WIKIPEDIA
The Free Encyclopedia

# Software design pattern

(Redirected from Design pattern (computer science))

In software engineering, a **software design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an 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 (c.f. "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.[1][2] In the following years, Beck, Cunningham and others followed up on this work.

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

- Gamma, Erich; 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.[3]

## Practice

Design patterns can speed up the development process by providing tested, proven development paradigms.[4] Effective software design requires considering issues that may not become visible 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 helps to prevent such subtle issues,[5] and it also improves code readability for coders and architects who are familiar with the patterns.

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

By definition, a pattern must be programmed anew into each application that uses it. Since some authors see this as a step backward from software reuse as provided by components, researchers have worked to turn patterns into components. Meyer and Arnout were able to provide full or partial componentization of two-thirds of the patterns they attempted.[6]

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

## Structure

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

### Domain-specific patterns

Efforts have also been made to codify design patterns in particular domains, including use of existing design patterns as well as domain-specific design patterns. Examples include user interface design patterns,[7] information visualization,[8] secure design,[9] "secure usability",[10] Web design [11] and business model design.[12]

The annual Pattern Languages of Programming Conference proceedings [13] include many examples of domain-specific patterns.

## Classification and list

Design patterns had originally been categorized into 3 sub-classifications based on what kind of problem they solve. Creational patterns provide the capability to create objects based on a required criterion and in a controlled way. Structural patterns are about organizing different classes and objects to form larger structures and provide new functionality. Finally, behavioral patterns are about identifying common communication patterns between objects and realizing these patterns.

### Creational patterns

| Name | Description | In *Design Patterns* | In *Code Complete*[14] | Other |
|---|---|---|---|---|
| Abstract factory | Provide an interface for creating *families* of related or dependent objects without specifying their concrete classes. | Yes | Yes | — |
| Builder | Separate the construction of a complex object from its representation, allowing the same construction process to create various representations. | Yes | No | — |
| Dependency Injection | A class accepts the objects it requires from an injector instead of creating the objects directly. | No | No | — |
| Factory method | Define an interface for creating a *single* object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. | Yes | Yes | — |
| Lazy initialization | Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed. This pattern appears in the GoF catalog as "virtual proxy", an implementation strategy for the Proxy pattern. | No | No | PoEAA[15] |
| Multiton | Ensure a class has only named instances, and provide a global point of access to them. | No | No | — |
| Object pool | Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalisation of connection pool and thread pool patterns. | No | No | — |
| Prototype | 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 | No | — |
| Resource acquisition is initialization (RAII) | Ensure that resources are properly released by tying them to the lifespan of suitable objects. | No | No | — |
| Singleton | Ensure a class has only one instance, and provide a global point of access to it. | Yes | Yes | — |

### Structural patterns

| Name | Description | In *Design Patterns* | In *Code Complete*[14] | Other |
|---|---|---|---|---|
| Adapter, Wrapper, or Translator | 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 | — |
| Bridge | Decouple an abstraction from its implementation allowing the two to vary independently. | Yes | Yes | — |
| Composite | Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. | Yes | Yes | — |
| Decorator | Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality. | Yes | Yes | — |
| Delegation | Extend a class by composition instead of subclassing. The object handles a request by delegating to a second object (the delegate) | — | — | — |
| Extension object | Adding functionality to a hierarchy without changing the hierarchy. | No | No | Agile Software Development, Principles, Patterns, and Practices[16] |
| Facade | Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. | Yes | Yes | — |
| Flyweight | Use sharing to support large numbers of similar objects efficiently. | Yes | No | — |
| Front controller | The pattern relates to the design of Web applications. It provides a centralized entry point for handling requests. | No | No | J2EE Patterns[17] PoEAA[18] |
| Marker | Empty interface to associate metadata with a class. | No | No | Effective Java[19] |
| Module | Group several related elements, such as classes, singletons, methods, globally used, into a single conceptual entity. | No | No | — |
| Proxy | Provide a surrogate or placeholder for another object to control access to it. | Yes | No | — |
| Twin[20] | Twin allows modeling of multiple inheritance in programming languages that do not support this feature. | No | No | — |

## Behavioral patterns

| Name | Description | In *Design Patterns* | In *Code Complete*[14] | Other |
|---|---|---|---|---|
| Blackboard | Artificial intelligence pattern for combining disparate sources of data (see blackboard system) | No | No | — |
| Chain of responsibility | Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it. | Yes | No | — |
| Command | Encapsulate a request as an object, thereby 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 | No | — |
| Fluent interface | 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. | No | No | — |
| Interpreter | Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language. | Yes | No | — |
| Iterator | Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. | Yes | Yes | — |
| Mediator | Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it allows their interaction to vary independently. | Yes | No | — |
| Memento | Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later. | Yes | No | — |
| Null object | Avoid null references by providing a default object. | No | No | — |
| Observer or Publish/subscribe | Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically. | Yes | Yes | — |
| Servant | 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. | No | No | — |
| Specification | Recombinable business logic in a Boolean fashion. | No | No | — |
| State | Allow an object to alter its behavior when its internal state changes. The object will appear to change its class. | Yes | No | — |
| Strategy | Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. | Yes | Yes | — |
| Template method | Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. | Yes | Yes | — |

| | | | | |
|---|---|---|---|---|
| Visitor | Represent an operation to be performed on 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 | No | — |

## Concurrency patterns

| Name | Description | In *POSA2*[21] | Other |
|---|---|---|---|
| Active Object | Decouples method execution from method invocation that reside in their own thread of control. The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests. | Yes | — |
| Balking | Only execute an action on an object when the object is in a particular state. | No | — |
| Binding properties | Combining multiple observers to force properties in different objects to be synchronized or coordinated in some way.[22] | No | — |
| Compute kernel | The same calculation many times in parallel, differing by integer parameters used with non-branching pointer math into shared arrays, such as GPU-optimized Matrix multiplication or Convolutional neural network. | No | — |
| Double-checked locking | Reduce the overhead of acquiring a lock by first testing the locking criterion (the 'lock hint') in an unsafe manner; only if that succeeds does the actual locking logic proceed.<br><br>Can be unsafe when implemented in some language/hardware combinations. It can therefore sometimes be considered an anti-pattern. | Yes | — |
| Event-based asynchronous | Addresses problems with the asynchronous pattern that occur in multithreaded programs.[23] | No | — |
| Guarded suspension | Manages operations that require both a lock to be acquired and a precondition to be satisfied before the operation can be executed. | No | — |
| Join | 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 | — |
| Lock | One thread puts a "lock" on a resource, preventing other threads from accessing or modifying it.[24] | No | PoEAA[15] |
| Messaging design pattern (MDP) | Allows the interchange of information (i.e. messages) between components and applications. | No | — |
| Monitor object | An object whose methods are subject to mutual exclusion, thus preventing multiple objects from erroneously trying to use it at the same time. | Yes | — |
| Reactor | A reactor object provides an asynchronous interface to resources that must be handled synchronously. | Yes | — |
| Read-write lock | Allows concurrent read access to an object, but requires exclusive access for write operations. An underlying semaphore might be used for writing, and a Copy-on-write mechanism may or may not be used. | No | — |
| Scheduler | Explicitly control when threads may execute single-threaded code. | No | — |
| Thread pool | A number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many | No | — |

| | | | |
|---|---|---|---|
| | more tasks than threads. Can be considered a special case of the object pool pattern. | | |
| Thread-specific storage | Static or "global" memory local to a thread. | Yes | — |
| Safe Concurrency with Exclusive Ownership | 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 | — |
| CPU atomic operation | 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.[25] 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.[26] 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

It has been observed that design patterns may just be a sign that some 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.[27] 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.[28] See also Paul Graham's essay "Revenge of the Nerds".[29]

Inappropriate use of patterns may unnecessarily increase complexity.[30]

## See also

- Abstraction principle
- Algorithmic skeleton
- Anti-pattern
- Architectural pattern
- Canonical protocol pattern
- Debugging patterns
- Design pattern
- Distributed design patterns
- Double-chance function
- Enterprise Architecture framework
- GRASP (object-oriented design)
- Helper class
- Idiom in programming
- Interaction design pattern
- 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. Smith, Reid (October 1987). *Panel on design methodology*. OOPSLA '87 Addendum to the Proceedings. doi:10.1145/62138.62151 (https://doi.org/10.1145%2F62138.62151). "Ward cautioned against requiring too much programming at, what he termed, 'the high level of wizards.' He pointed out that a written 'pattern language' can significantly improve the selection and application of abstractions. He proposed a 'radical shift in the burden of design and implementation' basing the new methodology on an adaptation of Christopher Alexander's

work in pattern languages and that programming-oriented pattern languages developed at Tektronix has significantly aided their software development efforts."

2. Beck, Kent; 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.
3. Baroni, Aline Lúcia; Guéhéneuc, Yann-Gaël; Albin-Amiot, Hervé (June 2003). "Design Patterns Formalization". 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).
4. 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."
5. Tiako, Pierre F. (31 March 2009). "Formal Modeling and Specification of Design Patterns Using RTPA" (https://books.google.com/books?id=_SklFgSidxQC&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%2F978-1-60566-060-8). ISBN 9781605660615.
6. 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).
7. 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.
8. 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%2FTVCG.2006.178). PMID 17080809 (https://pubmed.ncbi.nlm.nih.gov/17080809). S2CID 11634997 (https://api.semanticscholar.org/CorpusID:11634997).
9. 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.
10. 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).
11. "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.
12. "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.
13. Pattern Languages of Programming, Conference proceedings (annual, 1994—) [1] (http://hillside.net/plop/pastconferences.html)
14. McConnell, Steve (June 2004). "Design in Construction". *Code Complete* (2nd ed.). Microsoft Press. p. 104 (https://archive.org/details/codecomplete0000mcco/page/104). ISBN 978-0-7356-1967-8. "Table 5.1 Popular Design Patterns"
15. Fowler, Martin (2002). *Patterns of Enterprise Application Architecture* (http://martinfowler.com/books.html#eaa). Addison-Wesley. ISBN 978-0-321-12742-6.

16. C. Martin, Robert (2002). "28. Extension object" (https://archive.org/details/agilesoftwaredev00robe/page/408). *Agile Software Development, Principles, Patterns, and Practices*. p. 408 (https://archive.org/details/agilesoftwaredev00robe/page/408). ISBN 978-0135974445.
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. 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.
26. Fowler, Martin (2006-08-01). "Writing Software Patterns" (http://www.martinfowler.com/articles/writingPatterns.html). Retrieved 2007-03-06.
27. Norvig, Peter (1998). *Design Patterns in Dynamic Languages* (http://www.norvig.com/design-patterns/).
28. Hannemann, Jan; Kiczales, Gregor (2002). "Design pattern implementation in Java and AspectJ" (https://doi.org/10.1145/582419.582436). *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.
29. Graham, Paul (2002). "Revenge of the Nerds" (http://www.paulgraham.com/icad.html). Retrieved 2012-08-11.
30. McConnell, Steve (2004). *Code Complete: A Practical Handbook of Software Construction, 2nd Edition* (https://archive.org/details/codecomplete0000mcco). p. 105 (https://archive.org/details/codecomplete0000mcco/page/105). ISBN 9780735619678.

## 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/ejbdesignpattern00mari). 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/patternsforparal0000matt). 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.