

Subscribe to DeepL Pro to edit this document.  
Visit [www.DeepL.com/pro](https://www.deepl.com/pro?cta=edit-document) for more information.

Clean architectures

How to decouple business logic from infrastructure details

H1 Clean architectures

H2 How to decouple business logic from infrastructure details

An architecture is said to be clean when it achieves a successful separation between business logic and infrastructure details. Such separation is achieved through layering, strong application of SOLID principles and strict adherence to what we will call "The Dependency Rule". More on this later.

Coupling business logic and infrastructure details has been a problem since we started developing software, it is not new. This movement has not emerged in the last 10 years because of the purism of some developers. No, the senior is not trying to make it harder for you...

This problem already existed when programmers were writing software that read information from punched cards. What do you think happened when they were asked to change the software so that it would now read information from magnetic tape?

**Clean Architecture"** was born in 2012 with a strong influence of **Hexagonal Architecture**, proposed in 2005 by **Alistair Cockburn**.

These architectures generate systems, products, which are:

1. Independent of the framework used.
2. With business rules that must be measurable without relying on user interface, database, web server, etc.
3. Independent of such a user interface. In fact, the latter should be interchangeable without affecting the business rules, the aforementioned business logic.
4. Independent of the database used, if it is necessary to use it, independent of the data persistence system, local or online.
5. Independent, in general, of any external interference.

*In short: Business rules should know nothing of the infrastructure details, so as not to be dependent on them.*

Let's focus on our clean architecture in more detail...

H3 Business logic

The so-called business logic represents the domain. It is a series of rules, actions and data that represent our business, its activities, etc. For example, in a banking application, part of our business logic would be the models aimed at abstracting the user's current accounts, cards, movements, etc. What would also be part of our business logic, all those rules that help us to define it: preventing a user without funds from trying to make a transfer, verifying that the account provided by the user has a valid IBAN, the obfuscation logic so that the full account number is not shown but replacing some digits with asterisks...

*Important: The business logic is common to all departments of the company: marketing, design, product...*

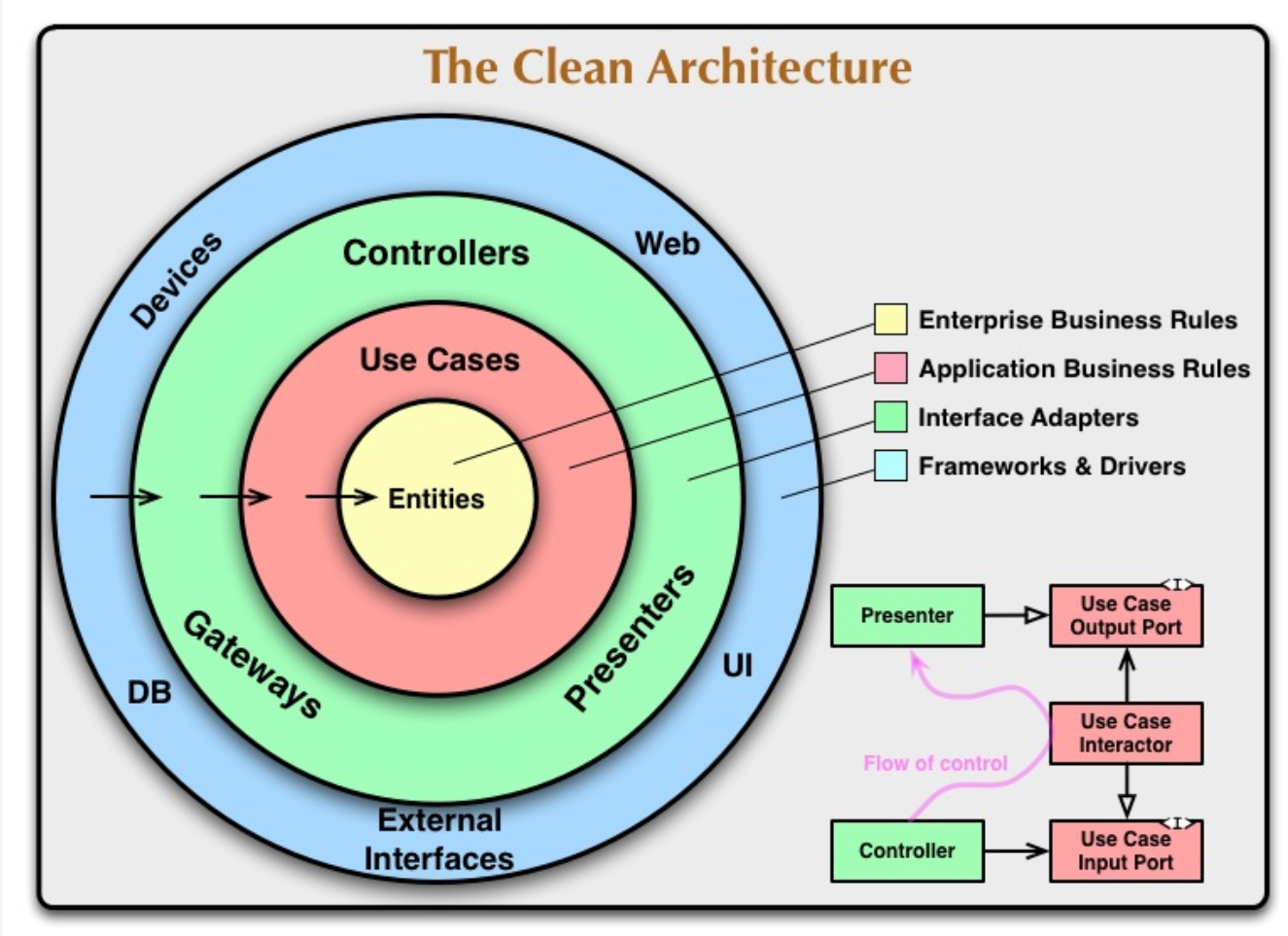
H3 Infrastructure details

Here we refer to the framework we use to design the views of our application, the network library, the dependency injector, the database, the persistence model, etc.

These components "have no life" outside our system, our app, our website. And they should be transparent to the rest of the company's departments.

While the business logic changes little over time, the infrastructure details may change. In an MVP we may decide to use a simple database management system that is not very scalable and quick to implement, as we only want to see if our product has traction in the market. If it does, we may need to change it. Or change the framework through which we present the interface to the client because it has been discontinued.  
  
This is why we said that the business rules should not know anything about the infrastructure details, because of the high possibility that they may change in the future. By implementing a clean architecture, we can ensure that if we need to change any part of our infrastructure, the part to be changed can be replaced by another part without affecting our business logic, thus minimising the impact of such changes.

In the following image, from **Robert C. Martin'**s (Uncle Bob) blog, we can see a representation of layers and responsibilities that should be fulfilled in any clean architecture.

ALT text: layered base representation in a clean architecture

In his reference article, "**The Clean Architecture**", Uncle Bob explains that the number of layers, the circles, are schematic, that there is no reason to have "only" four layers, these would be the minimum. When creating new layers, or splitting any of the current ones, we would only have to take into account that the dependency rule still applies.

Although the author speaks of a minimum of four layers, you will often find projects in which only three are used, while still maintaining a correct separation of responsibilities and complying with his aforementioned dependency rule.

H3 Layers of our architecture

How could we bring this clean architecture to our project with SwiftUI or Jetpack Compose, let's set it up step by step.

H4 Entities

It would contain the most static code, the one on which the rest will depend. All the outer layers will depend on it, they will be coupled to it, so any change in this layer will have repercussions in the rest of the layers of our architecture.

This layer does not have any coupling with an external layer, in fact, it must not know any component created outside its own layer. This is what we call the dependency rule and it is mandatory to keep our architecture clean. And it is not only mandatory for this layer, but also for all the outer layers. What does Uncle Bob tell us about the dependency rule?

*The primary rule that makes this architecture work is the dependency rule.*

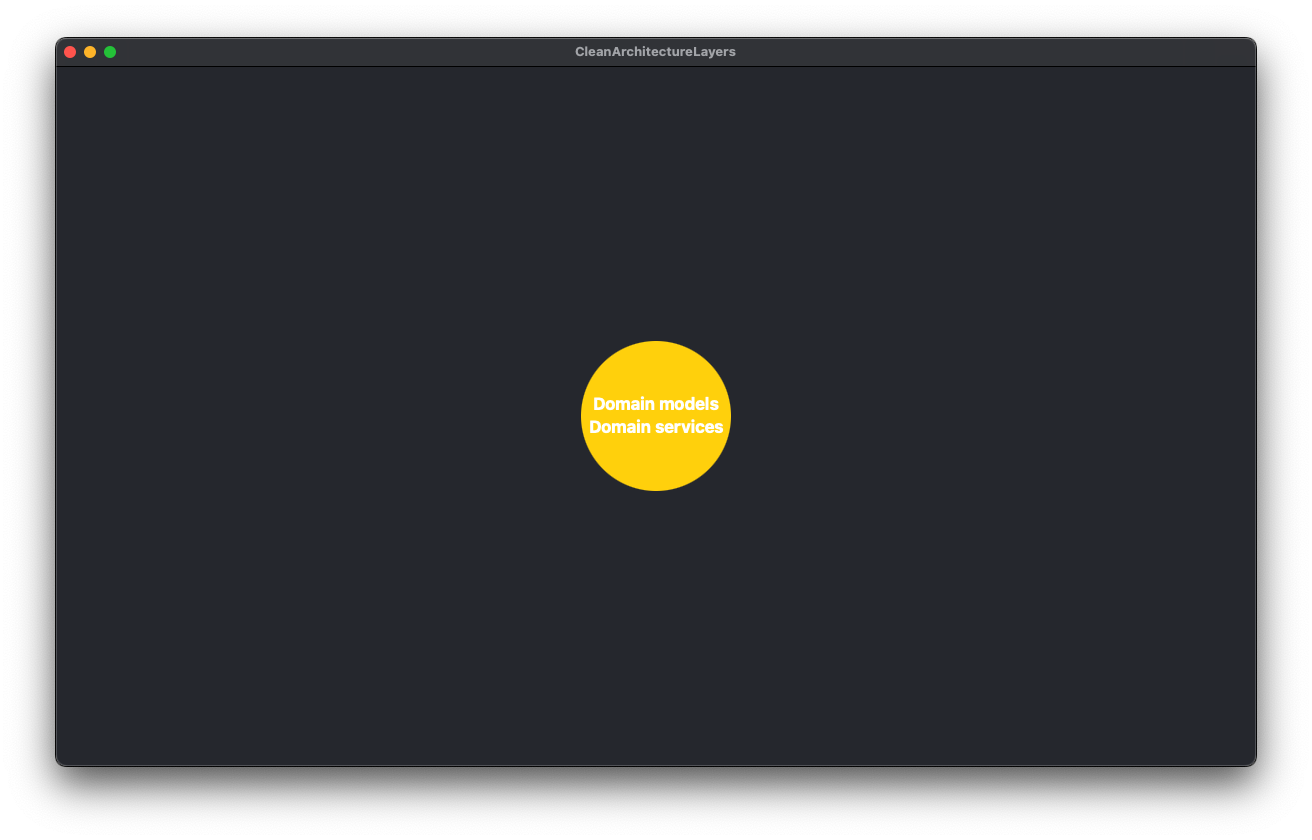
*This rule says that source code dependencies can only point inwards. Nothing in an inner circle can know anything about something in an outer circle. In particular, the name of something declared in an outer circle must not be mentioned in the code of the inner circle. That includes functions, classes, variables or any other software entity. - Uncle Bob*

What, for example, would this Core layer contain?

1. **Domain models:** abstraction of a current account, credit card, debit card, transaction, commission...
2. **Domain services:** classes related to domain models and that help to implement our **business rules, we will talk about them in the next section**. For example, the class that takes care of calculating the fees you are going to pay for a certain operation, the functionality that takes care of verifying that, a priori, there is no problem to make a transfer, the verification that you are not trying to send more money than you have in the account...

*Do not confuse these domain services with a class that, for example, performs a network request to receive approval for a loan, or a transfer, these domain services encompass and abstract our business logic, our rules. Don't worry if you are not familiar with these terms, we will end up understanding them better with the examples.*

You will also find our domain models cited here or in other articles, talks, etc., such as Entities or Data Objects. Let's see a representation of this layer.



ALT text: representation of the entity layer

Does this representation say little to you, maybe it does. But I want you to understand that this is the world for the components of this layer. The components of this layer can only have dependencies between them, both models and services, since they are in the same layer.

As we can see in this representation, our core layer does not know anything outside its layer, there is no lower layer on which it depends and no outer layer of which it is aware.

What kind of coupling could we have in this layer, for example: the ***CreditCard*** abstraction could have a dependency on ***Movements*** as it has to manage a collection of Movements.

H4 Business logic

This layer would contain the business rules specific to our application. By convention, the software we design in this layer is called **Use Cases**. We use them to direct the flow of data to and from our entities.

**Business rules, business rules, what are you talking about, let's see it better with examples?**

* We want the user to be able to make a transfer to another bank account, the business rules could be:
  + Verify that a customer has sufficient funds before attempting a transfer.
  + Verify that the destination account has a correct format (20 digits).
  + Verify that the IBAN of the account is correct (the IBAN is calculated on the basis of the account number).

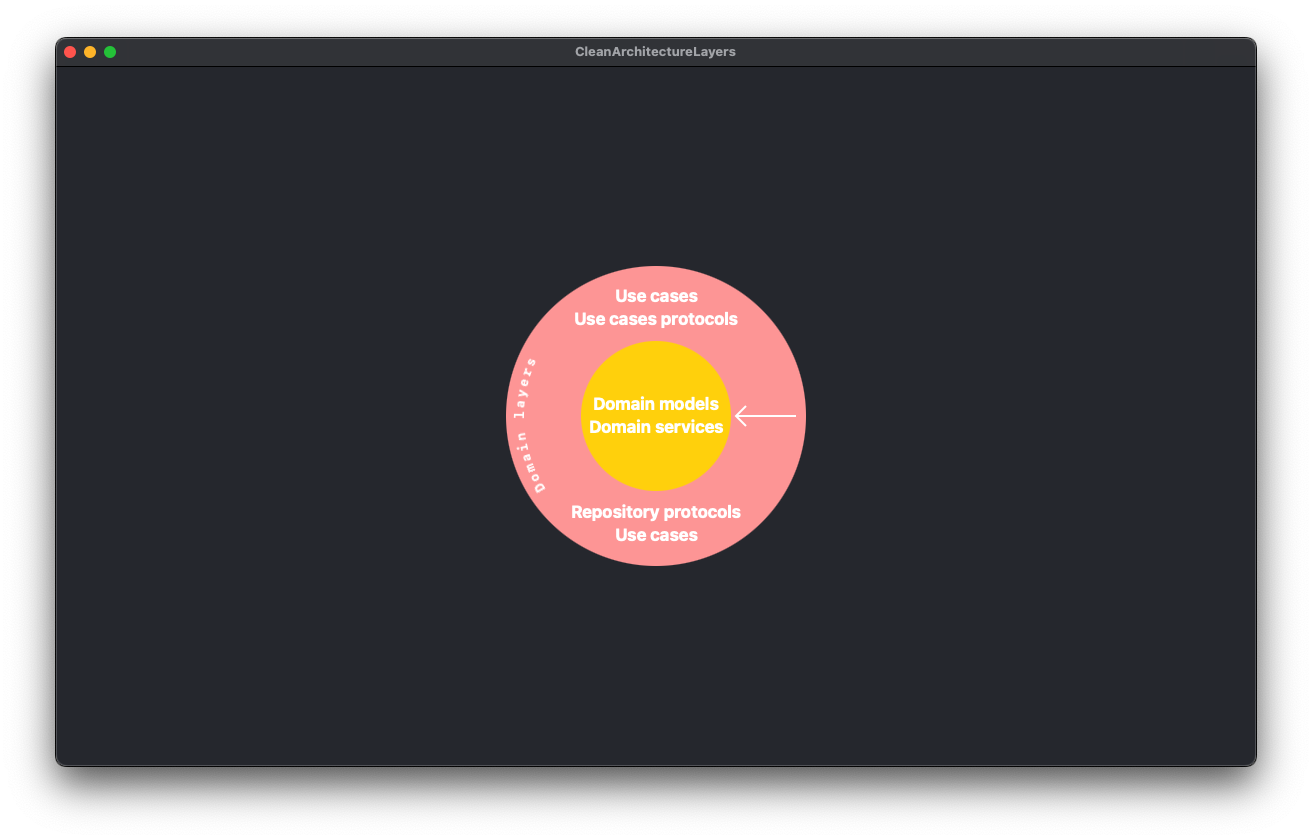
Changes in this layer would not affect our domain models and should not be affected by external changes either. How do we maintain our *dependency rule*? Use cases will establish their *contracts*, *protocols*, *interfaces*, whatever we want to call them, and will be implemented by the components of the upper layer.

These protocols inhabit the same layer as use cases, as they belong to our business logic.

That said, our use cases would have dependencies on:

1. Our domain models and services, which are at a lower layer.
2. Other use cases. One use case may need information obtained by another use case.
3. The protocols, interfaces, which they implement or on which they have dependencies by composition.

What would our layered representation look like?



ALT text: representation of the domain layer

I have drawn the dependence of this layer, with the lower layer, by means of an arrow towards the inside of the graphic. You will see it like this in the rest of the article, and also with the rest of the layers.

H4 Adapters

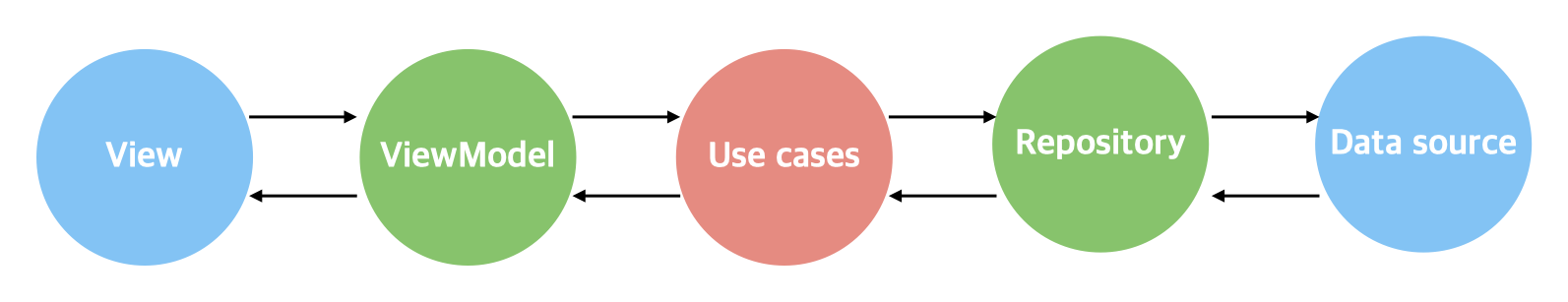
Adapters establish a bridge between the business logic and the infrastructure details. Here are the components that implement the contracts, protocols, interfaces, defined in the lower layer (Use Cases).

This layer is also in charge of "translating", mapping, the data we get from external layers to data in our business logic.

The adapter layer would be divided into several sections, at least these two:

1. **Presentation layer:** where we would find the viewmodels, presenters, etc.
2. **Data layer:** where we would find repositories, datasource protocols, mappers, Data Objects protocols, etc.

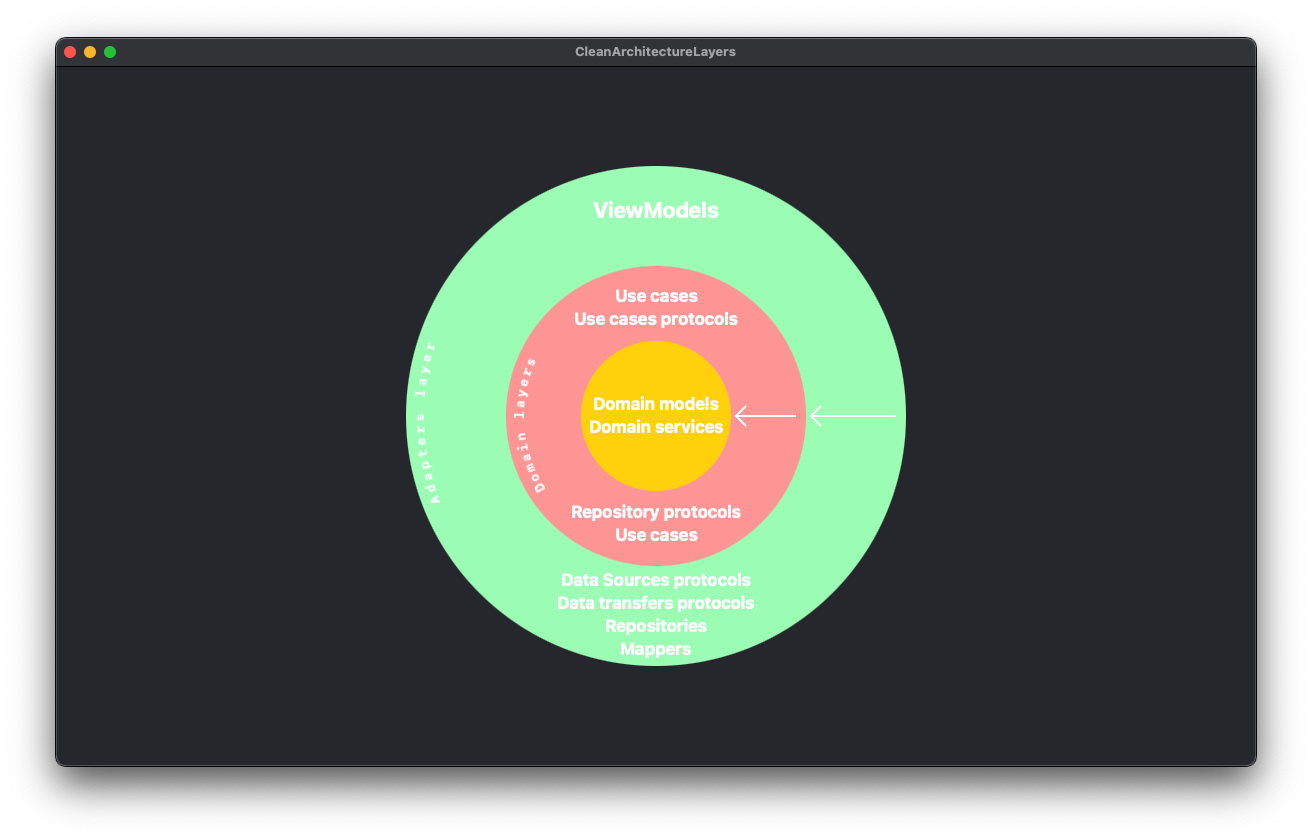
There is an important change from the lower layers. In these layers, all components "knew" each other and could depend on each other. In the **Adaptor** layer this is not possible, and it is very important that this rule is complied with. The components of the presentation layer cannot have any kind of dependency on components of the data layer and vice versa, they cannot be coupled.

A ViewModel could not, for example, instantiate a data layer object, it would have to do it through the use cases, through its lower layer. These are separate flows. Let's help ourselves with a graphical representation:

ALT text: representation of the data flow in the architecture

*The adapter layer could be divided into more parts, sections. I have mentioned these two because I consider that they are the most common to any clean architecture, regardless of the technology of the project, be it web, app, etc...*

*For example, a manager that manages the analytics of our business, application, etc., would also belong to this layer of adapters. And just like the presentation and data layers, it could not have dependencies with them.*



ALT text: representation of adaptor and domain layers

We can see how the adapter layer is coupled to the use cases layer, just as the adapter layer is coupled to the domain layer. Nothing defined here, in the adapter layer, could be "invoked" in the use cases layer. For that we use the principle of dependency inversion, we implement in our components the necessary protocols, which we have defined in the lower layer, and we inject them where necessary.

If you look at it, this layer has dependencies on the use case protocols it has to implement and the data models used by these protocols, in short, on its lower layers. But it does not know whether we are going to make network calls using URLSession or HttpURLConnection, or use a third-party library such as Alamofire or OkHttp. It doesn't know if we will persist sensitive data in UserDefaults - SharedPreferences or use Keychain - KeyStore (this is an example, for sensitive data use KeyStore - Keychain).

This means that this network layer is unaware of these implementations, so it avoids coupling to them. How does it communicate with the higher layers in the same way that use cases communicate with them, through contracts, interfaces, protocols and, again, the magic of dependency inversion.

H4 Infrastructure layer

At the beginning of the article we talked about separating the business logic from the details of the infrastructure. Until now, we had seen this logic through the layers containing the entities and use cases. We had also talked about a layer, adapters, that bridged between this and the infrastructure layer. Let's now talk about the latter.

This layer would contain all the components related to the user interface, for example the views in Jetpack Compose - SwiftUI or XML - UIKit. It would contain the classes needed to make a network call and get the data we need to inject into our repositories so that they, in turn, do so in the use cases. For this we would use URLSession, HttpURLConnection, Alamofire, OkHttp or similar. In case of having to persist data, this layer would be in charge of implementing the functionality through a local database (sqlite, room, coredata, swiftdata...).

Note the difference between this layer and the more internal ones. Here we are already talking about technologies, frameworks, etc. While before we were talking about accounts, movements, cards...

This part, although it may not enter our heads as developers, is transparent to the user and the rest of the company's stakeholders. As much as it pains us, this part is only of interest to us...

This is the part that is most susceptible to change during the life of a project. Through our clean architecture we ensure that a change in any component of this layer does not have any repercussions in the lower layers, in our business logic. If we have to change any of these components, it will be so isolated that the change will be limited to replacing one component with another, without the rest of the components having to undergo any alteration. If we do it right, these components, network, UI, storage, etc., will act as mere plugins to our business logic, replaceable.

Nobody expects to have to modify the wheels of a bicycle, or the braking system, when changing the saddle of a bicycle. Our architectures must work the same way. Again, we achieve this capability of easy substitutability by using SOLID principles and their inversion of dependencies.

Shall we take a look at the final representation of our architecture?



ALT text: graphical representation of the layers of the architecture

*This layer has the same particularity as the adapters layer, although the views are in the same layer as the data sources, they do not "know" each other, they have no dependencies between them, neither by inheritance nor by composition. In short, they are not coupled to each other.*

H3 Advantages and disadvantages

H4 Advantages

**Cost:** A significant improvement in the cost of new features by having the code better structured, more decoupled and with a clearer and broader vision of the elements you will need to implement each of the layers of your architecture.

**Maintainability:** The separation of "concerns", "responsibilities", facilitates the modification of one part of the system without affecting others.

**Testing:** Component independence makes unit testing easier to implement.

**Modularity:** Modular architecture allows new functionality to be added without major changes to existing code.

**Adaptability:** It is easier to adapt the application to new technologies as changes in the infrastructure details will not affect the business logic, domain layer, etc.

**Code reuse:** Independent components can be reused in other projects.

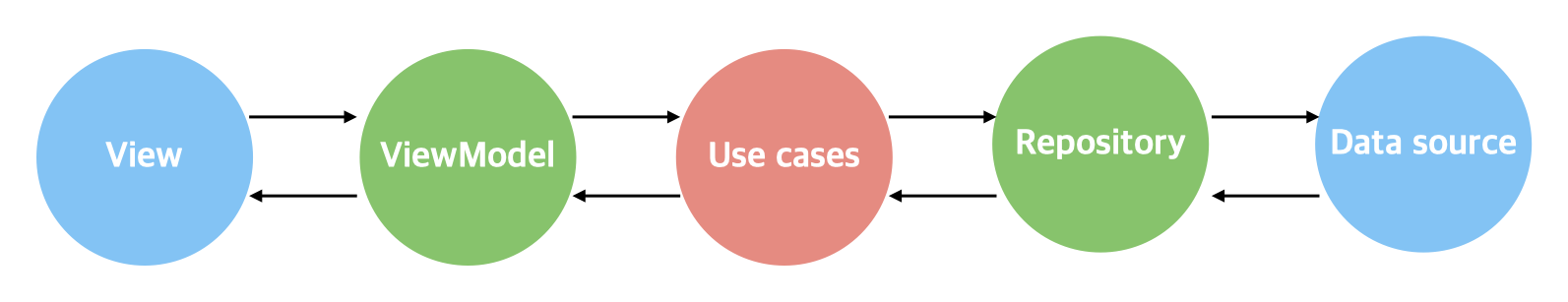
H4 Disadvantages

**Learning curve:** It can be difficult for junior developers to learn and understand.

**Initial overhead:** Initial configuration and structure can be more complex and time consuming than simpler architectures.

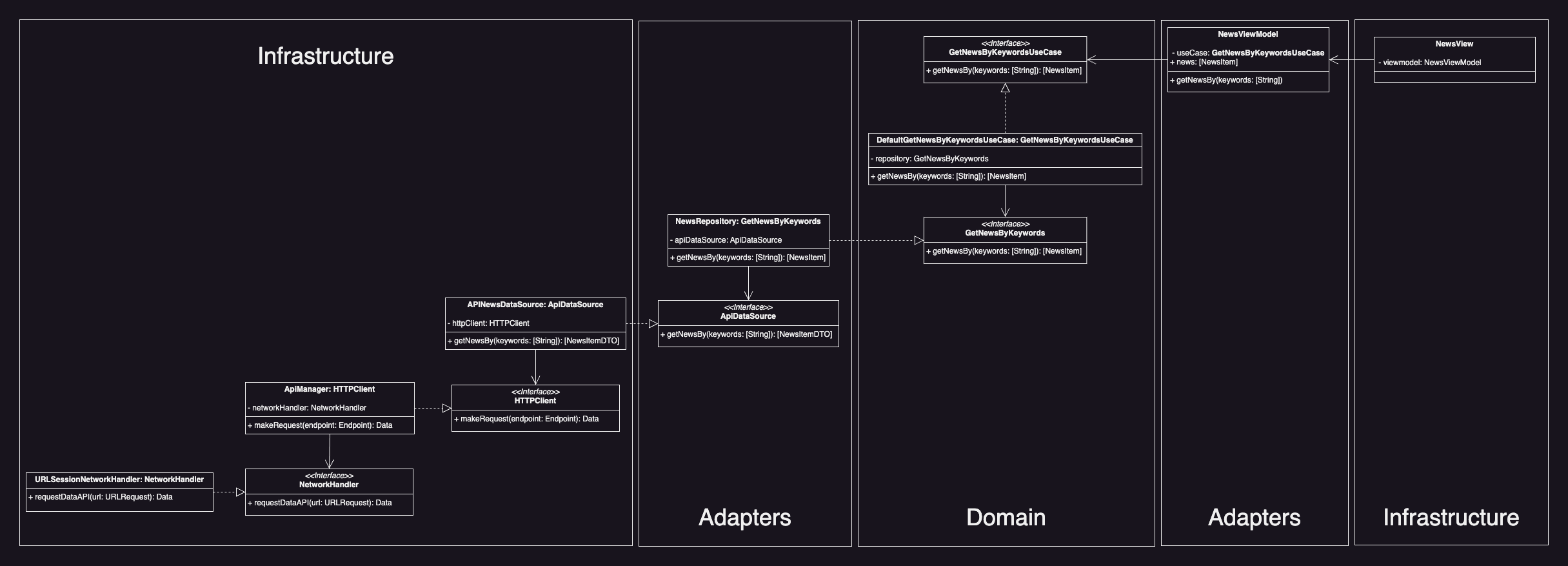
**Difficulty of implementation:** Requires experienced developers and a disciplined team that follows established practices and principles.

H3 Flows and responsibilities

When, in this same article in the Adapters section, we talked about the flow of information, we showed this image:

ALT text: representation of the data flow in the architecture

Let us now see, using a UML diagram, a simplified summary of a possible clean architecture for a particular use case. Let's imagine that we want to create a news application. One of our main use cases is to get a list of recent news based on keywords, how could we split it up, let's see an example:



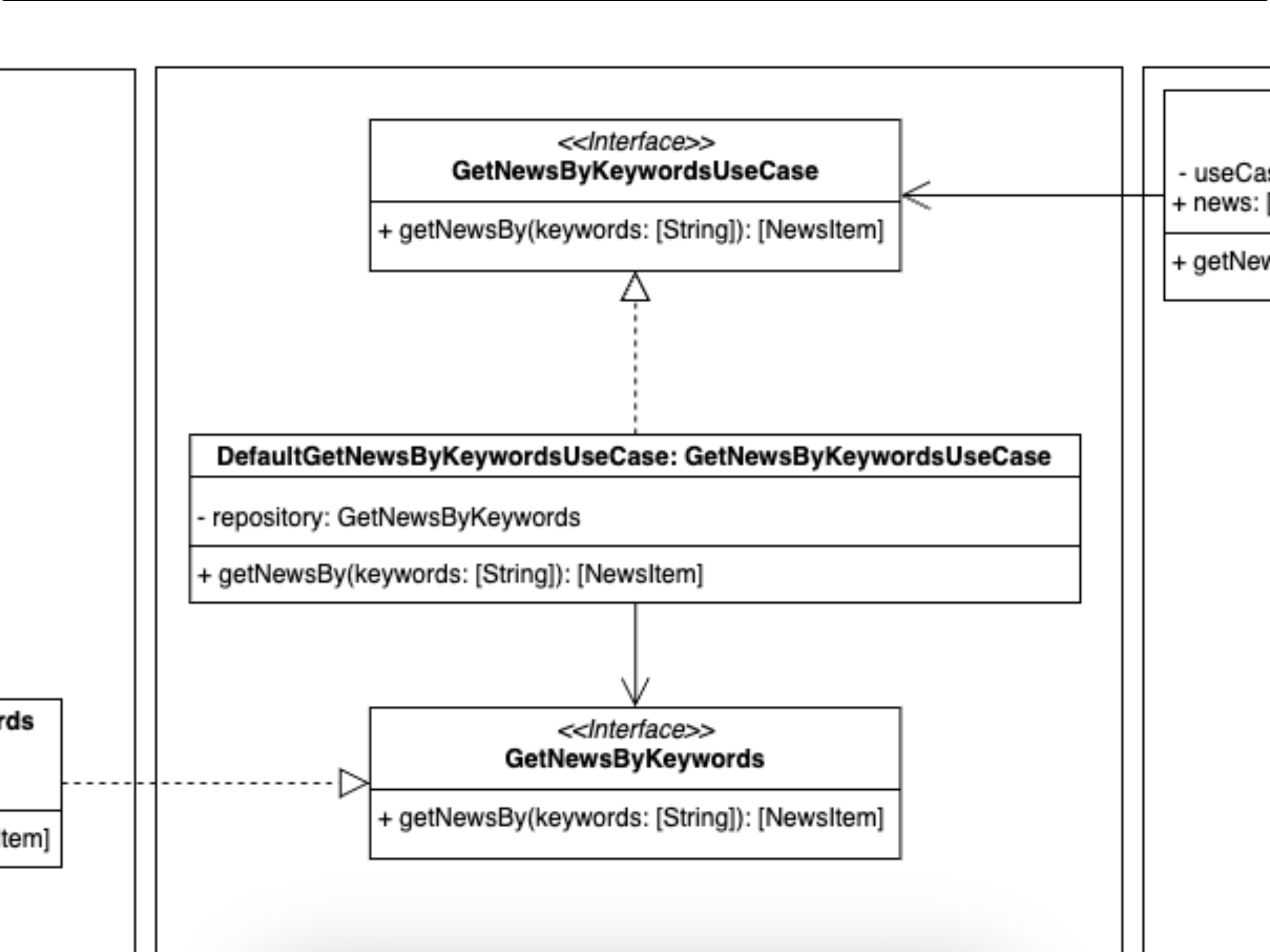
ALT text: representation of the components of the example architecture

**Important, based on the separation shown in the image above**:

*There is no single way of structuring a clean architecture; depending on the developer's experience, the requirements of the application, the tools to be used, etc., we may find one way or another. For example, DataSources, of which you will see an example below, some developers "implement" them in the adapter layer while others do it in the infrastructure layer. As this is a "logical" separation, it may not have any repercussions beyond being located in one hierarchical folder structure or another. As long as the dependency rule is respected, there should be no problem. In my humble opinion, if you know infrastructure information, such as very specific data on how to set up the api call endpoints, it should be in infrastructure.*

*And if instead of partial boundaries we use architectural boundaries, which we will talk about later, my choice would definitely be to implement them in the infrastructure package.   
  
Don't worry if any of these concepts are not clear to you, they will all be defined throughout the article.*

H4 Use case



ALT text: representation of the components of the part of the use cases

We are in the domain layer, we have created our **GetNewsByKeywordsUseCase** use case, what is its responsibility, this use case is in charge of preparing the data to correctly apply the business logic. For example, it could receive the data and, in order to comply with this logic, sort it based on a certain property: name, publication...

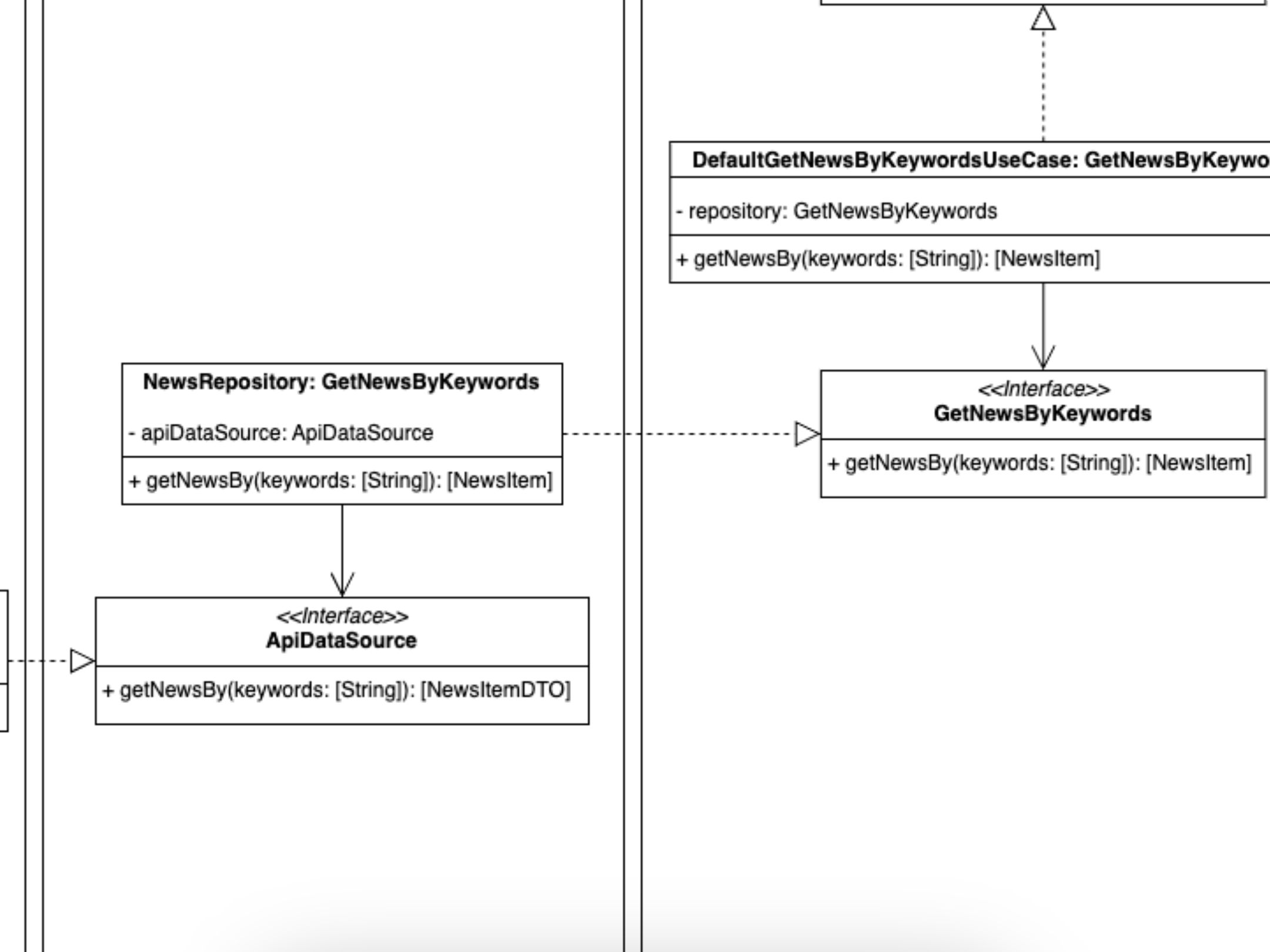
It could also take care of filtering the articles to exclude those whose content is deemed inappropriate, etc. In short, once the data had been obtained, it would be ready to be returned to whoever required it.

* Wait a minute, don't you have too many responsibilities, do you have to take care of receiving the data and prepare it based on our business rules...?
  + Well seen, but for data collection it relies on what we conventionally call a repository. It is from this repository that we will request the corresponding data, delegating this responsibility to the repository and maintaining our principle of single responsibility. Furthermore, our dependency on the repository is inverted, so we specify in an abstract way to the upper layer what we need, but without specifying any kind of implementation. In this way we depend on an abstraction, not on any concrete implementation that breaks our dependency rule.

*In the SOLID* ***Inversion*** *principle* ***of dependency****, it is often referred to that high-level modules should not depend on low-level modules and that both should depend on abstractions.*

*By high-level modules they refer to our domain logic, our business rules, our entities. They are those that are as far away from the input - output of data as possible. It tells us that these modules should not depend, for example, on infrastructure layer components such as the user interface or the database.*

H4 Adapters -> Repository



ALT Text: representation of the components of the adapter part and their repositories

We move on to the Adapters layer and find a class that implements the protocol, the interface, **GetNewsByKeywords**, which tells us that it can be injected as a dependency in the GetNewsByKeywordsUseCase class. And in this way we comply, once again, with our dependency rule. GetNewsByKeywordsUseCase doesn't "know" anything about its parent, it doesn't "know" anything about **NewsRepository** or **ApiDataSource**. It doesn't even need to, and that's great, because it's not coupled to anything in this Adapters layer. Any change you make to this layer may affect components in its own layer or in its shallowest layer, infrastructure, but never in the domain layer. It's not magic, it's good architecture.

In this layer, adapters, are the components that are in charge of converting the data provided by the infrastructure layer, conventionally called DTOs, into the data models required by our use cases, normally called Entities, Data Models, etc. Conversely, these components will be in charge of creating, by means of the domain data models, the most convenient data for the database, api rest or similar.

Responsibilities of this NewsRepository? class, to get the data from the external source it touches, in its models, and convert them into our domain models. We usually call this operation mapping.

Again, two responsibilities? no. To obtain the raw data from the service, database, local storage, etc., it relies on a DataSource, a class that takes care of that responsibility. And although I haven't specified it so as not to add more complexity to the example, also the mapping process would be delegated to another class...

*For more information on this type of classes, you can review the* ***Facade*** *and* ***Adapter*** *design patterns, as they share similarities with both of them.*

How do we realise the relationship between the **NewsRepository** class and the components of its outermost layer, also through dependency inversion. The **ApiDataSource** interface declares what the **NewsRepository** class needs and "forgets" about its upper layer, it is not your problem.

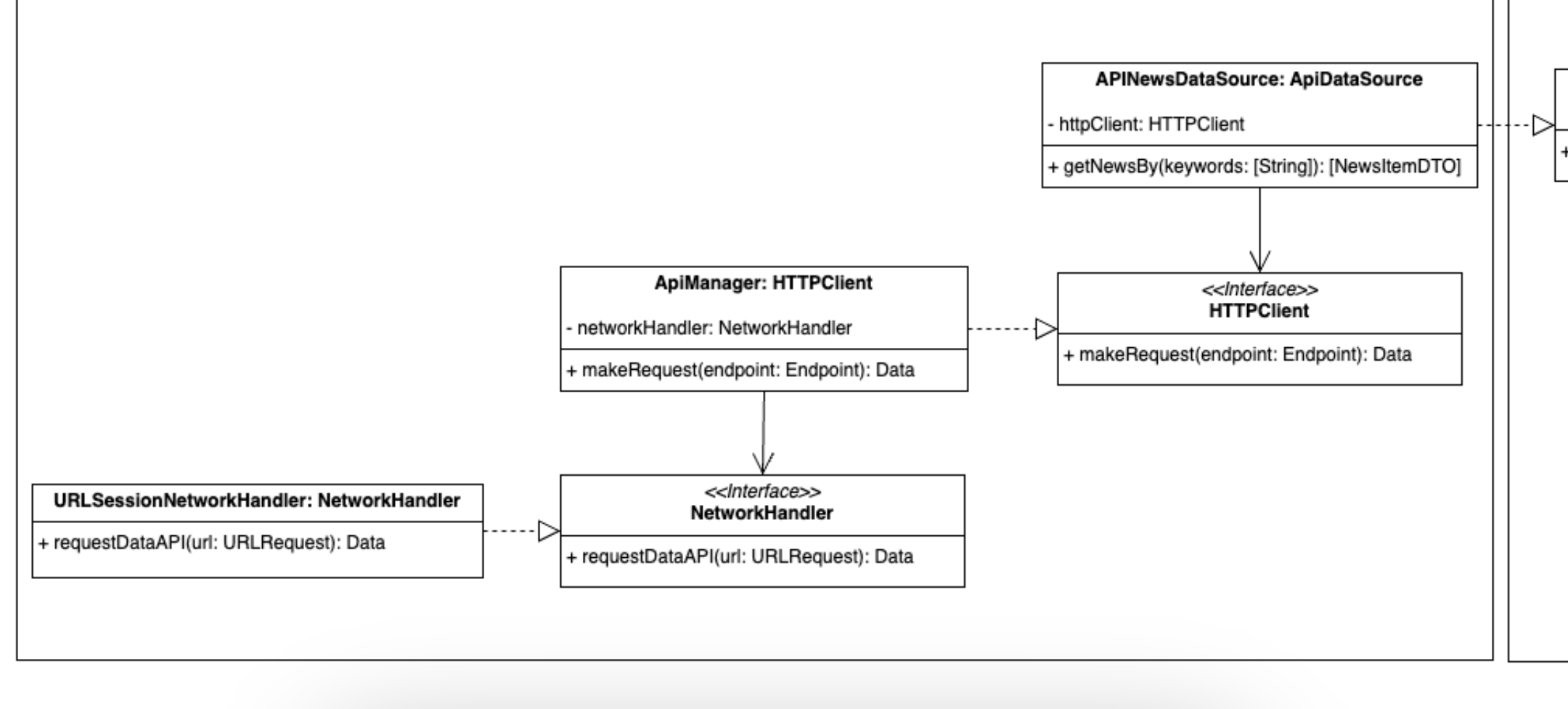
The upper layer will have the responsibility to create a component that can cover the need imposed by the **ApiDataSource** interface. And this component, whatever it is, can be injected as a dependency in **NewsRepository**. But **NewsRepository** will not know which concrete implementation is being injected, so it will not be coupled to it and a future modification of it will never be able to affect it. It won't be able to affect anything in the domain layer, no use cases, no entities. I'm repeating something I've said three paragraphs above, because it's vital that it's understood!

***Said Rehouni****, on his [youtube channel](https://www.youtube.com/@SaidRehouni) and his [course on clean architecture](https://www.youtube.com/watch?v=nILL3UXrMS0) warns against the creation of massive repositories in which the resolution of the needs of all the use cases associated with the same feature are accumulated. He talks about how this makes it more difficult to test these repositories, mock them up, and how they violate the principle of single responsibility.*

*This principle tells us that a class should only have one reason to change. If such a repository is managing all the use cases of the same feature, with its different data models, etc., it is very likely that it is susceptible to change with each of the modifications that can be made to them.*

*For these cases, it is recommended to create different* ***repositories*** *to serve the same feature.*

H4 Infrastructure -> Data



ALT text: representation of the data part components

Let's start talking about **APINewsDataSource**. This is a class that implements the **ApiDataSource** interface so it can be injected in **NewsRepository**. Among its responsibilities is to create the Endpoints that will be used by the client (URLSession, HttpConnection, Alamofire, OkHttp...) to obtain the data that it needs to "serve" in its fulfilment of the interface. It also takes care of converting the raw data received by the backend into our data objects (the DTOs expected by the repository).

We seem to be back to the common pattern, a class with more than one responsibility. And again we solve it in the same way. **APINewsDataSource** relies on the HTTPClient interface as a dependency. **APINewsDataSource** will take care of generating the endpoints mentioned in the previous paragraph and will delegate the obtaining of the data corresponding to them to the component injected by means of dependency inversion. In the same way, it will obtain, by means of dependency inversion, a component that will be in charge of deserialising the raw DATA in the corresponding DTOs. Again, I did not want to add these components to the UML diagram to simplify its understanding.

Importantly, if **httpClient** here were not an abstraction, but a concrete class with the same responsibility, we would not be in breach of the dependency rule since this class would be within its own layer, the infrastructure layer. But by adding, once again, a dependency inversion, we achieve that, in the event of having to change something in the concrete class, this change does not affect **APINewsDataSource**.

We are creating a new layer, although it would be more like a sub-layer, but in terms of benefits it is the same thing. APINewsDataSource is not being attached to a specific http client, so in the future, this component can change without affecting the components of its underlying "sub-layer".

Let's talk about **ApiManager**. This is a software component whose responsibility is to generate the appropriate network request, which can be a **URLRequest** in iOS or an OkHttp **Request** for Android, and make the call to the backend. In this case it will be supported by a component, which we have called **networkHandler**, which will make the request based on the request created by **ApiManager**. Obviously, this **networkHandler** is an abstraction that will be injected as it is the perfect candidate to change in the future.

In the outermost part of our diagram we find a class, **URLSessionNetworkHandler**, which would implement the NetworkHandler protocol and whose responsibility would be to make the http request based on the request received. I have called it **URLSessionNetworkHandler** but it could also have been called **HttpConnectionNetworkHandler** or **OkHttpNetworkHandler**, etc...

It is in this type of components that we can most easily see the performance as simple "plugins" of the infrastructure layer components. And we can see all the advantages of a good abstraction and separation of responsibilities.

Why is it so important that these behaviours are abstracted in their own adapters and placed in the most external part of our architecture, let's see it with an example:

Imagine that we decide that each viewmodel, using **URLSession** or the library of the moment, will be in charge of the calls to our API-REST. How many viewmodels can a medium-sized application with an MVVM architecture have? Can you imagine the cost of a possible **URLSession** deprecation?

And, mind you, the case of **URLSession** would be the least serious of all, since it is a library that is native to the system. Can you imagine the cost of changing the library in question if a change of Swift version, for example to Swift 6, were to cause it to stop working?

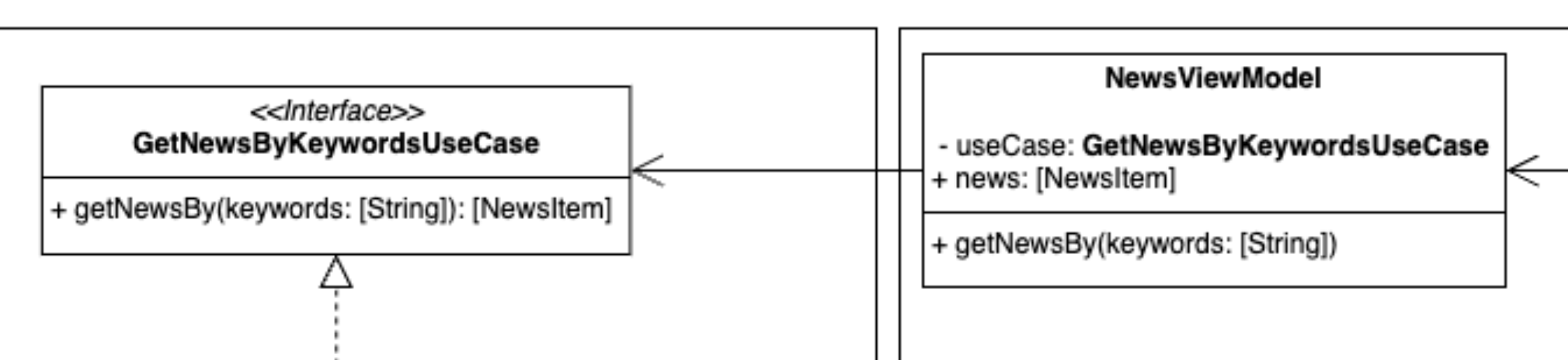
It is not a very remote case, I have worked with libraries that did not work with the latest version of Xcode, others whose components ended up coinciding with internal components of the language itself, and so on.

A good architecture will not mean that you will not have to replace that component, but it will considerably reduce the cost of replacement.

And we can go a little further, excessive coupling can make a change of framework almost impossible. For example, there are backend frameworks that, in order for your data models to be correctly interpreted, require them to inherit from a base class of the framework itself. In Uncle Bob's words, there is no stronger coupling than inheritance. Imagine, again, wanting to change the library by which you build your endpoints when you have each and every one of them tightly coupled to the framework...

OK, let's go back to the adaptor layer, but let's see what we have on the right side of the use cases, let's talk about the presentation layer...

H4 Adapters -> Presenters



ALT text: representation of the components of the presenters' side

On the other side of the data flow are the **presenters**. As they did in the repository layer, here they also act as intermediaries between the use cases and, in this case, the user interface.

The **presenters** receive the data from the use cases and prepare it to be "consumed" in the corresponding user interface, in our case SwiftUI views or Jetpack Compose Composables. This ensures that the view receives only the data it needs and in the format it needs it. In an MVVM architecture the role of the **presenter**, and its responsibilities, is taken over by the ViewModels.

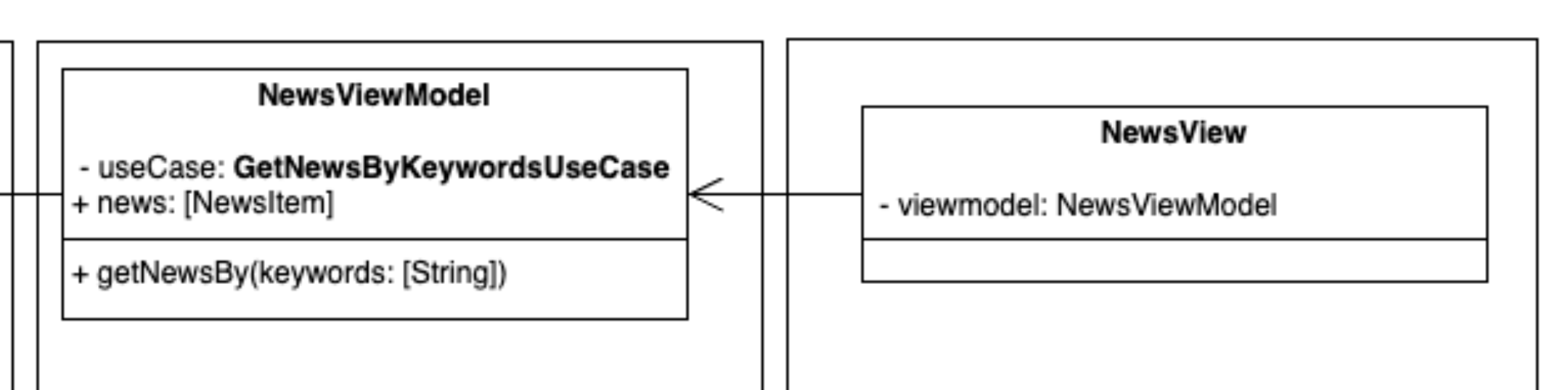
In order to continue complying with the rule of **depending on abstractions and not on implementations**, the dependency of the viewmodel on the use case also depends on an abstraction, not on a concrete implementation of the use case.

For working with these presenters we find the use of a pattern known as the **Modest Object Pattern**. According to **Uncle Bob**, in his book **Clean Architecture**:

*"The modest object pattern is a design pattern originally identified as a way to help unit testers separate hard-to-test behaviours from easy-to-test behaviours. The idea is very simple: it separates behaviours into two modules or classes. One of these modules is modest; it contains all the hard-to-test behaviours, reduced to their most basic essence. The other module contains all the testable behaviours that are extracted from the modest object."*

***xUnit Patterns****, Meszaros, Addison-Wesley, 2007, p 695.*

In other words, it is a concept that is applied in software design to improve testability and separation of responsibilities. In the context of a Jetpack Compose or SwiftUI application, our **Rich Object** would be our viewmodel, which would handle the business logic and whose code would be very easy to test independently. Any flag, publisher, routing, etc., that you want to test can be done much more easily in a viewmodel than in a view, so this, and its logic, is reduced to its minimum expression, acting as a **modest object**.

H4 Infrastructure -> UI

ALT text: representation of the UI part components

To complete our architecture we would reach the infrastructure layer on the user interface side. In this example we would deal with a SwiftUI View or a Jetpack Compose Composable. Both acting as the **Modest Object** of our pattern, quoted in the previous section.

Its functions would be to render the user interface and "capture" the user's action events, but then delegate their "consequences" to the innermost layers.

And with this we would have completed our clean architecture for a particular **feature.**

In the next section we will discuss some of the issues to be assessed.

H3 Problems and ruptures in architecture

Talking about clean architectures with one of my colleagues, Álvaro, we were wondering what to do with all those components that we reuse from one project to another. We gave the example of the typical wrapper for working with **Keychain**, which does not have to implement a specific protocol as it is only used as a helper in different projects.

*For those unfamiliar with Keychain, it is a password management and secure storage system provided by Apple. It is an API and storage service that allows applications to securely store sensitive information such as passwords, cryptographic keys, authentication tokens and other private data.*

From my point of view, I think the right thing to do would be to create an extension for this helper. This extension would be the one that would implement the necessary protocols to make our component injectable where necessary without having to break the dependency rule. In this way the Keychain Manager, or whatever you want to call it, would have been extended without modifying its original functionality and without affecting it, thus fulfilling our SOLID Open-Close principle.

Another point we discussed was whether to have a package of Swift extensions with different functionalities, and if so, do we use it in our domain layer? If we have it, do we use it in our domain layer, are we not breaking the dependency rule, do we declare the Swift extensions in our domain layer, thus complying with the dependency rule but making these extensions not portable, at least easily, to other projects...?

While documenting for this article I found the architecture break that I consider the most difficult to circumvent, that of the viewmodels in SwiftUI, let me explain:

In the GUI part, where the use case, the viewmodel and the view are located, the dependency rule is still strictly adhered to, in particular:

* The use case, in the domain layer, is not aware of the ViewModel and has no dependency on it.
* The ViewModel, in the adapter layer, does not know the View, it has no coupling to it.
* The View, in the infrastructure layer, uses the ViewModel it has in a more internal layer, so there is no problem.

But we do encounter a problem, at least in the case of a SwiftUI implementation. And that is the reactivity framework.

In early versions of SwiftUI it could be **Combine**, **Observation** for iOS 17 onwards or even, Steve Wozniak forbid, a third-party library.

The question to be considered is that this ViewModel in the Adaptor layer should not be coupled to an infrastructure detail. Could this be solved, yes, an intermediate component could be created that does not contain information about the reactive framework.

This component would use the use case to get the information, how would it communicate with the ViewModel, again through dependency inversion.

I can think of several ways, for example, using the delegate protocol pattern, making the ViewModel implement it. ViewModels, in SwiftUI, are classes, so it would be no problem for the composition root to use a reference to that ViewModel and inject it as a delegate.

We could also use the **observer** pattern, again a combination of steps by reference and protocols would make it possible to solve the problem.

But in this case I have chosen not to add it, why, as an architectural decision, I have not thought it appropriate. Or rather, I didn't see enough value in it to compensate for the extra complexity.

And this is where the bad faces come in, how much of a purist do we want to be, or as they say in my village, do we want to be more papist than the pope?

Every developer in charge of architecting a project will be faced with these decisions and will have to consider many things before choosing one path or the other. For example:

* **Domain complexity:** systems with complex business logic may require more layers to ensure a clear separation of responsibilities and ease of maintenance.
* **Volume and variability of requirements:** projects with changing or evolving requirements can benefit from a more modular architecture that allows rapid adaptations without affecting other parts of the system.
* **Experience and skills of the development team:** a simpler design may be preferable if the team does not have deep experience in multi-tier architectures.
* **Available development time:** the availability of development time may influence the complexity of the chosen architecture.
* **Budget:** the financial resources available may determine the tools and technologies accessible, as well as the possibility of implementing a more complex architecture.

And here you will allow me to digress and use a sporting simile. In volleyball, many coaches tend to make the mistake of adapting their teams to a specific tactical system when the most logical, useful and efficient thing to do is to adapt that tactical system to your team, thus enhancing its virtues and trying to correct, as far as possible, its defects.

I am a firm believer that the same is true for architecture. Within minimum quality standards, possibly the best architecture is the one that best suits your specific project, requirements, team, time availability, etc.

For example, [Julio César sar FernáFernández](https://www.linkedin.com/in/jcfmunoz/)of [Apple Coding](https://acoding.academy/)presented in one of his videos the architecture he considered best suited to a current project with SwiftUI.

He defined the basis of what he called his Apple Coding Clean Architecture, again based on his experience as a developer in Apple environments. Experience which, by the way, is no small amount.

Obviously he had to put up with the usual comments about it not being a clean architecture, breaking the rules, etc.

So rest assured that whatever you do, you will be criticised. Your perfect architecture will never be perfect for everyone.

Some of the patterns and behaviours defined here are already in use, often without knowing their **artistic name**. In my case, for example, I did not know **The modest object pattern**. But thanks to a debate with a **SNGULAR** colleague, Jorge Marciel, we used it some time ago in the implementation of a view and a viewmodel. None of us knew the pattern, but we thought it was the best way to solve that problem.

H3 Architectural Boundaries, Partial Boundaries and Boundary Interfaces

When implementing a clean architecture, it is necessary to know the difference between architectural boundaries and partial boundaries.

Architectural boundaries are the borders that separate the different layers of the system.

These require polymorphic Boundary interfaces, input-output data structures and, in short, all those components that make these layers independently compileable.

Partial boundaries refer to a separation of responsibilities within the same system, even in layers, but in a more granular and localised way. They do not necessarily require the complete division of layers as architectural boundaries do. When in doubt as to whether or not the creation of such an architectural boundary is necessary, the developer may choose to include a partial boundary, and may transform it in the future if necessary.

Obviously this type of boundary does not grant independence and any changes made to these boundaries affect the rest, having to be recompiled, etc. To convert these partial limits into architectural ones, we would have to use packages, libraries such as **CocoaPods**, **Carthage**, etc.

*Boundary interfaces are a concept specific to clean architectures that arises when defining the interactions between different layers across different architectural boundaries. They define the contracts that will regulate the communications between the different layers, respecting the dependency rule through inversion of control, i.e. the component that implements the interface is injected into the layer that defines it. We have several examples in the article, for example the* ***ApiDataSource*** *interface.*

H3 Architectural Boundaries, Partial Boundaries and Boundary Interfaces

In the context of Clean Architecture and modular software design, there are different ways to organise the packages or modules of a project. The three most common ways are to organise packages by **layer**, by **function** and by **component**.

H4 Package per layer

This is the one shown in the example in this article. As a design method it is the simplest and, according to Martin Fowler in "Presentation Domain Data Layering", a very good way to start a project.

This is the most traditional approach and is aligned with the layered architecture. The code is organised according to the logical layers of the application: domain, use cases, adapters, presenters, infrastructure, etc.

By implementing this design we would get a package for each of the layers that are divided by architectural boundaries, being able to leave partial boundaries within each of the packages.

H4 Package per function

In this approach, code is organised around application-specific functionality or features. Instead of grouping classes by their role in a specific layer, they are grouped by the functionality they provide.

By implementing this design, we would obtain a single package for each **feature** of the project, thus achieving a high cohesion within the project.

H4 Package per component

This approach can be seen as a mixture of the two previous approaches, as each component can have its own internal layered or feature structure. Here the code is organised into self-contained components that represent reusable subsystems or modules within the application.

To see it with an example, in the case we have been looking at during the development of this article, we could have two packages:

* A package with domain logic, adapters and infrastructure for the entire data part.
* A second package with the presentation and infrastructure layer of the view.

In this way we could encapsulate all the part of the first package that we are not interested in showing by privatising the visibility of the components, leaving the minimum necessary for public access.

The packages per layer, function and component are much more extensive than the simple summary I can provide here. There is probably enough content for another article as long as this one. I am satisfied that you know that there are at least these three types and that you can go deeper into them if you think it is convenient.

H3 Conclusion

In this article, we have explored the key concepts of clean architectures and how to apply them in a project. We have seen how the separation of responsibilities and decoupling of the different layers of the application can lead to more maintainable, scalable and flexible code.

While implementing clean architecture may seem complex at first, it is important to remember that it is an iterative process, requiring time and practice to master. However, the payoff is valuable: a clean and maintainable code base can save time and effort in the long run, and allow us to focus on creating innovative and effective solutions.

In short, clean architecture is a practice that allows us to create robust and scalable applications. I hope this article has been helpful to you, and that you have learned something new and valuable to apply in your future projects.

H3 Documentation

The Clean Code Blog, The Clean Architecture  
<https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

Clean Architecture course on iOS - Said Rehouni  
<https://www.youtube.com/playlist?list=PLy4xaLa5b6WPoXzJIwbrjQvbT5sPDJy3M>

Developing iOS applications with Uncle Bob's Clean Architecture

<https://medium.com/@jonas.schaude/developing-ios-applications-with-uncle-bobs-clean-architecture-572084853cdc>  
  
Clean Architecture and MVVM on iOS

<https://tech.olx.com/clean-architecture-and-mvvm-on-ios-c9d167d9f5b3>  
  
Clean Architecture, Author : Robert C. Martin

<https://anayamultimedia.es/libro/titulos-especiales/arquitectura-limpia-robert-c-martin-9788441539907/>

Talks and debates:  
Pablo García, Jorge Marciel, Álvaro Sánchez, Alberto Díaz, Tony Martínez