Who wants their applications to be maintainable?
(show of hands)
I would really like to talk to people who didn’t raise their hand right now. ;)

I think that maintainability is *the* most important thing we developers should be aiming for in our applications.

It opens up doors to everything else we want and care about.

If your application is maintainable, you can make changes to it quickly.
You can keep your business owners or your product managers happy by being able to quickly release new features.
You can also remove features or change how something works without too much sweat on your end.

At the same time, you can keep yourself and your fellow developers happy by working in a codebase which doesn't feel like a dumpster fire.
A codebase where technical debt is under control.
A codebase which has predictable, well-understood behaviour and dependencies.
A codebase where you can rely on the same business operation to be done the same way everywhere, not in four different ways in four different places.
A codebase which is super easy to test automatically, so you can have high confidence in it.
A world where you, as the maintainer, feel happy, not frustrated, to be working on it.
An application which you can proudly offer to your customers.

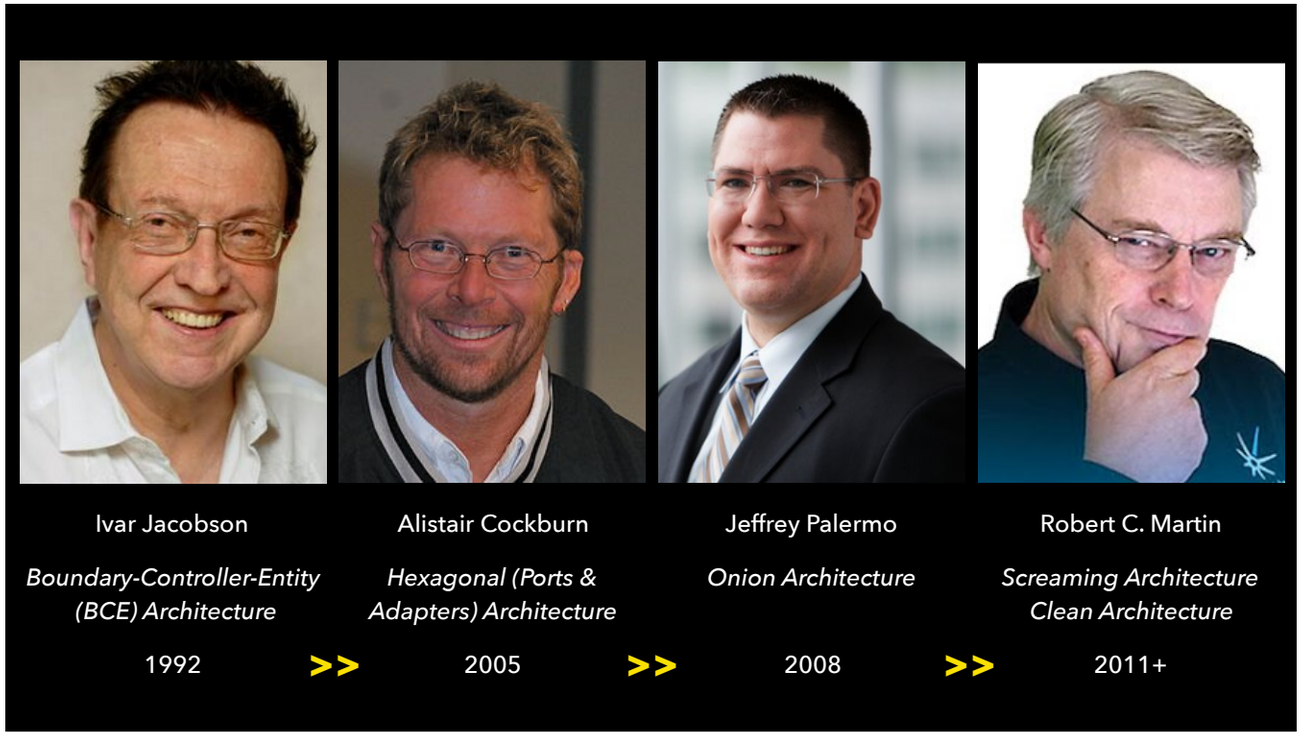
And it goes even further. Lack of maintainability will really hinder your ability to grow and scale. You simply won't be able to achieve maximum scalability and performance of an application which you're scared to touch. Or which simply cannot be modified in its current state without a signifiant rewrite.

Maintainability is the holy grail of good application design.

So this bring us to the focus of my talk, which is good software architecture which ensures maintainability.

This building is a concert hall in Reykjavík in Iceland, called Harpa, where we had Gophercon EU last year. And the clue to the name of the one particular pattern I will talk about today is in this building.

Hexagonal architecture. It’s actually the unofficial, more common name for the Ports & Adapters architecture described by Alastair Cockburn on his blog in 2005.



He was not the only person to come up with this idea.

As early as 1992, Ivar Jacobson published a pattern called the BCE - Boundary-Controller-Entity, or EBI - Entity-Boundary-Interactor as it was later called - in his book on *Object-Oriented Software Engineering*. He is the godfather of Object-Oriented programming.

Then Alistair Cockburn published the Ports & Adapters idea on his blog in 2005, giving it an alternative name of hexagonal architecture which is what it's widely known as today.

Around 2008 Jeffrey Palermo independently came up a very similar pattern which he called the onion architecture.

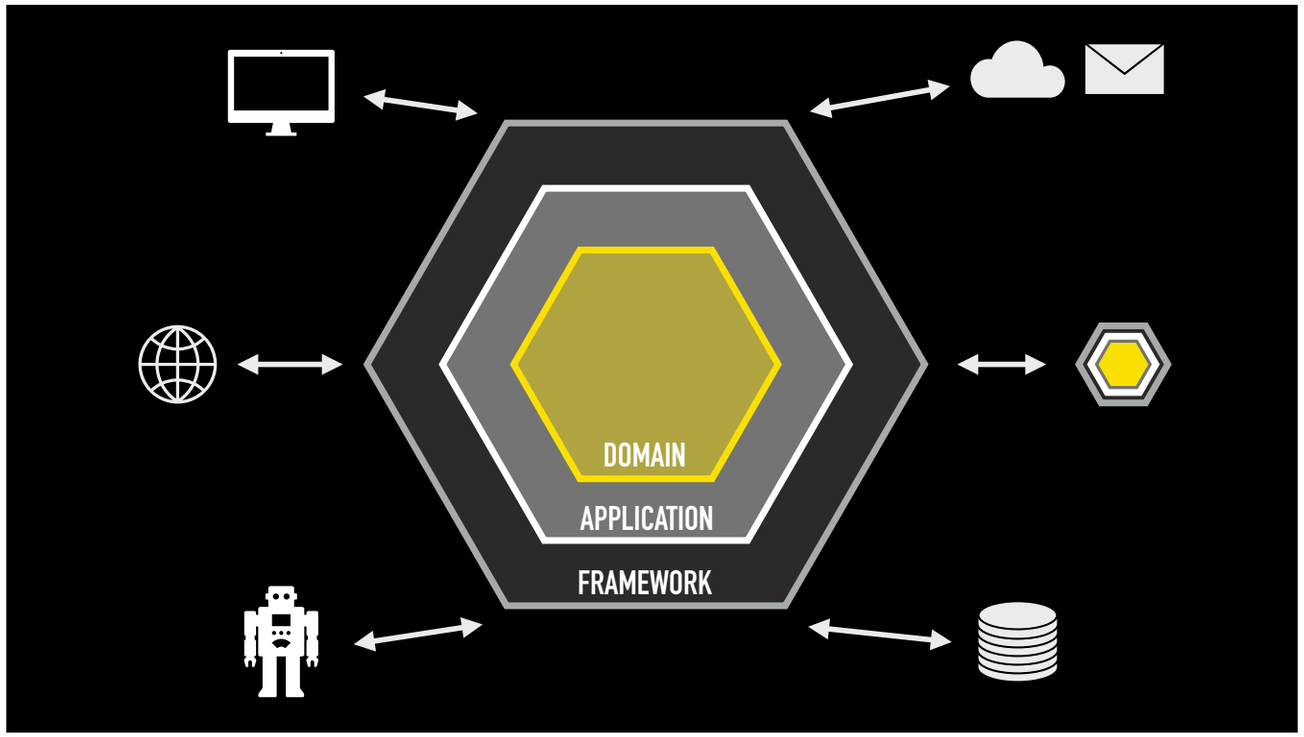
And then Robert Martin (Uncle Bob) referenced all those ideas and a few more in his book on Clean Architecture, but actually he referenced some of those ideas much earlier on his blog in 2011 and 2012 when he talked about the screaming architecture and then the clean architecture.

So this very same idea kept being reinvented and talked about independently over so many years. Maybe that's a clue that it's a good one.

image: https://www.flickr.com/photos/rebeccaanne/3719661594

So I thought I'll talk about the hex architecture simply because an image of Harpa is a little bit more captivating… than a bunch of onions. ;)

Also onions make you cry.



This is a diagram illustrating the ports & adapters architecture.

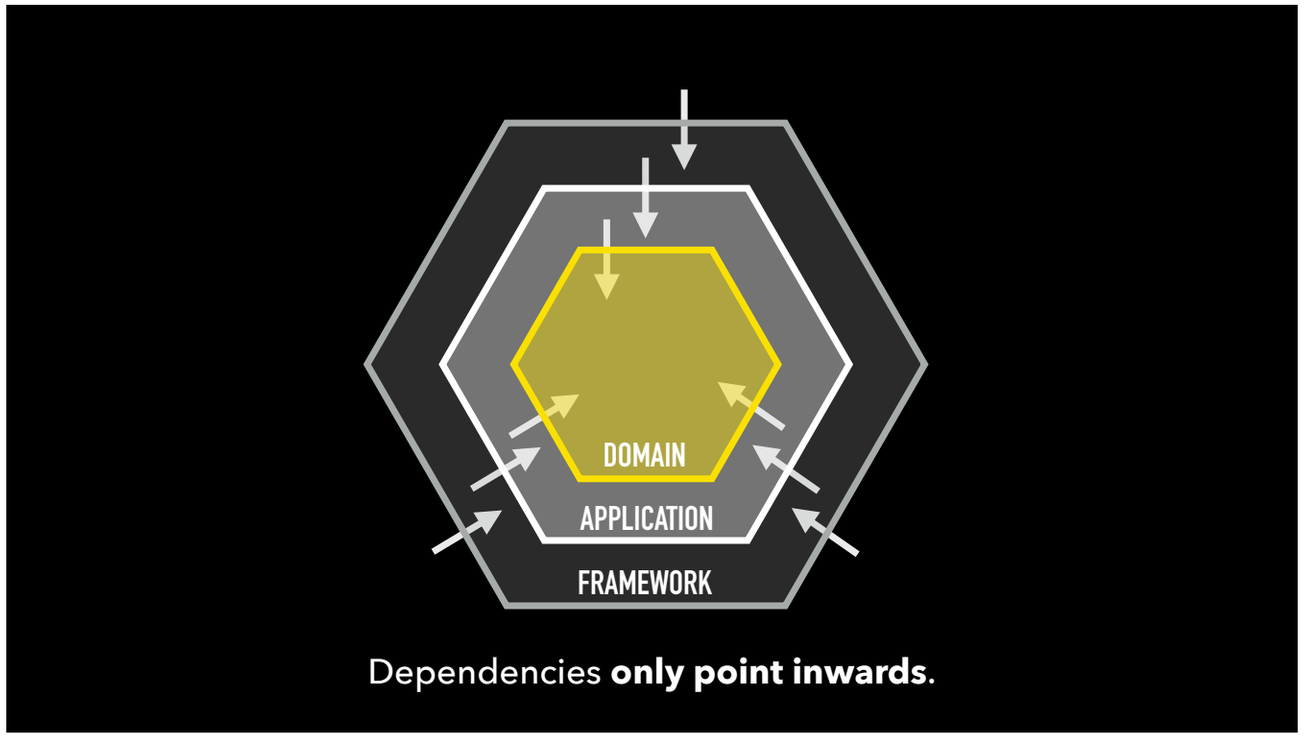
It shows that an application is composed of three layers.

The core layer, right in the centre, is the domain layer. It defines all the business logic of our application. No other layer is allowed to contain any business logic, it's all just in this one layer. The domain layer is not allowed to be aware of any implementation details.
Everything to do with how the business logic works needs to be defined in abstract terms and implemented in the outer layers.

A common choice is to define your core logic using Domain-Driven Design.

The application layer is the glue between your business logic and the very specific details of how your application communicates with the outside world.
It receives the raw input from the framework and does whatever is required to translate it into a form understandable by the domain to perform the operation we want.
For example, it may unmarshall the request body and validate all the values before passing it to the domain.
It will then receive the results from the domain layer and translate them into whatever format is expected by the outer layer. So this could be returning a specific response format, or data in the format expected by the database.

The framework layer is the furthest from the domain, and it is both the most abstract and the most detailed.
It treats the rest of the application as a black box, which just accepts some input and returns a response. It sees them as generic request and response objects. It doesn't know anything about how the business logic works. It is however full of specific implementation details for all the external interfaces the application interacts with.



So how does this help us achieve maintainability?
The key rule in this model is that dependencies are only allowed to point inwards.
What do I mean by that?

When we think about requests coming from the outside, the dependencies are easier to visualise.
The outer layers can call anything they like from the inner layers and reach out to the domain as much as they like, but they cannot reference anything outside of themselves.

For dependencies going out, like calling the database, we use Dependency Injection, or Inversion of Control.

TRADITIONALLY, EACH OBJECT IS RESPONSIBLE FOR OBTAINING ITS OWN REFERENCES TO THE OBJECTS IT COLLABORATES WITH (ITS DEPENDENCIES).

Craig Walls, Spring in Action

This is a quote from Craig Walls explaining dependency injection, or DI as it’s commonly abbreviated.
“Traditionally, each object is responsible for obtaining its own references to the objects it collaborates with (its dependencies).”

“When applying DI, the objects are given their dependencies at creation time by some external entity that coordinates each object in the system.”

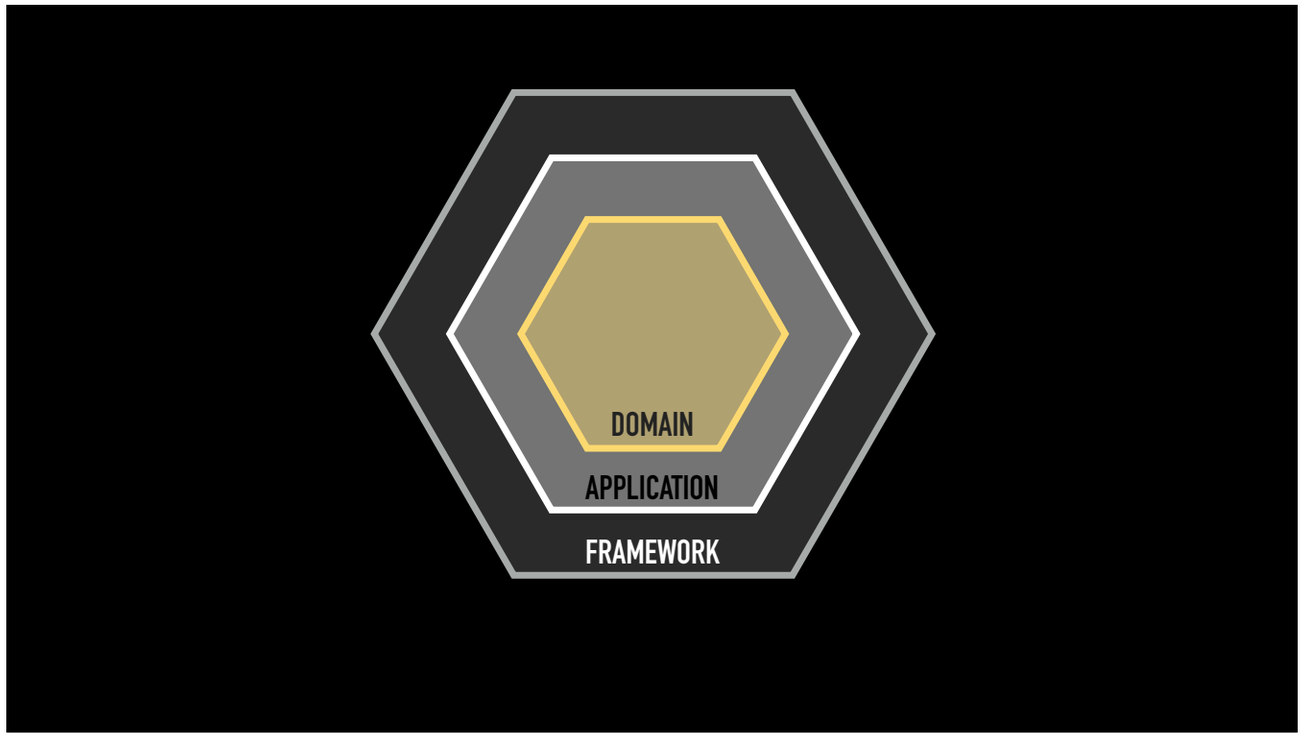
In other words, dependencies are injected into objects.

Inversion of Control uses interfaces to achieve dependency injection.

The domain will define all the functionality it requires from external dependencies in abstract terms as interfaces. It will then accept those interfaces in its constructors, and call their functions throughout the operation as if they were just some black boxes that do that stuff.
The actual, concrete implementations of those interfaces will be passed in at runtime, but the domain doesn’t really care what they are as long as they satisfy the required interfaces.

That way we can keep the dependencies pointing inwards only. The domain will not need to call out to the application or framework layers because it defines all its interfaces within itself.

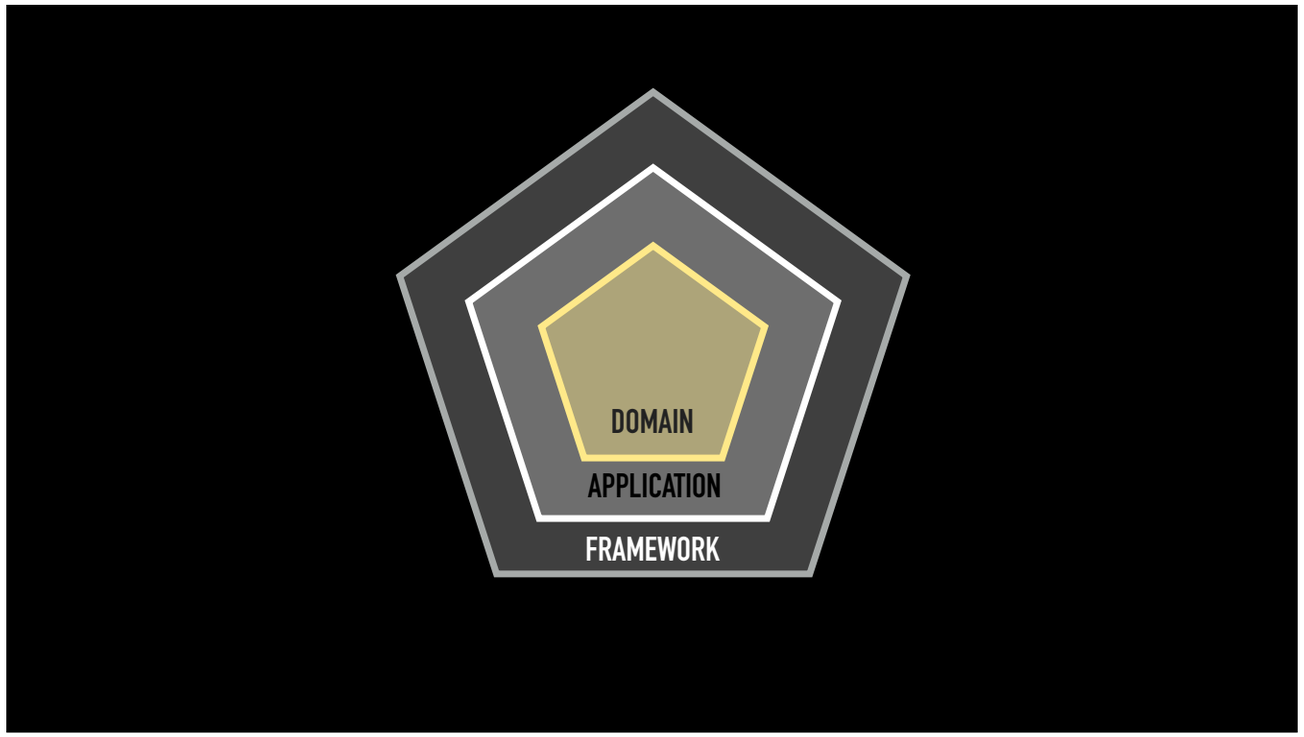
This in turn allows for maintainability.
Because of this organisation of dependencies, we can easily change one part of the application without affecting the rest of the application.
The point is to keep the changes local.
If the domain needs to change, we shouldn’t really need to touch any of the outer layers as long as the interfaces between them stay the same.
If the application or the framework needs to change, shouldn’t really need to touch the business logic as long as all interfaces defined at the boundaries are still satisfied.



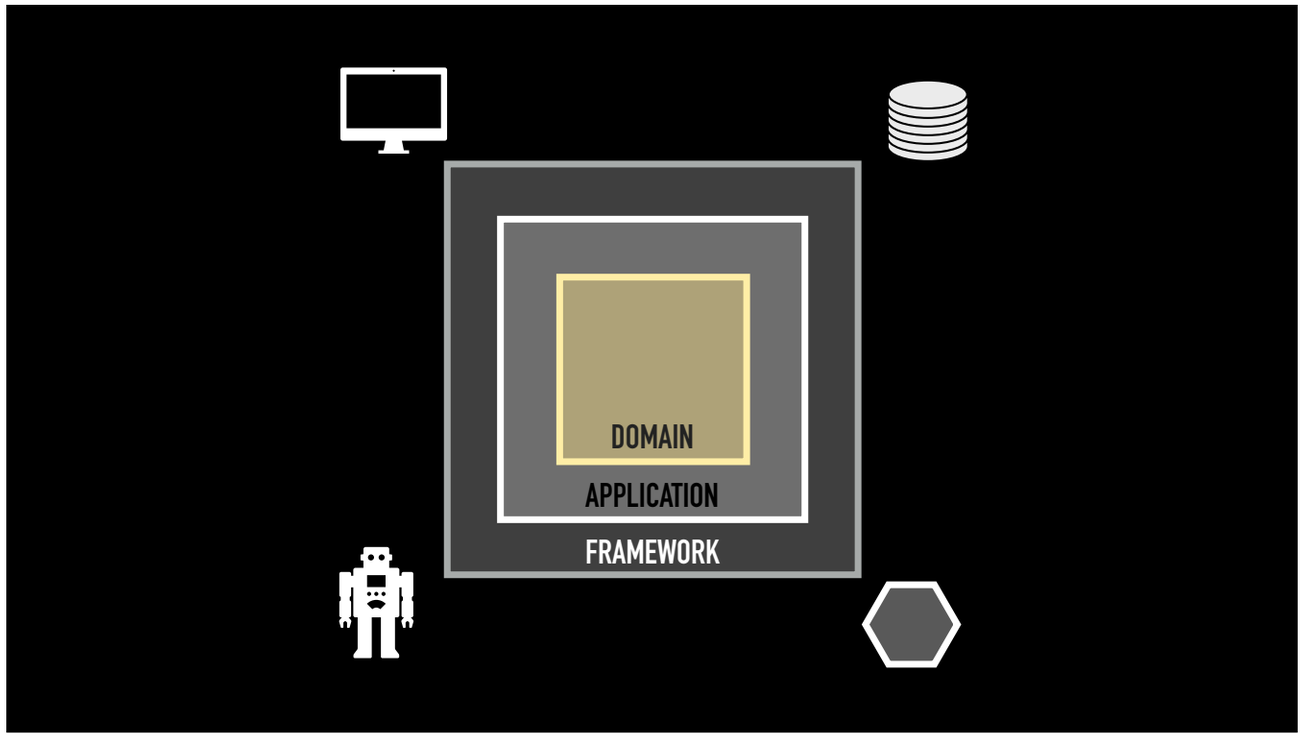
We define interfaces at every boundary.
Interfaces are ports, just like the ports on your laptop which allow different inputs to be plugged in. Any device which understands your laptop's port can be used.
For example, any headphones with a jack cable would work when you plug them in the jack port. Wouldn't they, Apple? :P
That gives you the flexibility to use many pairs of headphones over time, without having to buy lots of dongles.

Each layer will define its ports, or interfaces.
The outer layers provide adapters for those defined ports.
And this is why this architecture is called ports and adapters.

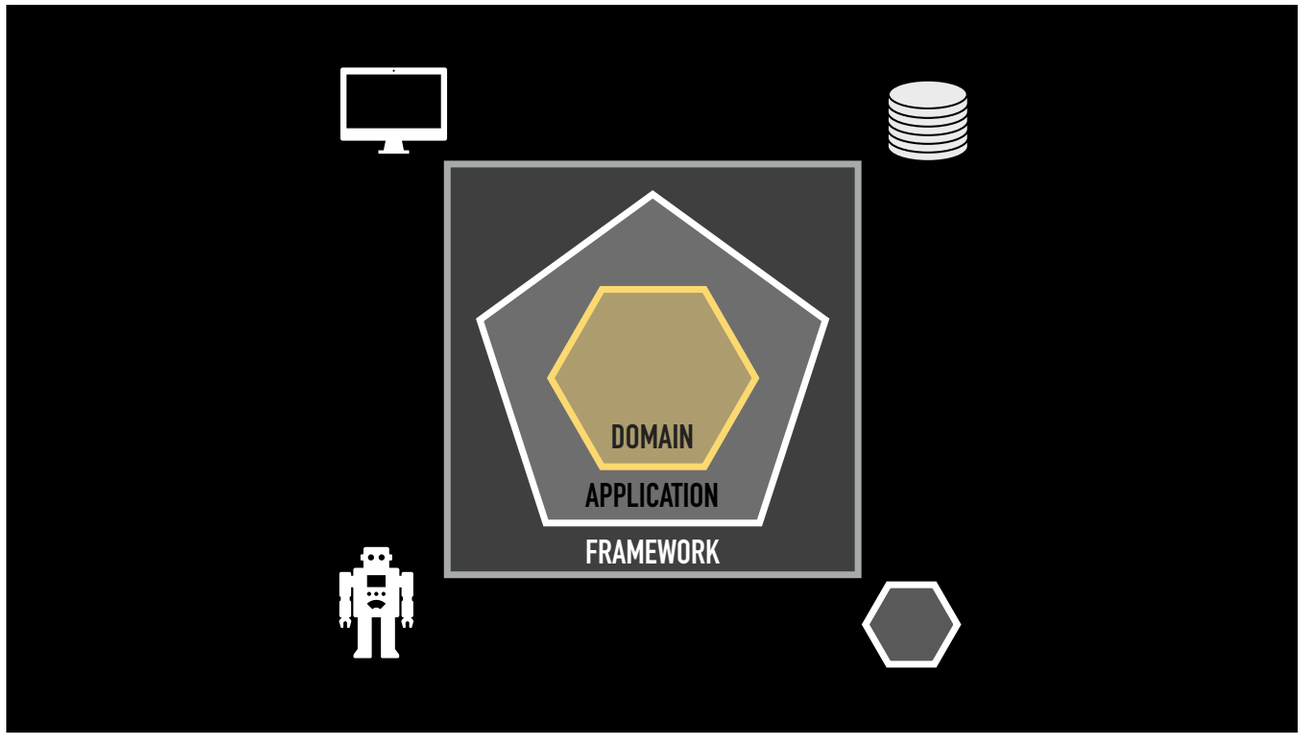
It's also worth pointing out that the hexagon is a random choice. The idea was that each vertex will represent a port, so in reality the shape will depend on how many ports the layers expose. Alistair Cockburn chose a hexagon because he felt that it leaves enough space on a whiteboard to then write the names of all ports into the diagram as you're designing your system.
He reckons we typically have about 3-5 ports in applications. There's no strict definition of what a port is, other that it's an abstract API exposed to the outer layers.



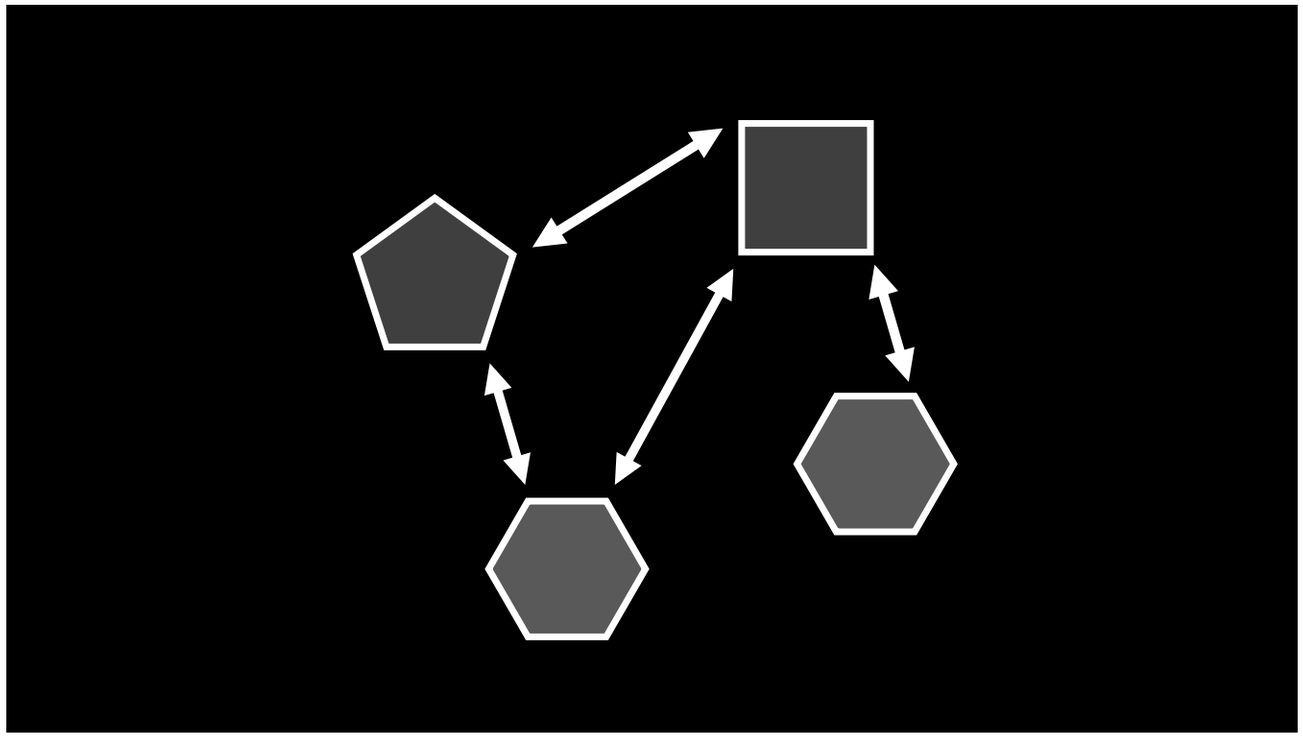
You could have a pentagon for example.



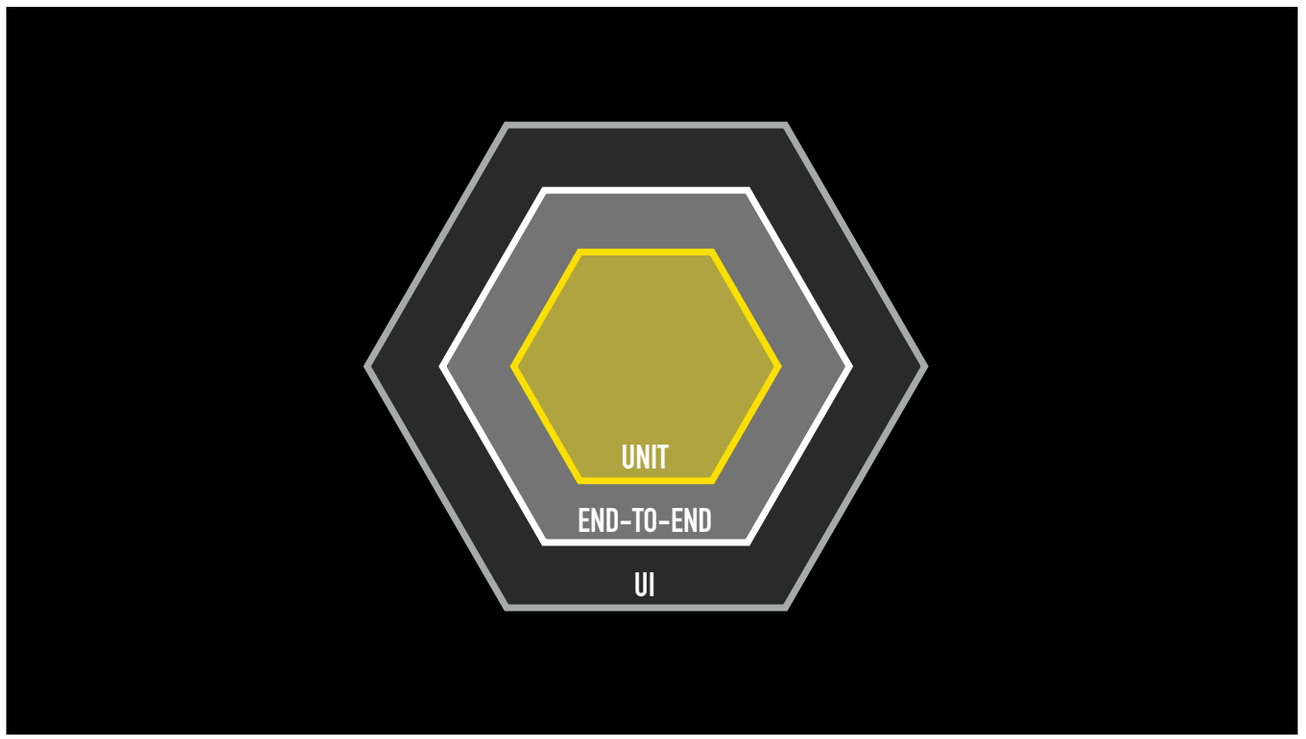
Or a square.



Or the actual representation of your system could even be something like this. There’s no rule that every layer needs to have the same number of ports.



This architecture works for single monolith services and for microservices. There’s no difference here.



Hexagonal architecture corresponds well to the test pyramid.
We can put lots and lots of unit tests in the domain layer, because the discrete units of domain and business logic should be easy and cheap to test. Unit tests are cheap to write and fast to run, so it makes sense to focus them on the domain so that our business logic is well protected and tested often.

The further we are from the centre, the more high-level we want to get.

The application layer will mostly use end-to-end (or integration) tests, because at this point we're interested in testing the end-to-end behaviour and the domain working together with the rest of the application.

And finally, we can add some tests which drive the application through its user- or command-line- interface, but as they are slow to run and expensive to maintain we do not want to have too many of those, and we certainly don't expect them to test every possible scenario.
Most things in the framework layer, like the HTTP router or the database adapter will probably come with their own unit tests, so there's no need to unit-test them ourselves. We can probably just rely on the UI tests or the integration tests to verify that they work as expected.

We already have interfaces in place because of dependency injection, and because those interfaces should define dependencies which don't contain any business logic, any test doubles you might need to use should be fairly simple.

Whether you end up having to use mocks or not will depend on what your rules are exactly.
If we were to be very rigorous about the domain having no knowledge of where the data comes from or where it goes to, then it won't be calling any functions to save to the database itself. The application layer will be in charge of deciding what happens with the output once it's received from the domain, so the domain layer is purely

PUTTING IT ALL TOGETHER

MAINTAINABLE

- Consistent.
- Easy to understand, navigate and reason about.
- Easy to change, loosely-coupled.
- Easy to test.
- Structure reflects design, design reflects how things work.

Hexagonal architecture introduces structure and consistency into your project.

It's easy to navigate, understand and reason about. Use-case centric architectures tend to make sense to people, and maybe this is why we keep coming back to this idea under different names.

Hexagonal architecture makes your applications easy to change, because the inside and outside layers are loosely-coupled.
It supports evolutionary design, because there's no limit to how many ports you can have.

It makes your code easy to test.

The design of your code reflects exactly how the application works. If something is placed in the application layer, you know that it most likely provides some sort of service for handling requests and that it won't define any business logic.
The responsibilities of each component are clear.
And the structure of your code, how your files are organised on disk, follow this design exactly.

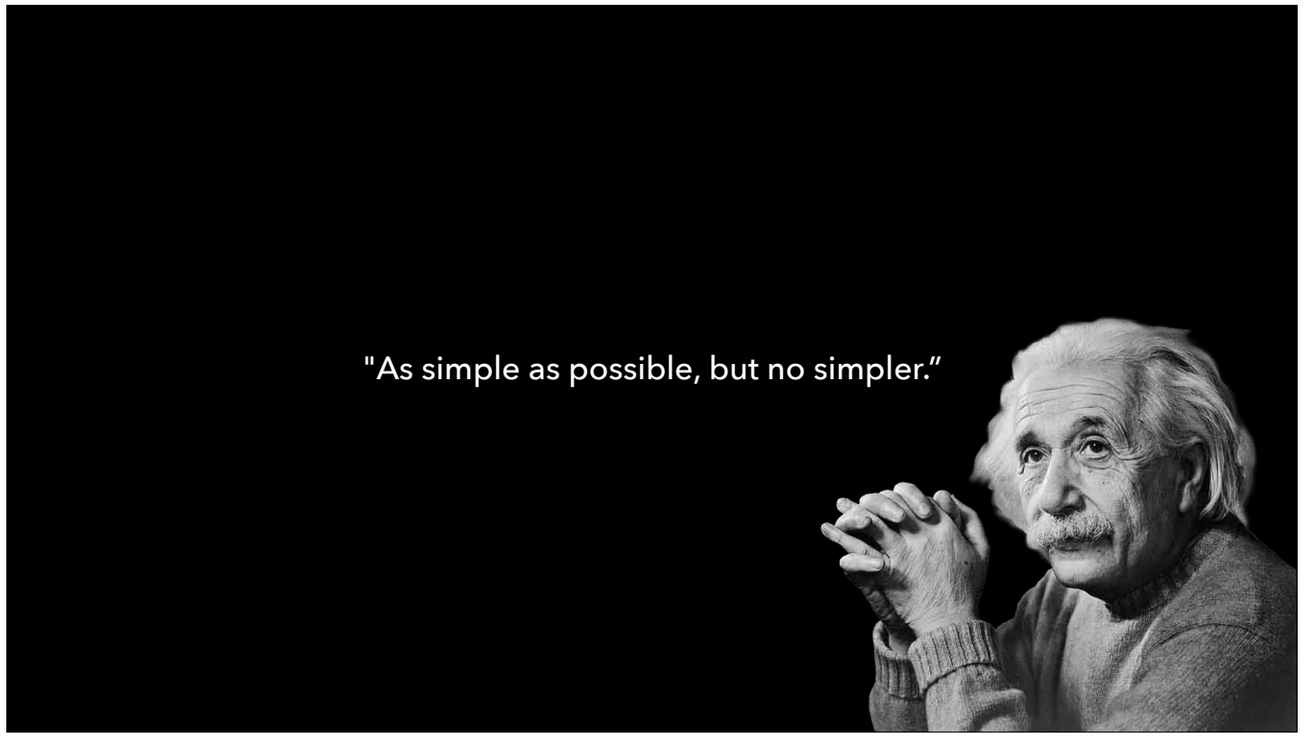
In any application, the code is ultimately the most up to date documentation of the business logic. Sometimes the intended design is not how the application works in practice.
In the hex architecture, we minimise the risk of that happening by keeping the core domain logic in one place, making it reusable and testable.

There is no single right answer to code structure.

End of the day, it’s whatever works for you and your use case. Sometimes the hex architecture might not be the right choice, and that’s fine. But if you don’t know where to start, especially with a commercial project, I think it’s a good one to try out. Or maybe you can start out with a flat structure and move to the hex architecture over time.

Remember that the logic in your app will expand and morph over time, so don’t expect to get everything right the first time.

As Bruce Lee said, be like water, expect to change and adapt over time.



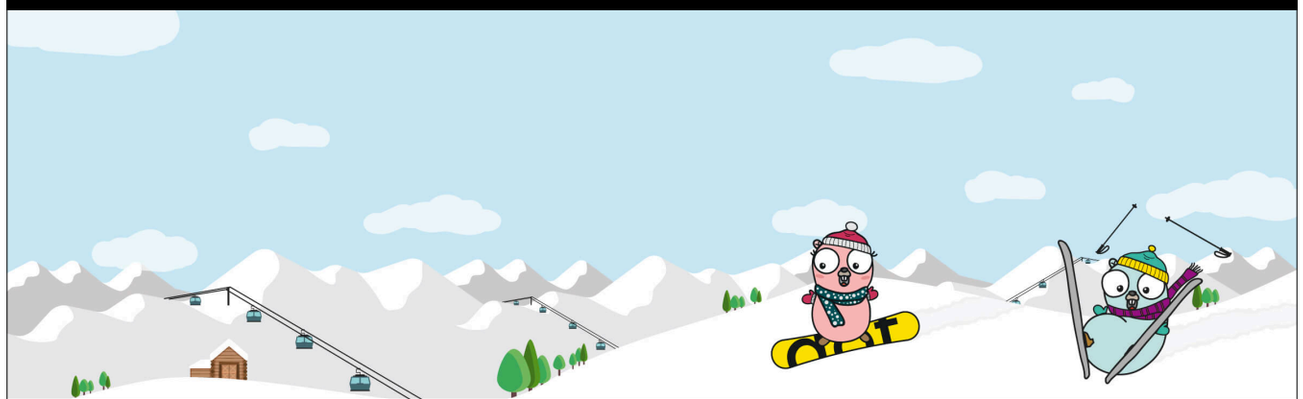
Whatever you do, aim for maintainability and don’t overcomplicate your structure. Keep it as simple as possible, but no simpler. Go for the simplest solution for today, don’t worry about the future.

Prototype. Experiment. It will help you raise questions you didn’t even know you should be asking.

Remember that good choices and best practices come with experience, so play around with your code and share your ideas with the community!

We’ve seen the idea of ports and adapters being reinvented over time, but I’m sure there’s plenty more to discover.

THANK YOU!



KAT ZIEŃ // @KASIAZIEN

Thank you.