Separation of Concerns in Software Design

2020, JAN 16

The separation of concerns (SoC) is one of the most fundamental principles in software development.

It is so crucial that 2 out of 5 SOLID principles (Single Responsibility and Interface Segregation) are direct derivations from this concept.

The principle is simple: don't write your program as one solid block, instead, break up the code into chunks that are finalized tiny pieces of the system each able to complete a simple distinct job.

In this article, I elaborate on applying this profound principle at all levels of abstraction: from the programming code inside each function and the design of the modules to the overall app's architecture, all for achieving the characteristics of what we call a qualitative software.

SoC for programming functions

If we take the lowest level (the actual programming code), SoC instructs us to avoid writing long complex functions. When the function starts to bloat up in size, this is the red flag that the method is possibly taking care of too many things at once.

In such a case SoC pushes us to refactor it, turning into a more laconic and descriptive revision. During this process, parts of the original algorithm get exported and encapsulated in separate smaller functions with a private access level. We gain the code clarity, and chunks of the algorithm eventually become reusable by other parts, even if we initially didn't expect this to happen.

SoC for modules

At a bit higher level, this principle tells us to group the functions under self-contained modules, each responsible for the fulfillment of a single set of tasks that have a clear logical correlation.

The process very much resembles what we had to do for functions: *estrange less-closely related functionality and group up the features serving the same distinct purpose*.

Cohesion and Coupling

The application of the Separation of Concerns involves two processes: reduction of coupling and increasing cohesion.

Cohesion is the measure of similarity by the set of duties, level of details, and locality. For example, functions `drawCircle` and `drawTriangle` are cohesive enough to belong to the same module responsible for drawing, and it *feels* natural to put these two functions close to each other in the code (high similarity ~ high cohesion).

Coupling, on the other hand, is the measure of dependence of the part on the rest of the system (low dependence ~ loose coupling).

The aforementioned `drawCircle` and `drawTriangle` can be used by another function `drawCybertruck`. We can be tempted to put this function in the drawing module as well, but `drawCyberthuck` may be dependant on the physics engine and the external state. So this will make the whole drawing module much less reusable and closely coupled with a few other components.

You can tell that primitive drawing functions and `drawCyberthuck` belong to different levels of abstraction and logic complexity, thus they need to reside in different modules.

And if at some point we decide to use the drawing module in another project - there will be no dependency on the physics engine, so we'll be able to extract it easier.

A quick way to remember which attribute should be increased or decreased:

- Decoupling is good - so we need to aim for a loose coupling
- Cohesive code is good - we need to aim for a high cohesion

A good example of the code with high cohesion (low dispersion) is the use of closure callbacks instead of the delegate methods. Consider the code for sending a networking request:

```
// configuring and sending the request
session.send(request: URLRequest) { response in
    // handling the response
}
```

Imagine if URLSession had a delegate-based API for making the requests: all the responses would be delivered to a single function `handle(response: URLResponse, for request: URLRequest)`

This would make the networking much more error-prone and tedious, as the logic for handling all the responses would now have to be tied to that one function.

With the callback-based API, the action and the result of the action are handled in one place, making it much easier to track the execution flow.

If we need to jump around between the functions or modules as we're following the logic of the algorithm, this means the code has **low cohesion**, which is often referred to as a Spaghetti code.

Benefits of the Loose Coupling and High Cohesion

Adherence to the principle of Separation of Concerns helps to improve numerous characteristics of the codebase:

1. Better code clarity. It is much easier to understand what is going on in the program when each module has a concise and clear API with a logically scoped set of methods.
2. Better code reusability (DRY principle). The main benefit of reusing the code is reduced maintenance costs. Whenever you need to extend the functionality or fix a bug - it's much less painful to do when you're certain it appears in one place only.
3. Better testability. Independent modules with properly scoped functionality and isolation from the rest of the app are a breeze to test. You don't need to set up the entire environment to see how your module works - it is sufficient to replace neighboring real modules with dummy mocks or fake data sources. This way you can test the module as the black box by verifying just the output, or as the white box by also seeing which methods are being called on the connected modules (BDD).
4. Faster project evolution. Whether it's a new feature or an update of the existing one, isolation of the modules helps with scoping out the areas of the program that may be affected by the change, thus speeding up the development.
5. It is easier to organize simultaneous development by multiple engineers. They just need to agree on which module they are working on to make sure they don't interfere with each other. Only the update of a

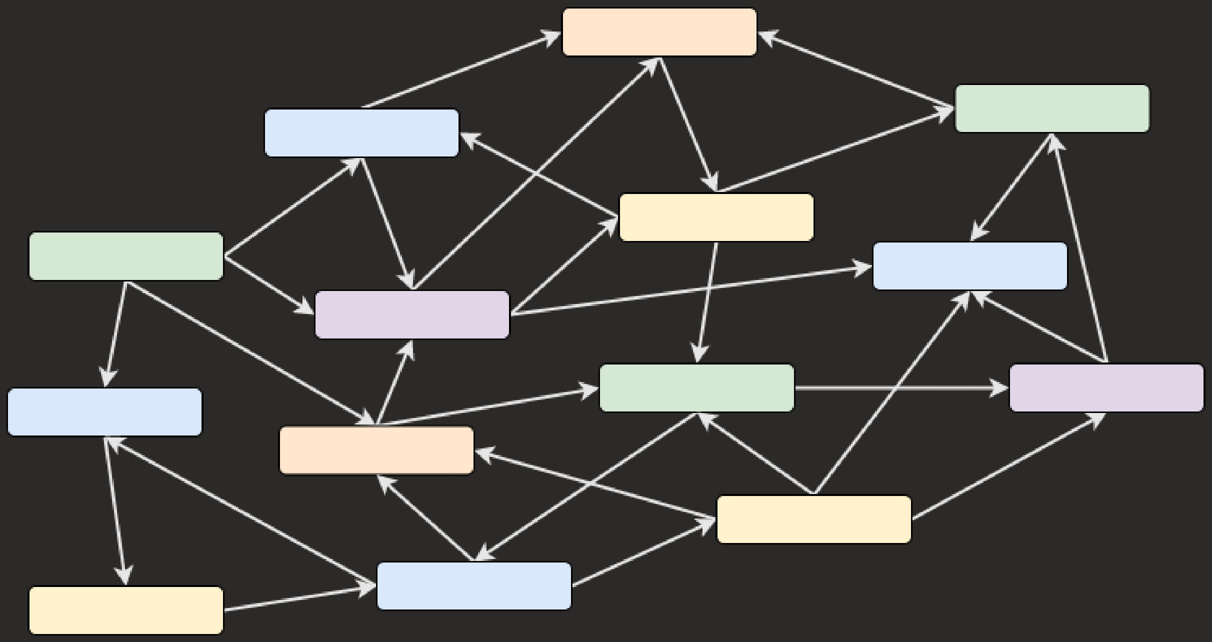
module's API can be a flag for explicit notifying other developers, while most of the changes can be added without immediate attention from the other contributors. When coupled with good test coverage, the parallel development becomes as efficient as the cumulative productivity of each individual engineer working solely (it is usually slower).

> *As you can see, coupling and cohesion are the characteristics that ultimately affect the **convenience** of working with the code from the programmer's perspective.*

SoC for the system's design

For a bunch of modules with distinct responsibilities and clear purpose we still need to outline a global strategy to how the modules should refer to each other.

If we don't introduce this strategy we may end up with a system with entangled relations and hard-to-track data flows.



The primary goal of the system's design is to outline the boundaries for the module's awareness of each other.

Every existing architectural pattern provides this strategy. Take we, for example, the Model-View-Controller, we would see that the View is not allowed to interact with the Model directly and should use the Controller as the intermediary.

It seems to me that these strategies often come from the general idea that permissiveness is not good. This, in my opinion, leads to either over-engineered solutions or conversely to the systems with insufficient decoupling of the responsibilities.

I tend to think that the system's design requires a much more formal approach with clear metrics and motivation.

We already saw that SoC, when applied to the functions and modules, always led to more reusable, testable and maintainable code. So why not take the cohesion and coupling as those metrics and apply the SoC at the application's level?

That's how we get to segregating the modules into layers. This is not a concrete architectural pattern, but rather a high-level specification for that strategy I was talking about.

The modules get grouped in layers, the same way we'd form a module from the set of distinct functions.

> *The resulting set of modules within one layer has high cohesion based on the similar duties in the system and the same level of abstraction, while communication and environment awareness between the layers is very much restricted to achieve loose coupling.*

We're not only constraining the communication - the layers with higher environment specifics at the bottom (Repositories, such as a database wrapper or a networking service) are forbidden to directly refer to anything defined in the higher layers (business logic or UI).

So if we take just the networking service that talks to the backend, it should know **nothing** about the rest of the system and only provide the API for sending the requests.

The business logic layer will be aware of and using that Repository, but it should have no idea if any UI is attached to the system.

The UI layer is aware of the business logic modules and uses their APIs to read the up-to-date data and trigger actions, but at the same time, it knows nothing about the Repository, as the business logic hides the factual underlying infrastructure from it.

> *This way we can guarantee intrinsic testability of the whole system, where each layer either doesn't even know the other exists or is decoupled to such a high degree that can easily be surrounded by mocks in tests.*

Repository

Although decoupling of the business logic and the UI is a standard move, I found it surprising that the majority of the popular patterns we have for iOS do not stress the importance of decoupling the business logic from the data gateways, such as a networking layer.

So many times did I see the requests being sent right from the view controllers or other business logic modules. Same thing with the database queries, UserDefaults, and any other local or remote data storages.

As you may guess, I don't like the tight coupling here. But it's not just coupling between the modules, which is more or less tolerable.

We're talking about a tight coupling between the inputs for the algorithm and the algorithm itself. Such code is nearly impossible to test or evolve.

There are multiple reasons why you don't want to embed direct read-write operations in your business logic, thus being unable to easily swap the real calls with mocked ones:

1. You may accidentally corrupt the valuable data as you run an unfinished algorithm

2. The access to the real data may be slow (large file sizes for local resources, slow network / test server when accessing remote resources)
3. The external data may not be available (local database is empty and needs to be pre-populated, the server is down or there is an Internet connection outage)
4. The backend may suddenly change the response format when you don't expect it

The latter case is notorious. Of course, in an ideal world, this should never happen, but it does, and more often than you may think. Even CI won't save you.

The app would stop working, and the first person to be blamed will be YOU, the mobile engineer. YOUR app broke. And YOU'll have to offer excuses and look pathetic during the first minutes after the failure was uncovered.

Imagine the CEO of your company is presenting the app on an important event for investors, and THIS happens.

The ideal way out would be: the app does not crash, but instead, gracefully shows a user-friendly error message. We're handing our boss another device with a build that runs in offline mode using mocked demo data, and the presentation continues, with the accident passed almost unnoticed.

Offline demo mode? It sounds like a lot of work! But it's not if you've decoupled and abstracted away from the data gateways.

When we have a business logic module that queries the data from elsewhere, we need to extract the concern of accessing that external data resources to a separate module and hide the unnecessary query details behind a facade.

This is how a Repository is formed.

Let's see an example. We have a ViewController that loads and displays a list of some items:

```
class ListViewController: UIViewController {

    var items: [Item] = []
    var tableView: UITableView?

    override func viewDidLoad() {
        let url = URL(string: "https://api.service.com/list")!
        let request = URLRequest(url: url)
        URLSession.shared.dataTask(with: request) { [weak self] (data, response, error) in
            if let list = try? JSONDecoder().decode([Item].self, from: data ?? Data()) {
                self?.items = list
                self?.tableView?.reloadData()
            }
        }
    }
}
```

The first thing to do is to introduce the `ListRepository` protocol and refactor the ViewController to use it:

```
protocol ListRepository {
    func loadList(completion: @escaping ([Item], Error?) -> Void)
}
```

```
 class ListViewController: UIViewController {

    var items: [Item] = []
    var tableView: UITableView?
    var repository: ListRepository

    override func viewDidLoad() {
        repository.loadList { [weak self] (list, error) in
            self?.items = list
```

```
                self?.tableView?.reloadData()
            }
        }
    }
```

And now we have the freedom to substitute the implementation that actually works with the backend:

```
struct RealListRepository: ListRepository {
    func loadList(completion: @escaping ([Item], Error?) -> Void) {
        // networking code
    }
}
```

or a dummy repository that serves the demo data even in offline mode:

```
struct DummyListRepository: ListRepository {
    func loadList(completion: @escaping ([Item], Error?) -> Void) {
        DispatchQueue.main.async {
            let list = [
                Item(id: "1", name: "First item"),
                Item(id: "2", name: "Second item")
            ]
            completion(list, nil)
        }
    }
}
```

With this setup, the app can be configured to work with either real networking API or with a mocked data, which we also can save in bundled resources instead of hardcoding.

For the above example I should also note that when we're implementing a stub for an asynchronous API call, we should always maintain it's asynchronicity (trigger the callback from inside `DispatchQueue.main.async`). Otherwise, we'll be releasing Zalgo.

You can see the Repository taking an intrinsic part in the Clean Architecture variant I proposed for the SwiftUI apps.

Conclusion

Separation of Concerns is that giant on whose shoulders stands many buzzword patterns we know today. Just that principle alone provides the required guidance for dramatic improvement of the software quality at all levels.

Don't overlook it when writing the code or designing the architecture. Loose coupling and high cohesion are your friends!

Separate the algorithms from the inputs and outputs for better testability, and your software will be rock-solid even without SOLID :)

Subscribe to RSS feed or follow my Twitter for the new articles alerts. And let's connect on LinkedIn as well! Don't hesitate to reach out to me - I'm always ready to help to anybody.

Your appreciation is my inspiration!

Many thanks to people supporting my work: Joseph Goodrick, David Roman, Pavel Sorokin, and folks on Venmo.

My other articles

Save your next app from rebuilding from scratch

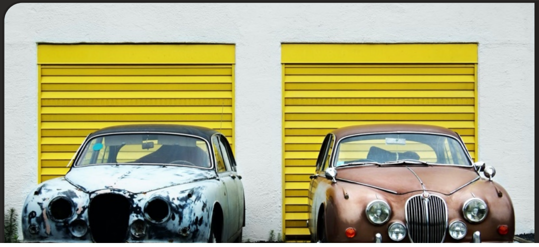
2020, Feb 27 — 7 minute read

Stranger things around SwiftUI's state

2020, Feb 20 — 5 minute read

Why I quit using the ObservableObject

2019, Dec 20 — 8 minute read



Performance Battle: AnyView vs Group

2019, Dec 05 — 9 minute read

Who said we cannot unit test SwiftUI views?

2019, Nov 21 — 12 minute read

Programmatic navigation in SwiftUI project

2019, Nov 08 — 8 minute read

Browse all articles