Design Principles

Separation of Concerns – Basics

The following extract from a blog by Hayim Makabee (https://effectivesoftwaredesign.com/2012/02/05/separation-of-concerns/) gives a nice introduction to the design principle known as "Separation of Concerns"

From the "Effective Software Design"

The Art of Separation of Concerns

Introduction:

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. 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.

Origin of the term:

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.

Why do we need SoC:

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. Thus the system as a whole becomes more stable.

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.

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.

Designing a software system which is extensible and maintainable is driven by the key principles of low coupling and high cohesion between the modules. Separation of concerns is one of the methods that helps in designing such a system.

To understand the benefits, let us look at examples of system designed with careful attention to separation of concerns. Consider a file reader which reads a file and displays the information (read only). If separation of concerns is emphasized then the reader would be a specific module and would be independent of display and process modules.

Maintainability: Consider the scenario where the large file is to be read efficiently. The existing design allows the change to be confined to one module (the reader) and hence, automated testing (for example) needs to be changed only for that module. This gives low cost of maintenance in terms of changes made and hours spent in verification.

Extensibility: If the system is to read a new file format, the reader could also be extended to support an additional format. A client designed to read xml can be modified to read JSON files with changes just to the reader. The other parts of the system are not concerned about the format of the input.

Reusability: Reader module used in the file reader could be used in the file editor (consider a suite built to read word, spreadsheets, and PowerPoint). As the modules become less coupled and more cohesive, they could be reused without much refactoring of code.

Separation of concerns, unlike other principles, can be applied at various design levels. It could be used in system level as well as the module level. In the system level, separation of concerns can be achieved by layering the system. A well-known example of layering would be the network OSI network model.

The following excerpt from “Aspiring Craftsman”, a blog by Derek Greer, discusses how different applications and varying perspectives can expose different decompositions of the same application – each of which focuses on different types of concerns, but still conforms to the principle of separation of concerns. The rest of the article by Greer, discusses SOC in even more advanced contexts such as aspect-oriented programming. The full article can be found at “http://aspiringcraftsman.com/2008/01/03/art-of-separation-of-concerns/”

From “Aspiring Craftsman”

How to achieve SoC:

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.

Horizontal Separation:

Horizontal Separation of Concerns refers to the process of dividing an application into logical layers of functionally 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:

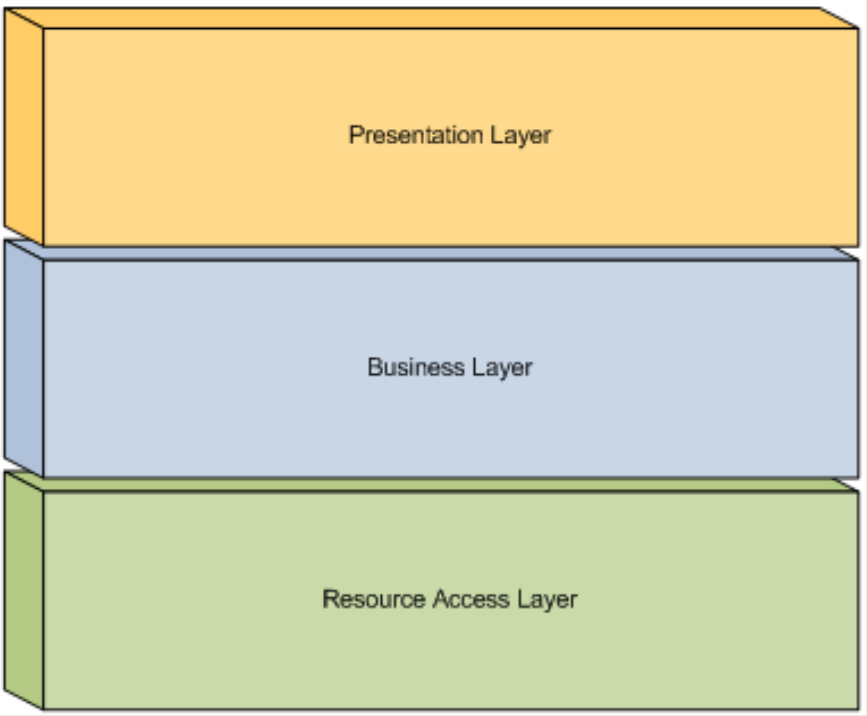

Figure 1

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.

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 sub-system 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:

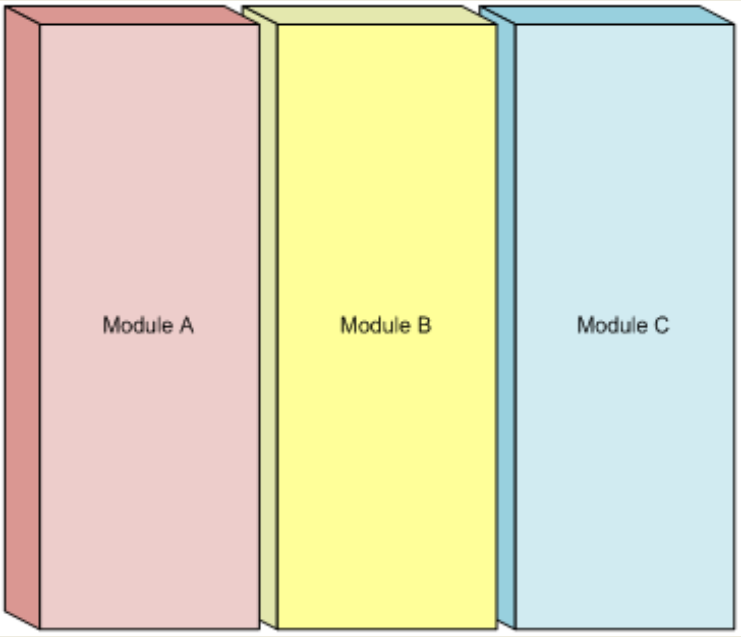

Figure 3

Separating the features of an application into modules clarifies the responsibly 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.

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:

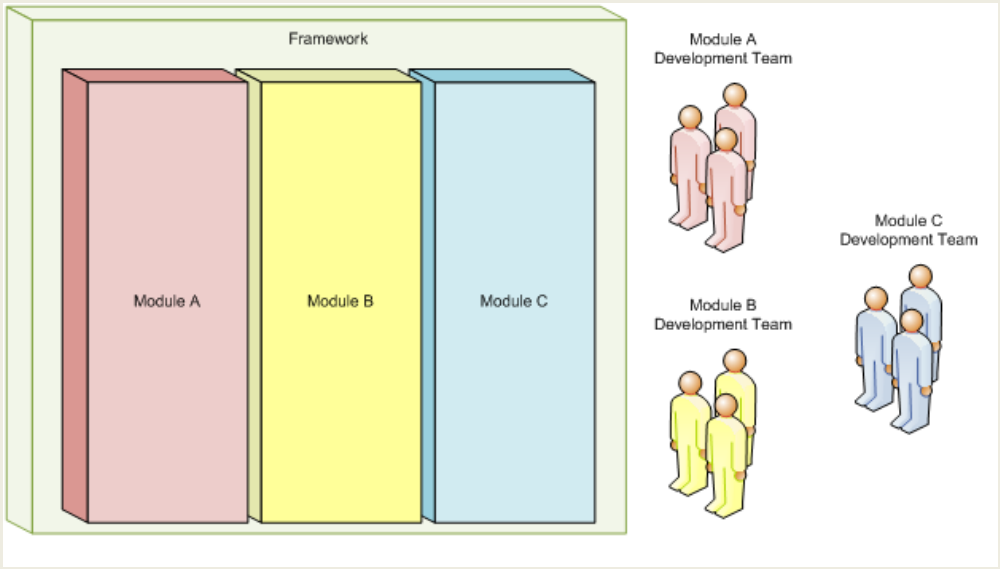

Figure 5

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:

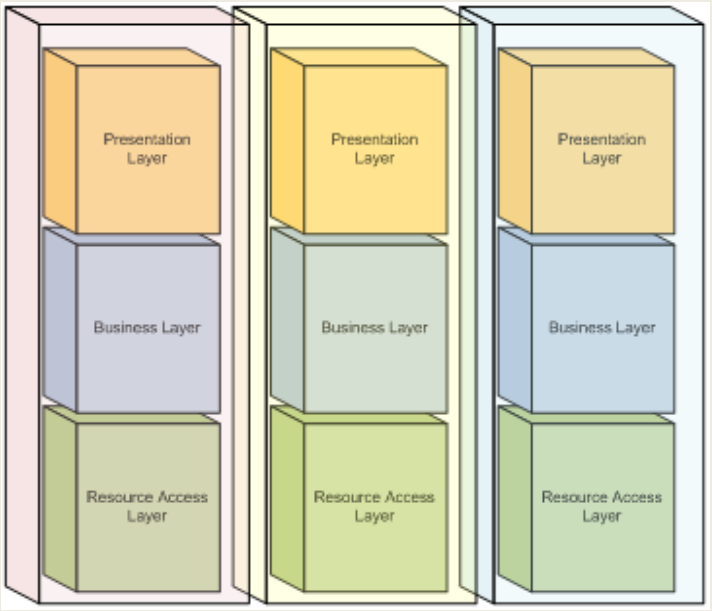

Figure 6