Cohesion & Coupling

Contents

Chapter 1

Information Hiding

1.1 Information hiding

This article is about the computer programming concept. For the practice of hiding data in a message or file, see Steganography. For data encryption, see Cryptography.

In computer science, **information hiding** is the principle of segregation of the *design decisions* in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed. The protection involves providing a stable interface which protects the remainder of the program from the implementation (the details that are most likely to change).

Written another way, information hiding is the ability to prevent certain aspects of a class or software component from being accessible to its clients, using either programming language features (like private variables) or an explicit exporting policy.

1.1.1 Overview

The term *encapsulation* is often used interchangeably with information hiding. Not all agree on the distinctions between the two though; one may think of information hiding as being the principle and encapsulation being the technique. A software module hides information by encapsulating the information into a module or other construct which presents an interface.[1]

A common use of information hiding is to hide the physical storage layout for data so that if it is changed, the change is restricted to a small subset of the total program. For example, if a three-dimensional point (x,y,z) is represented in a program with three floating point scalar variables and later, the representation is changed to a single array variable of size three, a module designed with information hiding in mind would protect the remainder of the program from such a change.

In object-oriented programming, information hiding (by way of nesting of types) reduces software development risk by shifting the code's dependency on an uncertain implementation (design decision) onto a well-defined interface. Clients of the interface perform operations purely through it so if the implementation changes, the clients do not have to change.

1.1.2 Encapsulation

See also: Encapsulation (object-oriented programming)

In his book on object-oriented design, Grady Booch defined encapsulation as "the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation."[2]

The purpose is to achieve potential for change: the internal mechanisms of the component can be improved without impact on other components, or the component can be replaced with a different one that supports the same public interface. Encapsulation also protects the integrity of the component, by preventing users from setting the internal data of the component into an invalid or inconsistent state. Another benefit of encapsulation is that it reduces system complexity and thus increases robustness, by limiting the interdependencies between software components.[2]

In this sense, the idea of encapsulation is more general than how it is applied in OOP: for example, a relational database is encapsulated in the sense that its only public interface is a Query language (SQL for example), which hides all the internal machinery and data structures of the database management system. As such, encapsulation is a core principle of good software architecture, at every level of granularity.

Encapsulating software behind an interface allows the construction of objects that mimic the behavior and interactions of objects in the real world. For example, a simple digital alarm clock is a real-world object that a lay person can use and understand. They can understand what the alarm clock does, and how to use it through the provided interface (buttons and screen), without having to understand every part inside of the clock. Similarly, if you replaced the clock with a different model, the lay person could continue to use it in the same way, provided that the interface works the same.

In the more concrete setting of an object-oriented programming language, the notion is used to mean either an information hiding mechanism, a bundling mechanism, or the combination of the two. (See Encapsulation (object-oriented programming) for details.)

1.1.3 History

The concept of information hiding was first described by David Parnas in Parnas (1972).[3] Before then, modularity was discussed by Richard Gauthier and Stephen Pont in their 1970 book *Designing Systems Programs* although modular programming itself had been used at many commercial sites for many years previously – especially in I/O sub-systems and software libraries – without acquiring the 'information hiding' tag – but for similar reasons, as well as the more obvious code reuse reason.

1.1.4 Example of information hiding

Information hiding serves as an effective criterion for dividing any piece of equipment, software or hardware, into modules of functionality. For instance a car is a complex piece of equipment. In order to make the design, manufacturing, and maintenance of a car reasonable, the complex piece of equipment is divided into modules with particular interfaces hiding design decisions. By designing a car in this fashion, a car manufacturer can also offer various options while still having a vehicle which is economical to manufacture.

For instance, a car manufacturer may have a luxury version of the car as well as a standard version. The luxury version comes with a more powerful engine than the standard version. The engineers designing the two different car engines, one for the luxury version and one for the standard version, provide the same interface for both engines. Both engines fit into the engine bay of the car which is the same between both versions. Both engines fit the same transmission, the same engine mounts, and the same controls. The differences in the engines are that the more powerful luxury version has a larger displacement with a fuel injection system that is programmed to provide the fuel air mixture that the larger displacement engine requires.

In addition to the more powerful engine, the luxury version may also offer other options such as a better radio with CD player, more comfortable seats, a better suspension system with wider tires, and different paint colors. With all of these changes, most of the car is the same between the standard version and the luxury version. The radio with CD player is a module which replaces the standard radio, also a module, in the luxury model. The more comfortable seats are installed into the same seat mounts as the standard types of seats. Whether the seats are leather or plastic, or offer lumbar support or not, doesn't matter.

The engineers design the car by dividing the task up into pieces of work which are assigned to teams. Each team then designs their component to a particular standard or interface which allows the team flexibility in the design of the component while at the same time ensuring that all of the components will fit together.

Motor vehicle manufacturers frequently use the same core structure for several different models, in part as a cost-control measure. Such a "platform" also provides an example of information hiding, since the floorpan can be built without knowing whether it is to be used in a sedan or a hatchback.

As can be seen by this example, information hiding provides flexibility. This flexibility allows a programmer to modify functionality of a computer program during normal evolution as the computer program is changed to better fit the needs of users. When a computer program is well designed decomposing the source code solution into modules using the principle of information hiding, evolutionary changes are much easier because the changes typically are local rather than global changes.

Cars provide another example of this in how they interface with drivers. They present a standard interface (pedals, wheel, shifter, signals, gauges, etc.) on which people are trained and licensed. Thus, people only have to learn to drive a car; they don't need to learn a completely different way of driving every time they drive a new model. (Granted, there are manual and automatic transmissions and other such differences, but on the whole cars maintain a unified interface.)

1.1.5 Relation to object-oriented programming

The authors of *Design Patterns* discuss the tension between inheritance and encapsulation at length and state that in their experience, designers overuse inheritance (Gang of Four 1995:20). The danger is stated as follows:

> “Because inheritance exposes a subclass to details of its parent’s implementation, it’s often said that 'inheritance breaks encapsulation'". (Gang of Four 1995:19)

1.1.6 See also

- Implementation inheritance
- Inheritance semantics
- Modularity (programming)
- Opaque data type
- Virtual inheritance
- Transparency (human–computer interaction)
- Scope (programming)
- Compartmentalization (information security)
- Law of Demeter

1.1.7 References

[1] “Encapsulation is not information hiding”. JavaWorld.

[2] Grady Booch, *Object-Oriented Analysis and Design with Applications*, . Addison-Wesley, 2007, ISBN 0-201-89551-X, p. 51-52

[3] Scott 2009, p. 173.

- Parnas, D.L. (December 1972). “On the Criteria To Be Used in Decomposing Systems into Modules” (PDF). *Communications of the ACM* **15** (12): 1053–58. doi:10.1145/361598.361623.
- Scott, Michael L. (2009) [2000]. *Programming Language Pragmatics* (Third ed.). Morgan Kaufmann Publishers. ISBN 978-0-12-374514-9.

Chapter 2

Patterns, Classes and Separation

Chapter 3

Cohesion & Coupling

3.1 Cohesion (computer science)

In computer programming, **cohesion** refers to the *degree to which the elements of a module belong together*.[1] Thus, cohesion measures the strength of relationship between pieces of functionality within a given module. For example, in highly cohesive systems functionality is strongly related.

Cohesion is an ordinal type of measurement and is usually described as “high cohesion” or “low cohesion”. Modules with high cohesion tend to be preferable, because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability. In contrast, low cohesion is associated with undesirable traits such as being difficult to maintain, test, reuse, or even understand.

Cohesion is often contrasted with coupling, a different concept. High cohesion often correlates with loose coupling, and vice versa. The software metrics of coupling and cohesion were invented by Larry Constantine in the late 1960s as part of Structured Design, based on characteristics of “good” programming practices that reduced maintenance and modification costs. Structured Design, cohesion and coupling were published in the article Stevens, Myers & Constantine (1974) and the book Yourdon & Constantine (1979); the latter two subsequently became standard terms in software engineering.

In object-oriented programming, if the methods that serve a class tend to be similar in many aspects, then the class is said to have high cohesion. In a highly cohesive system, code readability and reusability is increased, while complexity is kept manageable.

Cohesion is increased if:

- The functionalities embedded in a class, accessed through its methods, have much in common.
- Methods carry out a small number of related activities, by *avoiding* coarsely grained or unrelated sets of data.

Advantages of high cohesion (or “strong cohesion”) are:

- Reduced module complexity (they are simpler, having fewer operations).
- Increased system maintainability, because logical changes in the domain affect fewer modules, and because changes in one module require fewer changes in other modules.
- Increased module reusability, because application developers will find the component they need more easily among the cohesive set of operations provided by the module.

While in principle a module can have perfect cohesion by only consisting of a single, atomic element – having a single function, for example – in practice complex tasks are not expressible by a single, simple element. Thus a single-element module has an element that either is too complicated, in order to accomplish task, or is too narrow, and thus tightly coupled to other modules. Thus cohesion is balanced with both unit complexity and coupling.

3.1.1 Types of cohesion

Cohesion is a qualitative measure, meaning that the source code to be measured is examined using a rubric to determine a classification. Cohesion types, from the worst to the best, are as follows:

Coincidental cohesion (worst) Coincidental cohesion is when parts of a module are grouped arbitrarily; the only relationship between the parts is that they have been grouped together (e.g. a "Utilities" class).

Logical cohesion Logical cohesion is when parts of a module are grouped because they are logically categorized to do the same thing even though they are different by nature (e.g. grouping all mouse and keyboard input handling routines).

Temporal cohesion Temporal cohesion is when parts of a module are grouped by when they are processed - the parts are processed at a particular time in program execution (e.g. a function which is called after catching an exception which closes open files, creates an error log, and notifies the user).

Procedural cohesion Procedural cohesion is when parts of a module are grouped because they always follow a certain sequence of execution (e.g. a function which checks file permissions and then opens the file).

Communicational/informational cohesion Communicational cohesion is when parts of a module are grouped because they operate on the same data (e.g. a module which operates on the same record of information).

Sequential cohesion Sequential cohesion is when parts of a module are grouped because the output from one part is the input to another part like an assembly line (e.g. a function which reads data from a file and processes the data).

Functional cohesion (best) Functional cohesion is when parts of a module are grouped because they all contribute to a single well-defined task of the module (e.g. Lexical analysis of an XML string).

Although cohesion is a ranking type of scale, the ranks do not indicate a steady progression of improved cohesion. Studies by various people including Larry Constantine, Edward Yourdon, and Steve McConnell [2] indicate that the first two types of cohesion are inferior; communicational and sequential cohesion are very good; and functional cohesion is superior.

While functional cohesion is considered the most desirable type of cohesion for a software module, it may not be achievable. There are cases where communicational cohesion is the highest level of cohesion that can be attained under the circumstances.

3.1.2 See also

- List of object-oriented programming terms
- Static code analysis

3.1.3 References

[1] Yourdon & Constantine 1979.

[2] Code Complete 2nd Ed., p168-171

- Stevens, W. P.; Myers, G. J.; Constantine, L. L. (June 1974). "Structured design". *IBM Systems Journal* **13** (2): 115–139. doi:10.1147/sj.132.0115.
- Yourdon, Edward; Constantine, Larry L. (1979) [1975]. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Yourdon Press. ISBN 0-13-854471-9.

3.1.4 External links

- Definitions of Cohesion metrics
- Cohesion metrics

3.2 Coupling (computer programming)

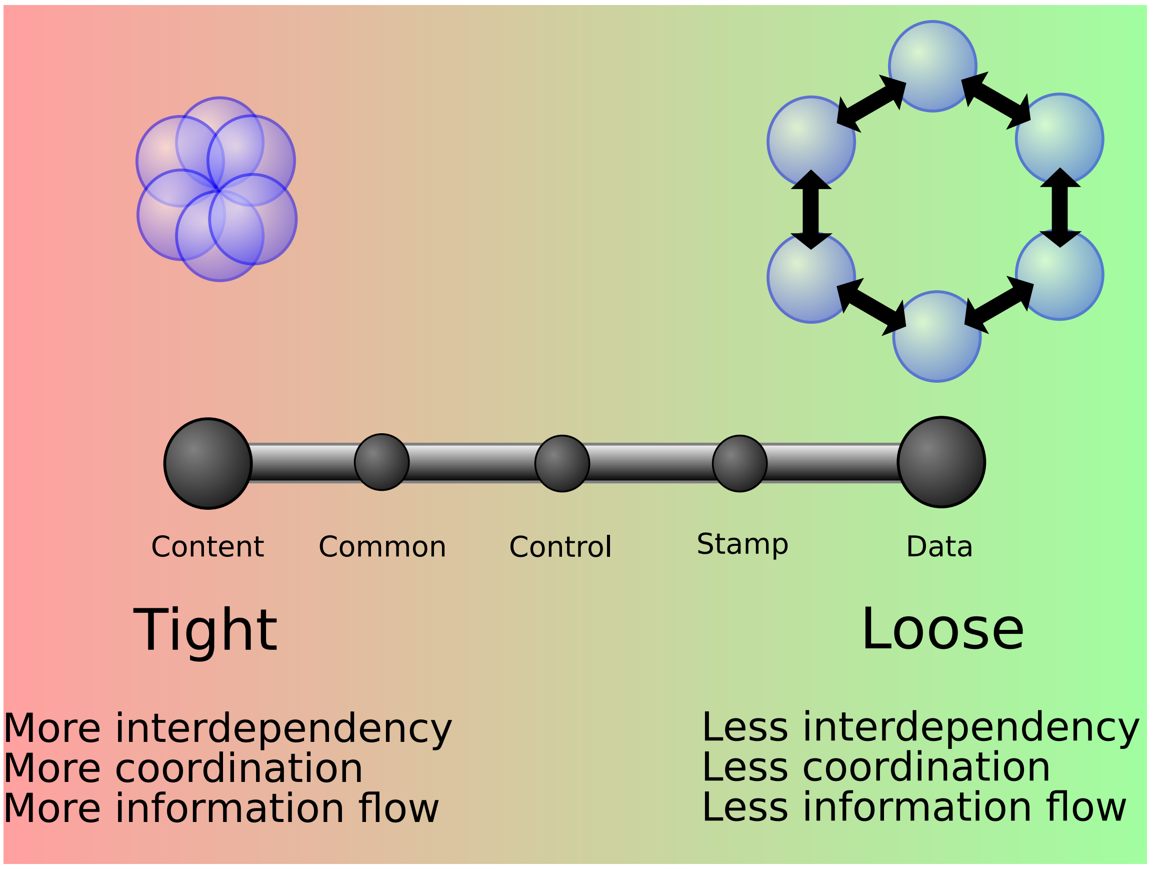
In software engineering, **coupling** is the manner and degree of interdependence between software modules; a measure of how closely connected two routines or modules are;[1] the strength of the relationships between modules.[2]

Coupling is usually contrasted with cohesion. Low coupling often correlates with high cohesion, and vice versa. Low coupling is often a sign of a well-structured computer system and a good design, and when combined with high cohesion, supports the general goals of high readability and maintainability.

3.2.1 History

The software quality metrics of coupling and cohesion were invented by Larry Constantine in the late 1960s as part of Structured Design, based on characteristics of "good" programming practices that reduced maintenance and modification costs. Structured Design, including cohesion and coupling, were published in the article Stevens, Myers & Constantine (1974) and the book Yourdon & Constantine (1979), and the latter subsequently became standard terms.

3.2.2 Types of coupling



Conceptual model of coupling

Coupling can be “low” (also “loose” and “weak”) or “high” (also “tight” and “strong”). Some types of coupling, in order of highest to lowest coupling, are as follows:

Procedural programming

A module here refers to a subroutine of any kind, i.e. a set of one or more statements having a name and preferably its own set of variable names.

Content coupling (high) Content coupling (also known as **Pathological coupling**) occurs when one module modifies or relies on the internal workings of another module (e.g., accessing local data of another module).

Therefore changing the way the second module produces data (location, type, timing) will lead to changing the dependent module.

Common coupling Common coupling (also known as **Global coupling**) occurs when two modules share the same global data (e.g., a global variable).

Changing the shared resource implies changing all the modules using it.

External coupling External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface. This is basically related to the communication to external tools and devices.

Control coupling Control coupling is one module controlling the flow of another, by passing it information on what to do (e.g., passing a what-to-do flag).

Stamp coupling (Data-structured coupling) Stamp coupling occurs when modules share a composite data structure and use only a part of it, possibly a different part (e.g., passing a whole record to a function that only needs one field of it).

This may lead to changing the way a module reads a record because a field that the module does not need has been modified.

Data coupling Data coupling occurs when modules share data through, for example, parameters. Each datum is an elementary piece, and these are the only data shared (e.g., passing an integer to a function that computes a square root).

Message coupling (low) This is the loosest type of coupling. It can be achieved by state decentralization (as in objects) and component communication is done via parameters or message passing (see Message passing).

No coupling Modules do not communicate at all with one another.

Object-oriented programming

Subclass Coupling Describes the relationship between a child and its parent. The child is connected to its parent, but the parent is not connected to the child.

Temporal coupling When two actions are bundled together into one module just because they happen to occur at the same time.

In recent work various other coupling concepts have been investigated and used as indicators for different modularization principles used in practice.[3]

3.2.3 Disadvantages

Tightly coupled systems tend to exhibit the following developmental characteristics, which are often seen as disadvantages:

1. A change in one module usually forces a ripple effect of changes in other modules.
2. Assembly of modules might require more effort and/or time due to the increased inter-module dependency.
3. A particular module might be harder to reuse and/or test because dependent modules must be included.

3.2.4 Performance issues

Whether loosely or tightly coupled, a system's performance is often reduced by message and parameter creation, transmission, translation (e.g. marshaling) and message interpretation (which might be a reference to a string, array or data structure), which require less overhead than creating a complicated message such as a SOAP message. Longer messages require more CPU and memory to produce. To optimize runtime performance, message length must be minimized and message meaning must be maximized.

Message Transmission Overhead and Performance Since a message must be transmitted in full to retain its complete meaning, message transmission must be optimized. Longer messages require more CPU and memory to transmit and receive. Also, when necessary, receivers must reassemble a message into its original state to completely receive it. Hence, to optimize runtime performance, message length must be minimized and message meaning must be maximized.

Message Translation Overhead and Performance Message protocols and messages themselves often contain extra information (i.e., packet, structure, definition and language information). Hence, the receiver often needs to translate a message into a more refined form by removing extra characters and structure information and/or by converting values from one type to another. Any sort of translation increases CPU and/or memory overhead. To optimize runtime performance, message form and content must be reduced and refined to maximize its meaning and reduce translation.

Message Interpretation Overhead and Performance All messages must be interpreted by the receiver. Simple messages such as integers might not require additional processing to be interpreted. However, complex messages such as SOAP messages require a parser and a string transformer for them to exhibit intended meanings. To optimize runtime performance, messages must be refined and reduced to minimize interpretation overhead.

3.2.5 Solutions

One approach to decreasing coupling is functional design, which seeks to limit the responsibilities of modules along functionality. Coupling increases between two classes ***A*** and ***B*** if:

- ***A*** has an attribute that refers to (is of type) ***B***.
- ***A*** calls on services of an object ***B***.
- ***A*** has a method that references ***B*** (via return type or parameter).
- ***A*** is a subclass of (or implements) class ***B***.

Low coupling refers to a relationship in which one module interacts with another module through a simple and stable interface and does not need to be concerned with the other module's internal implementation (see Information Hiding).

Systems such as CORBA or COM allow objects to communicate with each other without having to know anything about the other object's implementation. Both of these systems even allow for objects to communicate with objects written in other languages.

3.2.6 Coupling versus cohesion

Coupling and cohesion are terms which occur together very frequently. Coupling refers to the interdependencies between modules, while cohesion describes how related are the functions within a single module. Low cohesion implies that a given module performs tasks which are not very related to each other and hence can create problems as the module becomes large.

3.2.7 Module coupling

Coupling in Software Engineering[4] describes a version of metrics associated with this concept.

For data and control flow coupling:

- $\mathbf{d_i}$: number of input data parameters
- $\mathbf{c_i}$: number of input control parameters
- $\mathbf{d_o}$: number of output data parameters
- $\mathbf{c_o}$: number of output control parameters

For global coupling:

- $\mathbf{g_d}$: number of global variables used as data
- $\mathbf{g_c}$: number of global variables used as control

For environmental coupling:

- **w**: number of modules called (fan-out)
- **r**: number of modules calling the module under consideration (fan-in)

$\text{Coupling}(C) = 1 - \frac{1}{d_i + 2\times c_i + d_o + 2\times c_o + g_d + 2\times g_c + w + r}$

Coupling(C) makes the value larger the more coupled the module is. This number ranges from approximately 0.67 (low coupling) to 1.0 (highly coupled)

For example, if a module has only a single input and output data parameter

$C = 1 - \frac{1}{1+0+1+0+0+0+1+0} = 1 - \frac{1}{3} = 0.67$

If a module has 5 input and output data parameters, an equal number of control parameters, and accesses 10 items of global data, with a fan-in of 3 and a fan-out of 4,

$C = 1 - \frac{1}{5+2\times 5+5+2\times 5+10+0+3+4} = 0.98$

3.2.8 See also

- Cohesion (computer science)
- Connascence (computer science)
- Dependency hell
- Efferent coupling
- Inversion of control
- List of object-oriented programming terms
- Loose coupling
- Make (software)
- Static code analysis
- Coupling (physics)

3.2.9 References

[1] ISO/IEC/IEEE 24765:2010 Systems and software engineering — Vocabulary

[2] ISO/IEC TR 19759:2005, Software Engineering — Guide to the Software Engineering Body of Knowledge (SWEBOK)

[3] F. Beck, S. Diehl. On the Congruence of Modularity and Code Coupling. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (SIGSOFT/FSE '11), Szeged, Hungary, September 2011. doi:10.1145/2025113.2025162

[4] Pressman, Roger S. Ph.D. (1982). Software Engineering - A Practitioner's Approach - Fourth Edition. ISBN 0-07-052182-4

- Stevens, W. P.; Myers, G. J.; Constantine, L. L. (June 1974). "Structured design". *IBM Systems Journal* **13** (2): 115–139. doi:10.1147/sj.132.0115.
- Yourdon, Edward; Constantine, Larry L. (1979) [1975]. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Yourdon Press. ISBN 0-13-854471-9.

Chapter 4

Principles

4.1 Separation of mechanism and policy

The **separation of mechanism and policy**[1] is a design principle in computer science. It states that mechanisms (those parts of a system implementation that control the authorization of operations and the allocation of resources) should not dictate (or overly restrict) the policies according to which decisions are made about which operations to authorize, and which resources to allocate.

This is most commonly discussed in the context of security mechanisms (authentication and authorization), but is actually applicable to a much wider range of resource allocation problems (e.g. CPU scheduling, memory allocation, quality of service), and the general question of good object abstraction.

Per Brinch Hansen introduced the concept of separation of policy and mechanism in operating systems in the RC 4000 multiprogramming system.[2] Artsy and Livny, in a 1987 paper, discussed an approach for an operating system design having an "extreme separation of mechanism and policy".[3][4] In a 2000 article, Chervenak et al. described the principles of *mechanism neutrality* and *policy neutrality*.[5]

4.1.1 Rationale and implications

The separation of mechanism and policy is the fundamental approach of a microkernel that distinguishes it from a monolithic one. In a microkernel the majority of operating system services are provided by user-level server processes.[6] It is considered important for an operating system to have the flexibility of providing adequate mechanisms to support the broadest possible spectrum of real-world security policies.[7]

It is almost impossible to envision all of the different ways in which a system might be used by different types of users over the life of the product. This means that any hard-coded policies are likely to be inadequate or inappropriate for some (or perhaps even most) potential users. Decoupling the mechanism implementations from the policy specifications makes it possible for different applications to use the same mechanism implementations with different policies. This means that those mechanisms are likely to better meet the needs of a wider range of users, for a longer period of time.

If it is possible to enable new policies without changing the implementing mechanisms, the costs and risks of such policy changes can be greatly reduced. In the first instance, this could be accomplished merely by segregating mechanisms and their policies into distinct modules: by replacing the module which dictates a policy (e.g. CPU scheduling policy) without changing the module which executes this policy (e.g. the scheduling mechanism), we can change the behaviour of the system. Further, in cases where a wide or variable range of policies are anticipated depending on applications' needs, it makes sense to create some non-code means for specifying policies, i.e. policies are not hardcoded into executable code but can be specified as an independent description. For instance, file protection policies (e.g. Unix's *user/group/other read/write/execute*) might be parametrized. Alternatively an implementing mechanism could be designed to include an interpreter for a new policy specification language. In both cases, the systems are usually accompanied by a deferred binding mechanism (e.g. configuration files, or APIs) that permits policy specifications to be incorporated to the system or replaced by another after it has been delivered to the customer.

An everyday example of mechanism/policy separation is the use of card keys to gain access to locked doors. The mechanisms (magnetic card readers, remote controlled locks, connections to a security server) do not impose any limitations on entrance policy (which people should be allowed to enter which doors, at which times). These decisions are made by a centralized security server, which (in turn) probably makes its decisions by consulting a database of room access rules. Specific authorization decisions can be changed by updating a room access database. If the rule schema of that database proved too limiting, the entire security server could be replaced while leaving the fundamental mechanisms (readers, locks, and connections) unchanged.

Contrast this with issuing physical keys: if you want to change who can open a door, you have to issue new keys and change the lock. This intertwines the unlocking mechanisms with the access policies. For a hotel, this is significantly less effective than using key cards.

4.1.2 See also

- Separation of protection and security
- Separation of concerns

4.1.3 Notes

[1] Butler W. Lampson and Howard E. Sturgis. *Reflections on an Operating System Design* Communications of the ACM 19(5):251-265 (May 1976)

[2] "Per Brinch Hansen • IEEE Computer Society". *www.computer.org*. Retrieved 2016-02-05.

[3] Miller, M. S., & Drexler, K. E. (1988). "Markets and computation: Agoric open systems". In Huberman, B. A. (Ed.). (1988), pp. 133–176. *The Ecology of Computation*. North-Holland.

[4] Artsy, Yeshayahu *et al.*, 1987.

[5] Chervenak 2000 p.2

[6] Raphael Finkel, Michael L. Scott, Artsy Y. and Chang, H. *[www.cs.rochester.edu/u/scott/papers/1989_IEEETSE_Charlotte.pdf Experience with Charlotte: simplicity and function in a distributed operating system]. IEEE Trans. Software Engng 15:676-685; 1989. Extended abstract presented at the IEEE Workshop on Design Principles for Experimental Distributed Systems, Purdue University; 1986.*

[7] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau *The Flask Security Architecture: System Support for Diverse Security Policies* In Proceedings of the Eighth USENIX Security Symposium, pages 123–139, Aug. 1999.

4.1.4 References

- Per Brinch Hansen (2001). "The evolution of operating systems" (pdf). Retrieved 2006-10-24. included in book: Per Brinch Hansen, ed. (2001) [2001]. "1" (PDF). *Classic operating systems: from batch processing to distributed systems*. New York,: Springer-Verlag. pp. 1–36. ISBN 0-387-95113-X. (p.18)
- Wulf, W.; E. Cohen; W. Corwin; A. Jones; R. Levin; C. Pierson; F. Pollack (June 1974). "HYDRA: the kernel of a multiprocessor operating system". *Communications of the ACM* **17** (6): 337–345. doi:10.1145/355616.364017. ISSN 0001-0782.
- Hansen, Per Brinch (April 1970). "The nucleus of a Multiprogramming System". *Communications of the ACM* **13** (4): 238–241. doi:10.1145/362258.362278. ISSN 0001-0782. (pp.238–241)
- Levin, R.; E. Cohen; W. Corwin; F. Pollack; W. Wulf (1975). "Policy/mechanism separation in Hydra". *ACM Symposium on Operating Systems Principles / Proceedings of the fifth ACM symposium on Operating systems principles*: 132–140. doi:10.1145/800213.806531.
- Chervenak et al. *The data grid* Journal of Network and Computer Applications, Volume 23, Issue 3, July 2000, Pages 187-200
- Artsy, Yeshayahu, and Livny, Miron, An Approach to the Design of Fully Open Computing Systems (University of Wisconsin / Madison, March 1987) Computer Sciences Technical Report #689.

4.1.5 External links

- Raphael Finkel's "An operating system Vade Mecum"
- Mechanism and policy for HTC

4.2 Abstraction principle (computer programming)

In software engineering and programming language theory, the **abstraction principle** (or the **principle of abstraction**) is a basic dictum that aims to reduce duplication of information in a program (usually with emphasis on code duplication) whenever practical by making use of abstractions provided by the programming language or software libraries. The principle is sometimes stated as a recommendation to the programmer, but sometimes stated as requirement of the programming language, assuming it is self-understood why abstractions are desirable to use. The origins of the principle are uncertain; it has been reinvented a number of times, sometimes under a different name, with slight variations.

When read as recommendation to the programmer, the abstraction principle can be generalized as the "don't repeat yourself" principle, which recommends avoiding the duplication of information in general, and also avoiding the duplication of human effort involved in the software development process.

4.2.1 The principle

As a recommendation to the programmer, in its formulation by Benjamin C. Pierce in *Types and Programming Languages* (2002), the abstraction principle reads (emphasis in original):[1]

As a requirement of the programming language, in its formulation by David A. Schmidt in *The structure of typed programming languages* (1994), the abstraction principle reads:.[2]

4.2.2 History and variations

Under this very name, the abstraction principle appears into a long list of books. Here we give a necessarily incomplete list, together with the formulation if it is succinct:

- Alfred John Cole, Ronald Morrison (1982) *An introduction to programming with S-algol*: "[Abstraction] when applied to language design is to define all the semantically meaningful syntactic categories in the language and allow an abstraction over them”.[3]
- Bruce J. MacLennan (1983) *Principles of programming languages: design, evaluation, and implementation*: “Avoid requiring something to be stated more than once; factor out the recurring pattern”.[4]
- Jon Pearce (1998) *Programming and Meta-Programming in Scheme*: “Structure and function should be independent”.[5]

The principle plays a central role in design patterns in object-oriented programming, although most writings on that topic do not give a name to the principle. The influential book by the Gang of Four, states: “The focus here is *encapsulating the concept that varies*, a theme of many design patterns.” This statement has been rephrased by other authors as “Find what varies and encapsulate it.”[6]

In this century, the principle has been reinvented in extreme programming under the slogan “Once and Only Once”. The definition of this principle was rather succinct in its first appearance: “no duplicate code”.[7] It has later been elaborated as applicable to other issues in software development: “Automate every process that’s worth automating. If you find yourself performing a task many times, script it.”[8]

4.2.3 Implications

The abstraction principle is often stated in the context of some mechanism intended to facilitate abstraction. The basic mechanism of control abstraction is a function or subroutine. Data abstractions include various forms of type polymorphism. More elaborate mechanisms that may combine data and control abstractions include: abstract data types, including classes, polytypism etc. The quest for richer abstractions that allow less duplication in complex scenarios is one of the driving forces in programming language research and design.

Inexperienced programmers may be tempted to introduce too much abstraction in their program—abstraction that won't be used more than once. A complementary principle that emphasize this issue is "You Ain't Gonna Need It" and, more generally, the KISS principle.

Since code is usually subject to revisions, following the abstraction principle may entail refactoring of code. The effort of rewriting a piece of code generically needs to be amortized against the estimated future benefits of an abstraction. A rule of thumb governing this was devised by Martin Fowler, and popularized as the rule of three. It states that if a piece of code is copied more than once, i.e. it would end up having three or more copies, then it needs to be abstracted out.

4.2.4 Generalizations

"Don't repeat yourself", or the “DRY principle”, is a generalization developed in the context of multi-tier architectures, where related code is by necessity duplicated to some extent across tiers, usually in different languages. In practical terms, the recommendation here is to rely on automated tools, like code generators and data transformations to avoid repetition.

4.2.5 Hardware programming interfaces

In addition to optimizing code, a hierarchical/recursive meaning of Abstraction level in programming also refers to the interfaces between hardware communication layers, also called “abstraction levels” and “abstraction layers.” In this case, level of abstraction often is synonymous with interface. For example, in examining shellcode and the interface between higher and lower level languages, the level of abstraction changes from operating system commands (for example, in C) to register and circuit level calls and commands (for example, in assembly and binary). In the case of that example, the boundary or interface between the abstraction levels is the stack.[9]

4.2.6 References

[1] Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press. p. 339. ISBN 0-262-16209-1.

[2] David A. Schmidt, *The structure of typed programming languages*, MIT Press, 1994, ISBN 0-262-19349-3, p. 32

[3] Alfred John Cole, Ronald Morrison, *An introduction to programming with S-algol*, CUP Archive, 1982, ISBN 0-521-25001-3, p. 150

[4] Bruce J. MacLennan, *Principles of programming languages: design, evaluation, and implementation*, Holt, Rinehart, and Winston, 1983, p. 53

[5] Jon Pearce, *Programming and meta-programming in scheme*, Birkhäuser, 1998, ISBN 0-387-98320-1, p. 40

[6] Alan Shalloway, James Trott, *Design patterns explained: a new perspective on object-oriented design*, Addison-Wesley, 2002, ISBN 0-201-71594-5, p. 115

[7] Kent Beck, *Extreme programming explained: embrace change*, 2nd edition, Addison-Wesley, 2000, ISBN 0-201-61641-6, p. 61

[8] Chromatic, *Extreme programming pocket guide*, O'Reilly, 2003, ISBN 0-596-00485-0

[9] Koziol, *The Shellcoders Handbook”*, Wiley, 2004, p. 10, ISBN 0-7645-4468-3

4.3 Law of Demeter

The **Law of Demeter** (**LoD**) or **principle of least knowledge** is a design guideline for developing software, particularly object-oriented programs. In its general form, the LoD is a specific case of loose coupling. The guideline was proposed at Northeastern University towards the end of 1987, and can be succinctly summarized in each of the following ways:[1]

- Each unit should have only limited knowledge about other units: only units “closely” related to the current unit.
- Each unit should only talk to its friends; don't talk to strangers.
- Only talk to your immediate friends.

The fundamental notion is that a given object should assume as little as possible about the structure or properties of anything else (including its subcomponents), in accordance with the principle of "information hiding".

It is so named for its origin in the Demeter Project, an adaptive programming and aspect-oriented programming effort. The project was named in honor of Demeter, “distribution-mother” and the Greek goddess of agriculture, to signify a bottom-up philosophy of programming which is also embodied in the law itself.

4.3.1 In object-oriented programming

When applied to object-oriented programs, the Law of Demeter can be more precisely called the “Law of Demeter for Functions/Methods” (LoD-F). In this case, an object A can request a service (call a method) of an object instance B, but object A should not “reach through” object B to access yet another object, C, to request its services. Doing so would mean that object A implicitly requires greater knowledge of object B’s internal structure. Instead, B’s interface should be modified if necessary so it can directly serve object A’s request, propagating it to any relevant subcomponents. Alternatively, A might have a direct reference to object C and make the request directly to that. If the law is followed, only object B knows its own internal structure.

More formally, the Law of Demeter for functions requires that a method m of an object O may only invoke the methods of the following kinds of objects:[2]

1. O itself
2. m’s parameters
3. Any objects created/instantiated within m
4. O’s direct component objects
5. A global variable, accessible by O, in the scope of m

In particular, an object should avoid invoking methods of a member object returned by another method. For many modern object oriented languages that use a dot as field identifier, the law can be stated simply as “use only one dot”. That is, the code a.b.Method() breaks the law where a.Method() does not. As an analogy, when one wants a dog to walk, one does not command the dog’s legs to walk directly; instead one commands the dog which then commands its own legs.

4.3.2 Advantages

The advantage of following the Law of Demeter is that the resulting software tends to be more maintainable and adaptable. Since objects are less dependent on the internal structure of other objects, object containers can be changed without reworking their callers.

Basili et al.[3] published experimental results in 1996 suggesting that a lower *Response For a Class* (RFC, the number of methods potentially invoked in response to calling a method of that class) can reduce the probability of software bugs. Following the Law of Demeter can result in a lower RFC. However, the results also suggest that an increase in *Weighted Methods per Class* (WMC, the number of methods defined in each class) can increase the probability of software bugs. Following the Law of Demeter can also result in a higher WMC; see Disadvantages.

A multilayered architecture can be considered to be a systematic mechanism for implementing the Law of Demeter in a software system. In a layered architecture, code within each layer can only make calls to code within the layer and code within the next layer down. “Layer skipping” would violate the layered architecture.

4.3.3 Disadvantages

Although the LoD increases the adaptiveness of a software system, it may also result in having to write many wrapper methods to propagate calls to components; in some cases, this can add noticeable time and space overhead.[3][4][5]

At the method level, the LoD leads to narrow interfaces, giving access to only as much information as it needs to do its job, as each method needs to know about a small set of methods of closely related objects.[6] On the other hand, at the class level, the LoD leads to wide (i.e. enlarged) interfaces, because the LoD requires introducing many auxiliary methods instead of digging directly into the object structures. One solution to the problem of enlarged class interfaces is the aspect-oriented approach,[7] where the behavior of the method is specified as an aspect at a high level of abstraction. This is done by having an adaptive method that encapsulates the behaviour of an operation into a place, with which the scattering problem is solved. It also abstracts over the class structure that results in avoiding the tangling problem. The wide interfaces are managed through a language that specifies implementations. Both the traversal strategy and the adaptive visitor use only a minimal set of classes that participate in the operation, and the information about the connections between these classes is abstracted out.[4][7]

Since the LoD exemplifies a specific type of coupling, and does not specify a method of addressing this type of coupling, it is more suited as a metric for code smell as opposed to a methodology for building loosely coupled systems.

4.3.4 See also

- Single responsibility principle
- Principle of least astonishment

4.3.5 References

[1] Macedo, Emerson. "README.markdown: Demeter". GitHub. Retrieved 2012-07-05.

[2] Bock, David. "The Paperboy, The Wallet, and The Law Of Demeter" (PDF). College of Computer and Information Science, Northeastern University. p. 5. Retrieved 2012-07-05.

[3] Basili, Victor; Briand, L.; Melo, W. L. (October 1996). "A Validation of Object-Oriented Design Metrics as Quality Indicators". *IEEE Transactions on Software Engineering* **22** (10): 751–761. doi:10.1109/32.544352. As expected, the larger the WMC, the larger the probability of fault detection.

[4] Appleton, Brad. "Introducing Demeter and its Laws". Retrieved 6 July 2013. A side-effect of this is that if you conform to LoD, while it may quite increase the maintainability and "adaptiveness" of your software system, you also end up having to write lots of little wrapper methods to propagate methods calls to its components (which can add noticeable time and space overhead).

[5] "Tell, Don't Ask". The Pragmatic Programmers, LLC. Retrieved 6 July 2013. The disadvantage, of course, is that you end up writing many small wrapper methods that do very little but delegate container traversal and such. The cost tradeoff is between that inefficiency and higher class coupling.

[6] Lieberherr, K.; Holland, I.; Riel, A. (1988-09-25). "Object-Oriented Programming: An Objective Sense of Style" (PDF). OOPSLA '88 Proceedings. Archived from the original (PDF) on 1988-09-25. Retrieved 2012-07-05. Easier software maintenance, less coupling between your methods, better information hiding, narrower interfaces, methods which are easier to reuse, and easier correct.ness proofs using structural induction.

[7] Lieberherr, Karl; Orleans, Doug; Ovlinger, Johan (October 2001). "ASPECT-ORIENTED PROGRAMMING WITH ADAPTIVE METHODS" (PDF). COMMUNICATIONS OF THE ACM: 39–40. Retrieved 2012-07-05. An adaptive method encapsulates the behavior of an operation into one place, thus avoiding the scattering problem, but also abstracts over the class structure, thus avoiding the tangling problem as well.

4.3.6 Further reading

- Lieberherr, Karl; Holland, I. (September 1989). "Assuring good style for object-oriented programs". *IEEE Software* **6**: 38–48. doi:10.1109/52.35588.

- Lieberherr, Karl J. (1995). "Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns". Boston: PWS Publishing Company, International Thomson Publishing.
- Hunt, Andrew; Thomas, David (2002). *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley. pp. 140–141.
- Larman, Craig (2005). *Applying UML and Patterns* (3rd ed.). Prentice Hall. pp. 430–432. (from this book, "Law of Demeter" is also known as "Don't talk to strangers")
- McConnell, Steve (2004). *Code Complete* (2nd ed.). Microsoft Press. p. 150.
- Palermo, Jeffrey; Scheirman, Ben; Bogard, Jimmy (2009). *ASP.NET MVC in Action*. Manning Publications. p. 14.

4.3.7 External links

- Law of Demeter (LoD)
- "Object-Oriented Programming: An Objective Sense of Style" (OOPSLA '88 Proceedings) (PDF)
- The Paperboy, The Wallet,and The Law Of Demeter (PDF)
- Phil Haack: "The Law of Demeter is not a Dot Counting Exercise"
- Lieber: "Phil Holland's Law of Demeter"
- "Adaptive Object-Oriented Software, The Demeter Method"

4.4 Interface segregation principle

The **interface-segregation principle** (ISP) states that no client should be forced to depend on methods it does not use.[1] ISP splits interfaces which 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]

4.4.1 Importance in object-oriented design

Within object-oriented design, interfaces provide layers of abstraction that facilitate conceptual explanation of the code and create a barrier preventing dependencies.

According to many software experts who have signed the Manifesto for Software Craftsmanship, writing well-crafted and self-explanatory software is almost as important as writing working software.[4] Using interfaces to further describe the intent of the software is often a good idea.

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.

4.4.2 Origin

The ISP was first used and formulated by Robert C. Martin 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 called the Interface Segregation Principle today. 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 were all implemented by the Job class.

4.4.3 Typical violation

The Xerox case is an example of a clear violation (and resolution) of the Interface Segregation Principle, but not all violations are so clear cut.

A more commonly known example is the ATM Transaction example given in Agile Software Development: Principles, Patterns, and Practices [1] and in an article also written by Robert C. Martin specifically about the ISP.[5] This example is about an interface for the User Interface for an ATM, that 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.

4.4.4 See also

- SOLID - The “I” in SOLID stands for Interface segregation principle

4.4.5 References

[1] Martin, Robert (2002). Agile Software Development: Principles, Patterns and Practices. Pearson Education.

[2] Role Interface

[3] David Hayden, *Interface-Segregation Principle (ISP) - Principles of Object-Oriented Class Design*

[4] Manifesto of Software Craftsmanship

[5] Robert C. Martin,*The Interface Segregation Principle*, C++ Report, June 1996

4.4.6 External links

- *Principles Of OOD* – Description and links to detailed articles on SOLID.
- Object Oriented Design Quality Metrics: an analysis of dependencies Robert C. Martin, C++ Report, Sept/Oct 1995

4.5 Command–query separation

Command–query separation (**CQS**) is a principle of imperative computer programming. It was devised by Bertrand Meyer as part of his pioneering work on the Eiffel programming language.

It states that every method should either be a *command* that performs an action, or a *query* that returns data to the caller, but not both. In other words, *Asking a question should not change the answer*.[1] More formally, methods should return a value only if they are referentially transparent and hence possess no side effects.

4.5.1 Connection with design by contract

Command–query separation is particularly well suited to a design by contract (DbC) methodology, in which the design of a program is expressed as assertions embedded in the source code, describing the state of the program at certain critical times. In DbC, assertions are considered design annotations – not program logic – and as such, their execution should not affect the program state. CQS is beneficial to DbC because any value-returning method (any query) can be called by any assertion without fear of modifying program state.

In theoretical terms, this establishes a measure of sanity, whereby one can reason about a program's state without simultaneously modifying that state. In practical terms, CQS allows all assertion checks to be bypassed in a working system to improve its performance without inadvertently modifying its behaviour. CQS may also prevent the occurrence of certain kinds of heisenbugs.

4.5.2 Broader impact on software engineering

Even beyond the connection with design by contract, CQS is considered by its adherents to have a simplifying effect on a program, making its states (via queries) and state changes (via commands) more comprehensible.

CQS is well-suited to the object-oriented methodology, but can also be applied outside of object-oriented programming. Since the separation of side effects and return values is not inherently object-oriented, CQS can be profitably applied to any programming paradigm that requires reasoning about side effects.

4.5.3 Command Query Responsibility Segregation

Command Query Responsibility Segregation (CQRS) applies the CQS principle by using separate *Query* and *Command* objects to *retrieve* and *modify* data, respectively.[2][3]

4.5.4 Drawbacks

CQS can make it more difficult to implement re-entrant and multi-threaded software correctly. This usually occurs when a non-thread-safe pattern is used to implement the command–query separation.

Here is a simple example of a pattern that breaks CQS principles but is useful for multi-threaded software. It breaks CQS principles because the function both mutates state and returns it:

private int x; public int increment_and_return_x() { lock x; // by some mechanism x = x + 1; int x_copy = x; unlock x; // by some mechanism return x_copy; }

Here is a CQS-compliant pattern. However, it is safely usable only in single-threaded applications. In a multi-threaded program, there is a race condition in the caller, between where increment() and value() would be called:

private int x; public int value() { return x; } void increment() { x = x + 1; }

Even in single-threaded programs, it is sometimes arguably significantly more convenient to have a method that is a combined query and command. Martin Fowler cites the pop() method of a stack as an example.[4]

4.5.5 See also

- Fluent interface

4.5.6 References

[1] Meyer, Bertrand. "Eiffel: a language for software engineering" (PDF). p. 22. Retrieved 16 December 2014.

[2] Young, Greg. "CQRS Documents" (PDF). Retrieved 2012-12-28.

[3] Fowler, Martin. "CQRS". Retrieved 2011-07-14.

[4] Fowler, Martin. "CommandQuerySeparation". Retrieved 5 December 2005.

4.5.7 Further reading

- Meyer, Bertrand (1988). *Object-oriented Software Construction*. Prentice Hall. ISBN 0-13-629049-3.

4.5.8 External links

- Explanation on Martin Fowler's Bliki
- CQRS, Task Based UIs, Event Sourcing agh! by Greg Young
- Clarified CQRS by Udi Dahan
- CQRS Journey by Microsoft patterns & practices
- DDD/CQRS/Event Sourcing List
- The CQRS Frequently Asked Questions
- CQRS - a new architecture precept based on segregation of commands and queries
- Little CQRS book a collection of blog posts written by Mark Nijhof about his Fohjin CQRS example project from 2009
- CQRS Starter kit (.NET)
- Axon CQRS Framework for Java
- CS-CQRS - A C# CQRS Framework
- Commercially sponsors of CS-CQRS
- A blog focused on practical CQRS (.NET)

4.6 Separation of concerns

In computer science, **separation of concerns** (**SoC**) is a design principle for separating a computer program into distinct sections, such that each section addresses a separate concern. A concern is a set of information that affects the code of a computer program. A concern can be as general as the details of the hardware the code is being optimized for, or as specific as the name of a class to instantiate. A program that embodies SoC well is called a modular[1] program. Modularity, and hence separation of concerns, is achieved by encapsulating information inside a section of code that has a well-defined interface. Encapsulation is a means of information hiding.[2] Layered designs in information systems are another embodiment of separation of concerns (e.g., presentation layer, business logic layer, data access layer, persistence layer).[3]

The value of separation of concerns is simplifying development and maintenance of computer programs. When concerns are well-separated, individual sections can be reused, as well as developed and updated independently. Of special value is the ability to later improve or modify one section of code without having to know the details of other sections, and without having to make corresponding changes to those sections.

4.6.1 Implementation

The mechanisms for modular or object-oriented programming that are provided by a programming language are mechanisms that allow developers to provide SoC.[4] For example, object-oriented programming languages such as C#, C++, Delphi, and Java can separate concerns into objects, and architectural design patterns like MVC or MVP can separate content from presentation and the data-processing (model) from content. Service-oriented design can separate concerns into services. Procedural programming languages such as C and Pascal can separate concerns into procedures or functions. Aspect-oriented programming languages can separate concerns into aspects and objects.

Separation of concerns is an important design principle in many other areas as well, such as urban planning, architecture and information design.[5] The goal is to more effectively understand, design, and manage complex interdependent systems, so that functions can be reused, optimized independently of other functions, and insulated from the potential failure of other functions.

Common examples include separating a space into rooms, so that activity in one room does not affect people in other rooms, and keeping the stove on one circuit and the lights on another, so that overload by the stove does not turn the lights off. The example with rooms shows encapsulation, where information inside one room, such as how messy it is, is not available to the other rooms, except through the interface, which is the door. The example with circuits demonstrates that activity inside one module, which is a circuit with consumers of electricity attached, does not affect activity in a different module, so each module is not concerned with what happens in the other.

4.6.2 Origin

The term *separation of concerns* was probably coined by Edsger W. Dijkstra in his 1974 paper “On the role of scientific thought”.[6]

> Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one’s subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained —on the contrary!— by tackling these various aspects simultaneously. It is what I sometimes have called **“the separation of concerns”**, which, even if not perfectly possible, is yet the only available technique for effective ordering of one’s thoughts, that I know of. This is what I mean by “focusing one’s attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect’s point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.

Fifteen years later, it was evident the term **Separation of Concerns** was becoming an accepted idea. In 1989, Chris Reade wrote a book titled “Elements of Functional Programming”[7] that describes separation of concerns:

> The programmer is having to do several things at the same time, namely,
> 1. describe what is to be computed;
> 2. organise the computation sequencing into small steps;
> 3. organise memory management during the computation.

Reade continues to say,

> Ideally, the programmer should be able to concentrate on the first of the three tasks (describing what is to be computed) without being distracted by the other two, more administrative, tasks. Clearly, administration is important, but by separating it from the main task we are likely to get more reliable results and we can ease the programming problem by automating much of the administration.
>
> The separation of concerns has other advantages as well. For example, program proving becomes much more feasible when details of sequencing and memory management are absent from the program. Furthermore, descriptions of what is to be computed should be free of such detailed step-by-step descriptions of how to do it, if they are to be evaluated with different machine architectures. Sequences of small changes to a data object held in a store may be an inappropriate description of how to compute something when a highly parallel machine is being used with thousands of processors distributed throughout the machine and local rather than global storage facilities.
>
> Automating the administrative aspects means that the language implementor has to deal with them, but he/she has far more opportunity to make use of very different computation mechanisms with different machine architectures.

4.6.3 Examples

Internet protocol stack

Separation of concerns is crucial to the design of the Internet. In the Internet Protocol Suite, great efforts have been made to separate concerns into well-defined layers. This allows protocol designers to focus on the concerns in one layer, and ignore the other layers. The Application Layer protocol SMTP, for example, is concerned about all the details of conducting an email session over a reliable transport service (usually TCP), but not in the least concerned about how the transport service makes that service reliable. Similarly, TCP is not concerned about the routing of data packets, which is handled at the Internet Layer.

HTML, CSS, JavaScript

HyperText Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript (JS) are complementary languages used in the development of webpages and websites. HTML is mainly used for organization of webpage content, CSS is used for definition of content presentation style, and JS defines how the content interacts and behaves with the user. Historically, this was not the case: prior to the introduction of CSS, HTML performed both duties of defining semantics and style.

Subject-oriented programming

Subject-oriented programming allows separate concerns to be addressed as separate software constructs, each on an equal footing with the others. Each concern provides its own class-structure into which the objects in common are organized, and contributes state and methods to the composite result where they cut across one another. Correspondence rules describe how the classes and methods in the various concerns are related to each other at points where they interact, allowing composite behavior for a method to be derived from several concerns. Multi-dimensional Separation of Concerns allows the analysis and composition of concerns to be manipulated as a multi-dimensional "matrix" in which each concern provides a dimension in which different points of choice are enumerated, with the cells of the matrix occupied by the appropriate software artifacts.

Aspect-oriented programming

Aspect-oriented programming allows cross-cutting concerns to be addressed as secondary concerns. For example, most programs require some form of security and logging. Security and logging are often secondary concerns, whereas the primary concern is often on accomplishing business goals. However, when designing a program, its security must be built into the design from the beginning instead of being treated as a secondary concern. Applying security afterwards often results in an insufficient security model that leaves too many gaps for future attacks.[8]

Software build automation

Most project organization tasks are seen as secondary tasks. For example, build automation is an approach to automating the process of compiling source code into binary code. The primary goals in build automation are reducing the risk of human error and saving time.

Levels of analysis in artificial intelligence

In cognitive science and artificial intelligence, it is common to refer to David Marr's levels of analysis. At any given time, a researcher may be focusing on (1) what some aspect of intelligence needs to compute, (2) what algorithm it employs, or (3) how that algorithm is implemented in hardware. This separation of concerns is similar to the interface/implementation distinction in software and hardware engineering.

Normalized Systems

In Normalized Systems separation of concerns is one of the four guiding principles. Adhering to this principle is one of the tools that helps reduce the combinatorial effects that, over time, get introduced in software that is being maintained. In Normalized Systems separation of concerns is actively supported by the tools.

SoC via partial classes

Separation of concerns can be implemented and enforced via partial classes.[9]

SoC via partial classes in C# The following Bear class, written in C#, has different aspects implemented in different parts.

Bear_Hunting.cs

```
public partial class Bear { private IEdible Hunt() { // returns food... } }
```

Bear_Eating.cs

```
public partial class Bear { private int Eat(IEdible food) { return food.Nutrition.Value; } }
```

Bear_Hunger.cs

```
public partial class Bear { private int hunger; public void MonitorHunger() { // Here we can refer to members of the other partial definitions if (hunger > 50) hunger -= this.Eat(this.Hunt()); } }
```

For example, if we want to compile a version without support for hunger management (it could be a feature that costs extra for your customers), we simply remove the partial declaration in Bear_Hunger.cs.

Now, if a program also supported hunger management in other classes all those partial class definitions could go in a separate 'Hunger' directory. This is what is usually called multidimensional separation of concerns, and it helps programmers update the code and add new features, even the first time anyone started working with this code.

SoC via partial classes in Ruby

bear_hunting.rb

```
class Bear def hunt # TODO: return some food end end
```

bear_eating.rb

```
class Bear def eat( food ) raise "#{food} is not edible!" unless food.respond_to? :nutrition_value food.nutrition_value end end
```

bear_hunger.rb

```
class Bear attr_accessor :hunger def monitor_hunger if @hunger > 50 then @hunger -= self.eat( self.hunt ) end end end
```

4.6.4 See also

- Abstraction principle (programming)
- Aspect-oriented software development
- Concern (computer science)
- Core concern
- Coupling (computer science)
- Cross-cutting concern
- Holism
- Modular design
- Modular programming
- Orthogonality#Computer science
- Separation of presentation and content
- Single responsibility principle

4.6.5 References

[1] Laplante, Phillip (2007). *What Every Engineer Should Know About Software Engineering*. CRC Press. ISBN 0849372283.

[2] Mitchell, Dr. R. J. (1990). *Managing Complexity in Software Engineering*. IEE. p. 5. ISBN 0863411711.

[3] *Microsoft Application Architecture Guide*. Microsoft Press. 2009. ISBN 0-7356-2710-X.

[4] Painter, Robert Richard. “Software Plans: Multi-Dimensional Fine-Grained Separation of Concerns”. Penn State. CiteSeerX: 10.1.1.110.9227.

[5] Garofalo, Raffaele (2011). *Building Enterprise Applications with Windows Presentation Foundation and the Model View ViewModel Pattern*. Microsoft Press. p. 18. ISBN 0735650926.

[6] Dijkstra, Edsger W (1982). “On the role of scientific thought”. *Selected writings on Computing: A Personal Perspective*. New York, NY, USA: Springer-Verlag. pp. 60–66. ISBN 0-387-90652-5.

[7] Reade, Chris (1989). *Elements of Functional Programming*. Boston, MA, USA: Addison-Wesley Longman. ISBN 0-201-12915-9.

[8] Jess Nielsen (June 2006). “Building Secure Applications” (PDF). Retrieved 2012-02-08.

[9] Tiago Dias (October 2006). “Hyper/Net: MDSoC Support for .NET” (PDF). *DSOA 2006*. Retrieved 2007-09-25.

4.6.6 External links

- The Art of Separation of Concerns
- Multi-Dimensional Separation of Concerns
- TAOSAD
- Tutorial and Workshop on Aspect-Oriented Programming and Separation of Concerns

4.7 Uniform access principle

The **Uniform Access Principle** was put forth by Bertrand Meyer (originally in *Object-Oriented Software Construction*). It states “All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation”.[1] This principle applies generally to the syntax of object-oriented programming languages. In simpler form, it states that there should be no difference between working with an attribute, precomputed property, or method/query.

While most examples focus on the “read” aspect of the principle, Meyer shows that the “write” implications of the principle are harder to deal with in his monthly column on the Eiffel programming language official website.[2]

4.7.1 Explanation

The problem being addressed by Meyer involves the maintenance of large software projects or software libraries. Sometimes when developing or maintaining software it is necessary, after much code is in place, to change a class or object in a way that transforms what was simply an attribute access into a method call. Programming languages often use different syntax for attribute access and invoking a method, (e.g. obj.something versus obj.something()). The syntax change would require, in popular programming languages of the day, changing the source code in all the places where the attribute was used. This might require changing source code in many different locations throughout a very large volume of source code. Or worse, if the change is in an object library used by hundreds of customers, each of those customers would have to find and change all the places the attribute was used in their own code and recompile their programs.

Going the reverse way (from method to simple attribute) really wasn't a problem, as one can always just keep the function and have it simply return the attribute value.

Meyer recognized the need for software developers to write code in such a way as to minimize or eliminate cascading changes in code that result from changes which convert an object attribute to a method call or vice versa. For this he developed the Uniform Access Principle.

Many programming languages do not strictly support the UAP but do support forms of it. Properties, which are provided in a number of programming languages, address the problem Meyer was addressing with his UAP in a different way. Instead of providing a single uniform notation, properties provide a way to invoke a method of an object while using the same notation as is used for attribute access. The separate method invocation syntax is still available.

4.7.2 UAP Example

If the language uses the method invocation syntax it may look something like this.

//Assume print displays the variable passed to it, with or without parens //Set Foo's attribute 'bar' to value 5. Foo.bar(5) print Foo.bar()

When executed, should display :

5

Whether or not Foo.bar(5) invokes a function or simply sets an attribute is hidden from the caller. Likewise whether Foo.bar() simply retrieves the value of the attribute, or invokes a function to compute the value returned, is an implementation detail hidden from the caller.

If the language uses the attribute syntax the syntax may look like this.

Foo.bar = 5 print Foo.bar

Again, whether or not a method is invoked, or the value is simply assigned to an attribute is hidden from the calling method.

4.7.3 Problems

However, UAP itself can lead to problems, if used in places where the differences between access methods are *not* negligible, such as when the returned value is expensive to compute or will trigger cache operations.[1] It might not matter in principle to the client how the value of 42 was obtained, but if computing it requires running a planet-sized computer for 7.5 million years, the client ought to know what to expect.

4.7.4 Language Examples

Ruby

Consider the following

y = Egg.new(“Green”) y.color = “White” puts y.color

Now the Egg class could be defined as follows

class Egg attr_accessor :color def initialize(color) @color = color end end

The above initial code segment would work fine with the Egg being defined as such. The Egg class could also be defined as below, where color is instead a method. The calling code would still work, unchanged if Egg were to be defined as follows.

class Egg def initialize(color) @rgb_color = to_rgb(color) end def color to_color_name(@rgb_color) end def color=(color) @rgb_color = to_rgb(color) end private def to_rgb(color_name) ..... end def to_color_name(color) .... end end

Note how even though color looks like an attribute in one case and a pair of methods in the next, the interface to the class remains the same. The person maintaining the Egg class can switch from one form to the other without fear of breaking any caller's code. Ruby follows the revised UAP, the attr_accessor :color only acts as syntactic sugar for generating accessor/setter methods for color. There is no way in Ruby to retrieve an instance variable from an object without calling a method on it.

Strictly speaking, Ruby does not follow Meyer's original UAP in that the syntax for accessing an attribute is different from the syntax for invoking a method. But here, the access for an attribute will always actually be through a function which is often automatically generated. So in essence, either type of access invokes a function and the language does follow Meyer's revised Uniform Access Principle.

Python

Python properties may be used to allow a method to be invoked with the same syntax as accessing an attribute. Whereas Meyer's UAP would have a single notation for both attribute access and method invocation (method invocation syntax), a language with support for properties still supports separate notations for attribute and method access. Properties allow the attribute notation to be used, but to hide the fact that a method is being invoked instead of simply retrieving or setting a value.

In the strict sense, Python does NOT follow the UAP because there is a syntax difference between normal method invocations and attribute access.

In Python, we may have code that access an object as follows

>>> egg = Egg(4, "White") >>> egg.color = "Green" >>> print egg.weight, egg.color, egg.quack() 4 Green quack

A Egg object could be defined such that weight and color are simple attributes as in the following

class Egg(object): def __init__(self, weight, color): self.weight = weight self.color = color def quack(self): return "quack"

Or the Egg object could use properties, and invoke methods instead

class Egg(object): def __init__(self, weight, color): self.__weight = toGrams(weight) self.__color = toRGB(color) def setColor(self, colorname): self.__color = toRGB(colorname) def getColor(self): return toColorName(self.__color) color = property(getColor, setColor, doc="Color of the Egg") def setWeight(self, weightOz); self.__weight = 29.3*weightOz def getWeight(self): return self.__weight/29.3; weight = property(setWeight, getWeight, doc="Weight in Ounces") def quack(self): return "quack"

Regardless of which way Egg is defined, the calling code can remain the same. The implementation of Egg can switch from one form to the other without affecting code that uses the Egg class. Languages which implement the UAP have this property as well.

C++

C++ has neither the UAP nor properties, when an object is changed such that an attribute (color) becomes a pair of functions (getA, setA). Any place in that uses an instance of the object and either sets or gets the attribute value (x = obj.color or obj.color= x) must be changed to invoke one of the functions. (x = obj.getColor() or obj.setColor(x)). Using templates and operator overloading, it is possible to fake properties, but this is more complex than in languages which directly support properties. This complicates maintenance of C++ programs. Distributed libraries of C++ objects must be careful about how they provide access to member data.

4.7.5 References

[1] "The UniformAccessPrinciple". *c2 wiki*. Retrieved 6 August 2013.

[2] Meyer, Bertrand. "EiffelWorld Column: Business plus pleasure". Retrieved 6 August 2013.

4.8 Loose coupling

For other uses, see Loose coupling (disambiguation).

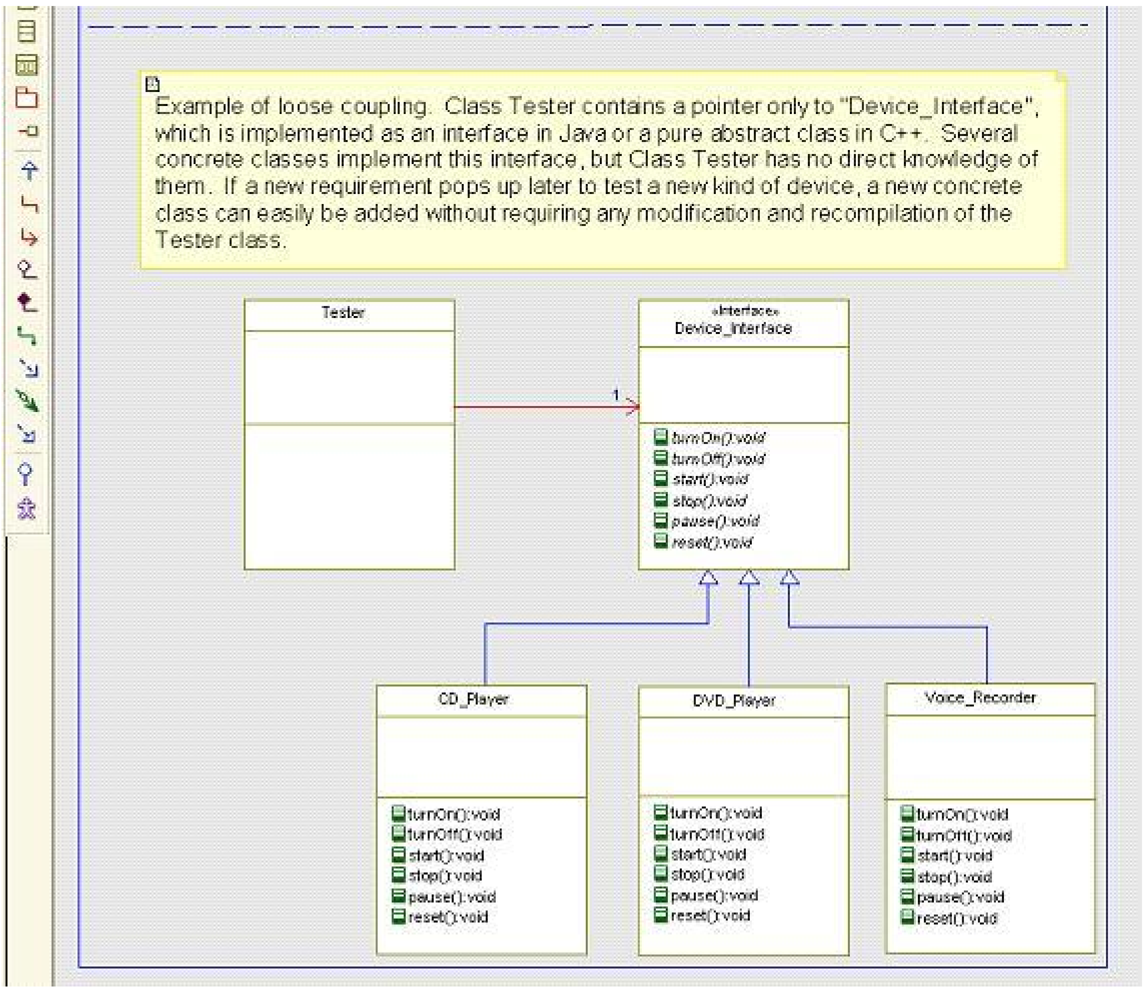
In computing and systems design a **loosely coupled** system is one in which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components. Sub-areas include the coupling of classes, interfaces, data, and services.

4.8.1 In computing

Coupling refers to the degree of direct knowledge that one component has of another. Loose coupling in computing is interpreted as encapsulation vs. non-encapsulation.

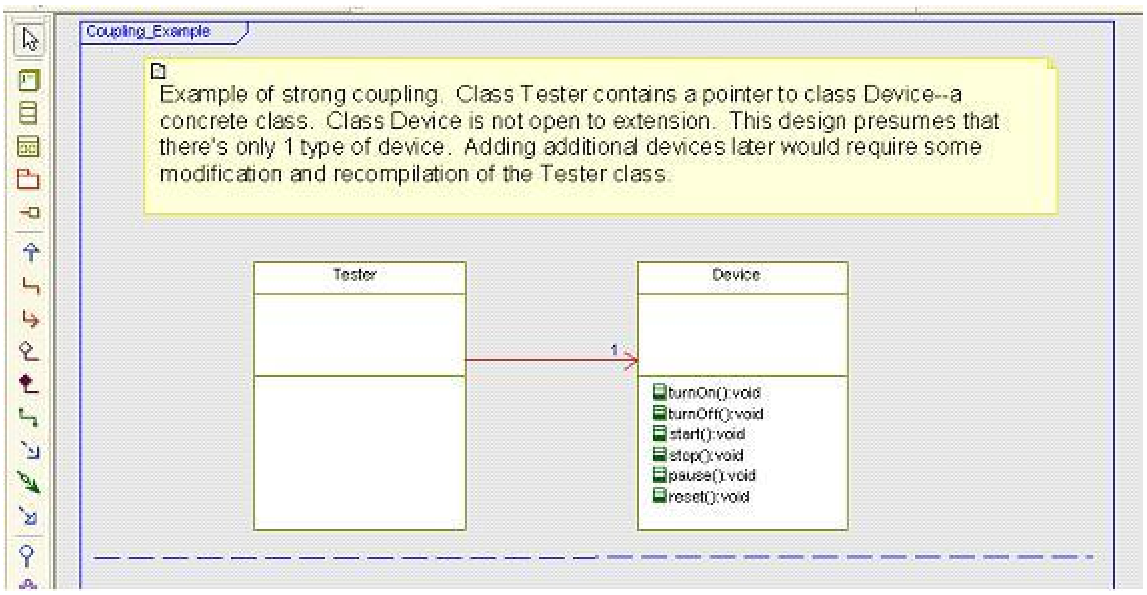
An example of tight coupling occurs when a dependent class contains a pointer directly to a concrete class which provides the required behavior. The dependency cannot be substituted, or its "signature" changed, without requiring a change to the dependent class. Loose coupling occurs when the dependent class contains a pointer only to an interface, which can then be implemented by one or many concrete classes. The dependent class's dependency is to a "contract" specified by the interface; a defined list of methods and/or properties that implementing classes must provide. Any class that implements the interface can thus satisfy the dependency of a dependent class without having to change the class. This allows for extensibility in software design; a new class implementing an interface can be written to replace a current dependency in some or all situations, without requiring a change to the dependent class; the new and old classes can be interchanged freely. Strong coupling does not allow this.

This is a UML diagram illustrating an example of *loose* coupling between a dependent class and a set of concrete classes, which provide the required behavior:



Example of loose coupling. Class Tester contains a pointer only to "Device_Interface", which is implemented as an interface in Java or a pure abstract class in C++. Several concrete classes implement this interface, but Class Tester has no direct knowledge of them. If a new requirement pops up later to test a new kind of device, a new concrete class can easily be added without requiring any modification and recompilation of the Tester class.

For comparison, this diagram illustrates the alternative design with *strong* coupling between the dependent class and a provider:



Example of strong coupling. Class Tester contains a pointer to class Device--a concrete class. Class Device is not open to extension. This design presumes that there's only 1 type of device. Adding additional devices later would require some modification and recompilation of the Tester class.

Other forms of loose coupling

Computer programming languages having notions of either functions as the core module (see Functional programming) or functions as objects provide excellent examples of loosely coupled programming. Functional languages have patterns of Continuations, Closure, or generators. See Clojure and Lisp as examples of function programming languages. Object oriented languages like Smalltalk and Ruby have code blocks, while Eiffel has agents. The basic idea is to objectify (encapsulate as an object) a function independent of any other enclosing concept (e.g. decoupling an object function from any direct knowledge of the enclosing object). See First-class function for further insight into functions as objects, which qualifies as one form of first-class function.

So, for example, in an object oriented language, when a function of an object is referenced as an object (freeing it from having any knowledge of its enclosing host object) the new function object can be passed, stored, and called at a later time. Recipient objects (to whom these functional objects are given) can safely execute (call) the contained function at their own convenience without any direct knowledge of the enclosing host object. In this way, a program can execute chains or groups of functional objects, while safely decoupled from having any direct reference to the enclosing host object.

Phone numbers are an excellent analog and can easily illustrate the degree of this decoupling.

For example: Some entity provides another with a phone number to call to get a particular job done. When the number is called, the calling entity is effectively saying, “Please do this job for me.” The decoupling or loose coupling is immediately apparent. The entity receiving the number to call may have no knowledge of where the number came from (e.g. a reference to the supplier of the number). On the other side, the caller is decoupled from specific knowledge of who they are calling, where they are, and knowing how the receiver of the call operates internally.

Carrying the example a step further, the caller might say to the receiver of the call, “Please do this job for me. Call me back at this number when you are finished.” The `number' being offered to the receiver is referred to as a “Call-back”. Again, the loose coupling or decoupled nature of this functional object is apparent. The receiver of the call-back is unaware of what or who is being called. It only knows that it can make the call and decides for itself when to call. In reality, the call-back may not even be to the one who provided the call-back in the first place. This level of indirection is what makes function objects an excellent technology for achieving loosely coupled programs.

Code Example

Measuring data element coupling

The degree of the loose coupling can be measured by noting the number of changes in data elements that could occur in the sending or receiving systems and determining if the computers would still continue communicating correctly. These changes include items such as:

1. adding new data elements to messages
2. changing the order of data elements
3. changing the names of data elements
4. changing the structures of data elements
5. omitting data elements

Methods for decreasing coupling

Loose coupling of interfaces can be enhanced by publishing data in a standard format (such as XML or JSON).

Loose coupling between program components can be enhanced by using standard data types in parameters. Passing customized data types or objects requires both components to have knowledge of the custom data definition.

Loose coupling of services can be enhanced by reducing the information passed into a service to the key data. For example, a service that sends a letter is most reusable when just the customer identifier is passed and the customer address is obtained within the service. This decouples services because services do not need to be called in a specific order (e.g. GetCustomerAddress, SendLetter)

Advantages

Components in a loosely coupled system can be replaced with alternative implementations that provide the same services.

Components in a loosely coupled system are less constrained to the same platform, language, operating system, or build environment.

Disadvantages

If systems are de-coupled in time using Message-oriented middleware, it is difficult to also provide transactional integrity. Data replication across different systems provides loose coupling (in availability), but creates issues in maintaining synchronization.

4.8.2 In systems design

Loose coupling in broader systems design is achieved by the use of transactions, queues, and standards.

4.8.3 See also

- Coupling (computer science)
- Cohesion (computer science)
- Connascence (computer programming)
- XML
- Web Services
- Design pattern (computer science)
- ISO/IEC 11179 - metadata registry specification
- Data element
- Enterprise service bus
- Enterprise Messaging System

- Space-based architecture (SBA)
- Tightly Coupled Systems
- Interface (computer programming)
- UFCS

4.8.4 References

- *Loosely Coupled: The Missing Pieces of Web Services* by Doug Kaye
- *Service-Oriented Architecture: A field Guide to Integrating XML and Web Services* by Thomas Erl

4.8.5 External links

- The Joy of Flex (2005) by John Hagel III and John Seely Brown
- About SOA and loose coupling: How EDA extends SOA and why it is important Jack van Hoof

Chapter 5

Software Metric

5.1 Software metric

A **software metric** is a standard of measure of a degree to which a software system or process possesses some property. Even if a metric is not a measurement (metrics are functions, while measurements are the numbers obtained by the application of metrics), often the two terms are used as synonymous. Since quantitative measurements are essential in all sciences, there is a continuous effort by computer science practitioners and theoreticians to bring similar approaches to software development. The goal is obtaining objective, reproducible and quantifiable measurements, which may have numerous valuable applications in schedule and budget planning, cost estimation, quality assurance testing, software debugging, software performance optimization, and optimal personnel task assignments.

5.1.1 Common software measurements

Common software measurements include:

- Balanced scorecard
- Bugs per line of code
- Code coverage
- Cohesion
- Comment density[1]
- Connascent software components
- Coupling
- Cyclomatic complexity (McCabe's complexity)
- DSQI (design structure quality index)
- Function Points and Automated Function Points, an Object Management Group standard[2]
- Halstead Complexity
- Instruction path length
- Maintainability index
- Number of classes and interfaces
- Number of lines of code
- Number of lines of customer requirements

- Program execution time
- Program load time
- Program size (binary)
- Robert Cecil Martin's software package metrics
- Weighted Micro Function Points
- CISQ automated quality characteristics measures

5.1.2 Limitations

As software development is a complex process, with high variance on both methodologies and objectives, it is difficult to define or measure software qualities and quantities and to determine a valid and concurrent measurement metric, especially when making such a prediction prior to the detail design. Another source of difficulty and debate is in determining which metrics matter, and what they mean.[3][4] The practical utility of software measurements has therefore been limited to the following domains:

- Scheduling
- Software sizing
- Programming complexity
- Software development effort estimation
- Software quality

A specific measurement may target one or more of the above aspects, or the balance between them, for example as an indicator of team motivation or project performance.

5.1.3 Acceptance and public opinion

Some software development practitioners point out that simplistic measurements can cause more harm than good.[5] Others have noted that metrics have become an integral part of the software development process.[3] Impact of measurement on programmers psychology have raised concerns for harmful effects to performance due to stress, performance anxiety, and attempts to cheat the metrics, while others find it to have positive impact on developers value towards their own work, and prevent them being undervalued.[6] Some argue that the definition of many measurement methodologies are imprecise, and consequently it is often unclear how tools for computing them arrive at a particular result,[7] while others argue that imperfect quantification is better than none ("You can't control what you can't measure.").[8] Evidence shows that software metrics are being widely used by government agencies, the US military, NASA,[9] IT consultants, academic institutions,[10] and commercial and academic development estimation software.

5.1.4 See also

- Goal Question-Metric
- Software crisis
- Software engineering
- Software package metrics
- Orthogonal Defect Classification
- List of tools for static code analysis

5.1.5 References

[1] "Descriptive Information (DI) Metric Thresholds". *Land Software Engineering Centre*. Retrieved 19 October 2010.

[2] "OMG Adopts Automated Function Point Specification". Omg.org. 2013-01-17. Retrieved 2013-05-19.

[3] Binstock, Andrew. "Integration Watch: Using metrics effectively". *SD Times*. BZ Media. Retrieved 19 October 2010.

[4] Kolawa, Adam. "When, Why, and How: Code Analysis". *The Code Project*. Retrieved 19 October 2010.

[5] Kaner, Dr. Cem, *Software Engineer Metrics: What do they measure and how do we know?*, CiteSeerX: 10.1.1.1.2542

[6] "ProjectCodeMeter (2010) "ProjectCodeMeter Users Manual" page 65" (PDF). Retrieved 2013-05-19.

[7] Lincke, Rüdiger; Lundberg, Jonas; Löwe, Welf (2008), "Comparing software metrics tools" (PDF), *International Symposium on Software Testing and Analysis 2008*, pp. 131–142

[8] DeMarco, Tom. *Controlling Software Projects: Management, Measurement and Estimation*. ISBN 0-13-171711-1.

[9] "NASA Metrics Planning and Reporting Working Group (MPARWG)". Earthdata.nasa.gov. Retrieved 2013-05-19.

[10] "USC Center for Systems and Software Engineering". Sunset.usc.edu. Retrieved 2013-05-19.

5.1.6 External links

- Definitions of software metrics in .NET
- Software Metrics
- Software Engineering Metrics: What do they measure and how do we know

Chapter 6

Text and image sources, contributors, and licenses

6.1 Text

6.2 Images

6.3 Content license