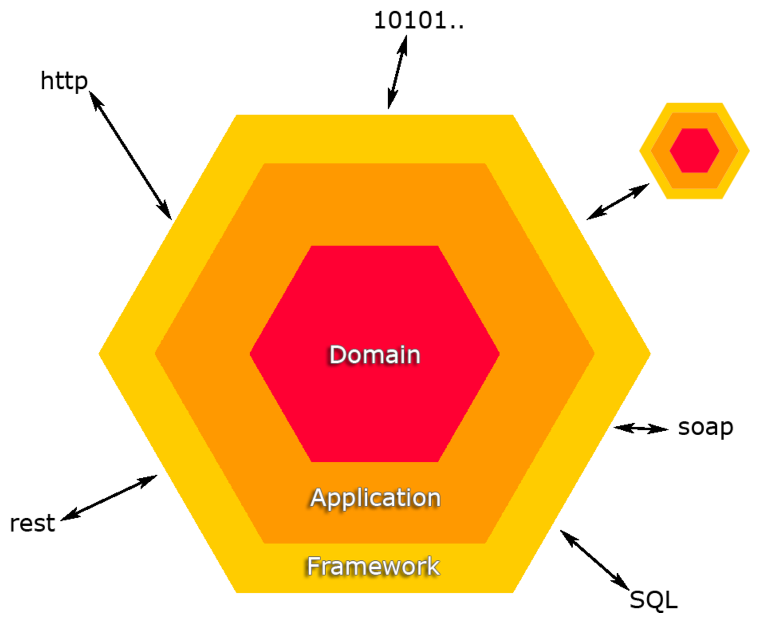
# Hexagonal Architecture

## Introduction

In this article I will discuss **Hexagonal Architecture**. This software architecture's main purpose is to improve maintainability and reduce how much time one will have to spend maintaining and modifying their code in the long run.

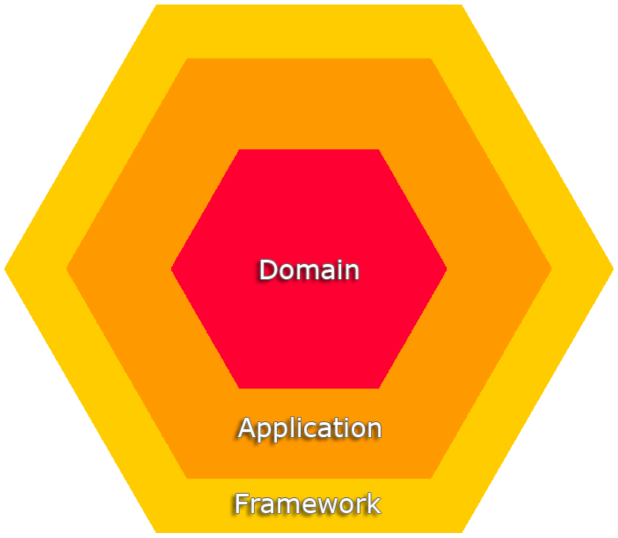
## Technical Debt and The Hexagonal Structure

A key concept in software development is technical debt. In software development, there is an important contrast between short term coding solutions and a long term coding plan. Usually code that is less planned out in the short run ends up leading to extra development work in the long run. This extra work is known as *technical debt,* and the goal of any good programmer is to reduce it as much as possible. *Maintainability* is the term used to describe how easy it is to maintain a system. The less technical debt you have, the higher your maintainability. **Hexagonal Architecture** works to increase maintainability so that our code will require less work in the long-term.



As the name implies, **Hexagonal Architecture** is represented by a hexagon. The figure above displays the entirety of the hexagon, with each of the sides representing a way that our system can communicate with other systems. For example, our system could communicate using an http, soap, rest, sql, or a binary protocol and even with other hexagonal systems. If we had a big system we could split it up into multiple hexagonal architectures that communicate remotely. By doing this, you can split the team of programmers and increase their ability to work in parallel. Another of the advantages of hexagonal development is the flexibility with which you can make changes or add new technologies. Since each layer is independent, you can make changes to the outer layer without having to change the inner layers at all.

## The Inside of the Hexagon



The inside of our hexagon consists of three layers: the domain, the application, and the framework. **Hexagonal Architecture** is an especially good fit for **Domain Driven Design** because it builds outwardly from the domain. In the next three sections I'll focus individually on each of the inner layers.

## The Domain

The domain is the central layer which contains all the business logic and business logic constraints. The domain layer responds in a technology independent way to whatever is being done in your architecture.
In Example 1 below, there is a constraint that checks if a certain gear is allowed. If it is not, an *InvalidGearException* is thrown. The exception will be passed all the way up the framework layer to be handled; the framework could, for example, log this exception and communicate it in a specific format. It could be a SOAP fault or a REST message indicating an error to the client.

```
public class Car implements Domain {
[...]
    public void drive(Gear gear) {
         [...]
         if (isNotAllowed(gear)) {
              throw new InvalidGearException();
        }
       [...]
   }
[...]
}
```
Example 1

## The Application

The application layer sits in between the domain and the framework and allows for communication between the two layers. Despite the name, it is NOT the actual application, but rather it applies the commands that the framework receives and sends them to the domain. One could imagine the application layer as the translator between what is received in the framework and the domain, or vice versa. In Example 2 below, the class CommandBusImpl is receiving a command that was forwarded from the framework layer and it is passing it onwards to the domain. In this example, CommandBusImpl is using a *registry*, a **Design Pattern**, to find which handler in the domain is going to execute a command, and then calls execute() on that handler.

```
public class CommandBusImpl implements CommandBus {
[...]
    public void execute(Command command) {
         registry.getHandler(command).execute();
    }
[...]
}
```

Example 2

An example of a possible command could be an abstract DriveCommand. For this command we would have a DriveHandler which would be forwarded that command if it was sent to the *execute* method. It is important to note that the application layer is not domain specific and it’s also not specific to a specific format of communication. Whether the command was sent in *http* or *sql*, the application handles it in the same way.

## The Framework

The framework layer is the outer layer of the hexagonal architecture. It handles communications that are coming from the outside and transfers outside communications into objects that can be utilized in the inner layers.
If we had http communications, we would want to create a controller in the framework that could handle http request and response objects. The function below in the HttpController class retrieves parameters from a HttpRequest object. Wrapped in a Command object, it forwards these parameters to the CommandBus. Not shown in this code excerpt is how the HttpResponse object will be updated at the end of the method.

```
public class HttpController extends BaseController {
[...]
    public void process(HttpRequest request, HttpResponse response) {
        [...]
        commandBus.execute(new Command(parameters));
        [...]
     }
[...]
}
```

## Brief Review of the Hexagonal Architecture

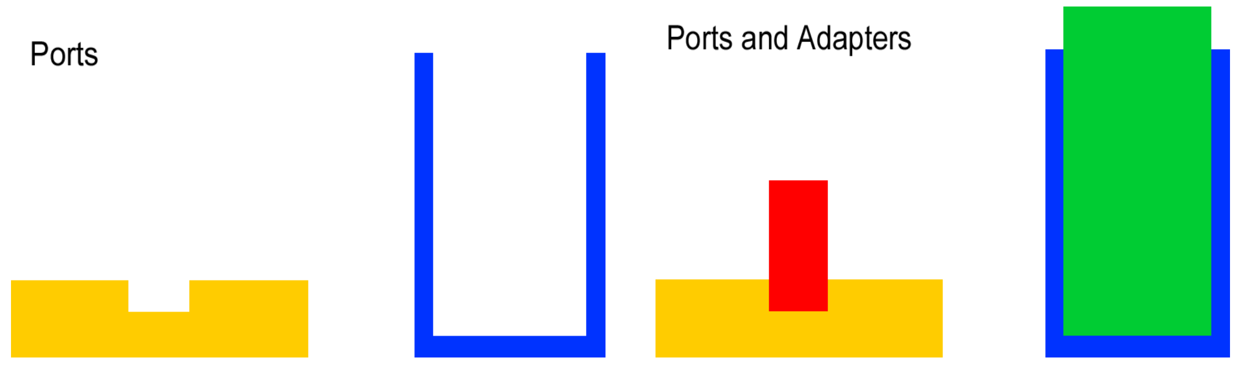
Since it can be confusing to look at each part of the **Hexagonal Architecture** separately, I'm going to take a moment to briefly go through the entire structure again. Outside the hexagon we have our communications with the system such as http, rest, binary, soap, or other hexagons. The communications these other systems send are received by the framework which transfers these communications into objects that can be worked on. The application then takes the command from the framework and sends it to its specific handler in the domain. The domain responds to the command in a technology independent way and sends its response all the way back to the framework. The framework will then convert this technology independent response into a technology dependent response.

## Commands (Use Cases)

A Command or Use Case is an operation that our system can perform, for example "MakeCall", "ReceiveCall" or "SendSMS", independent on a specific technology (soap, rest, http, binary…). While one team designs the use cases in an unfinished system, another team could implement the technical interfaces to handle them.

## Ports and Adapters

**Hexagonal Architecture** can also be called **Ports and Adapters** as illustrated in the figure below. Each protocol (http, rest, soap, sql, binary...) our system supports, or each side of the hexagon, represents a "Port" in and out of our application. A port is represented as an interface, its concrete implementation is an adapter of that port. The interface(port) is in one layer, its implementation (adapter) in the next inner layer.



## Boundaries

In hexagonal architecture each layer 'protects' itself from the layer around it with a boundary. A boundary divides the layers and allows each one to act independently, you could imagine it a wall. These boundaries force all of the dependencies to come from the outside in. So how do we actually achieve that?

We do this with inversion of control or dependendency injection, which means using yet more interfaces. In short, that means the Service classes are “plugged” externally, without any direct coupling - the communication is based only on technology independent interfaces.

In conclusion, hexagonal architecture allows us to increase the maintainability of our code by allowing different aspects of the code to be independent of one another. Communication with outside systems, the handling of commands, and the connection between the two can all be handled by independent teams (thanks to the boundaries between the layers) and can all be connected by a series of well constructed ports.

Thanks for reading!