index.php?title=State_pattern&oldid=1245339377"

Strategy pattern

In computer programming, the strategy pattern (also known as the policy pattern) is a behavioral software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives runtime instructions as to which in a family of algorithms to use.

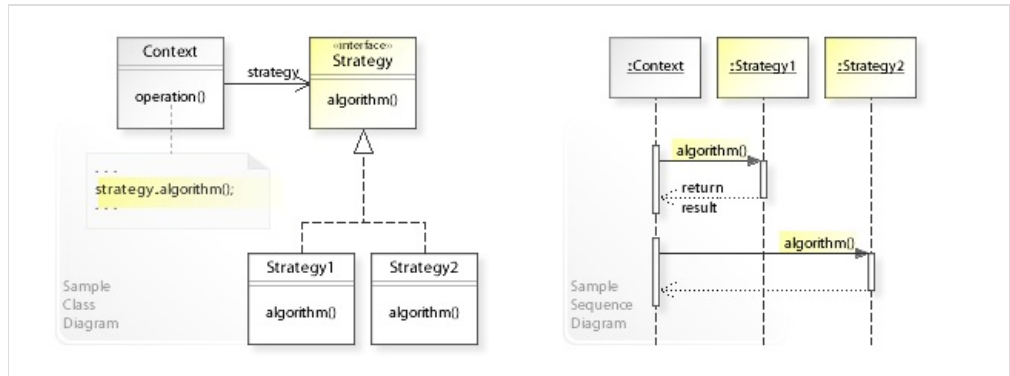
Strategy lets the algorithm vary independently from clients that use it. Strategy is one of the patterns included in the influential book Design Patterns by Gamma et al. that popularized the concept of using design patterns to describe how to design flexible and reusable object-oriented software. Deferring the decision about which algorithm to use until runtime allows the calling code to be more flexible and reusable.

For instance, a class that performs validation on incoming data may use the strategy pattern to select a validation algorithm depending on the type of data, the source of the data, user choice, or other discriminating factors. These factors are not known until runtime and may require radically different validation to be performed. The validation algorithms (strategies), encapsulated separately from the validating object, may be used by other validating objects in different areas of the system (or even different systems) without code duplication.

Typically, the strategy pattern stores a reference to code in a data structure and retrieves it. This can be achieved by mechanisms such as the native function pointer, the first-class function, classes or class instances in object-oriented programming languages, or accessing the language implementation's internal storage of code via reflection.

Structure

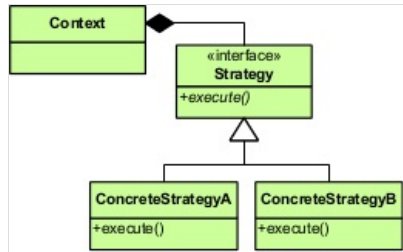
UML class and sequence diagram



A sample UML class and sequence diagram for the Strategy design pattern.

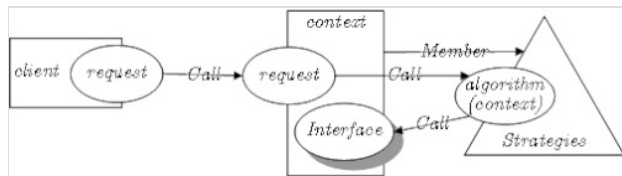
In the above UML class diagram, the Context class does not implement an algorithm directly. Instead, Context refers to the Strategy interface for performing an algorithm (strategy.algorithm()), which makes Context independent of how an algorithm is implemented. The Strategy1 and Strategy2 classes implement the Strategy interface, that is, implement (encapsulate) an algorithm. The UML sequence diagram shows the runtime interactions: The Context object delegates an algorithm to different Strategy objects. First, Context calls algorithm() on a Strategy1 object, which performs the algorithm and returns the result to Context. Thereafter, Context changes its strategy and calls algorithm() on a Strategy2 object, which performs the algorithm and returns the result to Context.

Class diagram



Strategy Pattern in UML

[5]

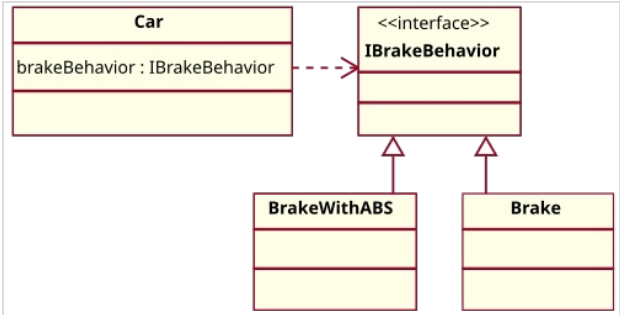


Strategy pattern in LePUS3 (legend (<http://lepus.org.uk/ref/legend/legend.xml>))

Strategy and open-closed principle

According to the strategy pattern, the behaviors of a class should not be inherited. Instead, they should be encapsulated using interfaces. This is compatible with the open-closed principle (OCP), which proposes that classes should be open for extension but closed for modification.

As an example, consider a car class. Two possible functionalities for car are *brake* and *accelerate*. Since accelerate and brake behaviors change frequently between models, a common approach is to implement these behaviors in subclasses. This approach has significant drawbacks; accelerate and brake behaviors must be declared in each new car model. The work of managing these behaviors increases greatly as the number of models increases, and requires code to be duplicated across models. Additionally, it is not easy to determine the exact nature of the behavior for each model without investigating the code in each.



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

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. 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 runtime 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:

```

Brake* brakeBehavior = new Brake();
  
```

```

package org.wikipedia.examples;

/* Encapsulated family of Algorithms
 * Interface and its implementations
 */
interface IBrakeBehavior {
    public void brake();
}

class BrakeWithABS implements IBrakeBehavior {
    public void brake() {
        System.out.println("Brake with ABS applied");
    }
}

class Brake implements IBrakeBehavior {
    public void brake() {
        System.out.println("Simple Brake applied");
    }
}

// Client that can use the algorithms above interchangeably
abstract class Car {
    private IBrakeBehavior brakeBehavior;

    public Car(IBrakeBehavior brakeBehavior) {
        this.brakeBehavior = brakeBehavior;
    }

    public void applyBrake() {
        brakeBehavior.brake();
    }

    public void setBrakeBehavior(IBrakeBehavior brakeType) {
        this.brakeBehavior = brakeType;
    }
}

// Client 1 uses one algorithm (Brake) in the constructor
class Sedan extends Car {
    public Sedan() {
        super(new Brake());
    }
}

// Client 2 uses another algorithm (BrakeWithABS) in the constructor
class SUV extends Car {
    public SUV() {
        super(new BrakeWithABS());
    }
}
  
```

```

    }
}

// Using the Car example
public class CarExample {
    public static void main(String[] arguments) {
        Car sedanCar = new Sedan();
        sedanCar.applyBrake(); // This will invoke class "Brake"

        Car suvCar = new SUV();
        suvCar.applyBrake(); // This will invoke class "BrakeWithABS"

        // set brake behavior dynamically
        suvCar.setBrakeBehavior(new Brake());
        suvCar.applyBrake(); // This will invoke class "Brake"
    }
}

```

See also

- [Dependency injection](#)
- [Higher-order function](#)
- [List of object-oriented programming terms](#)
- [Mixin](#)
- [Policy-based design](#)
- [Type class](#)
- [Entity–component–system](#)
- [Composition over inheritance](#)

References

1. "The Strategy design pattern - Problem, Solution, and Applicability" (<http://w3sdesign.com/?gr=b09&ugr=proble>). *w3sDesign.com*. Retrieved 2017-08-12.
2. Eric Freeman, Elisabeth Freeman, Kathy Sierra and Bert Bates, *Head First Design Patterns*, First Edition, Chapter 1, Page 24, O'Reilly Media, Inc, 2004. ISBN 978-0-596-00712-6
3. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (<https://archive.org/details/designpatternsel00gamm/page/315>). Addison Wesley. pp. 315ff (<https://archive.org/details/designpatternsel00gamm/page/315>). ISBN 0-201-63361-2.
4. "The Strategy design pattern - Structure and Collaboration" (<http://w3sdesign.com/?gr=b09&ugr=struct>). *w3sDesign.com*. Retrieved 2017-08-12.
5. "Design Patterns Quick Reference – McDonaldLand" (<http://www.mcdonaldland.info/2007/11/28/40/>).

External links

- [Strategy Pattern in UML](http://design-patterns-with-uml.blogspot.com.ar/2013/02/strategy-pattern.html) (<http://design-patterns-with-uml.blogspot.com.ar/2013/02/strategy-pattern.html>) (in Spanish)
- Geary, David (April 26, 2002). "Strategy for success" (<https://www.infoworld.com/article/2074195/strategy-for-success.html>). Java Design Patterns. *JavaWorld*. Retrieved 2020-07-20.
- [Strategy Pattern for C article](http://www.adamtornhill.com/Patterns%20in%20C%203,%20STRATEGY.pdf) (<http://www.adamtornhill.com/Patterns%20in%20C%203,%20STRATEGY.pdf>)
- [Refactoring: Replace Type Code with State/Strategy](http://martinfowler.com/refactoring/catalog/replaceTypeCodeWithStateStrategy.html) (<http://martinfowler.com/refactoring/catalog/replaceTypeCodeWithStateStrategy.html>)
- [The Strategy Design Pattern](https://web.archive.org/web/20170415154927/http://www.aleccove.com/2016/02/the-strategy-pattern-in-javascript) (<https://web.archive.org/web/20170415154927/http://www.aleccove.com/2016/02/the-strategy-pattern-in-javascript>) at the [Wayback Machine](#) (archived 2017-04-15) Implementation of the Strategy pattern in JavaScript

Retrieved from "https://en.wikipedia.org/w/index.php?title=Strategy_pattern&oldid=1316974000"

Template method pattern

In object-oriented programming, the **template method** is one of the behavioral design patterns identified by Gamma et al.^[1] in the book *Design Patterns*. The template method is a method in a superclass, usually an abstract superclass, and defines the skeleton of an operation in terms of a number of high-level steps. These steps are themselves implemented by additional *helper methods* in the same class as the *template method*.

The *helper methods* may be either *abstract methods*, in which case subclasses are required to provide concrete implementations, or *hook methods*, which have empty bodies in the superclass. Subclasses can (but are not required to) customize the operation by overriding the hook methods. The intent of the template method is to define the overall structure of the operation, while allowing subclasses to refine, or redefine, certain steps.^[2]

Overview

This pattern has two main parts:

- The "template method" is implemented as a method in a base class (usually an abstract class). This method contains code for the parts of the overall algorithm that are invariant. The template ensures that the overarching algorithm is always followed.^[1] In the template method, portions of the algorithm that may *vary* are implemented by sending self messages that request the execution of additional *helper methods*. In the base class, these helper methods are given a default implementation, or none at all (that is, they may be abstract methods).
- Subclasses of the base class "fill in" the empty or "variant" parts of the "template" with specific algorithms that vary from one subclass to another.^[3] It is important that subclasses do *not* override the *template method* itself.

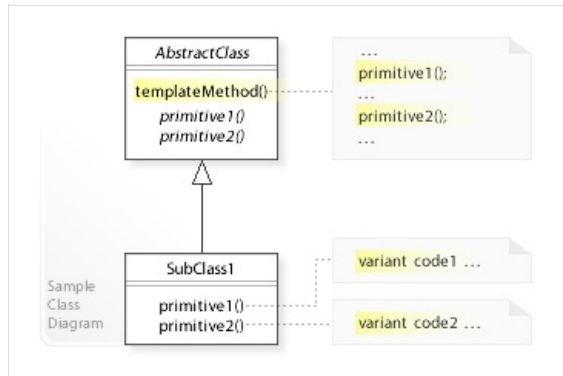
At run-time, the algorithm represented by the template method is executed by sending the template message to an instance of one of the concrete subclasses. Through inheritance, the template method in the base class starts to execute. When the template method sends a message to self requesting one of the helper methods, the message will be received by the concrete sub-instance. If the helper method has been overridden, the overriding implementation in the sub-instance will execute; if it has not been overridden, the inherited implementation in the base class will execute. This mechanism ensures that the overall algorithm follows the same steps every time while allowing the details of some steps to depend on which instance received the original request to execute the algorithm.

This pattern is an example of inversion of control because the high-level code no longer determines what algorithms to run; a lower-level algorithm is instead selected at run-time.

Some of the self-messages sent by the template method may be to *hook methods*. These methods are implemented in the same base class as the template method, but with empty bodies (i.e., they do nothing). Hook methods exist so that subclasses can override them, and can thus fine-tune the action of the algorithm *without* the need to override the template method itself. In other words, they provide a "hook" on which to "hang" variant implementations.

Structure

UML class diagram

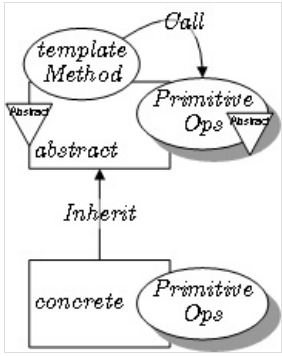


A sample UML class diagram for the Template Method design pattern.^[4]

In the above UML class diagram, the **AbstractClass** defines a `templateMethod()` operation that defines the skeleton (template) of a behavior by

- implementing the invariant parts of the behavior and
- sending to **self** the messages `primitive1()` and `primitive2()` , which, because they are implemented in **SubClass1** , allow

that subclass to provide a variant implementation of those parts of the algorithm.



Template Method in
LePUS3.^[5]

Usage

The *template method* is used in frameworks, where each implements the invariant parts of a domain's architecture, while providing hook methods for customization. This is an example of inversion of control. The template method is used for the following reasons.^[3]

- It lets subclasses implement varying behavior (through overriding of the hook methods).^[6]
- It avoids duplication in the code: the general workflow of the algorithm is implemented once in the abstract class's template method, and necessary variations are implemented in the subclasses.^[6]
- It controls the point(s) at which specialization is permitted. If the subclasses were to simply override the template method, they could make radical and arbitrary changes to the workflow. In contrast, by overriding only the hook methods, only certain specific details of the workflow can be changed,^[6] and the overall workflow is left intact.

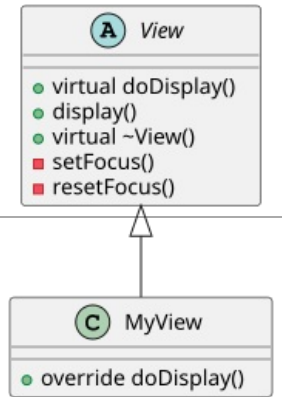
Use with code generators

The template pattern is useful when working with auto-generated code. The challenge of working with generated code is that changes to the source code will lead to changes in the generated code; if hand-written modifications have been made to the generated code, these will be lost. How, then, should the generated code be customized?

The Template pattern provides a solution. If the generated code follows the template method pattern, the generated code will all be an abstract superclass. Provided that hand-written customizations are confined to a subclass, the code generator can be run again without risk of over-writing these modifications. When used with code generation, this pattern is sometimes referred to as the generation gap pattern.^[7]

C++ example

This C++23 implementation is based on the pre C++98 implementation in the book.



```
import std;
using std::unique_ptr;

// abstract class
class View {
private:
    void setFocus() {
        std::println("View::setFocus called");
    }

    void resetFocus() {
        std::println("View::resetFocus called");
    }
}
```

```

    }
    public:
        // defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.
        virtual void doDisplay() = 0;

        // implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations
        // defined in AbstractClass or those of other objects.
        void display() {
            setFocus();
            doDisplay();
            resetFocus();
        }

        virtual ~View() = default;
};

// concrete class
class MyView : public View {
public:
    // implements the primitive operations to carry out subclass-specific steps of the algorithm.
    void doDisplay() override {
        // render the view's contents
        std::println("MyView::doDisplay called");
    }
};

int main(int argc, char* argv[]) {
    unique_ptr<View> myview = std::make_unique<MyView>();
    myview->display();
}

```

The program output is

```

View::setFocus called
MyView::doDisplay called
View::resetFocus called

```

See also

- [Inheritance \(object-oriented programming\)](#)
- [Method overriding \(programming\)](#)
- [GRASP \(object-oriented designer\)](#)
- [Adapter pattern](#)
- [Strategy pattern](#)

References

1. Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). "Template Method". *Design Patterns*. Addison-Wesley. pp. 325–330 (<https://archive.org/details/designpatternsel00gamm/page/325>). ISBN 0-201-63361-2.
2. Freeman, Eric; Freeman, Elisabeth; Sierra, Kathy; Bates, Bert (2004). Hendrickson, Mike; Loukides, Mike (eds.). *Head First Design Patterns* (<http://shop.oreilly.com/product/9780596007126.do>) (paperback). Vol. 1. O'REILLY. pp. 289, 311. ISBN 978-0-596-00712-6. Retrieved 2012-09-12.
3. "Template Method Design Pattern" (http://sourcemaking.com/design_patterns/template_method). Source Making - teaching IT professional. Retrieved 2012-09-12. "Template Method is used prominently in frameworks."
4. "The Template Method design pattern - Structure" (<http://w3sdesign.com/?gr=b10&ugr=struct>). *w3sDesign.com*. Retrieved 2017-08-12.
5. LePUS3 legend. Retrieved from <http://lepus.org.uk/ref/legend/legend.xml>.
6. Chung, Carlo (2011). *Pro Objective-C Design Patterns for iOS*. Berkeley, CA: Apress. p. 266. ISBN 978-1-4302-3331-2.
7. Vlissides, John (1998-06-22). *Pattern Hatching: Design Patterns Applied* (<http://www.informit.com/store/pattern-hatching-design-patterns-applied-9780201432930>). Addison-Wesley Professional. pp. 85–101. ISBN 978-0201432930.

External links

- [Six common uses of the template pattern](https://www.codeproject.com/Articles/307452/common-use-of-Template-Design-pattern-Design-pat) ([https://www.codeproject.com/Articles/307452/common-use-of-Template-Design-pat](https://www.codeproject.com/Articles/307452/common-use-of-Template-Design-pattern-Design-pat))
- [Template Method Design Pattern](http://sourcemaking.com/design_patterns/template_method) (http://sourcemaking.com/design_patterns/template_method)

Visitor pattern

A **visitor pattern** is a software design pattern that separates the algorithm from the object structure. Because of this separation, new operations can be added to existing object structures without modifying the structures. It is one way to follow the open/closed principle in object-oriented programming and software engineering.

In essence, the visitor allows adding new virtual functions to a family of classes, without modifying the classes. Instead, a visitor class is created 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.

Programming languages with sum types and pattern matching obviate many of the benefits of the visitor pattern, as the visitor class is able to both easily branch on the type of the object and generate a compiler error if a new object type is defined which the visitor does not yet handle.

Overview

The Visitor^[1] design pattern is one of the twenty-three well-known *Gang of Four design patterns* that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

Problems, the Visitor design pattern can solve

- It should be possible to define a new operation for (some) classes of an object structure without changing the classes.

When new operations are needed frequently and the object structure consists of many unrelated classes, it's inflexible to add new subclasses each time a new operation is required because "[..] distributing all these operations across the various node classes leads to a system that's hard to understand, maintain, and change."^[1]

Solution, the Visitor design pattern describes

- Define a separate (visitor) object that implements an operation to be performed on elements of an object structure.
- Clients traverse the object structure and call a *dispatching operation accept (visitor)* on an element — that "dispatches" (delegates) the request to the "accepted visitor object". The visitor object then performs the operation on the element ("visits the element").

This makes it possible to create new operations independently from the classes of an object structure by adding new visitor objects.

See also the UML class and sequence diagram below.

Definition

The Gang of Four defines the Visitor as:

Represent[ing] an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

The nature of the Visitor makes it an ideal pattern to plug into public APIs, thus allowing its clients to perform operations on a class using a "visiting" class without having to modify the source.^[2]

Advantages

Moving operations into visitor classes is beneficial when

- many unrelated operations on an object structure are required,
- the classes that make up the object structure are known and not expected to change,
- new operations need to be added frequently,
- an algorithm involves several classes of the object structure, but it is desired to manage it in one single location,
- an algorithm needs to work across several independent class hierarchies.

A drawback to this pattern, however, is that it makes extensions to the class hierarchy more difficult, as new classes typically require a new `visit` method to be added to each visitor.

Application

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

A fundamental operation on this type hierarchy is saving a drawing to the system's native file format. At first glance, it may seem acceptable to add local save methods to all types in the hierarchy. But it is also useful to be able to save drawings to other file formats. Adding ever more methods for saving into many different file formats soon clutters the relatively pure original geometric data structure.

A naive way to solve this would be to maintain separate functions for each file format. Such a save function would take a drawing as input, traverse it, and encode into that specific file format. As this is done for each added different format, duplication between the functions accumulates. For example, saving a circle shape in a raster format requires very similar code no matter what specific raster form is used, and is different from other primitive shapes. The case for other primitive shapes like lines and polygons is similar. Thus, the code becomes a large outer loop traversing through the objects, with a large decision tree inside the loop querying the type of the object. Another problem with this approach is that it is very easy to miss a shape in one or more savers, or a new primitive shape is introduced, but the save routine is implemented only for one file type and not others, leading to code extension and maintenance problems. As the versions of the same file grows it becomes more complicated to maintain it.

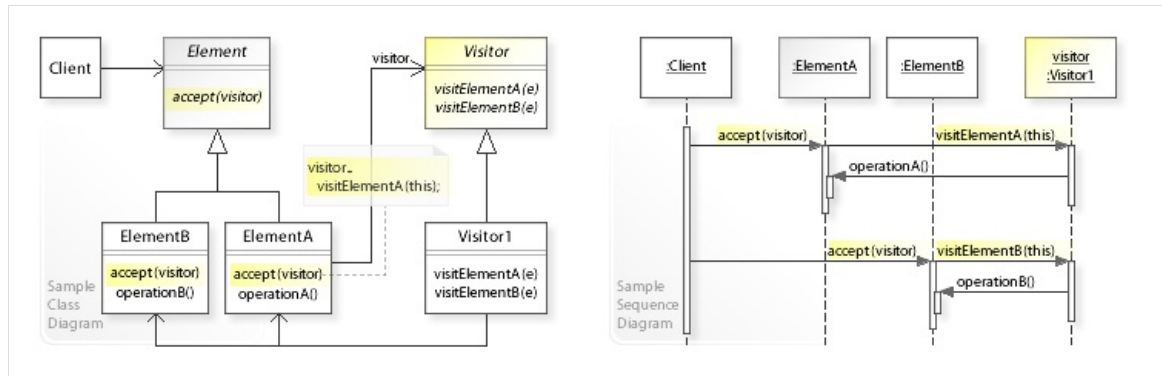
Instead, the visitor pattern can be applied. It encodes the logical operation (i.e. save(image_tree)) on the whole hierarchy into one class (i.e. Saver) that implements the common methods for traversing the tree and describes virtual helper methods (i.e. save_circle, save_square, etc.) to be implemented for format specific behaviors. In the case of the CAD example, such format specific behaviors would be implemented by a subclass of Visitor (i.e. SaverPNG). As such, all duplication of type checks and traversal steps is removed. Additionally, the compiler now complains if a shape is omitted since it is now expected by the common base traversal/save function.

Iteration loops

The visitor pattern may be used for iteration over container-like data structures just like Iterator pattern but with limited functionality.^{[3]:288} For example, iteration over a directory structure could be implemented by a function class instead of more conventional loop pattern. This would allow deriving various useful information from directories content by implementing a visitor functionality for every item while reusing the iteration code. It's widely employed in Smalltalk systems and can be found in C++ as well.^{[3]:289} A drawback of this approach, however, is that you can't break out of the loop easily or iterate concurrently (in parallel i.e. traversing two containers at the same time by a single i variable).^{[3]:289} The latter would require writing additional functionality for a visitor to support these features.^{[3]:289}

Structure

UML class and sequence diagram



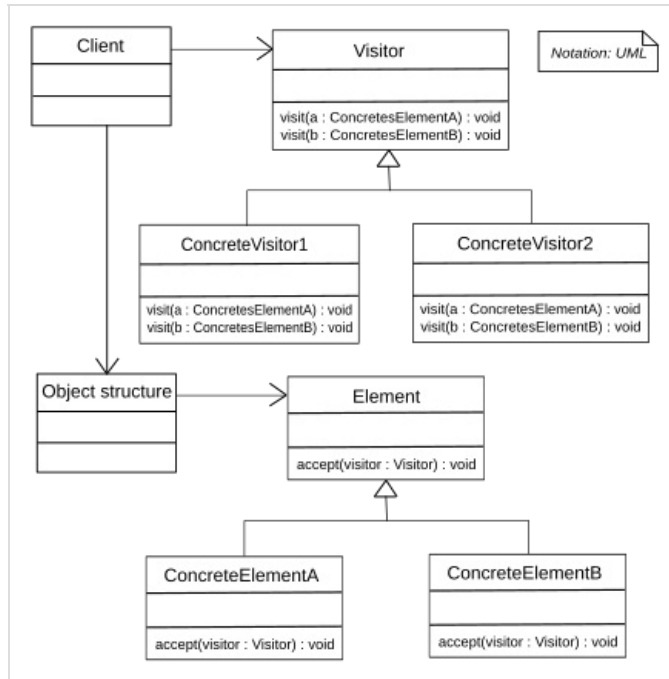
A sample UML class diagram and sequence diagram for the Visitor design pattern.^[4]

In the UML class diagram above, the ElementA class doesn't implement a new operation directly. Instead, ElementA implements a *dispatching operation* accept(visitor) that "dispatches" (delegates) a request to the "accepted visitor object" (visitor.visitElementA(this)). The Visitor1 class implements the operation (visitElementA(e:ElementA)). ElementB then implements accept(visitor) by dispatching to visitor.visitElementB(this). The Visitor1 class implements the operation (visitElementB(e:ElementB)).

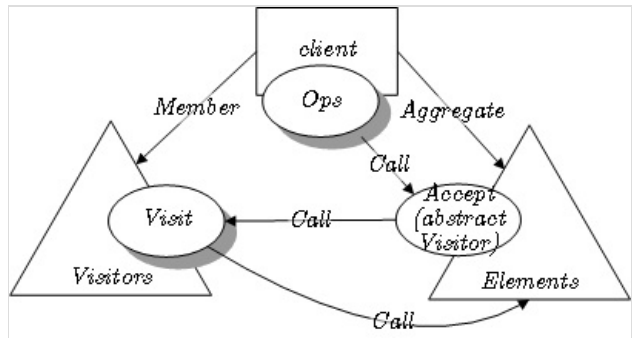
The UML sequence diagram shows the run-time interactions: The Client object traverses the elements of an object structure (ElementA, ElementB) and calls accept(visitor) on each element. First, the Client calls accept(visitor) on ElementA, which calls visitElementA(this) on the accepted visitor object.

The element itself (this) is passed to the visitor so that it can "visit" ElementA (call operationA()). Thereafter, the Client calls accept(visitor) on ElementB, which calls visitElementB(this) on the visitor that "visits" ElementB (calls operationB()).

Class diagram



Visitor in Unified Modeling Language (UML).[5]:381



Visitor in LePUS3 (legend (<http://lepus.org.uk/ref/legend/legend.xml>))

Details

The visitor pattern requires a programming language that supports single dispatch, as common object-oriented languages (such as C++, Java, Smalltalk, Objective-C, Swift, JavaScript, Python and C#) do. Under this condition, consider two objects, each of some class type; one is termed the *element*, and the other is *visitor*.

Objects

Visitor

The *visitor* declares a `visit` method, which takes the element as an argument, for each class of element. *Concrete visitors* are derived from the visitor class and implement these `visit` methods, each of which implements part of the algorithm operating on the object structure. The state of the algorithm is maintained locally by the concrete visitor class.

Element

The *element* declares an `accept` method to accept a visitor, taking the visitor as an argument. *Concrete elements*, derived from the element class, implement the `accept` method. In its simplest form, this is no more than a call to the visitor's `visit` method. Composite elements, which maintain a list of child objects, typically iterate over these, calling each child's `accept` method.

Client

The *client* creates the object structure, directly or indirectly, and instantiates the concrete visitors. When an operation is to be performed which is implemented using the Visitor pattern, it calls the `accept` method of the top-level element(s).

Methods

Accept

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

Visit

Thus, the implementation of the `visit` method is chosen based on both the dynamic type of the element and the dynamic type of the visitor. This effectively implements double dispatch. For languages whose object systems support multiple dispatch, not only single dispatch, such as Common Lisp or C# via the Dynamic Language Runtime (DLR), implementation of the visitor pattern is greatly simplified (a.k.a. Dynamic Visitor) by allowing use of simple function overloading to cover all the cases being visited. A dynamic visitor, provided it operates on public data only, conforms to the open/closed principle (since it does not modify extant structures) and to the single responsibility principle (since it implements the Visitor pattern in a separate component).

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

Examples

C#

This example declares a separate `ExpressionPrintingVisitor` class that takes care of the printing. If the introduction of a new concrete visitor is desired, a new class will be created to implement the Visitor interface, and new implementations for the `Visit` methods will be provided. The existing classes (`Literal` and `Addition`) will remain unchanged.

```
using System;

namespace Wikipedia;

public interface Visitor
{
    void Visit(Literal literal);
    void Visit(Addition addition);
}

public class ExpressionPrintingVisitor : Visitor
{
    public void Visit(Literal literal)
    {
        Console.WriteLine(literal.Value);
    }

    public void Visit(Addition addition)
    {
        double leftValue = addition.Left.GetValue();
        double rightValue = addition.Right.GetValue();
        var sum = addition.GetValue();
        Console.WriteLine($"{leftValue} + {rightValue} = {sum}");
    }
}

public abstract class Expression
{
    public abstract void Accept(Visitor v);

    public abstract double GetValue();
}

public class Literal : Expression
{
    public Literal(double value)
    {
        this.Value = value;
    }

    public double Value { get; set; }

    public override void Accept(Visitor v)
    {
        v.Visit(this);
    }

    public override double GetValue()
    {
        return Value;
    }
}

public class Addition : Expression
{

```

```

public Addition(Expression left, Expression right)
{
    Left = left;
    Right = right;
}

public Expression Left { get; set; }
public Expression Right { get; set; }

public override void Accept(Visitor v)
{
    Left.Accept(v);
    Right.Accept(v);
    v.Visit(this);
}

public override double GetValue()
{
    return Left.GetValue() + Right.GetValue();
}
}

public static class Program
{
    public static void Main(string[] args)
    {
        // Emulate 1 + 2 + 3
        var e = new Addition(
            new Addition(
                new Literal(1),
                new Literal(2)
            ),
            new Literal(3)
        );

        var printingVisitor = new ExpressionPrintingVisitor();
        e.Accept(printingVisitor);
        Console.ReadKey();
    }
}

```

Smalltalk

In this case, it is the object's responsibility to know how to print itself on a stream. The visitor here is then the object, not the stream.

"There's no syntax for creating a class. Classes are created by sending messages to other classes."

```

WriteStream subclass: #ExpressionPrinter
instanceVariableNames: ''
classVariableNames: ''
package: 'Wikipedia'.

```

```

ExpressionPrinter>>write: anObject
"Delegates the action to the object. The object doesn't need to be of any special
class; it only needs to be able to understand the message #putOn:"
anObject putOn: self.
^ anObject.

```

```

Object subclass: #Expression
instanceVariableNames: ''
classVariableNames: ''
package: 'Wikipedia'.

```

```

Expression subclass: #Literal
instanceVariableNames: 'value'
classVariableNames: ''
package: 'Wikipedia'.

```

```

Literal class>>with: aValue
"Class method for building an instance of the Literal class"
^ self new
    value: aValue;
    yourself.

```

```

Literal>>value: aValue
"Setter for value"
value := aValue.

```

```

Literal>>putOn: aStream
"A Literal object knows how to print itself"
aStream nextPutAll: value asString.

```

```

Expression subclass: #Addition
instanceVariableNames: 'left right'
classVariableNames: ''
package: 'Wikipedia'.

```

```

Addition class>>left: a right: b
"Class method for building an instance of the Addition class"
^ self new
    left: a;
    right: b;
    yourself.

```

```

Addition>>left: anExpression
"Setter for left"
left := anExpression.

```

```

Addition>>right: anExpression
"Setter for right"
right := anExpression.

```

```

Addition>>putOn: aStream
"An Addition object knows how to print itself"

```

```

aStream nextPut: $(.
left putOn: aStream.
aStream nextPut: $+.
right putOn: aStream.
aStream nextPut: $).

```

```

Object subclass: #Program
instanceVariableNames: ''
classVariableNames: ''
package: 'Wikipedia'.

```

```

Program>>main
| expression stream |
expression := Addition
    left: (Addition
        left: (Literal with: 1)
        right: (Literal with: 2))
    right: (Literal with: 3).
stream := ExpressionPrinter on: (String new: 100).
stream write: expression.
Transcript show: stream contents.
Transcript flush.

```

Go

Go does not support method overloading, so the visit methods need different names. A typical visitor interface might be

```

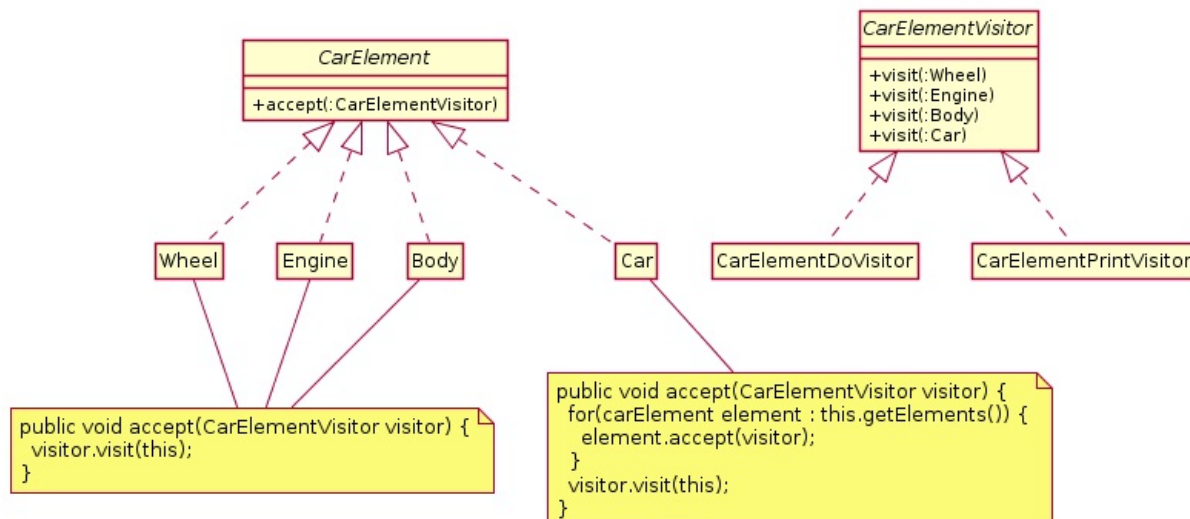
type Visitor interface {
visitWheel(wheel Wheel) string
visitEngine(engine Engine) string
visitBody(body Body) string
visitCar(car Car) string
}

```

Java

The following example is in the language Java, and shows how the contents of a tree of nodes (in this case describing the components of a car) can be printed. Instead of creating print methods for each node subclass (Wheel, Engine, Body, and Car), one visitor class (CarElementPrintVisitor) performs the required printing action. Because different node subclasses require slightly different actions to print properly, CarElementPrintVisitor dispatches actions based on the class of the argument passed to its visit method. CarElementDoVisitor, which is analogous to a save operation for a different file format, does likewise.

Diagram



Sources

```

import java.util.List;

interface CarElement {
    void accept(CarElementVisitor visitor);
}

interface CarElementVisitor {
    void visit(Body body);
    void visit(Car car);
    void visit(Engine engine);
    void visit(Wheel wheel);
}

class Wheel implements CarElement {
    private final String name;
}

```

```

    public Wheel(final String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

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

class Body implements CarElement {
    @Override
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Engine implements CarElement {
    @Override
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Car implements CarElement {
    private final List<CarElement> elements;

    public Car() {
        this.elements = List.of(
            new Wheel("front left"),
            new Wheel("front right"),
            new Wheel("back left"),
            new Wheel("back right"),
            new Body(),
            new Engine()
        );
    }

    @Override
    public void accept(CarElementVisitor visitor) {
        for (CarElement element : elements) {
            element.accept(visitor);
        }
        visitor.visit(this);
    }
}

class CarElementDoVisitor implements CarElementVisitor {
    @Override
    public void visit(Body body) {
        System.out.println("Moving my body");
    }

    @Override
    public void visit(Car car) {
        System.out.println("Starting my car");
    }

    @Override
    public void visit(Wheel wheel) {
        System.out.printf("Kicking my %s wheel%n", wheel.getName());
    }

    @Override
    public void visit(Engine engine) {
        System.out.println("Starting my engine");
    }
}

class CarElementPrintVisitor implements CarElementVisitor {
    @Override
    public void visit(Body body) {
        System.out.println("Visiting body");
    }

    @Override
    public void visit(Car car) {
        System.out.println("Visiting car");
    }

    @Override
    public void visit(Engine engine) {
        System.out.println("Visiting engine");
    }

    @Override
    public void visit(Wheel wheel) {
        System.out.println("Visiting %s wheel%n", wheel.getName());
    }
}

public class VisitorDemo {
    public static void main(String[] args) {

```

```

    Car car = new Car();

    car.accept(new CarElementPrintVisitor());
    car.accept(new CarElementDoVisitor());
  }
}

```

Output

```

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

```

Common Lisp

Sources

```

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

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

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

(defclass wheel (auto-part) ())

(defclass body (auto-part) ())

(defclass engine (auto-part) ())

(defgeneric traverse (function object other-object))

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

;; do-something visitations

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

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

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

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

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

(let ((a (make-instance 'auto
  :elements `((, (make-instance 'wheel :name "front-left-wheel")
    ,(make-instance 'wheel :name "front-right-wheel")
    ,(make-instance 'wheel :name "rear-left-wheel")
    ,(make-instance 'wheel :name "rear-right-wheel")
    ,(make-instance 'body :name "body")
    ,(make-instance 'engine :name "engine")))))

  ;; traverse to print elements
  ;; stream *standard-output* plays the role of other-object here
  (traverse #'print a *standard-output*)

  (terpri) ;; print newline

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

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

```

Output

```

"front-left-wheel"
"front-right-wheel"

```

```

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

```

Notes

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

```

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

;; ...

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

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

```

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

Python

Python does not support method overloading in the classical sense (polymorphic behavior according to type of passed parameters), so the "visit" methods for the different model types need to have different names.

Sources

```

"""
Visitor pattern example.
"""

from abc import ABCMeta, abstractmethod
from typing import NoReturn

NOT_IMPLEMENTED: str = "You should implement this."

class CarElement(metaclass = ABCMeta):
    @abstractmethod
    def accept(self, visitor) -> NoReturn:
        raise NotImplementedError(NOT_IMPLEMENTED)

class Body(CarElement):
    def accept(self, visitor: CarElementVisitor) -> None:
        visitor.visit_body(self)

class Engine(CarElement):
    def accept(self, visitor: CarElementVisitor) -> None:
        visitor.visit_engine(self)

class Wheel(CarElement):
    def __init__(self, name: str) -> None:
        self.name = name

    def accept(self, visitor: CarElementVisitor) -> None:
        visitor.visit_wheel(self)

class Car(CarElement):
    def __init__(self) -> None:
        self.elements: list[CarElement] = [
            Wheel("front left"),
            Wheel("front right"),
            Wheel("back left"),
            Wheel("back right"),
            Body(),
            Engine()
        ]

    def accept(self, visitor):
        for element in self.elements:
            element.accept(visitor)
        visitor.visit_car(self)

class CarElementVisitor(metaclass = ABCMeta):
    @abstractmethod

```

```

def visit_body(self, element: CarElement) -> NoReturn:
    raise NotImplementedError(NOT_IMPLEMENTED)

@abstractmethod
def visit_engine(self, element: CarElement) -> NoReturn:
    raise NotImplementedError(NOT_IMPLEMENTED)

@abstractmethod
def visit_wheel(self, element: CarElement) -> NoReturn:
    raise NotImplementedError(NOT_IMPLEMENTED)

@abstractmethod
def visit_car(self, element: CarElement) -> NoReturn:
    raise NotImplementedError(NOT_IMPLEMENTED)

class CarElementDoVisitor(CarElementVisitor):
    def visit_body(self, body: Body) -> None:
        print("Moving my body.")

    def visit_car(self, car: Car) -> None:
        print("Starting my car.")

    def visit_wheel(self, wheel: Wheel) -> None:
        print(f"Kicking my {wheel.name} wheel.")

    def visit_engine(self, engine: Engine) -> None:
        print("Starting my engine.")

class CarElementPrintVisitor(CarElementVisitor):
    def visit_body(self, body: Body) -> None:
        print("Visiting body.")

    def visit_car(self, car: Car) -> None:
        print("Visiting car.")

    def visit_wheel(self, wheel: Wheel) -> None:
        print(f"Visiting {wheel.name} wheel.")

    def visit_engine(self, engine: Engine) -> None:
        print("Visiting engine.")

if __name__ == "__main__":
    car: Car = Car()
    car.accept(CarElementPrintVisitor())
    car.accept(CarElementDoVisitor())

```

Output

```

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

```

Abstraction

Using Python 3 or above allows to make a general implementation of the accept method:

```

class Visitable:
    def accept(self, visitor: Visitor) -> Any:
        lookup: str = f"visit_{self.__qualname__.replace('.', '_')}"
        return getattr(visitor, lookup)(self)

```

One could extend this to iterate over the class's method resolution order if they would like to fall back on already-implemented classes. They could also use the subclass hook feature to define the lookup in advance.

Related design patterns

- Iterator pattern – defines a traversal principle like the visitor pattern, without making a type differentiation within the traversed objects
- Church encoding – a related concept from functional programming, in which tagged union/sum types may be modeled using the behaviors of "visitors" on such types, and which enables the visitor pattern to emulate variants and patterns.

See also

- Algebraic data type
- Double dispatch

- [Multiple dispatch](#)
- [Function object](#)

References

1. Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (<https://archive.org/details/designpatternsel00gamm/page/331>). Addison Wesley. pp. 331ff (<https://archive.org/details/designpatternsel00gamm/page/331>). ISBN 0-201-63361-2.
2. Coogan, Corey (June 16, 2009). "Visitor Pattern: A Real World Example" (<https://coreycoogan.wordpress.com/2009/06/16/visitor-pattern-real-world-example>) – via [WordPress.com](#).
3. Budd, Timothy (1997). *An introduction to object-oriented programming* (2nd ed.). Reading, Mass.: Addison-Wesley. ISBN 0-201-82419-1. OCLC 34788238 (<https://search.worldcat.org/oclc/34788238>).
4. "The Visitor design pattern - Structure and Collaboration" (<http://w3sdesign.com/?gr=b11&ugr=struct>). *w3sDesign.com*. Retrieved 2017-08-12.
5. Reddy, Martin (2011). *API design for C++*. Boston: Morgan Kaufmann. ISBN 978-0-12-385004-1. OCLC 704559821 (<https://search.worldcat.org/oclc/704559821>).

External links

- The Visitor Family of Design Patterns (<https://web.archive.org/web/20151022084246/http://objectmentor.com/resources/articles/visitor.pdf>) at the Wayback Machine (archived October 22, 2015). Additional archives: April 12, 2004 (<https://www.webcitation.org/66gH7HEVW?url=http://objectmentor.com/resources/articles/visitor.pdf>), March 5, 2002 (<https://drive.google.com/file/d/0BwhCYaYDn8EgNWlWZDg2YjUtNDRhNi00NzQ2LWFmNmMtYmYxNGI5ZjEwZDZj/view>). 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 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://web.cs.ucla.edu/~palsberg/paper/compsac98.pdf>)" by Jens Palsberg and C. Barry Jay. 1997 IEEE-CS COMPSAC paper showing that accept() methods are unnecessary when reflection is available; introduces term 'Walkabout' for the technique.
- Article "A Time for Reflection (<https://web.archive.org/web/20030408205824/http://www.polyglotinc.com/reflection.html>)" by Bruce Wallace – subtitled "*Java 1.2's reflection capabilities eliminate burdensome accept() methods from your Visitor pattern*"
- Visitor Pattern (http://www.oodeesign.com/oo_design_patterns/behavioral_patterns/visitor_pattern.html) using reflection(java).
- PerfectJPattern Open Source Project (<https://perfectjpattern.sourceforge.net/dp-visitor.html>), Provides a context-free and type-safe implementation of the Visitor Pattern in Java based on Delegates.
- Visitor Design Pattern (http://sourcemaking.com/design_patterns/visitor)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Visitor_pattern&oldid=1318932434"

SOLID

In object-oriented programming, **SOLID** is a mnemonic acronym for five principles intended to make source code more understandable, flexible, and maintainable. Although the principles apply to object-oriented programming, they can also form a core philosophy for methodologies such as agile software development and adaptive software development.^[1]

Software engineer and instructor Robert C. Martin^{[2][3][1]} introduced the basic principles of SOLID design in his 2000 paper *Design Principles and Design Patterns* about software rot.^{[3][4]:2–3} The *SOLID* acronym was coined around 2004 by Michael Feathers.^[5]

Principles

Single responsibility principle

The single-responsibility principle (SRP) states that there should never be more than one reason for a class to change.^[6] In other words, every class should have only one responsibility.^[7]

Importance:

- **Maintainability:** When classes have a single, well-defined responsibility, they're easier to understand and modify.
- **Testability:** It's easier to write unit tests for classes with a single focus.
- **Flexibility:** Changes to one responsibility don't affect unrelated parts of the system.^[7]

Open–closed principle

The open–closed principle (OCP) states that software entities should be open for extension, but closed for modification.^[8]

Importance:

- **Extensibility:** New features can be added without modifying existing code.
- **Stability:** Reduces the risk of introducing bugs when making changes.
- **Flexibility:** Adapts to changing requirements more easily.

Liskov substitution principle

The Liskov substitution principle (LSP) states that functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.^[9] ^[9]

Importance:

- **Polymorphism:** Enables the use of polymorphic behavior, making code more flexible and reusable.
- **Reliability:** Ensures that subclasses adhere to the contract defined by the superclass.
- **Predictability:** Guarantees that replacing a superclass object with a subclass object won't break the program.^[9]

Interface segregation principle

The interface segregation principle (ISP) states that "clients should not be forced to depend upon interfaces that they do not use."^{[10][4]}

Importance:

- **Decoupling:** Reduces dependencies between classes, making the code more modular and maintainable.
- **Flexibility:** Allows for more targeted implementations of interfaces.
- **Avoids unnecessary dependencies:** Clients don't have to depend on methods they don't use.

Dependency inversion principle

The dependency inversion principle (DIP) states to depend upon abstractions, not concretes.^{[11][4]}

Importance:

- **Loose coupling:** Reduces dependencies between modules, making the code more flexible and easier to test.

- **Flexibility:** Enables changes to implementations without affecting clients.
- **Maintainability:** Makes code easier to understand and modify.^{[1][4]}

See also

- [Code reuse](#)
- [GRASP \(object-oriented design\)](#)
- [Inheritance \(object-oriented programming\)](#)
- [List of software development philosophies](#)

References

1. Metz, Sandi (May 2009). "SOLID Object-Oriented Design" (<https://www.youtube.com/watch?v=v-2yFMzxqwU>). *YouTube*. Archived (<https://ghostarchive.org/varchive/youtube/20211221/v-2yFMzxqwU>) from the original on 2021-12-21. Retrieved 2019-08-13. Talk given at the 2009 Gotham Ruby Conference.
2. Martin, Robert C. "Principles Of OOD" (<http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>). *ButUncleBob.com*. Archived (<https://web.archive.org/web/20140910201842/http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>) from the original on Sep 10, 2014. Retrieved 2014-07-17.. (Note the reference to "the first five principles", although the acronym is not used in this article.) Dates back to at least 2003.
3. Martin, Robert C. (13 Feb 2009). "Getting a SOLID start" (<https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start>). *Uncle Bob Consulting LLC (Google Sites)*. Archived (<https://web.archive.org/web/20130917122741/https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start>) from the original on Sep 17, 2013. Retrieved 2013-08-19.
4. Martin, Robert C. (2000). "Design Principles and Design Patterns" (https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf) (PDF). *objectmentor.com*. Archived from the original on 2015-09-06.
5. Martin, Robert (2018). *Clean Architecture: A Craftsman's Guide to Software Structure and Design* (<https://books.google.com/books?id=uGE1DwAAQBAJ&q=2004+or+thereabouts+by+Michael+Feathers>). Pearson. p. 58. ISBN 978-0-13-449416-6.
6. "Single Responsibility Principle" (<https://web.archive.org/web/20150202200348/http://www.objectmentor.com/resources/articles/srp.pdf>) (PDF). *objectmentor.com*. Archived from the original on 2 February 2015.
7. Martin, Robert C. (2003). *Agile Software Development, Principles, Patterns, and Practices* (<https://books.google.com/books?id=0HYhAQAAIAAJ>). Prentice Hall. p. 95. ISBN 978-0135974445.
8. "Open/Closed Principle" (<https://web.archive.org/web/20150905081105/http://www.objectmentor.com/resources/articles/ocp.pdf>) (PDF). *objectmentor.com*. Archived from the original on 5 September 2015.
9. "Liskov Substitution Principle" (<https://web.archive.org/web/20150905081111/http://www.objectmentor.com/resources/articles/lsp.pdf>) (PDF). *objectmentor.com*. Archived from the original on 5 September 2015.
10. "Interface Segregation Principle" (<https://web.archive.org/web/20150905081110/http://www.objectmentor.com/resources/articles/isp.pdf>) (PDF). *objectmentor.com*. 1996. Archived from the original on 5 September 2015.
11. "Dependency Inversion Principle" (<https://web.archive.org/web/20150905081103/http://www.objectmentor.com/resources/articles/dip.pdf>) (PDF). *objectmentor.com*. Archived from the original on 5 September 2015.

Retrieved from "<https://en.wikipedia.org/w/index.php?title=SOLID&oldid=1317281842>"

Single-responsibility principle

The **single-responsibility principle** (**SRP**) is a computer programming principle that states that "A module should be responsible to one, and only one, actor."^[1] The term actor refers to a group (consisting of one or more stakeholders or users) that requires a change in the module.

Robert C. Martin, the originator of the term, expresses the principle as, "A class should have only one reason to change".^[2] Because of confusion around the word "reason", he later clarified his meaning in a blog post titled "The Single Responsibility Principle", in which he mentioned *Separation of Concerns* and stated that "Another wording for the Single Responsibility Principle is: Gather together the things that change for the same reasons. Separate those things that change for different reasons."^[3] In some of his talks, he also argues that the principle is, in particular, about roles or actors. For example, while they might be the same person, the role of an accountant is different from a database administrator. Hence, each module should be responsible for each role.^[4]

History

The term was introduced by Robert C. Martin in his article "The Principles of OOD" as part of his *Principles of Object Oriented Design*,^[5] made popular by his 2003 book *Agile Software Development, Principles, Patterns, and Practices*.^[6] Martin described it as being based on the principle of *cohesion*, as described by Tom DeMarco in his book *Structured Analysis and System Specification*,^[7] and Meilir Page-Jones in *The Practical Guide to Structured Systems Design*.^[8] In 2014 Martin published a blog post titled "The Single Responsibility Principle" with a goal to clarify what was meant by the phrase "reason for change."^[9]

Example

Martin defines a responsibility as a *reason to change*, and concludes that a class or module should have one, and only one, reason to be changed (e.g. rewritten).

As an example, consider a module that compiles and prints a report. Imagine such a module can be changed for two reasons. First, the content of the report could change. Second, the format of the report could change. These two things change for different causes. The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and should, therefore, be in separate classes or modules. It would be a bad design to couple two things that change for different reasons at different times.

The reason it is important to keep a class focused on a single concern is that it makes the class more robust. Continuing with the foregoing example, if there is a change to the report compilation process, there is a greater danger that the printing code will break if it is part of the same class.

See also

- Chain-of-responsibility pattern
- Coupling (computer programming)
- GRASP (object-oriented design)
- Information hiding
- SOLID—the "S" in "SOLID" represents the single responsibility principle
- Separation of concerns

References

- Martin, Robert C. (2018). *Clean architecture : a craftsman's guide to software structure and design*. Boston. ISBN 978-0-13-449432-6. OCLC 1003645626 (https://search.worldcat.org/oclc/1003645626).
- Martin, Robert C. (2003). *Agile Software Development, Principles, Patterns, and Practices* (https://books.google.com/books?id=0HYhAQAAIAAJ). Prentice Hall. p. 95. ISBN 978-0135974445.
- Martin, Robert C. (2014). "The Single Responsibility Principle" (https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleRepon sibilityPrinciple.html). *The Clean Code Blog*.
- Robert C. Martin (2018). *Clean Architecture: A Craftsman's Guide to Software Structure and Design* (https://books.google.com/book s?id=8ngAkAEACAAJ). Prentice Hall. ISBN 978-0-13-449416-6.
- Martin, Robert C. (2005). "The Principles of OOD" (http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod). *butunclebob.com*.

6. Martin 2003, pp. 95–98
7. DeMarco, Tom. (1979). *Structured Analysis and System Specification* (<https://archive.org/details/structuredanalys0000dema>). Prentice Hall. ISBN 0-13-854380-1.
8. Page-Jones, Meilir (1988). *The Practical Guide to Structured Systems Design*. Yourdon Press Computing Series. p. 82. ISBN 978-8120314825.
9. "Clean Coder Blog" (<https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html>). *blog.cleancoder.com*.

External links

- "The Principles of OOD (<http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>)" by Robert Martin
 - "The Single Responsibility Principle (2007) (https://drive.google.com/file/d/0ByOwmqah_nuGNHEtcU5OekdDMkk/view?resourcekey=0-AbuGpXQzwZcUGExkktKt0g)" by Robert Martin
 - "The Single Responsibility Principle (2014) (<https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html>)" by Robert Martin
-

Retrieved from "https://en.wikipedia.org/w/index.php?title=Single-responsibility_principle&oldid=1310638610"

Open–closed principle

In object-oriented programming, the **open–closed principle** (OCP) states "*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*";^[1] that is, such an entity can allow its behaviour to be extended without modifying its source code.

The name *open–closed principle* has been used in two ways. Both ways use generalizations (for instance, inheritance or delegate functions) to resolve the apparent dilemma, but the goals, techniques, and results are different.

The open–closed principle is one of the five SOLID principles of object-oriented design.

Meyer's open–closed principle

Bertrand Meyer is generally credited for having originated the term *open–closed principle*,^[2] which appeared in his 1988 book *Object-Oriented Software Construction*.^{[1]:23}

- A module will be said to be open if it is still available for extension. For example, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.
- A module will be said to be closed if [it] is available for use by other modules. This assumes that the module has been given a well-defined, stable description (the interface in the sense of information hiding).

At the time Meyer was writing, adding fields or functions to a library inevitably required changes to any programs depending on that library. Meyer's proposed solution to this problem relied on the notion of object-oriented inheritance (specifically implementation inheritance).^{[1]:229}

A class is closed, since it may be compiled, stored in a library, baselined, and used by client classes. But it is also open, since any new class may use it as parent, adding new features. When a descendant class is defined, there is no need to change the original or to disturb its clients.



The open–closed principle was introduced by Bertrand Meyer.

Polymorphic open–closed principle

During the 1990s, the open–closed principle became popularly redefined to refer to the use of abstracted interfaces, where the implementations can be changed and multiple implementations could be created and polymorphically substituted for each other.

In contrast to Meyer's usage, this definition advocates inheritance from abstract base classes. Interface specifications can be reused through inheritance but implementation need not be. The existing interface is closed to modifications and new implementations must, at a minimum, implement that interface.

Robert C. Martin's 1996 article "The Open-Closed Principle"^[2] was one of the seminal writings to take this approach. In 2001, Craig Larman related the open–closed principle to the pattern by Alistair Cockburn called *Protected Variations*, and to the David Parnas discussion of *information hiding*.^[3]

See also

- SOLID – the "O" in "SOLID" represents the open–closed principle
- Robustness principle

References

1. Meyer, Bertrand (1988). *Object-Oriented Software Construction*. Prentice Hall. ISBN 0-13-629049-3.
2. Robert C. Martin "The Open-Closed Principle", C++ Report, January 1996 (<https://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgN2M5MTkwM2EtNWfkZC00ZTI3LWFjZTUtNTFhZGZiYmUzODc1&hl=en>) Archived (<https://web.archive.org/web/20060822033314/http://www.objectmentor.com/resources/articles/ocp.pdf>) August 22, 2006, at the Wayback Machine

3. Larman, Craig (May–June 2001). "Protected Variation: The Importance of Being Closed" (<https://codecourse.sourceforge.net/materials/The-Importance-of-Being-Closed.pdf>) (PDF). *IEEE Software*. **18** (2). IEEE: 89–91. doi:10.1109/52.922731 (<https://doi.org/10.1109%2F52.922731>).

External links

- [The Principles of OOD \(http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod\)](http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod)
 - [The Open/Closed Principle: Concerns about Change in Software Design \(https://blog.symprise.net/articles/open-closed-principle-concerns-about-change-in-software-design\)](https://blog.symprise.net/articles/open-closed-principle-concerns-about-change-in-software-design)
 - [The Open-Closed Principle -- and What Hides Behind It \(https://medium.com/@wrong.about/the-open-closed-principle-c3dc45419784\)](https://medium.com/@wrong.about/the-open-closed-principle-c3dc45419784)
-

Retrieved from "https://en.wikipedia.org/w/index.php?title=Open-closed_principle&oldid=1305270760"

Liskov substitution principle

The **Liskov substitution principle** (**LSP**) is a particular definition of a subtyping relation, called strong behavioral subtyping, that was initially introduced by [Barbara Liskov](#) in a 1987 conference keynote address titled *Data abstraction and hierarchy*. It is based on the concept of "substitutability" – a principle in [object-oriented programming](#) stating that an object (such as a class) may be replaced by a sub-object (such as a class that extends the first class) without breaking the program. It is a semantic rather than merely syntactic relation, because it intends to guarantee semantic interoperability of types in a hierarchy, object types in particular. [Barbara Liskov](#) and [Jeannette Wing](#) described the principle succinctly in a 1994 paper as follows:^[1]

Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T . Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T .

Symbolically:

$$S \leq T \rightarrow (\forall x:T. \phi(x) \rightarrow \forall y:S. \phi(y))$$

That is, if S subtypes T , what holds for T -objects holds for S -objects. In the same paper, Liskov and Wing detailed their notion of behavioral subtyping in an extension of [Hoare logic](#), which bears a certain resemblance to [Bertrand Meyer's design by contract](#) in that it considers the interaction of subtyping with [preconditions](#), [postconditions](#) and [invariants](#).



Liskov substitution was introduced by [Barbara Liskov](#)

Principle

Liskov's notion of a behavioural subtype defines a notion of substitutability for objects; that is, if S is a subtype of T , then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program (e.g. [correctness](#)).

Behavioural subtyping is a stronger notion than typical subtyping of functions defined in type theory, which relies only on the [contravariance](#) of parameter types and [covariance](#) of the return type. Behavioural subtyping is undecidable in general: if q is the property "method for x always terminates", then it is impossible for a program (e.g. a compiler) to verify that it holds true for some subtype S of T , even if q does hold for T . Nonetheless, the principle is useful in reasoning about the design of class hierarchies.

Liskov substitution principle imposes some standard requirements on signatures that have been adopted in newer object-oriented programming languages (usually at the level of classes rather than types; see [nominal vs. structural subtyping](#) for the distinction):

- [Contravariance](#) of method parameter types in the subtype.
- [Covariance](#) of method return types in the subtype.
- New exceptions cannot be thrown by the methods in the subtype, except if they are subtypes of exceptions thrown by the methods of the supertype.

In addition to the signature requirements, the subtype must meet a number of behavioural conditions. These are detailed in a terminology resembling that of [design by contract](#) methodology, leading to some restrictions on how contracts can interact with [inheritance](#):

- [Preconditions](#) cannot be strengthened in the subtype.
- [Postconditions](#) cannot be weakened in the subtype.
- [Invariants](#) cannot be weakened in the subtype.
- History constraint (the "history rule"). Objects are regarded as being modifiable only through their methods ([encapsulation](#)). Because subtypes may introduce methods that are not present in the supertype, the introduction of these methods may allow state changes in the subtype that are not permissible in the supertype. The history constraint prohibits this. It was the novel element introduced by Liskov and Wing. A violation of this constraint is, for example, defining a *mutable point* as a subtype of an *immutable point*.^[2] This is a violation of the history constraint, because in the history of the *immutable point*, the state is always the same after creation, so it cannot include the history of a *mutable point* in general. Fields added to the subtype may, however, be safely modified because they are not observable through the supertype methods. Thus, one can define a *circle with immutable center and mutable radius* as a subtype of an *immutable point* without violating the history constraint.

Origins

The rules on pre- and postconditions are identical to those introduced by Bertrand Meyer in his 1988 book *Object-Oriented Software Construction*. Both Meyer, and later Pierre America, who was the first to use the term *behavioral subtyping*, gave proof-theoretic definitions of some behavioral subtyping notions, but their definitions did not take into account aliasing that may occur in programming languages that support references or pointers. Taking aliasing into account was the major improvement made by Liskov and Wing (1994), and a key ingredient is the history constraint. Under the definitions of Meyer and America, a mutable point would be a behavioral subtype of an immutable point, whereas Liskov substitution principle forbids this.

Violation

Liskov substitution principle explains a property, "*If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T .*"^[3]

Here is perhaps an example of violation of LSP:

```
class Rectangle {
private:
    double width;
    double height;
public:
    Rectangle(double width, double height):
        width{width}, height{height} {}

    // declared virtual for subclass Square
    virtual void setWidth(double width) noexcept {
        this->width = width;
    }

    // declared virtual for subclass Square
    virtual void setHeight(double height) noexcept {
        this->height = height;
    }

    [[nodiscard]]
    double getWidth() const noexcept {
        return width;
    }

    [[nodiscard]]
    double getHeight() const noexcept {
        return height;
    }

    [[nodiscard]]
    double getArea() const noexcept {
        return width * height;
    }
};
```

From a programming point of view, the Square class may be defined as extending the Rectangle class.

```
class Square : public Rectangle {
public:
    void setWidth(double width) noexcept override {
        Rectangle::setWidth(width);
        Rectangle::setHeight(width);
    }

    void setHeight(double height) noexcept override {
        Rectangle::setHeight(height);
        Rectangle::setWidth(height);
    }
};
```

However, this violates LSP even though the is-a relationship holds between Rectangle and Square.

Consider the following example, where function g does not work if a Square is passed in, and so the open-closed principle might be considered to have been violated.

```
void g(Rectangle& r) {
    r.setWidth(5).setHeight(4);
    assert(r.getArea() == 20); // assertion will fail
}
```

Conversely, if one considers that the type of a shape should only be a constraint on the relationship of its dimensions, then the assumption in g() that setHeight() will change height, and area, but not width is invalid. This assumption is invalid not only for squares, but even potentially for other rectangles that might be coded to preserve area or aspect ratio when height changes.^[4]

See also

- [Circle-ellipse problem](#)
- [Composition over inheritance](#)
- [Program refinement](#)
- [Referential transparency](#)
- [Type signature](#)
- [SOLID](#) – the "L" in "SOLID" stands for Liskov substitution principle

References

1. Liskov, Barbara; Wing, Jeannette (1994-11-01). "A behavioral notion of subtyping" (<https://doi.org/10.1145%2F197320.197383>). *ACM Transactions on Programming Languages and Systems*. **16** (6): 1811–41. doi:10.1145/197320.197383 (<https://doi.org/10.1145%2F197320.197383>). S2CID 999172 (<https://api.semanticscholar.org/CorpusID:999172>).
2. Kinsbruner, Elad; Itzhaky, Shachar; Peleg, Hila (2024). "Constrictor: Immutability as a Design Concept" (<https://doi.org/10.4230%2FLIPIcs.ECOOP.2024.22>). *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. Leibniz International Proceedings in Informatics (LIPIcs). **313**. Schloss Dagstuhl – Leibniz-Zentrum für Informatik: 22:1–22:29. doi:10.4230/LIPIcs.ECOOP.2024.22 (<https://doi.org/10.4230%2FLIPIcs.ECOOP.2024.22>). ISBN 978-3-95977-341-6.
3. Liskov, Barbara (May 1988). *Data Abstraction and Hierarchy* (<https://web.archive.org/web/20200621040810/https://klevas.mif.vu.lt/~plukas/resources/ODPrinciples/Liskov1987.pdf>) (PDF). SIGPLAN Notices. Archived from the original on Jun 21, 2020.
4. "The Liskov Substitution Principle" (<https://web.archive.org/web/20150905081111/http://www.objectmentor.com/resources/articles/lsp.pdf>) (PDF). Robert C. Martin, 1996. Archived from the original (<http://www.objectmentor.com/resources/articles/lsp.pdf>) (PDF) on 5 September 2015. Retrieved 2 October 2012.

Bibliography

Specific references

- [Liskov, B. \(1987\). "Keynote address - data abstraction and hierarchy". *Addendum to the proceedings on Object-oriented programming systems, languages and applications \(Addendum\) - OOPSLA '87*. pp. 17–34. doi:10.1145/62138.62141 \(<https://doi.org/10.1145%2F62138.62141>\). ISBN 0897912667](#). A keynote address in which Liskov first formulated the principle.
- [Meyer, B. \(1988\). *Object-oriented Software Construction*. Prentice Hall. ISBN 0-13-629031-0](#).

General reference

- [Leavens, Gary T.; Dhara, Krishna K. \(2000\). "Concepts of Behavioral Subtyping and a Sketch of Their Extension to Component-Bases Systems". In Leavens, Gary T.; Sitaraman, Murali \(eds.\). *Foundations of component-based systems*. Cambridge University Press. ISBN 0-521-77164-1](#). This paper surveys various notions of behavioral subtyping, including Liskov and Wing's.
- [Liskov, B. H.; Wing, J. M. \(November 1994\). "A behavioral notion of subtyping" \(<https://doi.org/10.1145%2F197320.197383>\). *ACM Trans. Program. Lang. Syst.* **16** \(6\): 1811–41. doi:10.1145/197320.197383 \(<https://doi.org/10.1145%2F197320.197383>\). S2CID 999172 \(<https://api.semanticscholar.org/CorpusID:999172>\)](#). An updated version appeared: [Liskov, Barbara; Wing, Jeannette \(July 1999\). *Behavioral Subtyping Using Invariants and Constraints* \(<http://reports-archive.adm.cs.cmu.edu/anon/1999/CMU-CS-99-156.ps>\) \(Technical report\). Carnegie Mellon University. CMU-CS-99-156](#). The formalization of the principle by its authors.
- [Plösch, Reinhold \(2004\). *Contracts, scenarios and prototypes: an integrated approach to high quality software*. Springer. ISBN 3-540-43486-0](#). Contains a gentler introduction to behavioral subtyping in its various forms in chapter 2.
- [Martin, Robert C. \(March 1996\). "The Liskov Substitution Principle" \(<https://web.archive.org/web/20151128004108/http://www.objectmentor.com/resources/articles/lsp.pdf>\) \(PDF\). *C++ Report*. Archived from the original \(<http://www.objectmentor.com/resources/articles/lsp.pdf>\) \(PDF\) on 2015-11-28](#). An article popular in the object-oriented programming community that gives several examples of LSP violations.
- [Majorinc, Kazimir. "Ellipse-Circle Dilemma and Inverse Inheritance" \(<https://www.researchgate.net/publication/323457799>\). *ITI 98, Proceedings of the 20th International Conference of Information Technology Interfaces, Pula, 1998*. Information Technology Interfaces, 2009. Iti '09. Proceedings of the Iti 2009 31st International Conference on. pp. 627–632. ISSN 1330-1012 \(<https://search.worldcat.org/issn/1330-1012>\). OCLC 894960131 \(<https://search.worldcat.org/oclc/894960131>\)](#). This paper discusses LSP in the mentioned context.

External links

- [Norvell, T.S. "The Liskov Substitution Principle" \(<https://www.engr.mun.ca/~theo/Courses/sd/5895-downloads/sd-principles-3.p>](#)

pt.pdf) (PDF). *Engineering Memorial University*.

- Samokhin, Vadim (2018-06-06). "Liskov Substitution Principle" (<https://medium.com/@wrong.about/liskov-substitution-principle-a982551d584a>). *Medium*.
 - "SOLID Class Design: The Liskov Substitution Principle" (<https://www.tomdalling.com/blog/software-design/solid-class-design-the-liskov-substitution-principle/>). *Tom Dalling*. 21 Nov 2009.
 - Jobaer, Abu (2023-05-31). "LSP: Liskov Substitution Principle" (<https://medium.com/swlh/lsp-liskov-substitution-principle-24311d1f854>). *The Startup*. Medium.
-

Retrieved from "https://en.wikipedia.org/w/index.php?title=Liskov_substitution_principle&oldid=1317976715"

Interface segregation principle

In the field of software engineering, the **interface segregation principle (ISP)** states that no code should be forced to depend on methods it does not use.^[1] ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them. Such shrunken interfaces are also called *role interfaces*.^[2] ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy. ISP is one of the five SOLID principles of object-oriented design, similar to the High Cohesion Principle of GRASP.^[3] Beyond object-oriented design, ISP is also a key principle in the design of distributed systems in general and one of the six IDEALS principles for microservice design.^[4]

Importance in object-oriented design

Within object-oriented design, interfaces provide layers of abstraction that simplify code and create a barrier preventing coupling to dependencies. A system may become so coupled at multiple levels that it is no longer possible to make a change in one place without necessitating many additional changes.^[1] Using an interface or an abstract class can prevent this side effect.

Origin

The ISP was first used and formulated by Robert C. Martin^[5] while consulting for Xerox. Xerox had created a new printer system that could perform a variety of tasks such as stapling and faxing. The software for this system was created from the ground up. As the software grew, making modifications became more and more difficult so that even the smallest change would take a redeployment cycle of an hour, which made development nearly impossible.

The design problem was that a single Job class was used by almost all of the tasks. Whenever a print job or a stapling job needed to be performed, a call was made to the Job class. This resulted in a 'fat' class with multitudes of methods specific to a variety of different clients. Because of this design, a staple job would know about all the methods of the print job, even though there was no use for them.

The solution suggested by Martin utilized what is today called the Interface Segregation Principle. Applied to the Xerox software, an interface layer between the Job class and its clients was added using the Dependency Inversion Principle. Instead of having one large Job class, a Staple Job interface or a Print Job interface was created that would be used by the Staple or Print classes, respectively, calling methods of the Job class. Therefore, one interface was created for each job type, which was all implemented by the Job class.

Typical violation

A typical violation of the Interface Segregation Principle is given in Agile Software Development: Principles, Patterns, and Practices^[1] in 'ATM Transaction example' and in an article also written by Robert C. Martin specifically about the ISP.^[6] This example discusses the User Interface for an ATM, which handles all requests such as a deposit request, or a withdrawal request, and how this interface needs to be segregated into individual and more specific interfaces.

See also

- SOLID – the "I" in SOLID stands for Interface segregation principle

References

- Martin, Robert (2002). *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education.
- Role Interface (<http://martinfowler.com/bliki/RoleInterface.html>)
- "David Hayden, *Interface-Segregation Principle (ISP) - Principles of Object-Oriented Class Design*" (<https://web.archive.org/web/20100820124217/http://codebetter.com/blogs/david.hayden/archive/2005/06/15/64635.aspx>). Archived from the original (<http://codebetter.com/blogs/david.hayden/archive/2005/06/15/64635.aspx>) on 2010-08-20. Retrieved 2009-11-07.
- Paulo Merson,*Principles for Microservice Design: Think IDEALS, Rather than SOLID*, The InfoQ eMag Issue #91, February 2021 (<http://www.infoq.com/minibooks/reexamining-microservices/>)
- "This principle was first defined by Robert C. Martin" (<https://www.baeldung.com/java-interface-segregation#:~:text=This%20principle%20was%20first%20defined%20by%20Robert%20C.%20Martin>).
- Robert C. Martin,*The Interface Segregation Principle*, C++ Report, June 1996 (<https://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgOTViYjJhYzMtMzYxMC00MzFjLWJjMzYtOGJiMDc5N2JkYmJi&hl=en>)

External links

- *Principles Of OOD* (<http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>) – Description and links to detailed articles on SOLID.
-

Retrieved from "https://en.wikipedia.org/w/index.php?title=Interface_segregation_principle&oldid=1280738205"

Dependency inversion principle

In object-oriented design, the **dependency inversion principle** is a specific methodology for loosely coupled software modules. When following this principle, the conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are reversed, thus rendering high-level modules independent of the low-level module implementation details. The principle states:^[1]

- A. High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces).
- B. Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

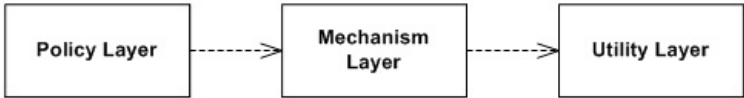
By dictating that *both* high-level and low-level objects must depend on the same abstraction, this design principle *inverts* the way some people may think about object-oriented programming.^[2]

The idea behind points A and B of this principle is that when designing the interaction between a high-level module and a low-level one, the interaction should be thought of as an abstract interaction between them. This has implications for the design of both the high-level and the low-level modules: the low-level one should be designed with the interaction in mind and it may be necessary to change its usage interface.

In many cases, thinking about the interaction itself as an abstract concept allows for reduction of the coupling between the components without introducing additional coding patterns and results in a lighter and less implementation-dependent interaction schema. When this abstract interaction schema is generic and clear, this design principle leads to the dependency inversion pattern described below.

Traditional layers pattern

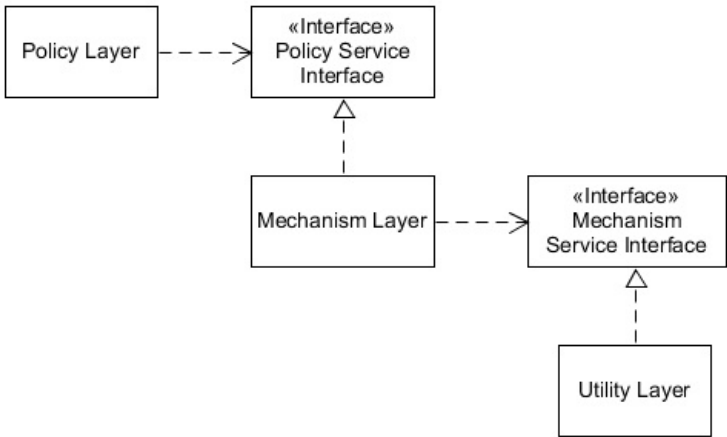
In conventional application architecture, lower-level components (e.g., Utility Layer) are designed to be consumed by higher-level components (e.g., Policy Layer) which enable highly composite systems to be built. Here higher-level components depend directly upon lower-level components to achieve some task. This dependency upon lower-level components limits the reuse opportunities of the higher-level components.^[1]



The goal of the dependency inversion pattern is to avoid this highly coupled distribution with the mediation of an abstract layer, and to increase the reusability of higher and policy layers.

Dependency inversion pattern

With the introduction of an abstract layer, both high- and lower-level layers reduce the traditional dependencies from top to bottom. Nevertheless, the "inversion" concept does not mean that lower-level layers *directly* depend on higher-level layers. Rather, *both* layers should depend on abstractions (interfaces) that expose the behavior needed by higher-level layers.



In a direct application of dependency inversion, the abstracts are owned by the higher/policy layers. This architecture groups the higher/policy components and the abstractions that define lower services together in the same package. The lower-level layers are created by inheritance/implementation of these abstract classes or interfaces.^[1]

The inversion of the dependencies and ownership encourages reuse of the higher and policy layers. Upper layers could use other implementations of the lower services. When the lower-level layer components are closed or when the application requires the reuse of existing services, it is common that an Adapter mediates between the services and the abstractions.

Dependency inversion pattern generalization

In many projects the dependency inversion principle and pattern are considered as a single concept that should be generalized, i.e., applied to all interfaces between software modules. There are at least two reasons for that:

1. It is simpler to see a good thinking principle as a coding pattern. Once an abstract class or an interface has been coded, the programmer may say: "I have done the job of abstraction".
2. Because many unit testing tools rely on inheritance to accomplish mocking, the usage of generic interfaces between classes (not only between modules when it makes sense to use generality) became the rule.

If the mocking tool used relies only on inheritance, it may become necessary to widely apply the dependency inversion pattern. This has major drawbacks:

1. Merely implementing an interface over a class isn't sufficient to reduce coupling; only thinking about the potential abstraction of interactions can lead to a less coupled design.
2. Implementing generic interfaces everywhere in a project makes it harder to understand and maintain. At each step the reader will ask themselves what are the other implementations of this interface and the response is generally: only mocks.
3. The interface generalization requires more plumbing code, in particular factories that generally rely on a dependency-injection framework.
4. Interface generalization also restricts the usage of the programming language.

Generalization restrictions

The presence of interfaces to accomplish the Dependency Inversion Pattern (DIP) has other design implications in an object-oriented program:

- All member variables in a class must be in interfaces or abstracts.
- All concrete class packages must connect only through interface or abstract class packages.
- No class should derive from a concrete class.
- No method should override an implemented method.^[1]
- All variable instantiation requires the implementation of a creational pattern such as the factory method or the factory pattern, or the use of a dependency-injection framework.

Interface mocking restrictions

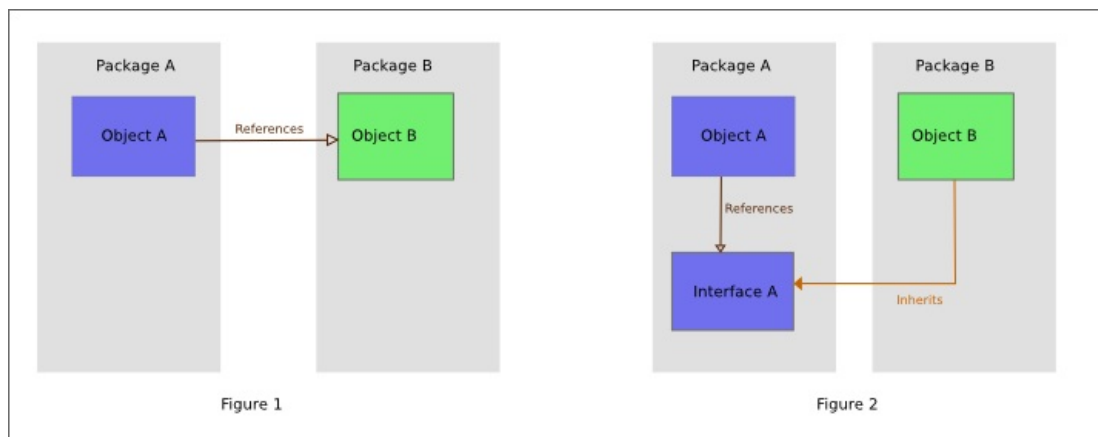
Using inheritance-based mocking tools also introduces restrictions:

- Static externally visible members should systematically rely on dependency injection making them far harder to implement.
- All testable methods should become an interface implementation or an override of an abstract definition.

Implementations

Two common implementations of DIP use similar logical architecture but with different implications.

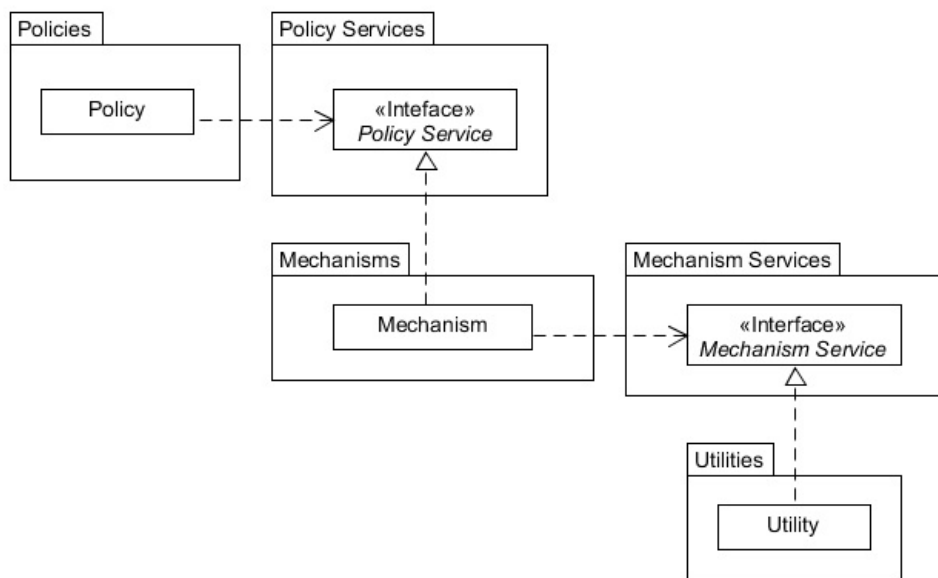
A direct implementation packages the policy classes with service abstracts classes in one library. In this implementation high-level components and low-level components are distributed into separate packages/libraries, where the interfaces defining the behavior/services required by the high-level component are owned by, and exist within the high-level component's library. The implementation of the high-level component's interface by the low-level component requires that the low-level component package depend upon the high-level component for compilation, thus inverting the conventional dependency relationship.



Figures 1 and 2 illustrate code with the same functionality, however in Figure 2, an interface has been used to invert the dependency. The direction of dependency can be chosen to maximize policy code reuse, and eliminate cyclic dependencies.

In this version of DIP, the lower layer component's dependency on the interfaces/abstracts in the higher-level layers makes reuse of the lower layer components difficult. This implementation instead "inverts" the traditional dependency from top-to-bottom to the opposite, bottom-to-top.

A more flexible solution extracts the abstract components into an independent set of packages/libraries:



The separation of each layer into its own package encourages reuse of any layer, providing robustness and mobility.^[1]

Examples

Genealogical module

A genealogical system may represent relationships between people as a graph of direct relationships between them (father-son, father-daughter, mother-son, mother-daughter, husband-wife, wife-husband, etc.). This is very efficient and extensible, as it is easy to add an ex-husband or a legal guardian.

But some higher-level modules may require a simpler way to browse the system: any person may have children, parents, siblings (including half-brothers and -sisters or not), grandparents, cousins, and so on.

Depending on the usage of the genealogical module, presenting common relationships as distinct direct properties (hiding the graph) will make the coupling between a higher-level module and the genealogical module much lighter and allow changing the internal representation of the direct relationships completely without any effect on the modules using them. It also permits embedding exact definitions of siblings or uncles in the genealogical module, thus enforcing the single responsibility principle.

Finally, if the first extensible generalized graph approach seems the most extensible, the usage of the genealogical module may show that a more specialized and simpler relationship implementation is sufficient for the application(s) and helps create a more efficient system.

In this example, abstracting the interaction between the modules leads to a simplified interface of the lower-level module and may lead to a simpler implementation of it.

Remote file server client

A remote file server (FTP, cloud storage ...) client can be modeled as a set of abstract interfaces:

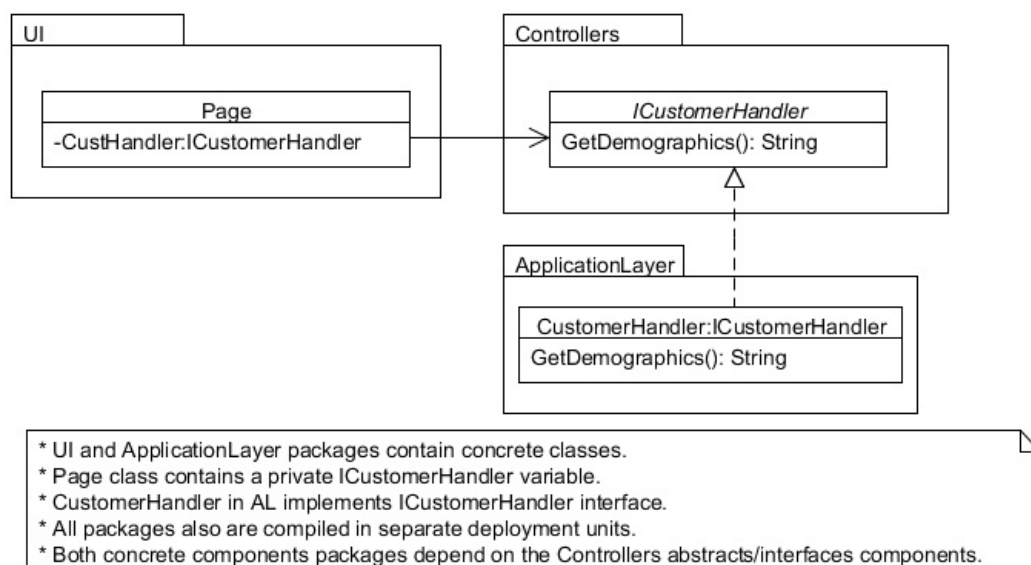
1. Connection/Disconnection (a connection persistence layer may be needed)
2. Folder/tags creation/rename/delete/list interface
3. File creation/replacement/rename/delete/read interface
4. File searching
5. Concurrent replacement or delete resolution
6. File history management ...

If local and remote files offers the same abstract interfaces, high-level modules that implement the dependency inversion pattern can use them indiscriminately. The application will be able to save its documents locally or remotely transparently.

The level of service required by high level modules should be considered.

Designing a module as a set of abstract interfaces, and adapting other modules to it, can provide a common interface for many systems.

Model-view-controller



UI and ApplicationLayer packages contain mainly concrete classes. Controllers contains abstracts/interface types. UI has an instance of ICustomerHandler. All packages are physically separated. In the ApplicationLayer there is a concrete implementation of CustomerHandler that Page class will use. Instances of the ICustomerHandler interface are created dynamically by a Factory (possibly in the same Controllers package). Concrete types Page and CustomerHandler depend on ICustomerHandler, not on each other.

Since the UI doesn't reference the ApplicationLayer or any other concrete package implementing ICustomerHandler, the concrete implementation of CustomerHandler can be replaced without changing the UI class. Also, the Page class implements interface IPageViewer which could be passed as an argument to ICustomerHandler methods, allowing the concrete implementation of CustomerHandler to communicate with UI without a concrete dependency. Again, both are linked by interfaces.

Related patterns

Applying the dependency inversion principle can also be seen as an example of the adapter pattern. That is, the high-level class defines its own adapter interface which is the abstraction on which the other high-level classes depend. The adapted implementation also depends necessarily on the same adapter interface abstraction, while it can be implemented by using code from within its own low-level module. The high-level module does not depend on the low-level module, since it only uses the low-level functionality indirectly through the adapter interface by invoking polymorphic methods to the interface which are implemented by the adapted implementation and its low-level module.

Various patterns such as Plugin, Service Locator,^[3] or Dependency injection^{[4][5]} are employed to facilitate the run-time provisioning of the chosen low-level component implementation to the high-level component.

History

The dependency inversion principle was postulated by Robert C. Martin and described in several publications including the paper *Object Oriented Design Quality Metrics: an analysis of dependencies*,^[6] an article appearing in the *C++ Report* in June 1996 entitled *The Dependency Inversion Principle*,^[7] and the books *Agile Software Development, Principles, Patterns, and Practices*,^[1] and *Agile Principles, Patterns, and Practices in C#*.

See also

- [Adapter pattern](#)
- [Dependency injection](#)
- [Design by contract](#)
- [Interface](#)
- [Inventor's Paradox](#)
- [Inversion of control](#)
- [Plug-in \(computing\)](#)
- [Service locator pattern](#)
- [SOLID](#) – the "D" in "SOLID" stands for the dependency inversion principle

References

1. Martin, Robert C. (2003). *Agile Software Development, Principles, Patterns, and Practices* (<https://books.google.com/books?id=0HYhAQAAIAAJ>). Prentice Hall. pp. 127–131. ISBN 978-0135974445.
2. Freeman, Eric; Freeman, Elisabeth; Kathy, Sierra; Bert, Bates (2004). Hendrickson, Mike; Loukides, Mike (eds.). *Head First Design Patterns* (<http://shop.oreilly.com/product/9780596007126.do>) (paperback). Vol. 1. O'REILLY. ISBN 978-0-596-00712-6. Retrieved 2012-06-21.
3. Aung, Nay Lin (2018-12-01). "Service Locator vs Dependency Injection" (<https://medium.com/@naylinaung/service-locator-vs-dependency-injection-5e7e6427111d>). *Medium*. Retrieved 2022-12-06.
4. Mathews, Sasha (2021-03-25). "You are Simply Injecting a Dependency, Thinking that You are Following the Dependency Inversion..." (<https://levelup.gitconnected.com/you-are-simply-injecting-a-dependency-thinking-that-you-are-following-the-dependency-inversion-32632954c208>). *Medium*. Retrieved 2022-12-06.
5. Erez, Guy (2022-03-09). "Dependency Inversion vs. Dependency Injection" (<https://betterprogramming.pub/straightforward-simple-dependency-inversion-vs-dependency-injection-7d8c0d0ed28e>). *Medium*. Retrieved 2022-12-06.
6. Martin, Robert C. (October 1994). "Object Oriented Design Quality Metrics: An analysis of dependencies" (<https://linux.ime.usp.br/~joaomm/mac499/arquivos/referencias/oodmetrics.pdf>) (PDF). Retrieved 2016-10-15.
7. Martin, Robert C. (June 1996). "The Dependency Inversion Principle" (<https://web.archive.org/web/20110714224327/http://www.objectmentor.com/resources/articles/dip.pdf>) (PDF). *C++ Report*. Vol. 8, no. 6. pp. 61–66. ISSN 1040-6042 (<https://search.worldcat.org/issn/1040-6042>). Archived from the original (<http://www.objectmentor.com/resources/articles/dip.pdf>) (PDF) on 2011-07-14.

External links

- [Object Oriented Design Quality Metrics: an analysis of dependencies Robert C. Martin, C++ Report, Sept/Oct 1995](https://linux.ime.usp.br/~joaomm/mac499/arquivos/referencias/oodmetrics.pdf) (<https://linux.ime.usp.br/~joaomm/mac499/arquivos/referencias/oodmetrics.pdf>)
- [The Dependency Inversion Principle, Robert C. Martin, C++ Report, May 1996](https://web.archive.org/web/20110714224327/http://www.objectmentor.com/resources/articles/dip.pdf) (<https://web.archive.org/web/20110714224327/http://www.objectmentor.com/resources/articles/dip.pdf>)
- [Examining the Dependency Inversion Principle, Derek Greer](https://web.archive.org/web/20110904131814/http://www.aspirinngcraftsman.com/2008/12/28/examining-dependency-inversion/) (<https://web.archive.org/web/20110904131814/http://www.aspirinngcraftsman.com/2008/12/28/examining-dependency-inversion/>)
- [DIP: The Dependency-Inversion Principle](https://medium.com/@unclexo/dip-dependency-inversion-principle-77ef85e6f656) (<https://medium.com/@unclexo/dip-dependency-inversion-principle-77ef85e6f656>)
- [DIP in the Wild, Brett L. Schuchert, May 2013](http://martinfowler.com/articles/dipInTheWild.html) (<http://martinfowler.com/articles/dipInTheWild.html>)
- [Dependency Inversion Principle](https://www.sebaslab.com/tag/dependency-injection/) (<https://www.sebaslab.com/tag/dependency-injection/>)
- [A Little Architecture, Robert C. Martin \(Uncle Bob\), Jan 2016 - a blog on significance of Dependency Inversion Principle in software architecture](https://blog.cleancoder.com/uncle-bob/2016/01/04/ALittleArchitecture.html) (<https://blog.cleancoder.com/uncle-bob/2016/01/04/ALittleArchitecture.html>)