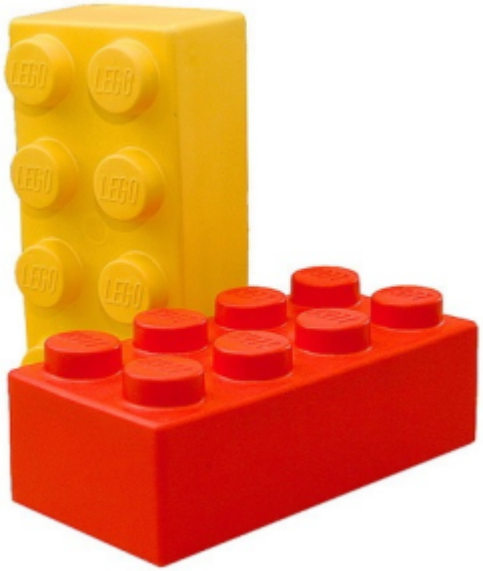
Effective Software Design

Doing the right thing.

Separation of Concerns

Posted on February 5, 2012



The most important principle in Software Engineering is the Separation of Concerns (SoC): The idea that a software system must be decomposed into parts that overlap in functionality as little as possible. It is so central that it appears in many different forms in the evolution of all methodologies, programming languages and best practices.

Dijkstra mentions it in 1974: *"**separation of concerns** ... even if not perfectly possible is yet the only available technique for effective ordering of one's thoughts"*. Information Hiding, defined by Parnas in 1972, focuses on reducing the dependency between modules through the definition of clear interfaces. A further improvement was Abstract Data Types (ADT), by Liskov in 1974, which integrated data and functions in a single definition.

In the case of Object Oriented Programming (OOP), encapsulation and inheritance proved to be essential mechanisms to support new levels of modularity. Design-by-Contract, proposed by Meyer in 1986, provides guidelines of how to improve interfaces using pre-conditions and post-conditions. Finally, the separation of cross-cutting concerns is the most important motivation for the proponents of Aspect Oriented Programming (AOP).

Since the first software systems were implemented, it was understood that it was important for them to be modular. It is necessary to follow a methodology when decomposing a system into modules and this is generally done by focusing on the software quality metrics of coupling and cohesion, originally defined by Constantine:

Coupling: The degree of dependency between two modules. We always want low coupling.

Cohesion: The measure of how strongly-related is the set of functions performed by a module. We always want high cohesion.

All methodologies try to reduce coupling and increase cohesion. Information Hiding reduces coupling by isolating the details of the implementation of state. ADTs reduce coupling by defining clear and abstract interfaces. An ADT that specifies in a single definition the set of functions that can be executed on a type is certainly more cohesive than a global data structure that is modified by external functions.

OOP adds another step in the reduction of coupling with the enforcement of encapsulation and the introduction of dynamic binding and polymorphism. Inheritance allows us to increase cohesion by defining hierarchies based on generalization and specialization, in which we can separate the functionality that belongs to the superclass from its subclasses. AOP provides a solution for the problem of cross-cutting concerns, so that both the aspects and the affected methods may become more cohesive.

There are many benefits that software developers expect to obtain when making a system more modular, reducing coupling and increasing cohesion:

Maintainability: A measure of how easy it is to maintain the system. As a consequence of low coupling, there is a reduced probability that a change in one module will be propagated to other modules. As a consequence of high

cohesion, there is an increased probability that a change in the system requirements will affect only a small number of modules.

Extensibility: A measure of how easily the system can be extended with new functionality. As a consequence of low coupling, it should be easier to introduce new modules, for example a new implementation for an existing interface. As a consequence of high cohesion, it should be easier to implement new modules without being concerned with aspects that are not directly related to their functionality.

Reusability: A measure of how easy it is to reuse a module in a different system. As a consequence of low coupling, it should be easier to reuse a module that was implemented in the past for a previous system, because that module should be less dependent on the rest of the system. Accordingly, it should be easier to reuse the modules of the current system in new future systems. As a consequence of high cohesion, the functionality provided by a module should be well-defined and complete, making it more useful as a reusable component.

As software developers, after we recognize the importance of SoC, we need to apply this principle in at least two ways: Understanding the power of our programming language tools and patterns, and learning how to evaluate and compare different designs in terms of coupling and cohesion.

Tools: For each mechanism in the programming languages we use, we should understand how it can be applied to reduce coupling and increase cohesion. For example: How encapsulation, dynamic binding, polymorphism and generic types can be used to separate concerns? Similarly, for each Design Pattern, we can analyze how it helps to make a system more modular.

Designs: When evaluating and comparing our own design alternatives, it is always useful to think in terms of coupling and cohesion. Sometimes a design seems to be more complex than others, but this may be a consequence of a better separation of concerns, with fewer dependencies between modules through the definition of additional

layers. Another design may at first appear to have too many classes, but this may be an advantage if each class becomes more cohesive.

Producing effective software designs requires lots of experience, but principles such as the Separation of Concerns are essential to perform a great work. So reduce coupling, increase cohesion and good luck!

Related

On Information Hiding and Encapsulation
In "OOD"

Do SOLID design principles make code slow?
In "Agile"

Antifragile Software Design: Abstraction and the Barbell Strategy
In "Antifragility"

About Hayim Makabee

Veteran software developer, enthusiastic programmer, author of a book on Object-Oriented Programming, co-founder and CEO at KashKlik, an innovative Influencer Marketing platform.

View all posts by Hayim Makabee →

This entry was posted in AOP, Design Patterns, OOD, OOP, Software Reuse and tagged AOP, Design Patterns, OOD, OOP, Software Reuse. Bookmark the permalink.

24 Responses to *Separation of Concerns*

Putcha V. Narasimham *says:*
February 5, 2012 at 4:58 pm

Very good and comprehensive introduction to Effective Software Design. I too have come to know this set of concepts / principles in different forms.

While studying General Systems Theory and Systems Thinking (particularly of Ackoff Russell), I felt that the holistic approach is either missing or not well utilized in SSAD / OOAD. There is too much of emphasis on partitioning, division, analysis which cannot be applied to the NEW SYSTEM to be created… there is NO READYMADE SYSTEM to be partitioned. It has to be conceived as a whole…more as a black-box with some capabilities. One can then think of possible composition (there can be many) of such a system…which is more of HYPOTHESIS than analysis.

I refer to Ackoff's video in which he said this http://www.youtube.com/watch?v=IJxWoZJAD8k. See all the 3 parts. This is very profound and it operates whether one knows it or not. I am interested in applying it to software design more rigorously. Feel free to reach me at putchavn@yahoo.com

Best wishes,

Reply

Pingback: *Four Myths of Software Evolution | Effective Software Design*

Pingback: *Separation of Concerns « Youry's Blog*

Pingback: *Layers – Separating Software Concerns | Software Patterns Demystified*

Pingback: *When Silos Make Sense | Form Follows Function*

Pingback: *Separation of concerns | Wordpress XML*

Pingback: *Attention Agile Programmers: Project Management is not Software Engineering | Effective Software Design*

Pingback: *Separation of Concerns | Richards Innovation Blog*

Pingback: *Patterns for the separation of concerns | Project Ramon*

Pingback: *Coping with Change in Agile Software Development | Effective Software Design*

Pingback: *The End of Agile: Death by Over-Simplification | Effective Software Design*

Pingback: *Why language choice doesn't matter for your development project : EURA NOVA Blog*

Pingback: *When Silos Make Sense | Iasa Global*

Pingback: *"When Silos Make Sense" on Iasa Global Blog | Form Follows Function*

Pingback: *Antifragility and Component-Based Software Development | Effective Software Design*

Pingback: *Re-Post: The End of Agile: Death by Over-Simplification | Youry's Blog*

Pingback: *Quora*

Pingback: *Do SOLID design principles make code slow? | Effective Software Design*

Pingback: *The SOLID Principles Illustrated by Design Patterns | Effective Software Design*

Pingback: *Unit testing part 2: Getting started with TDD | Ken Bonny's Blog*

Pingback: *Step into OO paradigm | ISA*

Pingback: *Building Better Entity Framework Applications | OnCall DBA*

Pingback: *Six ways to build better Entity Framework (Core and EF6) applications – The Reformed Programmer*

Pingback: *Six things I learnt about using ASP.NET Core's Razor Pages – The Reformed Programmer*

Effective Software Design