Introduction to Software Engineering/Architecture/Design Patterns

Design Patterns

If you remember, software engineers speak a common language called UML. And if we use this analogy of language, then *design patterns* are the common stories our culture shares, like for instance fairy tales. They are stories about commonly occurring problems in software design and their solutions. And as young children learn about good and evil from fairy tales, beginning software engineers learn about good design (design patterns) and bad design (anti-patterns).

History

Patterns originated as an architectural concept by Christopher Alexander (1977/79). In 1987, Kent Beck and Ward Cunningham began experimenting with the idea of applying patterns to programming and presented their results at the OOPSLA conference that year.[1][2] In the following years, Beck, Cunningham and others followed up on this work.

Design patterns gained popularity in computer science after the book *Design Patterns: Elements of Reusable Object-Oriented Software* was published in 1994 by the so-called "Gang of Four" (Gamma et al.).[3] 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.

Definition of a Design Pattern

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

Design patterns reside in the domain of modules and interconnections. At a higher level there are architectural patterns that are larger in scope, usually describing an overall pattern followed by an entire system.[4]

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

Examples of Design Patterns

Design patterns are easiest understood when looking at concrete examples. For beginners the following ten patterns may suffice. However, you should make it a habit to learn about some of the other patterns mentioned below. The more patterns you know, the better.

Factory Method

The Factory pattern creates an object from a set of similar classes, based on some parameter, usually a string. An example, is the creation of a MessageDigest object in Java:

```
MessageDigest md = MessageDigest.getInstance("SHA-1");
```

If one changes the parameter to "MD5" for instance, one gets an object that calculates the message digest based on the MD5 algorithm instead. The advantage of using a parameter is that changing the algorithm does not require us to re-compile our code. Other examples of this pattern are the loading of the database connection driver in Java using `Class.forName("jdbc.idbDriver")`, which admittedly is some very odd syntax, but the idea is the same.

Abstract Factory

Where the Factory pattern only affects one class, the Abstract Factory pattern affects a whole bunch of classes. A well known example from Java is the Swing Look-and-Feel:

```
UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
```

This looks quite similar to the Factory pattern, but the difference is that now every Swing class that is being loaded is affected, not just one.

Singleton

This is one of the most dangerous design patterns, when in doubt don't use it. Its main purpose is to guarantee that only one instance of a particular object exists. Possible applications are a printer manager or a database connection manager. It is useful when access to a limited resource needs to be controlled.

Iterator

Nowadays, the Iterator pattern is trivial: it allows you to go through a list of objects, starting at the beginning, iterating through the list one element after the other, until reaching the end.

Template Method

Also the Template Method pattern is rather simple: as soon as you define an abstract class, that forces its subclasses to implement some method, you are using a simple form of the Template pattern.

Command

To understand the idea behind the Command pattern consider the following restaurant example: A customer goes to a restaurant and orders some food. The waiter takes the order (command, in this case) and hands it to the cook in the kitchen. In the kitchen the command is executed, and depending on the command different food or drink is being prepared.

Observer

The Observer pattern is one of the most popular patterns, and it has many variants. Assume you have a table in a spreadsheet. That data can be displayed in table form, but also in form of some graph or histogram. If the underlying data changes, not only the table view has to change, but you also expect the histogram to change. To communicate these changes you can use the Observer pattern: the underlying data is the *observable* and the table view as well as the histogram view are *observers* that observe the observable. Another example of the Observer pattern is a button in Swing for instance: here the JButton is the observable, and if something happens to the button (usually this means it was pressed by the user) then the listener (the observer) gets notified.

Composite

The Composite pattern is very wide spread. Basically, it is a list that may contain objects, but also lists. A typical example is a file system, which may consist of directories and files. Here directories may contain files, but also may contain other directories. Other examples of the Composite pattern are menus that may contain other menus, or in user management one often has users and groups, where groups may contain users, but also other groups.

State

In the State pattern, an internal state of the object influences its behavior. Assume you have some drawing program in which you want to be able to draw straight lines and dotted lines. Instead of creating different classes for lines, you have one Line class that has an internal state called 'dotted' or 'straight' and depending on this internal state either dotted lines or straight lines are drawn. This pattern is also implicitly used by Java, when setting the font via ’setFont()’ or the color via 'setColor()', for instance.

Proxy

The idea behind the Proxy pattern is that we have some complex object and we need to make it simpler. One typical application is an object that exists on another machine, but you want to give the impression as if the user is dealing with a local object. Another application is when an object would take a long time to create (like loading a large image/video), but the actual object may never be needed. In this case a proxy represents the object until it is needed.

Patterns in Practice

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems, and it also improves code readability for coders and architects who are familiar with the patterns. In addition to this, patterns allow developers to communicate using well-known, well understood names for software interactions.

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.[5]

Classification and List of Patterns

Design patterns were originally grouped into the categories: creational patterns, structural patterns, and behavioral patterns, and described using the concepts of delegation, aggregation, and consultation. Another classification has also introduced the notion of architectural design pattern that may be applied at the architecture level of the software such as the Model-View-Controller pattern. The following patterns are taken from *Design Patterns* [3] and *Code Complete*,[6] unless otherwise stated.

Creational patterns

- **Abstract factory:** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Builder:** Separate the construction of a complex object from its representation allowing the same construction process to create various representations.
- **Factory method:** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **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.[7]
- **Multiton:** Ensure a class has only named instances, and provide global point of access to them.
- **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.
- **Prototype:** Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- **Resource acquisition is initialization:** Ensure that resources are properly released by tying them to the

lifespan of suitable objects.

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

Structural Patterns

- **Adapter or Wrapper:** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.
- **Bridge:** Decouple an abstraction from its implementation allowing the two to vary independently.
- **Composite:** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- **Decorator:** Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.
- **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.
- **Front Controller:** Provide a unified interface to a set of interfaces in a subsystem. Front Controller defines a higher-level interface that makes the subsystem easier to use.
- **Flyweight:** Use sharing to support large numbers of fine-grained objects efficiently.
- **Proxy:** Provide a surrogate or placeholder for another object to control access to it.

Behavioral Patterns

- **Blackboard:** Generalized observer, which allows multiple readers and writers. Communicates information system-wide.
- **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.
- **Command:** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- **Interpreter:** Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- **Iterator:** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- **Mediator:** Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- **Memento:** Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later.
- **Null object:** Avoid null references by providing a default object.
- **Observer or Publish/subscribe:** Define a one-to-many dependency between objects where a state change in one object results with all its dependents being notified and updated automatically.
- **Servant:** Define common functionality for a group of classes
- **Specification:** Recombinable business logic in a boolean fashion
- **State:** Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **Strategy:** Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- **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.
- **Visitor:** Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Concurrency Patterns

Most of the following concurrency patterns are taken from *POSA2*[8]

- **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.
- **Balking:** Only execute an action on an object when the object is in a particular state.
- **Binding Properties:** Combining multiple observers to force properties in different objects to be synchronized or coordinated in some way.[9]
- **Messaging pattern:** The messaging design pattern (MDP) allows the interchange of information (i.e. messages) between components and applications.
- **Double-checked locking:** Reduce the overhead of acquiring a lock by first testing the locking criterion (the

'lock hint') in an unsafe manner; only if that succeeds does the actual lock proceed. Can be unsafe when implemented in some language/hardware combinations. It can therefore sometimes be considered an anti-pattern.

- **Event-based asynchronous:** Addresses problems with the Asynchronous pattern that occur in multithreaded programs.[10]
- **Guarded suspension:** Manages operations that require both a lock to be acquired and a precondition to be satisfied before the operation can be executed.
- **Lock:** One thread puts a "lock" on a resource, preventing other threads from accessing or modifying it.[11][7]
- **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.
- **Reactor:** A reactor object provides an asynchronous interface to resources that must be handled synchronously.
- **Read-write lock:** Allows concurrent read access to an object but requires exclusive access for write operations.
- **Scheduler:** Explicitly control when threads may execute single-threaded code.
- **Thread pool:** A number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads. Can be considered a special case of the object pool pattern.
- **Thread-specific storage:** Static or "global" memory local to a thread.

Data Access Patterns

Another interesting area where patterns have a wide application is the area of *data access patterns*. Clifton Nock [12] lists the following patterns:

- **ORM Patterns:** Domain Object Factory, Object/Relational Map, Update Factory
- **Resource Management Patterns:** Resource Pool, Resource Timer, Retryer, Paging Iterator
- **Cache Patterns:** Cache Accessor, Demand Cache, Primed Cache, Cache Collector, Cache Replicator
- **Concurrency Patterns:** Transaction, Optimistic Lock, Pessimistic Lock

Enterprise Patterns

If you deal with J2EE or with .Net Enterprise applications, the problems that occur and the solutions to them are similar. These solutions are the Enterprise patterns. The book *Core J2EE Patterns* [13] lists these patterns:

- **Presentation Tier Patterns:** Intercepting Filter, Front Controller, View Helper, Composite View, Service to Worker, Dispatcher View
- **Business Tier Patterns:** Business Delegate, Value Object, Session Facade, Composite Entity, Value Object Assembler, Value List Handler, Service Locator
- **Integration Tier Patterns:** Data Access Object, Service Activator

Real-Time Patterns

Finally, in the area of real-time and embedded software development a vast number of patterns have been identified.[14][15] In his book *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*[16][17] Bruce Powel Douglass lists some very intriguing patterns:

- **Architecture Patterns:** Layered Pattern, Channel Architecture Pattern, Component-Based Architecture, Recursive Containment Pattern and Hierarchical Control Pattern, Microkernel Architecture Pattern, Virtual Machine Pattern
- **Concurrency Patterns:** Message Queuing Pattern, Interrupt Pattern, Guarded Call Pattern, Rendezvous Pattern, Cyclic Executive Pattern, Round Robin Pattern
- **Memory Patterns:** Static Allocation Pattern, Pool Allocation Pattern, Fixed Sized Buffer Pattern, Smart Pointer Pattern, Garbage Collection Pattern, Garbage Compactor Pattern
- **Resource Patterns:** Critical Section Pattern, Priority Inheritance Pattern, Priority Ceiling Pattern, Simultaneous Locking Pattern, Ordered Locking Pattern
- **Distribution Patterns:** Shared Memory Pattern, Remote Method Call Pattern, Observer Pattern, Data Bus Pattern, Proxy Pattern, Broker Pattern
- **Safety and Reliability Patterns:** Monitor-Actuator Pattern, Sanity Check Pattern, Watchdog Pattern, Safety Executive Pattern, Protected Single Channel Pattern, Homogeneous Redundancy Pattern, Triple Modular Redundancy Pattern, Heterogeneous Redundancy Pattern

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,[18] information visualization [19], secure design[20], "secure usability"[21], web design [22] and business model design.[23] The annual Pattern Languages of Programming Conference proceedings [24] include many examples of domain specific patterns.

Documenting and Describing Patterns

Assume you discovered a new design pattern. Or your friend wants to explain to you this cool pattern she found in a pattern repository. How do we describe patterns? There is no single, standard format for documenting design patterns. Rather, a variety of different formats have been used by different pattern authors.[25] 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 the book *Design Patterns*.[3] 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.

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.

References

1. ↑ Smith, Reid (October 1987). "Panel on design methodology". *OOPSLA '87 Addendum to the Proceedings*. OOPSLA '87. doi:10.1145/62138.62151., *"Ward cautioned against requiring too much programming at, what he termed, 'the high level of wizards.' He pointed out that a written 'pattern language' can significantly improve the selection and application of abstractions. He proposed a 'radical shift in the burden of design and implementation' basing the new methodology on an adaptation of Christopher Alexander's work in pattern languages and that programming-oriented pattern languages developed at Tektronix has significantly aided their software development efforts."*

2. ↑ Beck, Kent; Ward Cunningham (September 1987). "Using Pattern Languages for Object-Oriented Program". *OOPSLA '87 workshop on* Specification and Design for Object-Oriented Programming. OOPSLA '87. http://c2.com/doc/oopsla87.html. Retrieved 2006-05-26.
3. ↑ ***a b c*** Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.
4. ↑ Martin, Robert C.. "Design Principles and Design Patterns". http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf. Retrieved 2000.
5. ↑ Meyer, Bertrand; Karine Arnout (July 2006). "Componentization: The Visitor Example". *IEEE Computer* (IEEE) **39** (7): 23–30. http://se.ethz.ch/~meyer/publications/computer/visitor.pdf.
6. ↑ McConnell, Steve (June 2004). "Design in Construction". *Code Complete* (2nd ed.). Microsoft Press. pp. 104. ISBN 978-0735619678. "Table 5.1 Popular Design Patterns"
7. ↑ ***a b*** Fowler, Martin (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley. ISBN 978-0321127426. http://martinfowler.com/.
8. ↑ , unless stated otherwise. Schmidt, Douglas C.; Michael Stal, Hans Rohnert, Frank Buschmann (2000). *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons. ISBN 0-471-60695-2.
9. ↑ http://c2.com/cgi/wiki?BindingProperties
10. ↑ Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, and Morgan Skinner (2008). "Event-based Asynchronous Pattern". *Professional C# 2008*. Wiley. pp. 570–571. ISBN 0470191376.
11. ↑ http://c2.com/cgi/wiki?LockPattern
12. ↑ Nock, Clifton (2003). *Data Access Patterns: Database Interactions in Object-Oriented Applications*. Addison Wesley. ISBN 0131401572.
13. ↑ Alur, Deepak; John Crupi, Dan Malks (May 2003). *Core J2EE Patterns: Best Practices and Design Strategies (2nd Edition)*. Prentice Hall. ISBN 0131422464.
14. ↑ "STL Design Patterns II,". EventHelix.com Inc.. http://www.eventhelix.com/RealtimeMantra/Patterns/stl_design_patterns_2.htm. Retrieved 2011-03-08.
15. ↑ "Embedded Design Patterns,". EventHelix.com Inc.. http://www.eventhelix.com/RealtimeMantra/Patterns/. Retrieved 2011-03-08.
16. ↑ Douglass, Bruce Powel (2002). *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison Wesley. ISBN 0201699567.
17. ↑ Douglass, Bruce Powel (1999). *Doing Hard Time: Developing Real-Time Systems with UML, Objects,* Frameworks and Patterns. *Addison Wesley. ISBN 0201498375.*
18. ↑ Laakso, Sari A. (2003-09-16). "Collection of User Interface Design Patterns". University of Helsinki, Dept. of Computer Science. http://www.cs.helsinki.fi/u/salaakso/patterns/index.html. Retrieved 2008-01-31.
19. ↑ Heer, J.; M. Agrawala (2006). "Software Design Patterns for Information Visualization". *IEEE Transactions on Visualization and Computer Graphics* **12** (5): 853. doi:10.1109/TVCG.2006.178. PMID 17080809. http://vis.berkeley.edu/papers/infovis_design_patterns/.
20. ↑ Chad Dougherty et al (2009). *Secure Design Patterns*. http://www.cert.org/archive/pdf/09tr010.pdf.
21. ↑ Simson L. Garfinkel (2005). *Design Principles and Patterns for Computer Systems That Are Simultaneously Secure and Usable*. http://www.simson.net/thesis/.
22. ↑ "Yahoo! Design Pattern Library". http://developer.yahoo.com/ypatterns/. Retrieved 2008-01-31.
23. ↑ "How to design your Business Model as a Lean Startup?". http://torgronsund.wordpress.com/2010/01/06/lean-startup-business-model-pattern/. Retrieved 2010-01-06.
24. ↑ Pattern Languages of Programming, Conference proceedings (annual, 1994—) [1]

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

Further Reading

Books

- Alexander, Christopher; Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel (1977). *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press. ISBN 978-0195019193.
- Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.
- Buschmann, Frank; Regine Meunier, Hans Rohnert, Peter Sommerlad (1996). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons. ISBN 0-471-95869-7.
- Schmidt, Douglas C.; Michael Stal, Hans Rohnert, Frank Buschmann (2000). *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons. ISBN 0-471-60695-2.
- Fowler, Martin (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley. ISBN 978-0321127426.
- Hohpe, Gregor; Bobby Woolf (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley. ISBN 0-321-20068-3.
- Freeman, Eric T; Elisabeth Robson, Bert Bates, Kathy Sierra (2004). *Head First Design Patterns*. O'Reilly Media. ISBN 0-596-00712-4.
- Alur, Deepak; John Crupi, Dan Malks (May 2003). *Core J2EE Patterns: Best Practices and Design Strategies (2nd Edition)*. Prentice Hall. ISBN 0131422464.
- Beck, Kent (October 2007). *Implementation Patterns*. Addison-Wesley. ISBN 978-0321413093.
- Beck, Kent; R. Crocker, G. Meszaros, J.O. Coplien, L. Dominick, F. Paulisch, and J. Vlissides (March 1996). *Proceedings of the 18th International Conference on Software Engineering*. pp. 25–30.
- Nock, Clifton (2003). *Data Access Patterns: Database Interactions in Object-Oriented Applications*. Addison Wesley. ISBN 0131401572.
- Borchers, Jan (2001). *A Pattern Approach to Interaction Design*. John Wiley & Sons. ISBN 0-471-49828-9.
- Coplien, James O.; Douglas C. Schmidt (1995). *Pattern Languages of Program Design*. Addison-Wesley. ISBN 0-201-60734-4.
- Coplien, James O.; John M. Vlissides, and Norman L. Kerth (1996). *Pattern Languages of Program Design 2*. Addison-Wesley. ISBN 0-201-89527-7.
- Fowler, Martin (1997). *Analysis Patterns: Reusable Object Models*. Addison-Wesley. ISBN 0-201-89542-0.
- Fowler, Martin (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley. ISBN 978-0321127426.
- Freeman, Eric; Elisabeth Freeman, Kathy Sierra, and Bert Bates (2004). *Head First Design Patterns*.

O'Reilly Media. ISBN 0-596-00712-4.
- Hohmann, Luke; Martin Fowler and Guy Kawasaki (2003). *Beyond Software Architecture*. Addison-Wesley. ISBN 0-201-77594-8.
- Alur, Deepak; Elisabeth Freeman, Kathy Sierra, and Bert Bates (2004). *Head First Design Patterns*. O'Reilly Media. ISBN 0-596-00712-4.
- Gabriel, Richard (1996) (PDF). *Patterns of Software: Tales From The Software Community*. Oxford University Press. pp. 235. ISBN 0-19-512123-6. http://www.dreamsongs.com/NewFiles/PatternsOfSoftware.pdf.
- Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.
- Hohpe, Gregor; Bobby Woolf (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley. ISBN 0-321-20068-3.
- Holub, Allen (2004). *Holub on Patterns*. Apress. ISBN 1-59059-388-X.
- Kircher, Michael; Markus Völter and Uwe Zdun (2005). *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*. John Wiley & Sons. ISBN 0-470-85662-9.
- Larman, Craig (2005). *Applying UML and Patterns*. Prentice Hall. ISBN 0-13-148906-2.
- Liskov, Barbara; John Guttag (2000). *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.*. Addison-Wesley. ISBN 0-201-65768-6.
- Manolescu, Dragos; Markus Voelter and James Noble (2006). *Pattern Languages of Program Design 5*. Addison-Wesley. ISBN 0-321-32194-4.
- Marinescu, Floyd (2002). *EJB Design Patterns: Advanced Patterns, Processes and Idioms*. John Wiley & Sons. ISBN 0-471-20831-0.
- Martin, Robert Cecil; Dirk Riehle and Frank Buschmann (1997). *Pattern Languages of Program Design 3*. Addison-Wesley. ISBN 0-201-31011-2.
- Mattson, Timothy G; Beverly A. Sanders and Berna L. Massingill (2005). *Patterns for Parallel Programming*. Addison-Wesley. ISBN 0-321-22811-1.
- Shalloway, Alan; James R. Trott (2001). *Design Patterns Explained, Second Edition: A New Perspective on Object-Oriented Design*. Addison-Wesley. ISBN 0-321-24714-0.
- Vlissides, John M. (1998). *Pattern Hatching: Design Patterns Applied*. Addison-Wesley. ISBN 0-201-43293-5.
- Weir, Charles; James Noble (2000). *Small Memory Software: Patterns for systems with limited memory*. Addison-Wesley. ISBN 0201596075. http://www.cix.co.uk/~smallmemory/.

Web sites

- "History of Patterns". *Portland Pattern Repository*. http://www.c2.com/cgi-bin/wiki?HistoryOfPatterns. Retrieved 2005-07-28.
- "Are Design Patterns Missing Language Features?". Cunningham & Cunningham, Inc.. http://www.c2.com/cgi/wiki?AreDesignPatternsMissingLanguageFeatures. Retrieved 2006-01-20.
- "Show Trial of the Gang of Four". Cunningham & Cunningham, Inc.. http://www.c2.com/cgi/wiki?ShowTrialOfTheGangOfFour. Retrieved 2006-01-20.
- "Design Patterns in Modern Day Software Factories (WCSF)". XO Software, Ltd. http://www.xosoftware.co.uk/Articles/WCSFDesignPatterns/. Retrieved 2010-02-22.
- "STL Design Patterns II,". EventHelix.com Inc.. http://www.eventhelix.com/RealtimeMantra/Patterns/stl_design_patterns_2.htm. Retrieved 2011-03-08.
- "Embedded Design Patterns,". EventHelix.com Inc.. http://www.eventhelix.com/RealtimeMantra/Patterns/.

Retrieved 2011-03-08.
- "Enterprise Integration Patterns". Gregor Hohpe and Bobby Woolf, Addison-Wesley. http://www.enterpriseintegrationpatterns.com/. Retrieved 2011-03-08.

External Links

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