a Clean Architecture in .Net

Dec 9, 2015

A few years ago I stumbled upon on an inspiring talk by Robert Martin about separation of concerns.

This post is about how I’ve tried to put some of his ideas into practice using ASP.NET MVC web applications.

**Once upon a time**

*Project manager: “the client wants this online shop where you can order products.”*

*Me: “Hmm, ok but we have to save the orders somewhere. I guess we need a database with columns. And I will create a webpage to show them”.*

*Project manager: “Oh, and if the product is not available, you should not be able to order it, obviously.”*

*Me: “Right, you could have told me earlier. OK, I’ll add a column to the database and filter on that.”*

Three months later…

*Project Manager:  
“Why do we have so many bugs?  
Why is the website so slow?  
Why does it take you so long to make changes to the system?”*

I knew these complaints were fair, and I knew there were problems with the code that I produced.

Projects would always start out simple.  
Then stuff would happen and things would get complicated.  
I ended up with a big ball of mud at the end.

I had some hunches… I recognized that the database would often become a bottleneck. that presentation logic was mixed up with business logic and data access code. And that testing was so painful that we never properly tested anything.

**Another way**

Then I saw a talk by Uncle Bob in which he describes a different approach. It was about a ‘clean architecture’. In this architecture business logic was central and “the web” and “the database” are details.  
This spoke directly to the concerns I had before and it even offered a solution. Great!

But there was a problem.

Uncle Bob *didn’t show me any source code*.  
At the time I didn’t grasp the ideas well enough to use them in practices. I needed a bite-size example.

So I dug in and read his books and studied video courses. I learned a lot in the process. But I was still struggling trying to apply these principles in real world projects.

After a while I figured out an approach that adheres to some of these principles and works well for me in practice.

In this blog-post I will clarify some of the implementation details. I will explain how these work *for me.*

I will share some code examples I wish my younger self would have had access to years ago.

**Clean architecture in a nutshell**

Before I start, I will summarize some key elements of the architecture.

If you are completely unfamiliar with the ‘clean architecture’, do not expect me to explain it in detail here. Watch the video above and read [the original blog post](https://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html) by Uncle Bob first.

**The Dependency Rule**

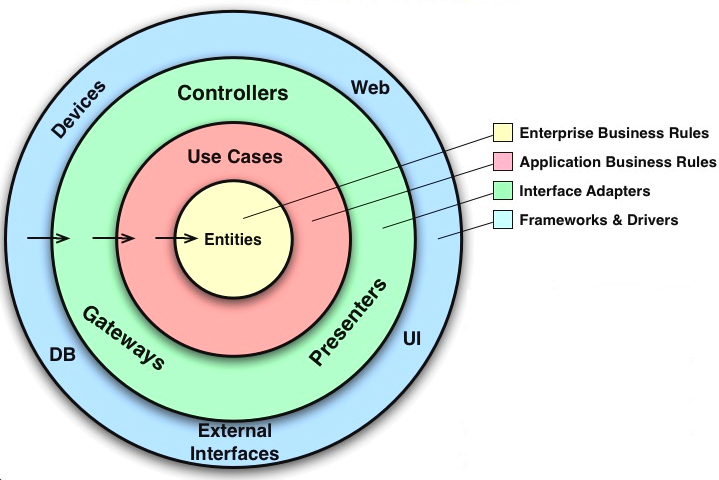


Diagram by Robert C. Martin

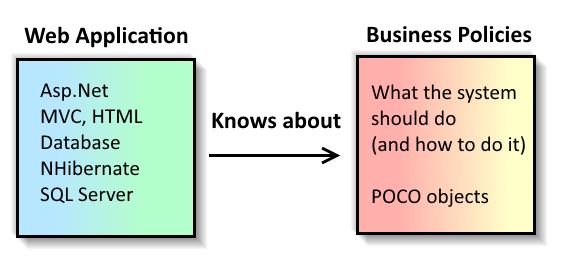
One of the goals of the clean architecture is to encapsulate the business logic of the application / enterprise in a clean way.

The diagram above shows this. The domain and use cases are positioned in the center of the ‘onion’.

*Uncle Bob:  
The concentric circles represent different areas of software. In general, the further in you go, the higher level the software becomes. The outer circles are mechanisms. The inner circles are policies.*

Notice that the dependencies in this diagram always point ***inwards*** when crossing architectural boundaries.

In practice this means that you have the code which describes business logic separated from your ‘delivery mechanism’.



über-simplified version of the same diagram, with the concrete .Net parts ‘filled’ in.

This ensures that your business rules are independent of your choices in framework. The more general ‘policies’ and business rules do not depend on the more concrete implementations and frameworks.

This has has some benefits:

* Separation of concerns.
* Business rules *just* describe how your business domain actually works, nothing else.
* Easy to change data storage, or web framework architecture, without having to change the business rule code.
* business rules can be unit tested easily
* safe refactoring of your business rules.

The ‘inwards dependency’ rule is not limited to the business rules, and applies everywhere. But the isolation of the business rules is one of the most valuable things this design has to offer.

How do I try to achieve this in practice?

**Implementation**

Let’s have a look at an implementation and see some code.

We’ll start with ‘use cases’ and an example because this is a natural way to start. Usually we begin a project with a purpose. The software should slve some real world problem that the user has.

What should the software do and how does the user interact with it?



[Funda](http://www.funda.nl/) (where I work) is an online real estate platform in the Netherlands.

Let’s image we want to develop a feature that allows future could-be home-owners to contact a real estate agent about a specific house.

At Funda we do scrum. We write user stories which try to capture the business value of the functionality that we’re going to build.

***User story: Contact real estate agent*** *As a consumer I want to contact the real estate agent  
so that I can visit a specific house.*

We can express this like a formal use-case.  
For example:

***Use case: Contact real estate agent***

*Data:  
- Customer email address(required, must be valid)  
- Customer phone number (required)  
- HouseId — every house on funda.nl has an Id.  
Primary Course  
- The customer issues a “Contact Real Estate Agent” command with above data  
- System validates all data  
- System records the fact that the user is interested in the house, so we can notify the agent at a later time.  
- System confirms to the user that their request has been processed.  
Exception Course: Validation Error  
- Processing of the request is cancelled.  
- System delivers the error to customer.*

Our developers do not necessarily write down the use cases in such a formal way. But I have included the formal use-case because it is useful to illustrate some aspects of the architecture.

Notice that this use case just describes what is needed in the interaction between the user and the system.

It does not specify any details about how the contact information is saved (to a data store?), or even that the functionality is exposed through form on the funda.nl website.

This use case might apply to console application, or a phone-in call centre. Or a website. You couldn’t tell just by looking at the formal use-case. Indeed, that’s the point — to describe what the system does, without mentioning the technical bits and pieces. (Uncle Bob: “The business rules are agnostic of the delivery mechanism.”)

**Interactor Objects**

In the clean architecture, the ‘primary course’ of a Use Case like this one maps nicely onto what uncle bob calls an *Interactor Object*.

Interactor objects are objects which model the interactions of the user with the system (the use cases) in terms of manipulating the domain entities.

Entities are the business objects of your application, for example a ‘House’

Notice a couple of things about the interactor class:

1. It describes the use-case we saw earlier, pretty much 1:1.

2. It uses plain datastructures (‘request-/response-messages’) as input-/output-ports and has no references to ASP.NET.

3. It has a dependency on IRepository

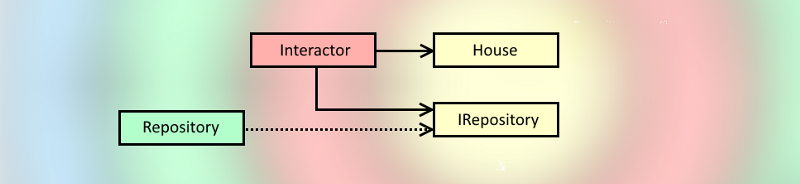
The interactor concisely describes how the system is supposed to behave according to the functional specification. That’s pretty clean code!

The interactor accepts a plain request message, and returns a plain message object. These form a boundary which decouples the Interactor object from whatever application it is being used in. The Interactor object just knows about the messages, that’s all it needs to know.

The Interactor object uses a Repository to “get”, manipulate, then “save” an entity (in this case a ‘House’) using the [Repository pattern](http://martinfowler.com/eaaCatalog/repository.html). (Uncle Bob calls these ‘Gateways’).

At first this dependency might seem like a violation of the Dependency Rule, shouldn’t the dependencies always ***point inwards***?

We can use the [Inversion of Control](http://martinfowler.com/bliki/InversionOfControl.html) principle to turn the dependency on it’s head by introducing an interface.



The Interactor uses the IRepository interface, which is defined in the ‘Entities’ layer. The interfaces describes *that*the entity can be persisted, but not how.

The Interactor does *not* know about the concrete implementation of the Repository, which is defined in the ‘Interface Adapters’ layer.

I’ve mentioned Entities in passing, in this example, we have a House. Let’s have a closer look.

**Entities**

*Uncle Bob:  
Entities encapsulate Enterprise wide business rules. An entity can be an object with methods, or it can be a set of data structures and functions. It doesn’t matter so long as the entities could be used by many different applications in the enterprise.*

*If you don’t have an enterprise, and are just writing a single application, then these entities are the business objects of the application. They encapsulate the most general and high-level rules. They are the least likely to change when something external changes.*

*For example, you would not expect these objects to be affected by a change to page navigation, or security. No operational change to any particular application should affect the entity layer.*

There is a lot that can be said about how to model a business domain that I will not go into here.

Bbut the low level implementation is pretty straightforward: I use ‘plain old c# objects’ that model how the most general objects and policies in the application.

For Funda, the fact that we have information on houses and that people can express interest in particular houses is a very general and ‘deep’ truth about *what Funda is about*.

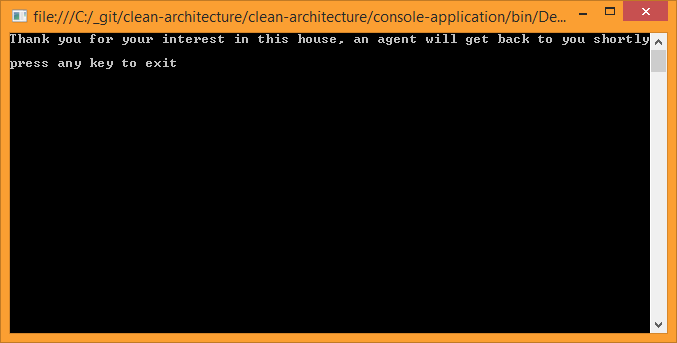
To be able to express this in a domain model without referencing all the bits and pieces that it takes to make a website ‘tick’ is a very pleasant and elegant thing.

Keeping this part of the codebase simple and concise will give us a lot of freedom to refactor and evolve the design when we start adding more features.

*Note: I will admit that this is a ridiculously simple (perhaps too simplistic, some might say ‘*anemic’*) example of a domain model, but I hope it illustrates the general idea:*

*That you can model the most general behaviours of your business in pure OO classes. And that you can use them in your applications without needing to mix them with the logic that deals with web frameworks, databases, message queues or other ‘delivery mechanisms’.*

**Done!**





Yes, really. Done. Kind of…

With these parts:

* The interactor classes,
* request-response messages, and
* the ‘business domain’ Entities

you have everything you need to model, express, and unit test the problems that are unique (and interesting) to your project. Of course it’s not a working MVC application yet.

**But I’m not joking**that *in a way*, the business rules can be ‘done’ before implementing anything related to the web, or persistence.

A key insight for me was that with these parts you already have every tool you need to model your project or company’s unique and interesting business rules and problems.

You could write unit tests or a small console app to talk to these classes, and prove that in principle the business logic works.

In fact, I have added an [example of a working console](https://github.com/stephanhoekstra/clean-architecture) application at the end of this article.

Below I will briefly touch upon the way these parts are tied together in a web application. But I hope by now I have convinced you that the website can built *on top of*(as opposed to intertwined with*)*the clean architecture that I have demonstrated.

**Putting an ASP.NET MVC web app together with clean architecture**

If we want to expose our features through an ASP.NETMVC web application, the code we write will fit firmly into the ‘Interface Adapater’ layer of the clean architecture.

***Uncle Bob:*** *The software in this layer is a set of adapters that convert data from the format most convenient for the use cases and entities, to the format most convenient for some external agency such as the Database or the Web. It is this layer, for example, that will wholly contain the MVC architecture of a GUI.*

*The Presenters, Views, and Controllers all belong in here. The models are likely just data structures that are passed from the controllers to the use cases, and then back from the use cases to the presenters and views.*

Indeed this layer is where most of the web application ‘stuff’ is found.

If our customer wants to contact the real estate agent through a website, they will likely have to fill out some HTML form.

A **model binder**converts the form-data is converted to a strongly typed **ContactAgentRequestMessage** and the request is handled by the **AgentController:**

Now we must cross the boundary into the inner layers of the application where “the business logic happens”.

We do this by invoking the **Handle**method on the **Interactor**, and it is important to stop and think about what datastructure we pass into this method.

***Uncle Bob:****Typically the data that crosses the boundaries is simple data structures. You can use basic structs or simple Data Transfer objects if you like. Or the data can simply be arguments in function calls. Or you can pack it into a hashmap, or construct it into an object.*

*The important thing is that isolated, simple, data structures are passed across the boundaries. We don’t want to cheat and pass Entities or Database rows. We don’t want the data structures to have any kind of dependency that violates The Dependency Rule.*

*[…] when we pass data across a boundary, it is always in the form that is most convenient for the inner circle.*

This is an essential insight. Repeat:

*“When we pass data across a boundary, it is always in the form that is most convenient for the inner circle”*

In the ASP.NET world it is tempting to use ‘Entity Framework’ classes, which Visual Studio encourages. And then to use those classes all over the place (as your view models, as your entities, in your model binders, etcetera.)

It would also be tempting to pass these across the boundary between Controller and Interactor, but that would violate the Dependency Rule.

Instead, the inner circle should dictate the shape of the input, which is the case in our example. the **ContactAgentRequestMessage**is defined inside the business logic layer alongside the interactor it corresponds with. The dependency points inwards, and ‘the web is a detail’.

Finally, the interactor does it’s stuff. After evaluating some validation logic, it saves the data somewhere (invokes an **IRepository** which has an implementation in the outer layer).

***Uncle Bob:****Similarly, data is converted, in [the interface adapter layer], from the form most convenient for entities and use cases, into the form most convenient for whatever persistence framework is being used. i.e. The Database.*

*Also in this layer is any other adapter necessary to convert data from some external form, such as an external service, to the internal form used by the use cases and entities.*

In this example I have included a dummy **Repository** so you don’t have to install anything when you run it on your own machine.

But in practice, this is usually where Object Relational Mappers come in handy. I like to user NHibernate or sometimes Dapper, depending on the situation.

**Presentation**

Finally the interactor returns a **ContactAgentResponseMessage**message to the controller and the use case has been completed!

This message contains the bare minimum information that the end-user needs. It is a data structure formatted in a way that matches the functional requirements from the use case, and that suits the Interactor.

Of course we want to show something nice to the end-user, so we user a **Presenter**object to convert the raw ResponseMessage into a **ViewModel.**

The ViewModel is also a plain data structure, but this one is structured in a way that suits the **View.**

For example, we might want to show them a thank you message or a friendly error message:

Finally, the Controller passes the viewmodel to the viewengine, which renders and returns the result to the end-user.