

The Wayback Machine - <https://web.archive.org/web/20111110152814/http://www.aspiringcraftsman.com:80/2008/01/03/art-of-separation-of-conc...>

Aspiring Craftsman

pursuing well-crafted software



- [Home](#)
- [About](#)

Search

Go

[Interactive Application Architecture Patterns](#) [An Early Glimpse of Future Composite Smart Client Guidance](#)

The Art of Separation of Concerns

On January 3, 2008, in [Uncategorized](#), by derekgreer



Introduction

In software engineering, Separation of Concerns refers to the delineation and correlation of software elements to achieve order within a system. Through proper separation of concerns, complexity becomes manageable. The goal of this article is to promote the understanding of the principle of Separation of Concerns and to provide a set of foundational concepts to aid software engineers in the development of maintainable systems.

The Principle of Separation of Concerns

The Principle of Separation of Concerns states that system elements should have exclusivity and singularity of purpose. That is to say, no element should share in the responsibilities of another or encompass unrelated responsibilities. Separation of concerns is achieved by the establishment of boundaries. A boundary is any logical or physical constraint which delineates a given set of responsibilities. Some examples of boundaries would include the use of methods, objects, components, and services to define core behavior within an application; projects, solutions, and folder hierarchies for source organization; application layers and tiers for processing organization; and versioned libraries and installers for product release organization. Though the process of achieving separation of concerns often involves the division of a set of responsibilities, the goal is not to reduce a system into its indivisible parts, but to organize the system into elements of non-repeating sets of cohesive responsibilities. As Albert Einstein stated, "Make everything as simple as possible, but not simpler." At its essence, Separation of Concerns is about order. The overall goal of Separation of Concerns is to establish a well organized system where each part fulfills a meaningful and intuitive role while maximizing its ability to adapt to change.

The Value of Separation of Concerns

Applying the principle of separation of concerns to software design can result in a number of residual benefits. First, the lack of duplication and singularity of purpose of the individual components render the overall system easier to maintain. Second, the system as a whole becomes more stable as a byproduct of the increased maintainability. Third, the strategies required to ensure each component only concerns itself with a single set of cohesive responsibilities often result in natural extensibility points. Forth, the decoupling which results from requiring components to focus on a single purpose leads to components which are more easily reused in other systems, or different contexts within the same system. Fifth, the increase in maintainability and extensibility can have a major impact on the marketability and adoption rate of the system. The principle of separation of concerns can also be of benefit when applied to business organizations. Within large companies, ensuring that groups and sub-organizations are assigned a unique set of cohesive responsibilities helps to facilitate overall business goals by minimizing the coordination necessary between teams and maximizing the potential of each team to focus on their collective responsibility and center of competency. The principle of separation of concerns can also improve problem resolution in enterprise wide systems. When responsibilities are properly delineated, problem identification becomes easier, resolution becomes faster, and personal accountability is increased. Each of these areas in turn contributes to an improved quality control process. Whether organizations of people or software systems are in view, applying the principle of separation of concerns can aid in the management of complexity by eliminating unnecessary duplication and proper responsibility allocation. In the next sections, various techniques will be discussed for achieving separation of concerns within application design.

Horizontal Separation

Horizontal Separation of Concerns refers to the process of dividing an application into logical layers of functionality that fulfill the same role within the application. One common division for graphical user interface applications is the separation of processes into the layers of Presentation, Business, and Resource Access. These categories encompass the main types of concerns for most application needs, and represent an organization of concerns which minimizes the level of dependencies within an application. Figure 1 depicts a typical three-layered application:

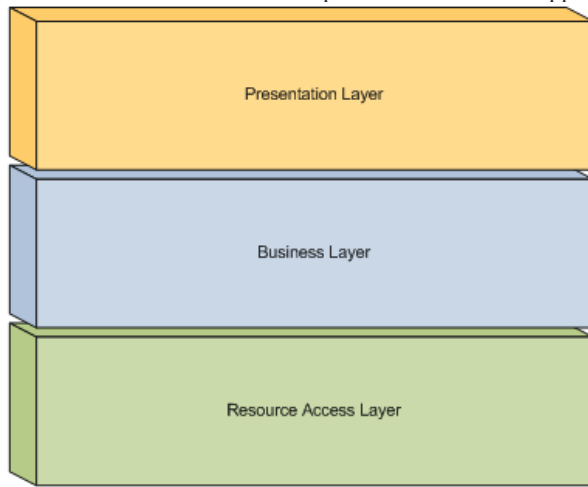


Figure 1

The Presentation Layer encompasses processes and components related to an application's user interface. This includes components which define the visual display of an application, and may include advanced design concepts such as controllers, presenters, or a presentation model. The primary goal of the Presentation Layer is to encapsulate all user interface concerns in order to allow the application domain to be varied independently. The Presentation Layer should include all components and processes exclusively related to the visual display needs of an application, and should exclude all other components and processes. This allows other layers within the application to vary independently from its display concerns. The Business Layer encompasses processes and components related to the application domain. This includes components which define the object model, govern business logic, and control the workflow of the system. The business layer may be represented through specialized components which represent the workflow, business processes, and entities used within the application, or through a traditional object-oriented domain model which encapsulates both data and behavior. The primary goal of the Business Layer is to encapsulate the core business concerns of an application exclusive of how data and behavior is exposed, or how data is specifically obtained. The Business Layer should include all components and processes exclusively related to the business domain of the application, and should exclude all other components and processes. The Resource Access Layer encompasses processes and components related to the access of external information. This includes components which interface with a local data store or remote service. The goal of the Resource Access Layer is to provide a layer of abstraction around the details specific to data access. This includes tasks such as the establishing of database and service connections, maintaining knowledge about database schemas or stored procedures, knowledge about service protocols, and the marshalling of data between service entities and business entities. The Resource Access Layer should include all components and processes exclusively related to accessing data external to the system, and should exclude all other components and processes. Another common division used

by Service Oriented Applications is the division of the application into the layers of Service Interface, Business, and Resource Access as depicted in Figure 2:

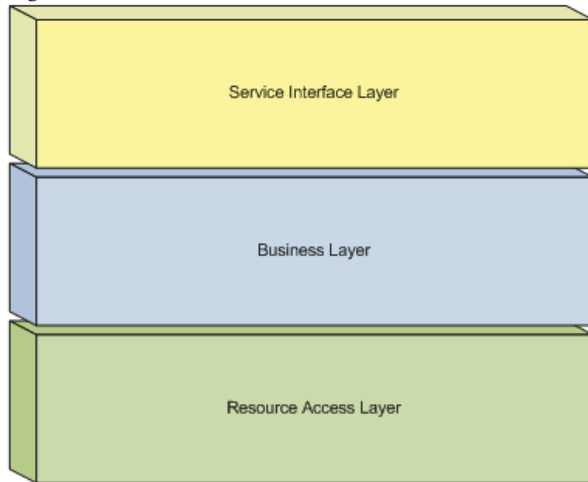


Figure 2

In this division, the Business and Resource Access layers serve the same purposes as previously discussed, with the service exposure concerns encapsulated into a Service Interface Layer. This layer encompasses service interface concerns such as the exposure of business processes through various protocols and the management of service specific contracts and data types. The primary goal of the Service Interface Layer is to encapsulate all service interface concerns in order to allow the application domain to be varied independently. The Service Interface Layer should include all components and processes exclusively related to the exposure of the application as a service, and should exclude all other components and processes. By grouping processing concerns based on their role within the application, a number of benefits are gained which improve the overall system manageability. These benefits include ease of maintenance through consistent architecture and isolation of process, increased insulation from change impact, increased adaptability to change, and increased potential for reuse.

Vertical Separation

Vertical Separation of Concerns refers to the process of dividing an application into modules of functionality that relate to the same feature or subsystem within an application. Vertical separation divides the features of an application holistically, associating any interface, business, and resource access concerns within a single boundary. Figure 3 depicts an application separated into three modules:

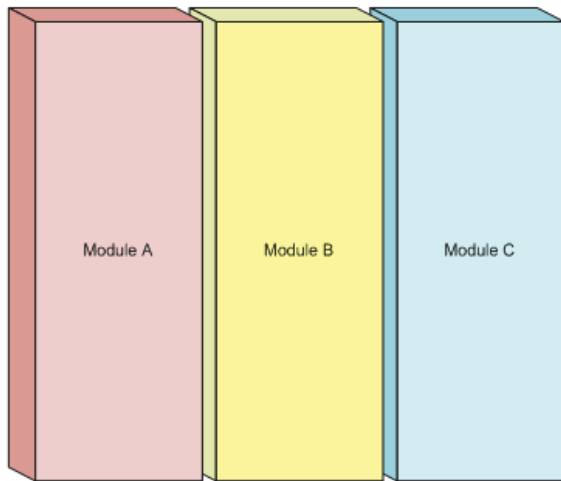


Figure 3

Separating the features of an application into modules clarifies the responsibilities and dependencies of each feature which can aid in testing and overall maintenance. Boundaries may be defined logically to aid in organization, or physically to enable independent development and maintenance. Logical boundaries imply the existence of modularity, though the methods used to denote separation may have no bearing on the actual deployment or runtime behavior of an application. This can be useful for improving the maintainability of an application as well easing any future efforts for physically separating features. Figure 4 depicts an application containing logical boundaries:

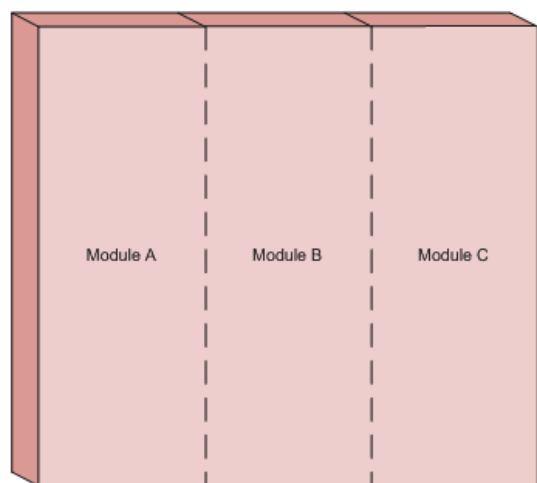


Figure 4

Physical boundaries are generally used in the context of developing add-ins or composite applications, and can enable features to be managed by disparate development teams. Applications supporting add-in modules often employ techniques such as auto-discovery, or initializing modules based on an external configuration source. Figure 5 depicts a hosting framework containing modules developed by separate development teams:

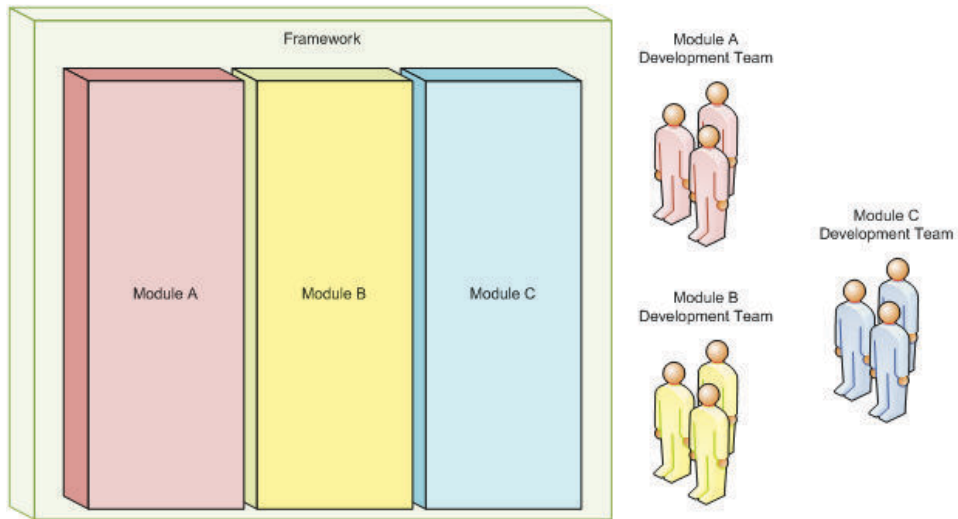


Figure 5

While vertical separation groups a set of concerns based on their relevance to the total fulfillment of a specific feature within an application, this does not preclude the use of other separation of concerns strategies. For example, each module may itself be designed using layers to delineate the role of components within the module. Figure 6 depicts an application using both horizontal and vertical separation of concerns strategies:

3/28/24, 3:58 PM

The Art of Separation of Concerns

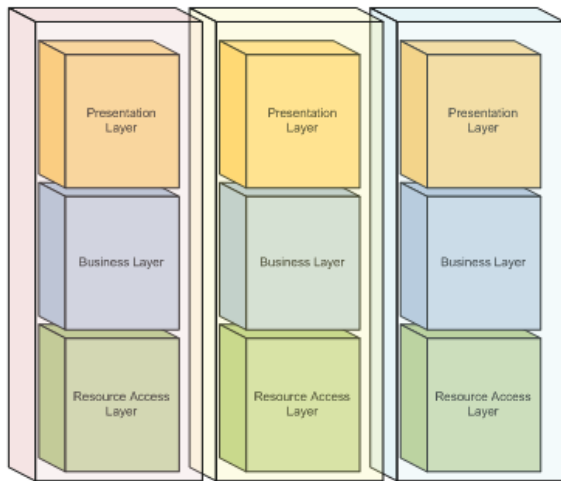


Figure 6

Aspect Separation

Aspect Separation of Concerns, better known as Aspect-Oriented Programming, refers to the process of segregating an application's cross-cutting concerns from its core concerns. Cross-cutting concerns, or aspects, are concerns which are interspersed across multiple boundaries within an application. Logging is one example of an activity performed across many system components. Figure 7 depicts an application with several cross-cutting concerns:

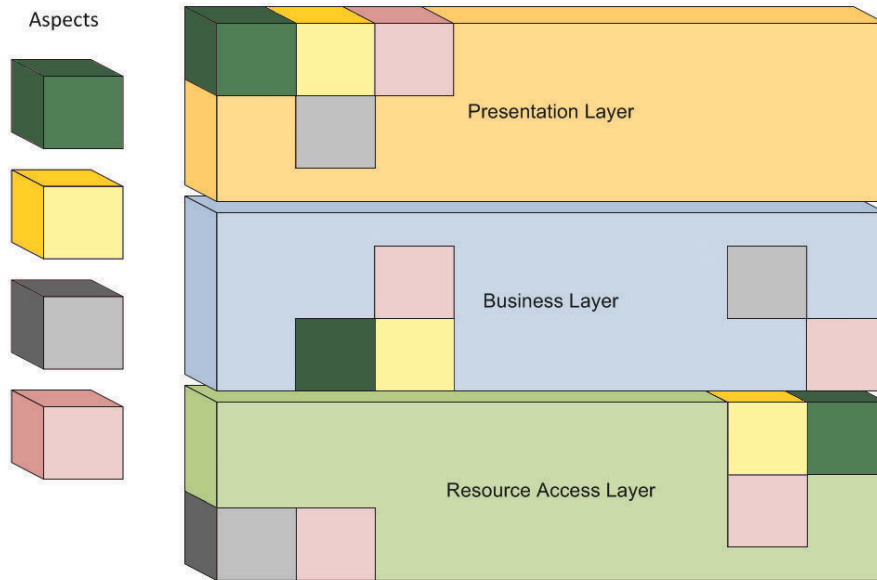


Figure 7

Cross-cutting concerns can be difficult to maintain due to their widespread placement throughout an application. Their mixture with core concerns also adds complexity, rendering the application more difficult to maintain. By separating these concerns, both core concerns and cross-cutting concerns are made easier to manage. Aspect separation differs from other separation techniques in that it relies on a pre-processing, compile time, or run-time fusion of cross-cutting concerns with the application. The process of fusing the two concerns back together is referred to as “weaving”. By using various strategies, cross-cutting concerns can be separated and woven back into the application for run time cohesion. Aspect separation tools are available for a wide range of programming languages. For more information on this topic including a list of available tools for facilitating aspect separation see [Aspect-Oriented Programming](https://web.archive.org/web/20111110152814/https://aspiringcraftsman.com/2008/01/03/art-of-separation-of-concerns/).

Dependency Direction

One characteristic of good Separation of Concerns is the ideal establishment of dependency direction. An ideal dependency direction establishes the roles of consumer and dependency such that the role of dependency is occupied by the entity possessing the highest potential for reuse. A simple example illustrating the concept of dependency direction is the common relationship between a business component and a utility component. Consider a system which provides an order inquiry process which requires that frequently requested data be cached for greater efficiency. To facilitate the caching of order inquiries, a caching utility component may be developed to separate the concerns of caching from the rest of the order inquiry process. Figure 8 depicts this system with its two components:

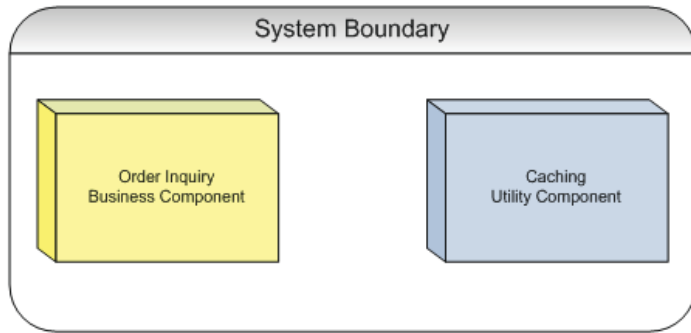


Figure 8

Because the function of caching is a more generic behavior than an order inquiry process, the caching component has the higher potential for reuse among the two. Therefore, the best dependency direction in this case would be from the business component to the utility component. This dependency is depicted in Figure 9:

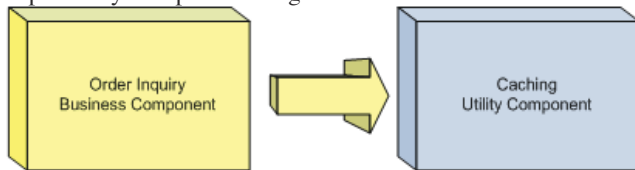


Figure 9

Such a relationship allows the caching component to remain agnostic of the business inquiry process enabling it to be reused by future processes.

Data Concerns

Applying the principle of Separation of Concerns to data involves properly modeling information managed by a system. While the aspects of data modeling apply to both object-oriented and database design, this discussion focuses on object-oriented data concerns since the primary needs of a database often require form to follow function. When organizing data within an object-model, the exposed information should be inherent to the entity being represented. For example, given a system which sells products to customers, an object which defines the product should not contain customer related information. This is because products are not inherently concerned with who may be purchasing the product. A better approach would be the creation of a conceptual order object which composes both customer and product information. This enables the product object to be reused by other processes in the future which may not be concerned with customer information. In addition to considering the potential reuse of a data model within differing contexts, the intuitive organization of data is also beneficial when maintaining highly complex systems. For example, if a new developer were tasked with accessing the serial number of a particular part composed by a product object already in memory, the developer would likely first

attempt to locate the particular product part and then examine the part for a “SerialNumber” or similarly named property. The developer would probably not think to look for something such as a “product number to serial number” dictionary located on the product object because this would not be a natural representation of the data. This would be similar to considering a person to have a serial number by virtue of their wearing a wrist-watch that possessed a serial number. There are times, however, when the natural organization of data does not present an efficient mechanism for information inquiry. For example, an exceptionally complex product object which needs to be inspected frequently to derive the total count of all copper elements may not be efficiently handled through the examination or cross-referencing of each of its composed elements. In cases where natural modeling is not itself sufficient, the integrity of an object-model can be maintained by supplementing conceptual types to satisfy the specialized need. For example, if the product’s composition remains static once assembled, a conceptual model representing the product’s precious metal information (e.g. “ProductPreciousMetalManifest”) might be composed at a peer level to the product information. If the product’s composition changes frequently and the composition processes are centralized then the precious metal information might be updated as part of this process. Otherwise, a specialized component could be conceived (e.g. “PreciousMetalDetector”) to dynamically return the product’s precious metal information. As with the first example, the benefits of separating conceptual needs from an otherwise natural model are that other processes may reuse the model without incurring the overhead of non-inherent characteristics, and the model is kept easily maintainable.

Behavior Concerns

Separating behavior involves the division of system processes into logical, manageable, and reusable units of code. This represents the most fundamental type of Separation of Concerns. Within object-oriented systems, fine-grained behavior is separated using methods while course-grained behavior is separated using objects, components, applications, and services. As with the separation of data, encapsulated behavior should be inherent to its containing boundaries. For instance, a method named `CreateCustomer()` would be expected to only contain behavior relevant to the creation of a new customer. It wouldn’t be expected to, for example, place orders for a new customer. Similarly, a component named `ProductAssembler` would be expected to contain data and behavior relevant to the assembly of products. Similarly, it wouldn’t be expected to contain data or behavior related to customers. Achieving good separation of behavior is often an iterative process. The primary behavior of a system is generally conceived during a design phase, but the specific implementation of a system design often requires several iterations of refactoring as fine-grained concerns become more apparent. When organizing behavior, the following goals should be sought: * Eliminate the duplication of functionality. * Restrict the scope of work to a maintainable size. * Restrict the scope of work to the description of the containing boundary. * Restrict the scope of work to the inherent behavior of the containing boundary. * Minimize external dependencies. * Maximize the potential for reuse.

Extending Concerns

Extensions are a separation of concerns strategy which enables the addition of new behavior to an existing set of concerns. Extensions are used to enhance existing concerns where the desired behavior cannot be added to the targeted system, is not an inherent behavior of the system, or is otherwise impractical for inclusion as part of the system’s core set of features. Figure 10 depicts the dependency relationship of an extension to a target system:

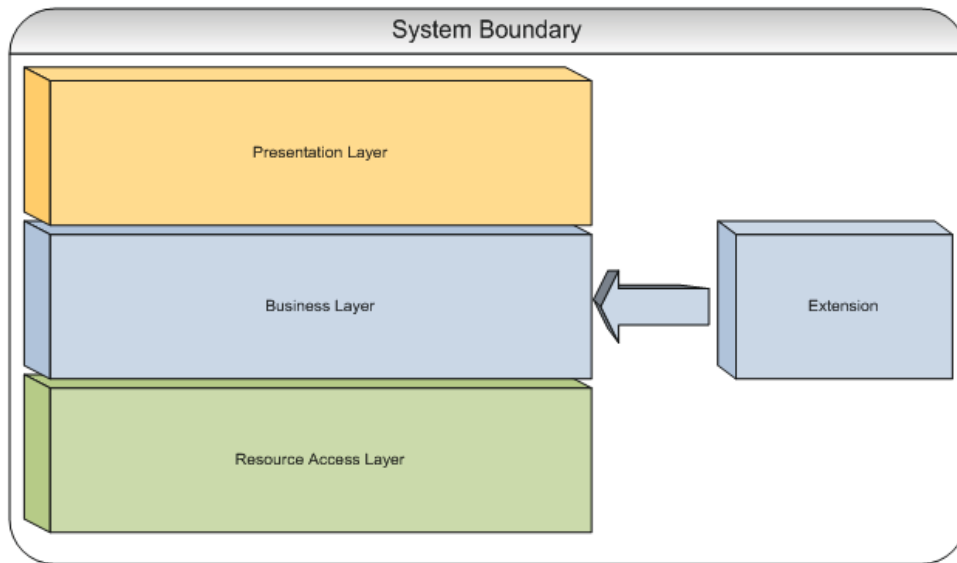


Figure 10

Extensions interact with the target system in response to notifications of key events. Examples of behavior provided by extensions would include the display of new views or controls, the alteration of normal processing, or the alteration of data within the target system. Extensions generally take the form of a hosted add-in, and are typically registered through configuration or a dynamic discovery process. When multiple client applications are maintained within an enterprise, it may be discovered that behavior provided by one client's extension would be valuable to add to another client as well. If the extension is highly customized to the target application and provides a relatively simplistic behavior, the best course of action would be to reproduce the functionality in the new application. However, if the extension's behavior is substantial in size and complexity then it may be more appropriate to provide the behavior generically through an enterprise service. If the behavior provided by the service is still inappropriate to include as a core dependency of the host system then a new extension can be developed to interact with the service on behalf of the application.

Delegating Concerns

Delegating concerns refers to the process of assigning the responsibility for fulfilling behavior to a subordinate component. This strategy separates the concerns of responsibility from execution, and is beneficial for designing components whose implementation details may vary depending on external conditions. Using this strategy, components proxy some or all data requests and method invocations to another component designed to fulfill the request in a specialized way. For example, a component designed to return a list of roles assigned to the current user may delegate the request to one or more subordinate components designed to retrieve roles from a local XML file, database, and/or a remote service. Figure 11 illustrates the delegation of authorization concerns to components specialized for differing data sources:

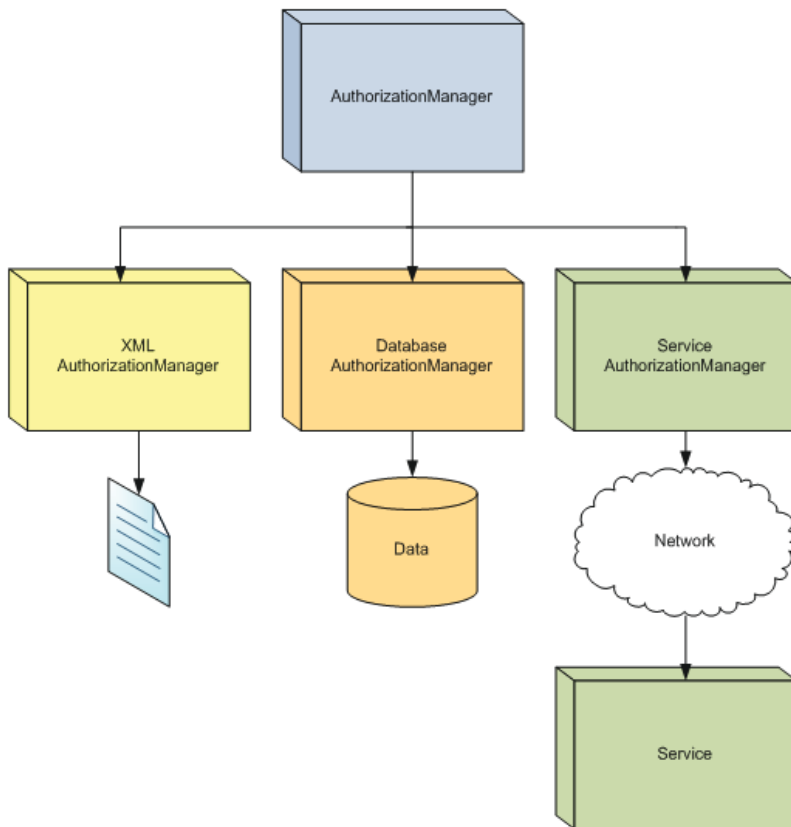


Figure 11

Delegation can be facilitated using the [Strategy Pattern](#), the [Plug-in Pattern](#), the [Microsoft Provider Model](#), and other variant patterns which enable portions of a component's behavior to be fulfilled externally.

Inverting Concerns

Inverting concerns, better known as Inversion of Control, refers to the process of moving a concern outside of an established boundary. Some uses for inversion of concerns include effecting aspect separation, minimizing non-essential processes, decoupling components from specific abstraction

strategies, or relocating responsibilities to infrastructure components. Some specific applications would include alleviating responsibilities pertaining to hardware interaction, work-flow management, registration processes, or obtaining dependencies. Figure 12 depicts an inversion of concerns process where both a presentation component and a domain component have had concerns moved to infrastructure level components:

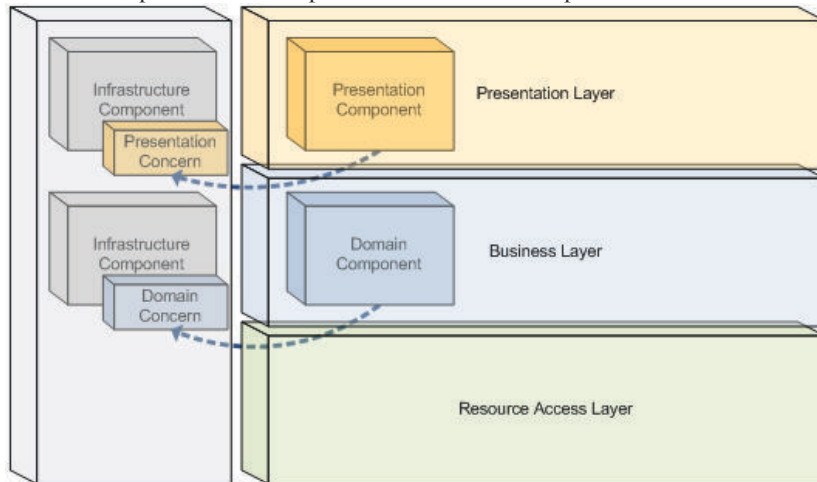


Figure 12

One example where inversion of concerns can be seen is in the use of the [Template Method](#) pattern. This pattern is used to generalize the behavior of a process in order to allow variation through inheritance. When applying this pattern to an existing component, the steps performed by the desired process would be generalized and encapsulated in a base class. The existing component would then inherit from the new base class, maintaining only specialized behavior. Other types would then be free to inherit from the base class to provide variant implementations. This would be an example of inverting the concerns of an algorithm sequence. Another example of inverting concerns can be observed when converting an interactive console application to a GUI application. A typical interactive console application provides a main loop which prompts the user for information and waits for data to be entered. When converted to a GUI application, the main control of the program is generally provided by infrastructure components, and notification of user interaction is accomplished using the [Observer Pattern](#). This inverts the main control of the user interaction process by relying on the application infrastructure or hosting environment for these concerns. Dependency Injection is a term applied to the inversion of concerns which relate to obtaining external dependencies. This term was conceived to more clearly describe the behavior of a class of frameworks concerned with this form of inversion. The goal of dependency injection is to separate the concerns of how a dependency is obtained from the core concerns of a boundary. This improves reusability by enabling components to be supplied with dependencies which may vary depending on context. The process of dependency injection generally involves the roles of container, dependency, and recipient. The role of container is occupied by a component responsible for assigning dependencies to a recipient component. Containers are generally implemented using the [Factory Pattern](#) to allow creation of the recipient and dependency components. The role of dependency is occupied by the resource being provided to the recipient. Containers are often implemented to allow existing objects to be registered for use as a dependency, or to create new instances when required. The role of recipient is occupied by the component receiving a dependency from the container. Recipients are required to declare dependencies using a strategy supported by the container. Strategies for determining dependency information generally employ the use of reflection to examine one or more of the recipient's

member types, and may require the use of attributes/annotations to implicitly or explicitly specify dependencies. A sequence diagram representing these roles is presented in Figure 13.

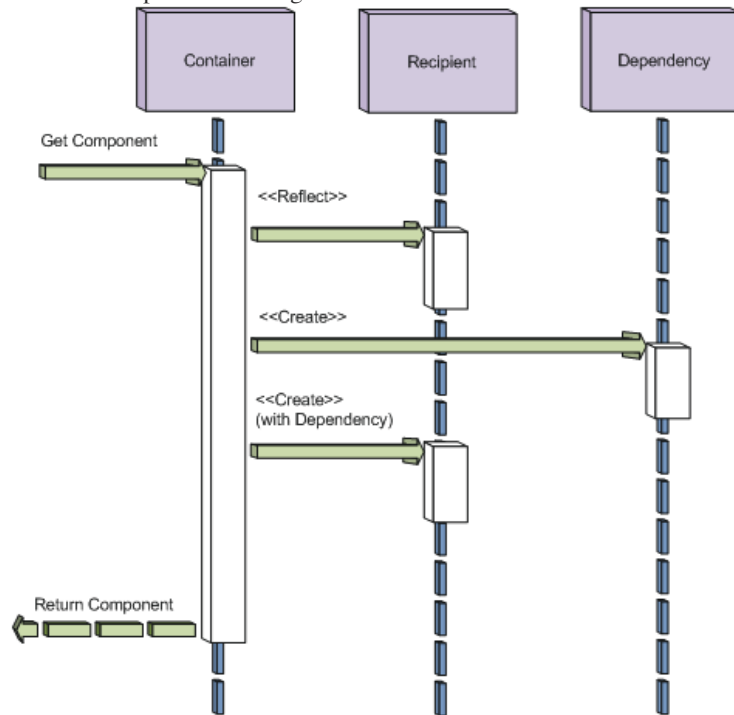


Figure 13

One example use of dependency injection would be to abstract caching functionality from a business component in lieu of using the [Factory](#) or [Factory Method](#) patterns. By supplying a recipient with a caching component using dependency injection, the recipient remains loosely coupled to the caching component without the added coupling to a specific factory. Another example would be the use of dependency injection to control the number of caching components created in lieu of using the [Singleton Pattern](#). This enables the application to ensure that only a single instance is used throughout the lifetime of an application, or to control which recipients receive which instances of the component. This also increases the ability to test components in isolation of their dependencies by enabling mock dependencies to be supplied rather than requiring singletons to be tested along with their consumers. Components using patterns such as [Factory](#), [Factory Method](#), [Abstract Factory](#), [Singleton](#), [Plug-in](#), [Service Locator](#) to abstract the concerns of object creation or dependency retrieval are good candidates to consider for supplementing or replacement with dependency injection. By using dependency injection in lieu of, or in cooperation with these and other such patterns, consuming components can be completely abstracted from the particular abstraction strategy.

The Exaggeration Exercise

It is often the case that the negative consequences of a design choice, particularly those relating to scalability and reuse, do not become apparent until long after a system has become established. Problems with scalability and reuse are usually rooted in a lack of adherence to the principle of Separation of Concerns. One process that can aid in optimizing concerns is considering the impact of a design when applied under exaggerated circumstances. Through this exercise, the use of a system is hypothetically exaggerated beyond the system's known expectations in order to reveal potential weaknesses in a design approach. For example, when designing an object model to be used by two existing systems, one might consider what the consequences would be if the object model were shared by fifty systems. By exaggerating the use of a design, poorly organized responsibilities will often become easier to identify. To demonstrate this exercise, consider the following example concerning the creation of a new composite customer relationship management system: Within a corporation, an IT department has been requested to create a custom CRM application which will allow disparate development teams to contribute specialized modules of functionality. The main customer related segments within the corporation will be sales, billing, and technical support, though there may be multiple development teams assigned to support the functions of each segment. It has been requested that the application present a main screen allowing customers to be queried by a number of different criteria including name, address, phone number, order number, and any registered product serial numbers. However, the resulting views and workflows should depend upon which business function is being carried out at a given time. For example, upon submitting customer search criteria, sales users may be displayed with a view pertaining to past purchasing trends, credit rating scores, and suggested up-selling scripts; while a billing user may be presented with a view displaying customer payment history, billing dispute history, and outstanding balances. An initial analysis reveals that there will be a total of three workflow variations resulting from a total of five different search criteria fields, and that there will be three backend systems involved in obtaining the information needed by all segments. Because the variations are low, the choice is made to centralize the main view, customer search functionality, and workflow initiation within a search module. It is understood that the addition of new search criteria and workflow needs will require modifications by the search module development team, but it is believed that these needs will be infrequent. Applying the exaggeration exercise to this design choice might entail considering what consequences might ensue if the number of business segments or number of backend systems were increased to fifty. As a result of centralizing the search functionality and workflow initiation, increasing the scope of these concerns will also increase the responsibility and workload of the search module development team. This would include the scoping, analysis, design, coding, and testing of all new features and change requests related to these concerns. In turn, this would likely result in the need to increase the number of development resources required for the search module team. It is also likely that the same initial assumptions that lead to the consolidation of these concerns also informed other design decisions made for the search module. This may have led to design choices that would not readily accommodate such an increase in responsibilities, requiring some level of internal redesign to handle new concerns. An outstanding problem that can be observed through this exercise is that the amount of work required by the search team is proportionally related to the number of business segments or workflows supported by the system. The fact that the initial solution doesn't scale as the need increases can be an indication that concerns may have been inappropriately distributed throughout the system. The decision for consolidating search functionality and the resulting workflow decisions within the search module was the result of recognizing similarities between each use case. What was not given equal consideration was the fact that each use case had to be treated in a specialized way, and the fact that the number of possibilities has no true boundaries. Because the search screen provides a central function for many modules, it can be said to possess inherent infrastructure concerns. It should, therefore, be expected to provide the behavior that is common to all modules. However, the details of each case are not common, and can be considered the inherent concerns of each respective module. Upon recognizing the inherent responsibilities, an alternate approach might include the development of a framework to consolidate the common concerns, but enabling the distribution of each domain specific concern. Such an approach could be accomplished by requiring each module to provide an add-in which registers available search criteria and provides handling capabilities to invoke the corresponding workflow. A framework would then be responsible for presenting a consolidated view of the search criteria, and providing a generic infrastructure for associating each search criteria to its corresponding workflow handler. By using this approach, the search module can be designed to accommodate an unlimited number of use cases.

While the exaggeration exercise can be useful in creating highly scalable designs, this is only a byproduct of its primary goal which is the achievement of optimal Separation of Concerns. The exaggeration exercise might be compared to the medical practice of taking a patient's temperature to test for symptoms of a possible underlying problem. This exercise attempts to identify problems pertaining to separation of concerns by examining the scalability aspect of a potential design. This might be compared to using a magnifying glass to look at the details of a design as depicted in Figure 14.

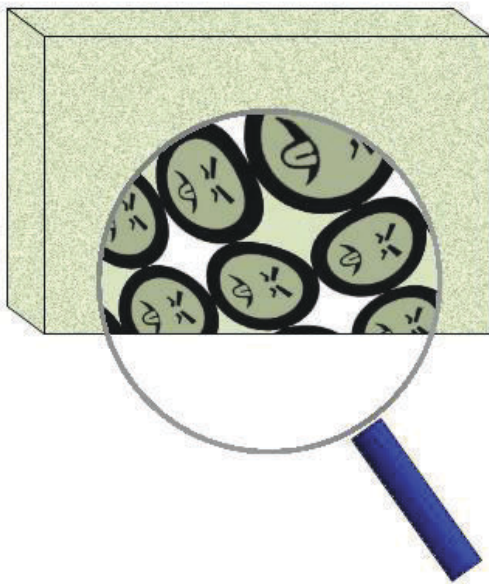


Figure 14

It is through the exaggeration of a design's actual scope that small problems are magnified for easier identification. Once identified, it may then be determined what course of action should be taken.

Separation Anxiety

Applying the principle of separation of concerns often involves advanced concepts and constructs which bring a certain level of complexity to the application beyond that of merely addressing the domain concerns of the application. For developers inexperienced in these programming techniques the reaction falls within the spectrum of enthusiastic excitement over the opportunity to add to one's repertoire of design skills, and an adverse reaction to the additional amount of complexity involved in getting the job done in the most expedient way possible. These techniques often lead

inexperienced or more tactical-minded developers to characterize such designs as “overly-complex” or “over-engineered” based upon their frustrations in first learning, and then maneuvering through such architectures on a day-to-day basis. Furthermore, there is often an ever-present pressure from project managers, product managers, upper level management, marketing, or end users for instant gratification which tends to encourage and reward expediency over thoughtful design. These conditions can present obstacles to developing good designs beyond merely solving the technical problems. While designs which promote separation of concerns often add complexity to an application, it should be pointed out that they also remove the complexity that is generally associated with a lack of separation of concerns. For many applications, the trade-off is often between ordered complexity and disordered complexity. Applications which do not exhibit an appropriate amount of separation of concerns are often difficult to learn due to the need to understand the whole before understanding the part, and difficult to maintain and extend. Highly complex, yet poorly modularized applications also have the effect of causing high turnover in development staff (which tends to further compound design and implementation problems), or attracting individuals who have an aversion to change and seek to build their career by being the indispensable “master of the maze” within their organization. Development teams should certainly not seek complexity for complexity’s sake (unless entering an obfuscation contest), but the notion that avoiding advanced design concepts equates to avoiding complexity should be dispelled.

Conclusion

Put simply, the goal of the principle of separation of concerns is order. By ensuring elements within a system adhere to a single and unique purpose, complex systems can be designed which maximize productivity and maintainability.

If you enjoyed this article, please consider sharing it!

 [Facebook](#)  [Twitter](#)  [Delicious](#)  [Digg](#)

Tagged with: [Separation of Concerns](#)

10 Responses to *The Art of Separation of Concerns*

1.  *maarten* says:
[October 27, 2008 at 6:52 am](#)

great article!

[Reply](#)

2.  *David Roh* says:
[November 22, 2008 at 7:16 am](#)

Very, very nice article Derek.

Thank you for sharing this.

3/28/24, 3:58 PM

The Art of Separation of Concerns

David Roh

JK@SilverlightAzure.com

[Reply](#)



Dave says:

[January 19, 2009 at 6:46 am](#)

Best Article i have read on SOC.
Very well defined and based on solid principals.

Thank you!!

[Reply](#)



Anonymous says:

[September 4, 2009 at 12:35 pm](#)

One of the best article on SOC.

[Reply](#)



Anonymous says:

[November 6, 2009 at 2:38 am](#)

I used this for teaching post-grads at the university. Well done.

[Reply](#)



Anonymous says:

[January 14, 2010 at 5:03 am](#)

Excellent article! Thank you for sharing it.

[Reply](#)



Anonymous says:

[February 27, 2010 at 12:02 pm](#)

Simple, clear, cohesive. In one word "Excellent" !

[Reply](#)

<https://web.archive.org/web/20111110152814/https://aspiringcraftsman.com/2008/01/03/art-of-separation-of-concerns/>

19/23

3/28/24, 3:58 PM

The Art of Separation of Concerns

8.  *Iskender Yigitel* says:
[March 17, 2011 at 5:32 pm](#)

Thanks for sharing. Seperation of concerns is one of the most fundamental concepts in software engineering. Your article presents the concept very well.

[Reply](#)

9.  *Ivan Troy* says:
[July 25, 2011 at 2:51 pm](#)

Excelent article, just a comment, please mention your sources and references.

[Reply](#)

10.  *Anonymous* says:
[August 26, 2011 at 2:45 pm](#)

Thank you for sharing. It's a great article explaining one of the most important things on the road to building maintainable applications.

[Reply](#)

Leave a Reply

Your email address will not be published. Required fields are marked *

Name *



Email *

Website

Comment



3/28/24, 3:58 PM

The Art of Separation of Concerns

You may use these **HTML** tags and attributes: `<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code> <del datetime=""> <em> <i> <q cite=""> <strike> <strong>`

Post Comment



SE HABLA CODE

Search for:

Search

Recent Posts

- [How To Produce Bug-Free Software](#)
- [Dependency Management in .Net: install2](#)
- [Dependency Management in .Net: Get](#)
- [Dependency Management in .Net: Using NuGet without Visual Studio](#)
- [Dependency Management in .Net](#)

Series

- [Effective Tests](#) (17)

Tags

[acceptance testing](#) [Application Structure](#) [Arrow Anti-pattern](#) [ASP.NET MVC](#) [Assembly Resolution](#) [Auto-mocking Containers](#) [BDD](#) [C#](#) [Composite Application Development](#) [Convention over Configuration](#) [Custom Assertions](#) [Cygwin](#) [Dependency Injection](#) [Dependency Inversion Principle](#) [Double Dispatch](#) [Expected Object Pattern](#) [ExtensionMethods](#) [Git](#) [Law of Demeter](#) [Machine.Specifications](#) [Managed Extensibility Framework](#) [Model-View-Controller](#) [Model-View-Presenter](#) [NCommons](#) [NHibernate](#) [NuGet](#) [Object-Oriented Design](#) [Presentation-Abstracton-Control](#) [Resharper](#) [Separation of Concerns](#) [TDD](#) [Test Doubles](#) [Testing](#) [Unity](#) [Visitor](#)

Archives

- [October 2011](#)
- [September 2011](#)

<https://web.archive.org/web/20111110152814/https://aspiringcraftsman.com/2008/01/03/art-of-separation-of-concerns/>

21/23

3/28/24, 3:58 PM

The Art of Separation of Concerns

- [July 2011](#)
- [June 2011](#)
- [May 2011](#)
- [April 2011](#)
- [March 2011](#)
- [October 2010](#)
- [July 2010](#)
- [June 2010](#)
- [April 2010](#)
- [March 2010](#)
- [February 2010](#)
- [January 2010](#)
- [October 2009](#)
- [June 2009](#)
- [May 2009](#)
- [March 2009](#)
- [December 2008](#)
- [November 2008](#)
- [October 2008](#)
- [August 2008](#)
- [June 2008](#)
- [January 2008](#)
- [August 2007](#)
- [June 2007](#)
- [April 2006](#)
- [January 2006](#)
- [October 2005](#)

Meta

- [Log in](#)
- [Entries RSS](#)
- [Comments RSS](#)
- [WordPress.org](#)

[Aspiring Craftsman](#)

<https://web.archive.org/web/20111110152814/https://aspiringcraftsman.com/2008/01/03/art-of-separation-of-concerns/>

22/23

3/28/24, 3:58 PM

The Art of Separation of Concerns

Pages

- [About](#)

Stay In Touch

- [About](#)

More

© 2011 Aspiring Craftsman

 [iBlog by PageLines](#)
