JULY 26, 2018 / #CLEAN ARCHITECTURE

# A TypeScript Stab at Clean Architecture

By Warren Bell

#

## Clean Architecture

There are many videos and articles explaining clean architecture. Most of these go over the concepts from a 20,000 foot view. I don't know about you, but I don't learn things very well at that elevation.

## Bob's Your Uncle

The term "Clean Architecture" was made popular by Robert Martin (Uncle Bob) and his book "Clean Architecture: A Craftsman's Guide to Software Structure and Design." Now I don't proclaim to be an expert in this field and I haven't read his book, though I intend to. But I can completely relate to the problems it is trying to solve.

How do you write a software system that is not dependent on anything other than a primary language ? We were promised this in the past with interfaces and other OO principles, but I had never before seen a "clean", pun intended, explanation on how to do this regarding the whole system. And yes, I am a bit late to this party, being that Uncle Bob started to talk about these concepts in 2012, which is a century ago in software years.

## The Diagram That Baffled Me

Here is the original diagram Uncle Bob and others used in their presentations when explaining Clean Architecture. This simple little diagram became an obsession of mine. I had long ago purged my memory of anything UML related and was struggling with the has-a, and uses-a relationships indicated by the open and close arrow heads. The only way I was going to figure this out was by writing some code.

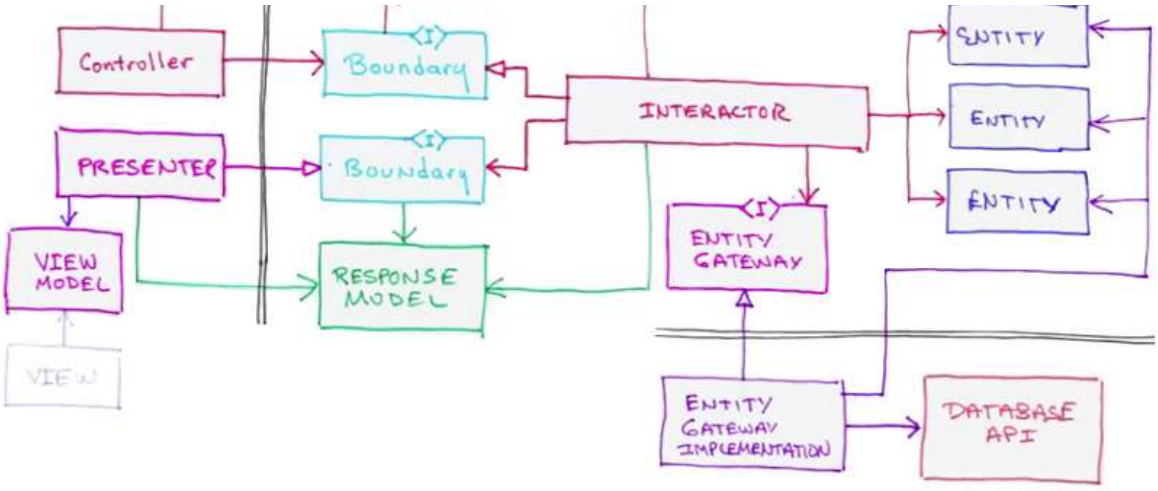
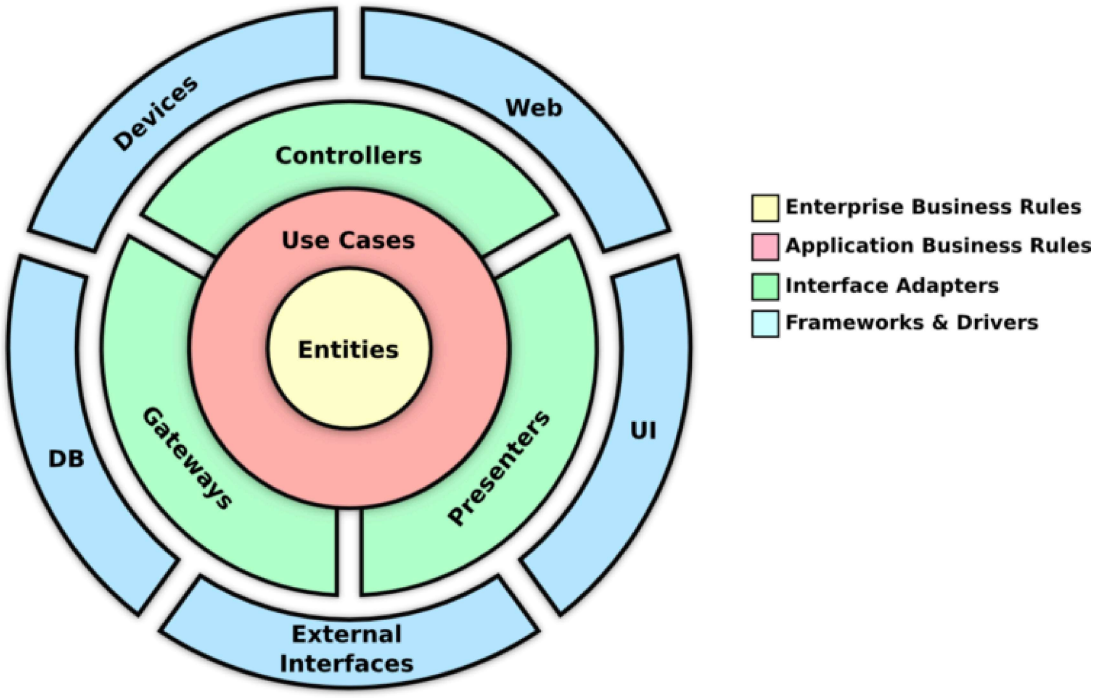



*Image Credit: Uncle Bob*

## Know Your Onions

One way to look at Clean Architecture is as an onion with layers. All layers can only depend on a layer that is closer to the center. That is, all dependencies point inward and not outward.



## One of These Days, I'm Gonna Get Organized

In our example, there are 4 modules that correspond with each layer of this onion. Eventually these could be separate npm modules. For readability's sake, I tried to name things according to Uncle Bob's original Clean Architecture diagram at the top of this article. In the real world you would probably preface all names with what ever use case they represented.

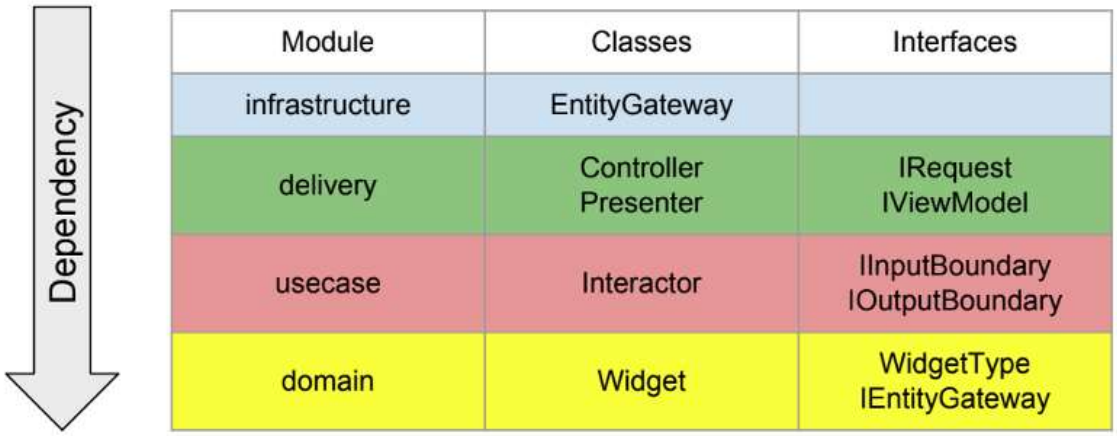
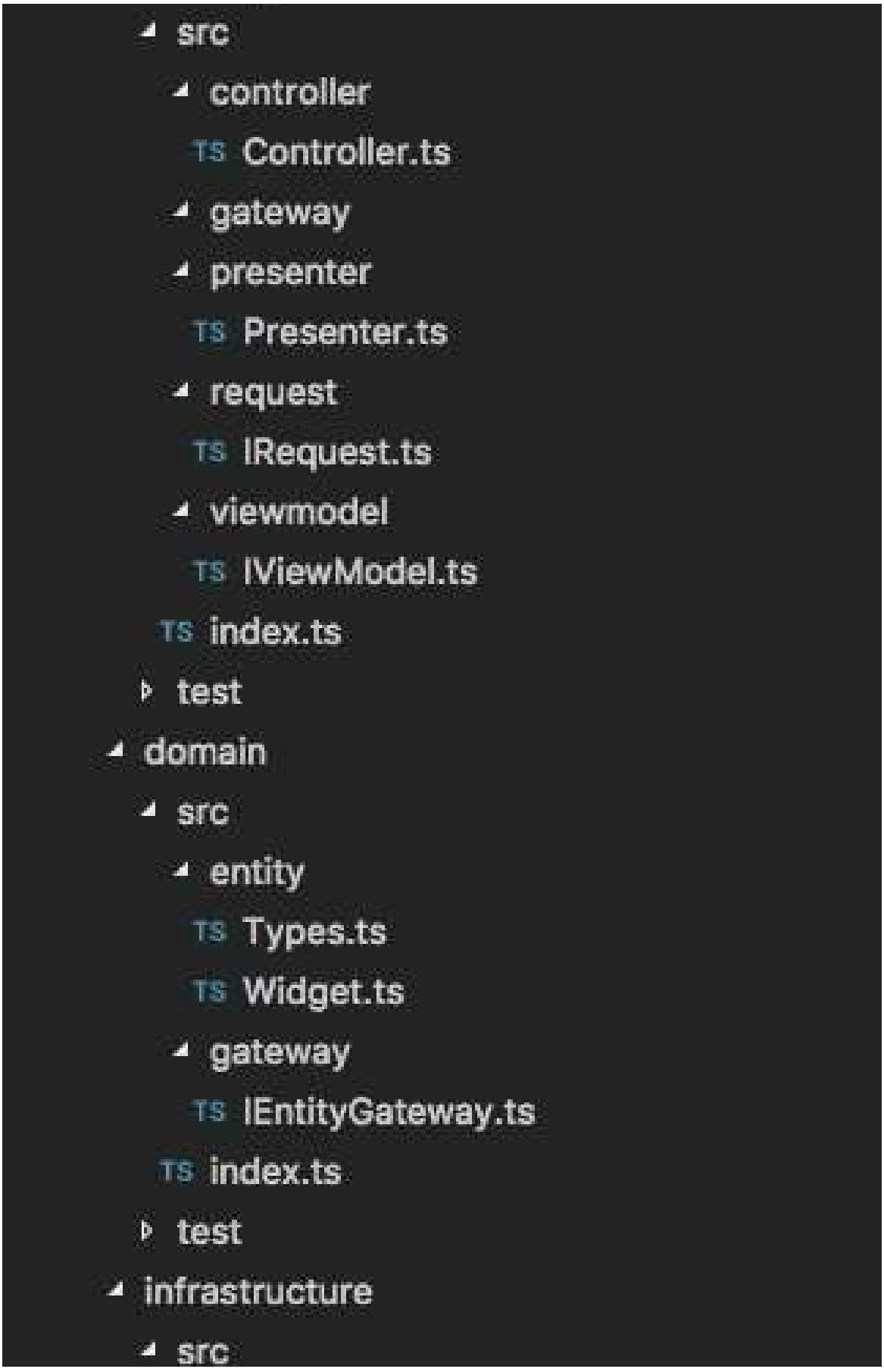



*Image Credit: Myself*

The infrastructure (blue) layer is where all of our outside pluggable systems live. These outside systems such as devices, web, and UIs, shown in the onion diagram, will use our IRequest and IViewModel interfaces to communicate with our Controller and Presenter while the db and external interfaces, shown in the onion diagram, will use the IEntityGateway interface to communicate with our Interactor. Our example will have one entity called a Widget with three properties. It also uses one use case "create widget" which takes a widget from the UI, saves the widget to some sort of storage, and

## More Visual Aids

Here is the directory structure. Everything gets wired together in each module's index.ts file. The entry point is demonstrated in a test located in infrastructure/test/TestEntryPoint.spec.ts .



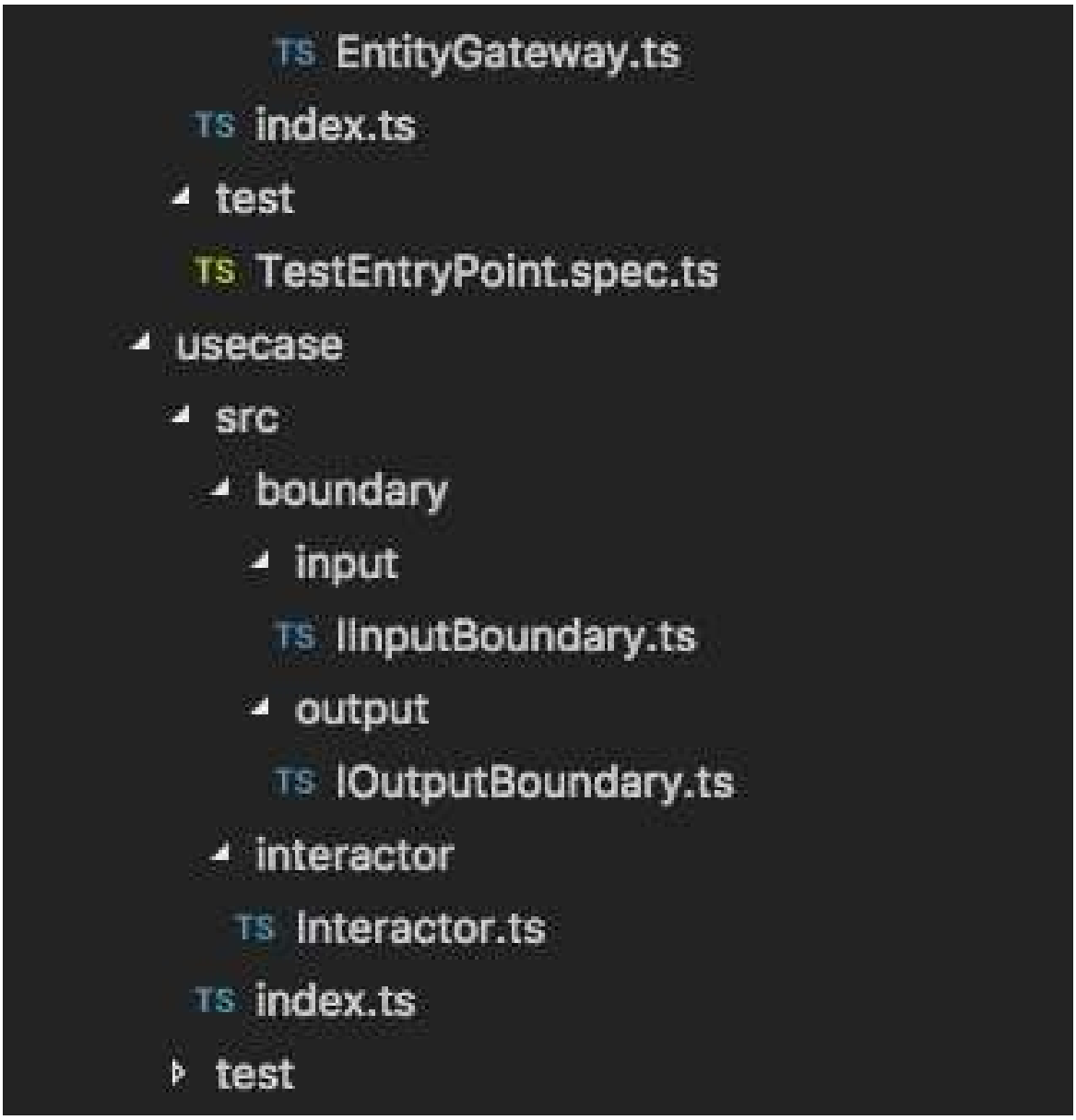
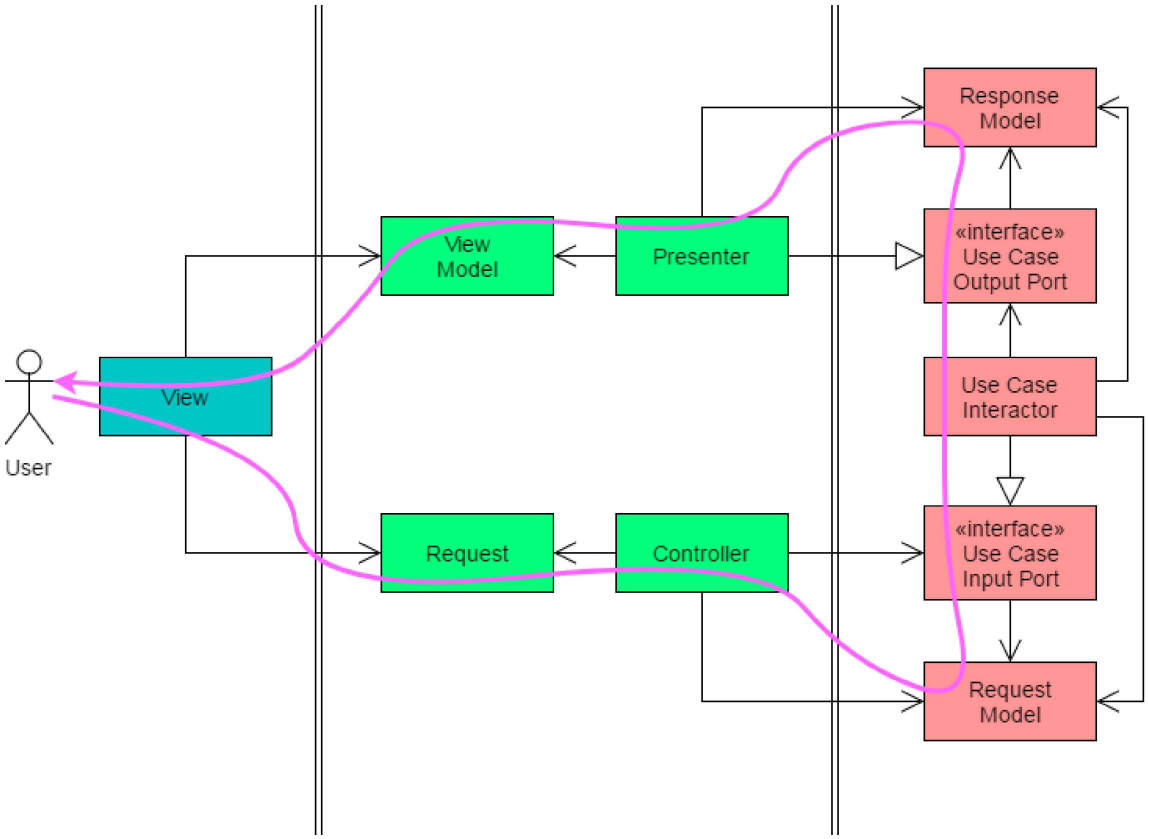



*Image Credit: Shift Command 4*

## Which Way Did He Go?

One of my original hangups was how the outer most layer communicates with the inner layers. I thought all you had to do is call some createWidget() function, for example, on a Controller and you would get a nice shiny new widget returned back to you. Wrong. What you want to do is send the widget to be created down to the use case (Interactor) on a certain path and have the use case (Interactor) send the new widget back up to you on a different path. This is similar

implemented a RequestModel or a ResponseModel.




_[Image Credit](https://plainionist.github.io/Implementing-Clean-Architecture-Controller-Presenter/" rel="noopener" target="*blank*" *title=")*

## Step Into, Step Over, and Step Out, the OO Way

So let's first create a widget with classes and interfaces.

## OO Step 1

This is the entry point. This code would be located in the infrastructure (blue) layer. This layer is where your mobile app, web app, API, CLI, etc. lives. Also all of your outside systems like external APIs, frameworks, libraries and databases live here too. Everything is

interface where a new widget will appear and you can update your UI within the implemented function presentWidget(widget).
You then create a Controller that implements the IRequest interface by passing the EntityGateway and the ViewModel you created above to a constructor. Finally your UI calls createWidget(widget) on the Controller where your new widget begins its journey to the Interactor.

## What's an EntityGateway?

An EntityGateway implements the IEntityGateway interface and is where you implement specific code that persists your widget. It lives in the infrastructure (blue) layer. This could be any type of existing or future external API or persistence system such as a database.
To change out to a different system, you would just simply wire together the new EntityGateway implementation with the IEntityGateway interface. In this example, I use a Promise to simulate some sort of persistence operation.

## Wire up What?

The file infrastructure/src/index.ts in the infrastructure module is where you can wire up your different implementations of your IEntityGateway interface. The "from" path of the import statement points to the correct implementation. In this case it is a persistence system named AnyDB.
Uncle Bob also talks about the use of a Main class where you can do this type of wiring up or do other initializing code. The Main class would also live in the infrastructure module and be pluggable. It would also communicate in the same manner as the other systems in the infrastructure module do. For example, you would initiate this class in your UI's initializing code and pass it down into your more inner layers to be used via some sort of configuration interface.

done to make our example easier to read.

## OO Step 2

The Controller is a very busy place. First, the EntityGateway passes through unchanged to our Interactor constructor. Then our ViewModel gets passed to the constructor of our Presenter which in turn also gets passed to our Interactor constructor. This all happens in the createWidget(widget) function of the Controller which was called by our UI in step 1 via the IRequest interface. We will talk about our Presenter in step 4 when the newly created widget travels back up to the UI.

## OO Step 3

Finally we are at the most inner layer of our journey, the usecase layer where our Interactor lives. Or better known as our home for all of our app's use case logic. There is one more inner layer, the domain. This is where all of our business entities and business specific logic lives. In this example, we really don't have any need to go there except to borrow the WidgetType and IEntityGateway interfaces.

## Movin On Up

Here in our Interactor we take the EntityGateway that was passed through from the Controller and call its saveWidget(widget) function via the IEntityGateway interface. This function returns a Promise from the EntityGateway which resolves in .then() with a newly created widget. We then call the Presenter's presentWidget(widget) function via the IOutputBoundary interface which starts the newly created widget back up to the UI. This all happens in the Interactor's createWidget(widget) function which was called by our Controller via the IInputBoundary interface in step 2.

presentWidget(widget) function we created in our UI. This all happens in the Presenter's presentWidget(widget) function via the IOutputBoundary interface which was called in the Interactor's createWidget(widget) function in step 3. More can happen here, but not in our example.

## OO Step 5

Finally our newly created widget is back home ready to be displayed in our UI. This is the exact spot (code) where we started in step 1.
Updating the UI happens in the ViewModel's presentWidget(widget) function via the IViewModel interface which was called in the Presenter's presentWidget(widget) function in step 4.

## OO Supporting Cast Members

Here are all the remaining interfaces and type definitions clumped together in one file.

## 2 Men Enter 1 Man Leaves

I wrote the class and interface version of this project first. I wanted to try and make it match Uncle Bob's original diagram as close as I could. When I finished that project, I realized I could have done the same thing with functions and type definitions. So I created an identical project and replaced Classes with Functions and Interfaces with Type definitions.
And here is the difference between a Controller class and a Controller function.

## Step Into, Step Over, and Step Out, the Function Way

Now lets give a stab at creating widgets with functions and type definitions.

IRequest, IViewModel, IInputBoundary, and IOutputBoundary are now type definitions instead of interfaces.

## Function Step 1

Every thing is the same as OO step 1 above, other than that we are now importing a function named “controllerConstructor” instead of a class named “Controller.” And importing a function named “entityGateway” instead of a class named EntityGateway. Last but not least, the ViewModel we created is now an object with a presentWidget() function in it instead of a class with a presentWidget() function.

## EntityGateway Again?

The EntityGateway does the same task as the OO version above. It is now a function instead of a class. It returns a saveWidget() function wrapped in an object.

## More Wiring

Same as OO version above except we are now exporting a function instead of a class.

## Function Step 2

Our Controller is still a busy place and does the same tasks as the OO version. We are now importing a function named interactorConstructor instead of a class named Interactor. We are exporting a function named “controllerConstructor” instead of a class named “Controller.” It returns a function named “createWidget wrapped in an object.

## Function Step 3

It returns a function named “createWidget wrapped in an object.

## Function Step 4

We are now passing the newly created widget back up in our Presenter where we are executing the same tasks as the OO version above. We export a function named “presenterConstructor” instead of a class named “Presenter.” It returns a function named “presentWidget wrapped in an object.

## Function Step 5

Again we have come full circle and we are back in the exact spot (code) where we started in step 1. Our UI gets updated with our newly created widget in the ViewModel’s presentWidget() function.

## Function Supporting Cast Members

Here are all the remaining type definitions clumped together in one file. These are our interfaces.

## All That for a Damn Widget?

Yes, but you also get the promise of a completely decoupled system where you can plug in different implementations of your outside (infrastructure blue layer) systems, including different types of UIs, external APIs, databases, libraries, frameworks and more.

## We Don’t Need No Stinkin Profilers

My original hunch was that the class and interface version would be slower than the function version. So I ran both projects through my advance profiling tools of typing “npm test” and pressing enter until my finger cramped up.

all of the important functions wrapped in objects so I could enforce the function names. I then ran both versions through my advance profilers and they were about the same speed.

I have no idea why wrapping a function in an object would slow it down that much. Maybe I didn't actually get Adobe Flash completely uninstalled from my laptop and it decided to interfere. Anyways, it would be interesting to get a more accurate measure of speed using the correct tools against the compiled JavaScript.

## The Take Away

The OO version has more code but may be easier to read and follow. The function version has less code but may be harder to read and follow.

Personally I like the function version, being that I have done a lot of programming in Java and I am tired of writing so many classes. One of the things I like the most about TypeScript/JavaScript is the ability to use object literals. And with TypeScript type definitions, you can now apply some safety to using object literals.

Another take away is that you don't need to rigidly conform to the clean architecture as diagrammed above to achieve a decoupled system. For example, you could just as easily have your UI communicate directly with your use case layer bypassing the delivery layer if it's not needed. All of these layers may physically live in different places and have different ways of communicating with each other.

## Give it a Try

Here are some of the things I intend to enforce in my next project.

1. Dependencies should always go one way.

3. Your inner layers (delivery, use case, and business entities) need to expose interfaces for the more outer layers to use.
4. You should always start developing from the most inner layer out. Start with the business entities and logic first and test. Create the interfaces that will be used and then test these interfaces. I am guilty of working the other way around. I think we all like to start with the UI, because it immediately lets us visually see how our system will look to a user. Plus the UI is where a lot of the "cool" technologies live.
5. Use TDD (Test Driven Development). Clean architecture allows you to do this much more easily. Everything is more compartmentalized and easier to mock. The implementation of the IEntityGateway above is basically a mock of a database.
6. Last but not least, be flexible. Don't knock yourself out trying to adhere to clean architecture when that library or framework you want to use just won't work with it. But be warned that this is probably a good indication that you will eventually have some sort of problems regarding that library or framework, especially if it wants you to extend their classes. Decoupling should be your goal.

## But, But, What About...

Please ask questions and give feedback, there is no better way to learn than to get constructive criticism from your peers. And it is highly probable that I missed something somewhere.

## Resources:

## Code for the function version is located at:

https://github.com/warrenbell/cleanarch-tsfun

## ts-node

Handy little TypeScript tool.
https://github.com/TypeStrong/ts-node

## The Clean Architecture by Uncle Bob

https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

## The Book

https://www.amazon.com/Clean-Architecture-Craftsmans-Software-Structure/dp/0134494164

## One of Many Videos

They are all basically the same except for the the first 5 minutes where Uncle Bob likes to muse about something loosely related and then makes a hard segue into clean architecture.
https://www.youtube.com/watch?v=Nltqi7ODZTM

## Implementing Clean Architecture — Of controllers and presenters

https://plainionist.github.io/Implementing-Clean-Architecture-Controller-Presenter/

## Clean Architecture: Standing on the shoulders of giants

https://herbertograca.com/2017/09/28/clean-architecture-standing-on-the-shoulders-of-giants/

https://softwareengineering.stackexchange.com/questions/339392/clean-architecture-use-case-containing-the-presenter-or-returning-data

## Clean architecture. What are the jobs of presenter?

https://stackoverflow.com/questions/46510550/clean-architecture-what-are-the-jobs-of-presenter

---