# Separation of Concerns - Behavior Concerns

## What is it?

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.

#### Example:

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.

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.

## How to implement it

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.

## Reasons to do it

1. 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.
2. Helps you focus on one particular function or responsibility at a time.
3. Applying the principle of separation of concerns can contribute to an improved quality control process.
4. Supports parallel work and separation of programmer responsibilities.
5. *Maintainability*: 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.

## Conclusion

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.