

# Hexagonal Architecture

---

4 WEEKS AGO

code

---

I recently [gave a talk on Hexagonal Architecture](#) at Laracon NYC. The feedback was great, but seemed to have left people wanting for some extra explanation and of course examples. This is an attempt to expand on the ideas of that presentation.

- [Video of Talk](#)
- [Slides for Talk](#)

I found Hexagonal Architecture to be a good expression of how I think about code. In fact, when I wrote Implementing Laravel, I was actually espousing some ideals of Hexagonal Architecture without knowing it.

Hexagonal Architecture defines conceptual layers of code responsibility, and then points out ways to decouple code between those layers. It's helped clarify when, how and why we use interfaces (among other ideas).

Hexagonal Architecture is **NOT** a new way to think about programming within a framework. Instead, it's a way of describing "best practices" - practices that are both old and new. I use quotes because that's a bit of a loaded phrase. Best practices for me might not be best practices for you - it depends on what technical circles we engage in.

However, Hexagonal Architecture espouses common themes we'll always come across: decoupling of code from our framework, letting our application express itself, using a framework as a means to accomplish tasks in our application, instead of being our application itself.

## Beginnings

The name for Hexagonal Architecture is brought to us (so far as I can tell) by Alistair Cockburn. He [outlines the architecture](#) very well on his website.

It's intent:

|

Allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases.

# Why a Hexagon

The article takes on a shape of a hexagon. The number of sides is actually arbitrary. The point is that it has many sides. Each side represents a "port" into or out of our application.

A port can be thought of as a vector for accepting requests (or data) into an application. For example, an HTTP port (browser requests, API) can make requests on our application. Similarly, a queue worker or other messaging protocol (perhaps AMQP) can also make a request on our application. These are different ports into our application, but are also part of the "request port". Other ports could include those for data access, such as a database port.

## Architecture

Why do we even talk about Architecture?

We talk about architecture because we want our applications to contain two attributes:

1. High Maintainability
2. Low Technical Debt

These are, in fact, the same thing. To word this succinctly: We want our applications to be easy to work with. We want to make future changes easy.

## Maintainability

Maintainability is the absence (reduction) of technical debt. A maintainable application is one that increases technical debt at the slowest rate we can feasibly achieve.

Maintainability is a long-term concept. Applications in their early form are easy to work with - they haven't yet been formed and molded by the early decisions of the developers working on them. New features and libraries are added quickly and easily.

However, as time goes on, applications can get harder to work on. Adding features might conflict with current functionality. Bugs might hint at systemic issues, which may require large changes in code to fix (and to help clarify overly complex code).

A good architecture early on in a project can help prevent such issues.

What kinds of maintainability are we looking for? What are measures of a highly maintainable application?

1. Changes in one area of an application should affect as few other places as possible
2. Adding features should not require large code-base changes
3. Adding new ways to interact with the application should require as few changes as possible
4. Debugging should require as few work-arounds and "just this once" hacks as possible
5. Testing should be relatively easy

I use the word "should" because there's no perfectly coded application in existence. We want to make our applications easy to work with, but trying for "perfect" becomes a waste of time and an over-exertion of mental energy.

If you think you're spinning your wheels over "the right way" to do something, then just "get it done". Come back to the problem later, or keep your code in its "it just works" state. There's no perfectly coded application in existence.

## Technical Debt

Technical debt is the debt we pay for our (bad) decisions, and it's paid back in time and frustration.

Applications all incur a base-line technical debt. We need to work within the confines and limitations of our chosen persistence mechanisms, language, frameworks, tooling, teams and organizations!

Bad architectural decisions made early on compound themselves into larger and larger issues.

For every bad decision, we end up making work-arounds and hacks. Some of these bad decisions aren't blatantly obvious - we may simply make a class that "does too much" or mixes multiple concerns.

Smaller, yet equally bad decisions during development similarly also create issues. Luckily, these don't necessarily compound themselves like early architectural "mistakes" can. A solid basis reduces technical debt's rate of growth!


So, we want to reduce as many bad decisions as possible, especially early on in a project.

**We make a discussion of architecture so that we can focus on increasing maintainability and decreasing technical debt.**

How do we make maintainable applications?

We make them easy to change.

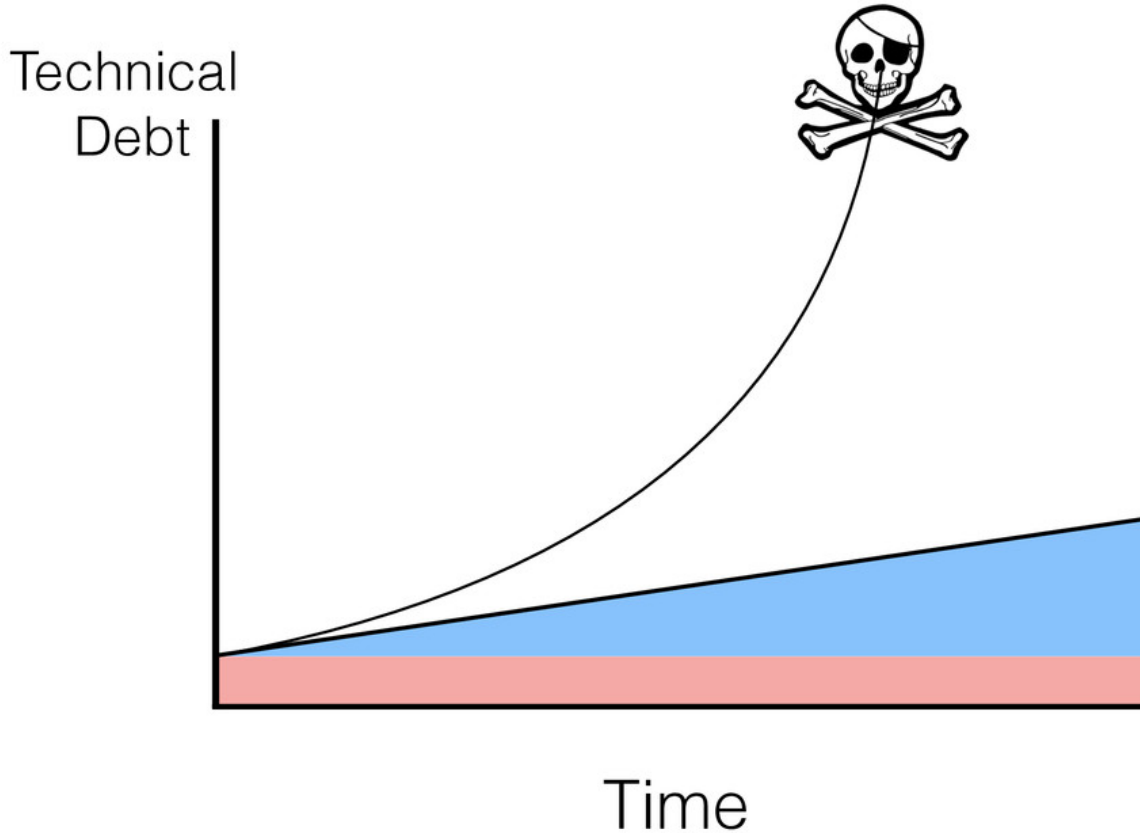
How do we make our applications easy to change? We...



Identify the aspects  
that **vary** and  
separate them from  
what stays **the**  
**same**

We'll go back to this point quite a few times in the following explanations.

# Maintainability



## Interfaces and Implementations

Let's take some time to discuss something (seemingly) basic in the world of OOP: Interfaces.

Not all languages (notably: Python & Ruby) have explicit Interfaces, however conceptually the same goals can be accomplished in such languages.

You can think of an interface as contract, which defines an application need. If the application need can be or must be fulfilled by multiple implementations, than an interface can be used.

In other words, we use interfaces when we plan on having or needing multiple *implementations* of an *interface*.

For example, if our application sends notifications, we might define a notification interface. Then we can implement an SES notifier to use Amazon SES, a Mandrill notifier to use Mandrill and others implementations for other mail systems.

The interface ensures that particular methods are available for our application to use, no matter what implementation is decided upon.

For example, the notifier interface might look like this:

```
interface Notifier {  
  
    public function notify(Message $message);  
}
```

We know any implementation of this interface **must** have the `notify` method. This lets us define the **interface** as a dependency in other places of our application.

The application doesn't care which implementation it uses. It just cares that the `notify` method exists for it to use.

```
class SomeClass {  
  
    public function __construct(Notifier $notifier)  
    {  
        $this->notifier = $notifier;  
    }  
  
    public function doStuff()  
    {  
        $to = 'some@email.com';  
        $body = 'This is a message';  
        $message = new Message($to, $body);  
  
        $this->notifier->notify($message);  
    }  
}
```

See how our `SomeClass` class doesn't specify a specific implementation, but rather simply requires a subclass of `Notifier`. This means we can use our SES, Mandrill or any other implementation.

This highlights an important way interfaces can add maintainability to your application. Interfaces make changing our application notifier easier - we can simply add a new implementation and be done

with it.

```
class SesNotifier implements Notifier {  
  
    public function __construct(SesClient $client)  
    {  
        $this->client = $client;  
    }  
  
    public function notify(Message $message)  
    {  
        $this->client->send([  
            'to' => $message->to,  
            'body' => $message->body]);  
    }  
}
```

In the above example, we've used an implementation making use of Amazon's Simple Email Service (SES). However, what if we need to switch to Mandrill to send emails, or even switch to Twilio, to send SMS?

As we've seen, we can easily make additional implementations and switch between those implementations as needed.

```
// Using SES Notifier  
$sesNotifier = new SesNotifier(...);  
$someClass = new SomeClass($sesNotifier);  
  
// Or we can use MandrillNotifier  
  
$mandrillNotifier = new MandrillNotifier(...);  
$someClass = new SomeClass($mandrillNotifier);  
  
// This will work no matter which implementation we use  
$someClass->doStuff();
```

Our frameworks make liberal use of interfaces in a similar fashion. In fact, frameworks are useful *because* they handle many possible implementations we developers may need - for example,

different SQL servers, email systems, cache drivers and other services.

Frameworks use interfaces because they increases the maintainability of the framework - it becomes easier to add or modify features, and easier for us developers to extend the frameworks should we need to.

The use of an interface helps us properly **encapsulate change** here. We can simply make new implementations as needed!

## Going Further

Now, what if we need to add some functionality around individual (or all) implementations? For example, we may need to add logging to our SEE implementation, perhaps to help debug an issue we're having.

The most obvious way, of course, is to add code directly to the implementations.

```
class SesNotifier implements Notifier {  
  
    public function __construct(SesClient $client, Logger $logger)  
    {  
        $this->logger = $logger;  
        $this->client = $client;  
    }  
  
    public function notify(Message $message)  
    {  
        $this->logger->logMessage($message);  
        $this->client->send([...]);  
    }  
}
```

Adding the logger directly to the concrete implementation may be OK, but our implementation is now doing two things instead of one - we're mixing concerns. Furthermore, what if we need to add logging to all implementations? We'd end up with very similar code in each implementation, which is hardly DRY. A change in how we add logging means making changes in each implementation. Is there an easier way to add this functionality in a way that's more maintainable? Yes!

Do you recognize some of the SOLID principles being implicitly discussed here?



To clean this up, we can make use of one of my personal favorite design patterns - the [Decorator Pattern](#). This makes clever use of interfaces in order to "wrap" a decorating class around an implementation in order to add in our desired functionality. Let's see an example.

```
// A class wrapping a Notifier with some Logging behavior
class NotifierLogger implements Notifier {

    public function __construct(Notifier $next, Logger $logger)
    {
        $this->next = $next;
        $this->logger = $logger;
    }

    public function notify(Message $message)
    {
        $this->logger->logMessage($message);
        return $this->next->notify($message);
    }
}
```

Similar to our other Notifier implementations, the `NotifierLogger` class **also** implements the `Notifier` interface. We can see, however, that it doesn't actually notify anything. Instead, it accepts another Notifier implementation in its constructor, calling it "\$next". When run, `NotifierLogger` will log the Message data and *then* pass the message onto the real notifier implementation.

We can see that the logger decorator logs the message, and then passes the message off to the notifier to actually do the notifying! If you need to, you can reverse the order of these so the logging is done *after* the notification is actually sent, so you can also log the results of the sent notification, instead of simply logging the message being sent.

In this way, the `NotifierLogger` "decorates" the actual notifier implementation with the logging functionality.

The best part is that the consuming class (our `SomeClass` example above) doesn't care that we pass in a decorated object. The decorator also implements the expected interface, so the requirements set by `SomeClass` are still fulfilled!

We can chain together multiple decorators also. Perhaps, for example, we want wrap the email notifier with an SMS notifier that sends a text message in addition to sending an email. In that example, we're

adding an additional notifier implementation (SMS) on top of an emailing implementation.

We aren't limited to adding additional concrete notifying implementations, as our logging example shows. A few additional examples can include updating the database, or adding in some metric gathering code. The possibilities are endless!

The ability to add additional behaviors, while keeping each class only doing one thing, **and** still giving us the freedom to add additional implementations, is very powerful - changing our code becomes much easier!

The Decorator Pattern is just one design pattern of *many* that make excellent use of **interfaces to encapsulate change**. In fact, almost all of the classic design patterns make use of interfaces.

Furthermore, almost all design patterns exist to make future changes easier. This is not a coincidence. Making a study of design patterns (and *when* to use them) is a critical step towards making good architectural decisions. I suggest the Head First Design Patterns book for further reading on design patterns.

I'll repeat: **Interfaces are a central way of encapsulating change**. We can add functionality by creating a new implementation, and we can add behaviors onto existing implementations - all without affecting other areas of our codebase!

Once properly encapsulated, functionality can more easily be changed. Easily changed codebases increase application maintainability (they're easier to change) by reducing technical debt (we've invested time in making changes easier to accomplish).

That was quite a lot on the topic of interfaces. Hopefully that helped clarify some of the important use cases of interfaces, and gave you a taste of how some design patterns make use of them to help make our applications more maintainable.

## Ports and Adapters

Now, finally we can begin to discuss the meat of Hexagonal Architecture.

Hexagonal Architecture, a layered architecture, is also called the Ports and Adapters architecture. This is because it has the concept of different ports, which can be adapted for any given layer.

For example our framework will "adapt" a SQL "port" to any number of different SQL servers for our application to use. Similarly, we can create interfaces at key points in our application for other layers to

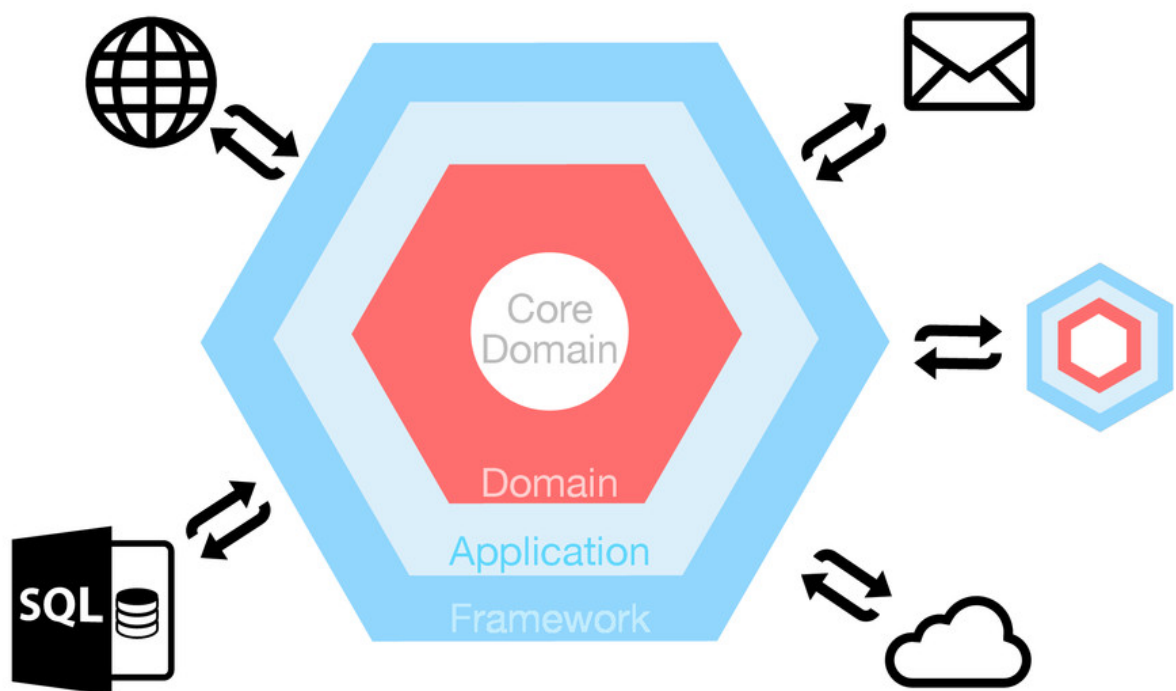
implement. This lets us create multiple adaptations for those interfaces as needs change, and for testing. This is also the basis for decoupling our code between layers.

Creating interfaces for portions of our application that may change is a way to encapsulate change. We can create a new implementations or add more features around an existing implementation as needed with strategic use of interfaces.

Before returning to the concept of Ports and Adapters, let's go over the layers of the Hexagonal Architecture.



# The Hexagon



## Layers: Code

The Hexagonal Architecture can describe an application in multiple layers.

The goal of describing the architecture in layers is to make *conceptual* divisions across functional areas of an application.

The code within the layers (and it their boundaries) should describe how the layers communicate with each other. Because layers act as ports and adapters for the other layers inside and surrounding them, describing the communication between them is important.

Layers communicate with each other using interfaces (ports) and implementations (adapters).

Each layer has two elements:

1. The Code
2. The Boundary

The **code** inside of a layer is just what it sounds like - actual code, doing things. Often times this code acts as adapters to ports defined in other layers, but it can also be any code we need (business logic or other services).

Each layer also has a **boundary** between itself and an outside layer. At the boundary we find our "ports". These ports are interfaces that the layer defines. These interfaces define how outside layers can communicate to the current layer. We'll go into this in more detail.

## Domain Layer

The inner-most layer is the Domain Layer. This layer contains your business logic and defines how the layer outside of it can interact with it.

Business logic is central to your application. It can also be described as 'policy' - rules your code must follow.

The domain layer and its business logic define the behavior and constraints of your application. It's what makes your application different from others. It's what gives your application value.

If you have an application with a lot of behavior, your application can have a rich domain layer. If your application is more of a thin layer on top of a database (many are!), this layer might be "thinner".

In addition to business logic (the Core Domain), we often also find supporting domain logic within the Domain Layer, such as Domain Events (events fired at important points in the business logic) and use-cases (definitions of what actions can be taken on our applications).

What goes inside of Domain Layer is the subject of books by themselves - especially if you are interested in Domain Driven Design, which goes into much detail on how to create applications which closely match the real business processes you are codifying.

Some "Core" Domain Logic:

```
<?php namespace Hex\Tickets;

class Ticket extends Model {

    public function assignStaffer(Staffer $staffer)
    {
        if( ! $staffer->categories->contains( $this->category ) )
        {
            throw new DomainException("Staffer can't be assigned to ".$this->category);
        }

        $this->staffer()->associate($staffer); // Set Relationship

        return $this;
    }

    public function setCategory(Category $category)
    {
        if( $this->staffer instanceof Staffer &&
            ! $this->staffer->categories->contains( $category ) )
        {
            // Unset staffer if can't be assigned to set category
            $this->staffer = null;
        }

        $this->category()->associate($category); // Set Relationship

        return $this;
    }
}
```

Above we can see a **constraint** within the `assignStaffer` method. If the provided Staffer is not assigned a Category under which this Ticket falls, we throw an exception.

We also see some **behavior**. If the Category of the Ticket is changed, and the current Staffer is unable to be assigned a Ticket of this Category, we unset the Staffer. We do not throw an exception - instead we allow the opportunity to set a new Staffer when the Category is changed.

These are both examples of business logic being enforced. In one scenario, we set a constraint by

throwing an error when something is set incorrectly. In another scenario, we provide behavior - once a Category is changed, users must have the opportunity to re-assign a Staffer who is able to handle that Category of Ticket.

Inside of the Domain layer, we may also see some supporting Domain Logic:

```
class RegisterUserCommand {  
  
    protected $email;  
  
    protected $password;  
  
    public function __construct($email, $password)  
    {  
        // Setter email/password  
    }  
  
    public function getEmail() { ... } // return $email  
  
    public function getPassword() { ... } // return $password  
}  
  
class UserCreatedEvent {  
  
    public function __construct(User $user) { ... }  
}
```

Above we have some supporting (but very important) domain logic. One is a Command (aka a Use Case) which defines a way in which our application can be used. It simply takes in the data needed to create a new user. We'll see how that's used later.

Another is an example Domain Event, which our application might dispatch after a user is created. These are important to the things that occur within a domain and so belong in the domain layer. They are not system events often found within the plumbing of our frameworks such as "pre-dispatch", often used for hooks in case framework behavior needs to be extended.

## Application Layer

Just outside of the Domain Layer sits the Application layer. This layer orchestrates the use of the

entities found in the Domain Layer. It also adapts requests from the Framework Layer to the Domain Layer by sitting between the two.

For example, it might have a handler class handle a use-case. This handler class in the Application Layer would accept input data brought in from the Framework Layer and perform the actions needed to accomplish the use-case.

It might also dispatch Domain Events raised in the Domain Layer.

This layer represents the outside layer of the code that makes up the application.

Of course, you can see that outside of the Application Layer sits the "Framework Layer". The Framework layer contains code that helps your application (perhaps by accepting an HTTP request or sending an email), but is **not** your application itself.

## Framework Layer

The Framework Layer sits outside of the Application Layer. It contains code that your application uses but it is not actually your application. This is often literally your framework, but can also include any third-party libraries, SDKs or other code used. Think of all the libraries you bring in with Composer (assuming you use PHP). They are not your framework, but they do act in the same layer - performing tasks to handle application needs.

The Framework Layer implements services defined by the application layer. For example, it might implement a notification interface to send emails or SMS. Your application knows it needs to send notifications, but it may not need to care how they are sent (email vs SMS for example).

```
class SesEmailNotifier implements Notifier {  
  
    public function __construct(SesClient $client) { ... }  
  
    public function notify(Message $message)  
    {  
        $this->client->sendEmail([ ... ]); // Send email with SES particulars  
    }  
}
```

Another example is an event dispatcher. Inside the Framework Layer might be code to implement an event dispatcher interface defined in the application layer. Again, the application knows it has events to

dispatch, but it doesn't necessarily need to have its own dispatcher - our framework likely already has one, or we might pull in a library to handle the implementation details of dispatch events.

The Framework Layer also adapts requests from the outside to our Application Layer. For example, it's responsible for accepting HTTP requests, gathering user input and routing this request/data to a controller. The Framework Layer can then call an application use-case, pass it the input data, and have the application handle the use case (rather than handling it itself inside of a controller).

In that way, the framework can sit between all requests made on the application externally (us, setting there using a browser) and the application itself (Application Layer and deeper). The Framework Layer is adapting raw requests into our application.

## Communication Between Layers: Boundaries

Now that we've seen what code goes inside of each layer, let's talk about an interesting part of each layer: How they communication with each other.

As mentioned, each layer also defines how other layers can communicate with it. Specifically, each layer is responsible for defining how the next outside layer *can* communicate with it.

The tool for this is the interface. **At each layer boundary, we find interfaces..** These interfaces are the ports for the next layer to create adapters for.

We saw this in the notifier and event dispatcher examples above.

The Application Layer will implement interfaces (make adapters of the ports) defined in the Domain Layer. It will also contain code for other concerns it may have.

Let's go through each layer's boundary and see how this works.

## Domain Layer

At the boundary of our Domain Layer, we find definitions in how the outside layer (the Application Layer) can communicate with the domain objects/entities found in the Domain Layer.

For example, our Domain Layer might contain a command (use case). Above, we saw an example `RegisterUserCommand`. This command is pretty simple - you might call it a simple DTO (Data Transfer Object).

Our Domain Layer defines Use Cases, but it's job is just to say "This is how you can use me".

Remember, the Application Layer is responsible for orchestrating the Domain Layer code in order to



accomplish a task. So, we have communication across boundaries - the Domain Layer defines how it should be used, and the Application Layer uses those definitions (in part) to accomplish the defined use cases.

Our Application Layer, therefore, needs to know how to handle this command to register a user. Since we have communication across these layers, let's define an interface "at the boundary" of the Domain Layer:

```
interface CommandBus {  
  
    public function execute($command);  
}
```

So, we've told our Application Layer how to "execute" a command, using a Command Bus. The Command Bus is simple - it just needs to have a method `execute` available so that implementations can process a Command.

Our Domain Layer contains this CommandBus interface, so that our Application Layer can implement the CommandBus interface. The interface is the port, and the implementations of it are the adapters to that port.

Cognitively, we've done a few things:

1. Recognized that CommandBus may be processed in a few ways
2. Recognized that because of that, we may have multiple implementations of the CommandBus
3. Recognized that since our Application Layer orchestrates the use of the Domain Layer, it makes sense for the CommandBus implementation(s) to exist in the Application Layer (they'll orchestrate the use of the Commands defined in the Domain Layer).
4. We've decoupled the Domain Layer from the Application layer by using an interface. This also defines communication between layers. Command Buses have a defined way to execute commands.

## Application Layer

So our Application Layer can implement a Command Bus. That's right in the middle of this layer - implementations (adapters) to other layers.

However the Application Layer has its own needs to communicate. The Application Layer might need to send a notification to a user. The Framework has the tools to do so - it can send emails, and we can

pull in libraries to send SMS messages or other notification transports. The framework is a good place to implement our notification needs.

So, we have communication between layers. The Application Layer needs to send a notification, and we know it can use libraries in the Framework layer to do so. You know what's coming up: another interface!

```
interface Notifier {  
  
    public function notify(Message $message);  
}
```

The Application Layer is defining how it will be communicating to the Framework Layer. In fact, it's defining how it will use the Framework Layer, without actually coupling to it. Interfaces (ports) and implementations (adapters) give us the freedom to change the adapters. We don't tie our application to the Framework Layer in this way.

The interface defined "at the boundary" of the Application Layer is defining how the Application Layer will communicate with the Framework Layer.

This is very much conceptual and is not meant to be taken as concrete rules. If you find yourself asking "What if my **Domain Layer** needs a third party library found in the framework?", fear not!

If that's a need, then define an interface and implement it using that library! You have to make your code work after all - worrying about breaking the "rules" from some dude or dudette on the internet won't get you anywhere!

The key point is to make sure to decouple concerns (hint: You're doing so by defining an interface) so functionality is easy to switch/modify later. There's no need to live and die by what I write here. There's no "doing it wrong". To repeat: **There's no doing it wrong**. There's just varying levels of severity in how you shoot yourself in the foot.

A good starting place to read up on the topic of requiring [third party libraries in your "domain layer"](#) is in this linked thread.

## Framework Layer

So far we've seen the boundary in our Domain Layer and in our Application Layer. These two layers

both communicate with layers under our control. The Domain Layer communicates with the Application Layer. The Application Layer communicates with the Framework Layer. Who does the Framework Layer communicate with?

The outside world! That world is one filled with protocols - mostly TCP based protocols (such as HTTP/HTTPS). There is certainly lots of code in the framework layer (all the libraries we use), as well as some code we write ourselves, such as controller code and implementations of interfaces defined in the Application Layer.

What exactly is at the boundary of the Framework Layer and the "layer" outside of it, however? Well more interfaces (AND implementations of those interfaces) of course!

Most of our frameworks have code that takes care of talking to the outside work - HTTP implementations, various SQL implementations, various email implementations, and so on.

Luckily, for the most part, we don't have to care about the boundary between the framework layer and the outside world. That's the framework's concern; our benevolent framework creators have taken care of this for us.

This is, arguably, the whole point of a framework. Frameworks provide tools for us to communicate to the world outside of our application, so that we don't need to write that boiler plate code ourselves.

We don't usually need to add to the Framework Layer at its boundary, but of course this isn't always the case. If we're building an API, HTTP level concerns become an issue we need to work through. This usually means implementing [CORS](#), HTTP caching, HATEOS and other specifics in how our application handles HTTP level requests - concerns that are important to our application, but aren't likely concerns of the Domain Layer or even the Application Layer.

## Use-Cases/Commands

Earlier in this writing, I've made mention of "Use Cases" and "Commands." Let's go deeper into what these are.

Hexagonal Architecture isn't just about communication between layers on the micro level (interfaces, implementations for ports and adapters). There's also a concept of the **Application Boundary**, a macro-level concept.

This boundary separates our application as a whole from everything else (both framework and communication with the outside world).

We can strictly define how the outside world can communicate with our application. We do this explicitly by creating "Use Cases" (also called "Commands"). These essentially are classes which name actions that can be taken. For example, our `RegisterUserCommand` defines that our application can register a user. A `UpdateBillingCommand` might be the code path defined for us to update a user's billing information.

A Use Case (Command) is an explicitly defined way in which an application can be used.

Defining Use Cases has some useful side-effects. For example, we clearly and explicitly can see how our application "wants" to be interacted with. This can strictly follow the business logic that our application needs to perform. Use Cases are also useful for clarity amongst a team of developers. We can plan use cases ahead of time, or add them as needed, but we find it harder to create odd logic outside of use cases, which don't seem to fit business logic.

## How do we define use cases?

We saw some examples already - What we can do is create objects representing an application use case. We'll call such an object a "Command." These commands can then be processed by our application "Command Bus", which will call a command "Handler" to orchestrate the execution of the use case.

So, we have three actors in command processing:

1. A Command
2. The Command Bus
3. A Handler

A Command Bus accepts a Command in its `execute` method. It then does some logic to find and instantiate a Handler for that Command. Finally, the Handler's `handle` method is called, running the logic to fulfill the Command.

```
class SimpleCommandBus implements CommandBus {  
  
    // Other methods removed for brevity  
  
    public function execute($command)  
    {  
        return $this->resolveHandler($command)->handle($command);  
    }  
}
```

Notice that we are taking coordinating logic we often see within a controller and moving it into a Handler. This is good, as we want to decouple from our framework layer, giving us the benefit of protecting our application from changes in the framework as much as possible (another form of maintainability), and allowing us to run the same code in other contexts (CLI, API calls, etc).

The main benefit of use cases is that we create an avenue to re-use code run in multiple contexts (web, API, CLI, workers, etc).

For example, the code to create a new user in web, API and CLI can be almost exactly the same:

```
public function handleSomeRequest()
{
    try {
        $registerUserCommand = new RegisterUserCommand(
            $this->request->username, $this->request->email, $this->request->password );

        $result = $this->commandBus->execute($registerUserCommand);

        return Redirect::to('/account')->with([ 'message' => 'success' ]);
    } catch( \Exception $e )
    {
        return Redirect::to('/user/add')->with( [ 'message' => $e->getMessage() ] );
    }
}
```

What might change between contexts is how we get user input and pass it into the command, as well as how we handle errors - but those are mostly framework-level concerns. Our application code doesn't need to care if its being used in an HTTP browser request, an HTTP api request or any other request type.

That's where we see the potential of Use Cases. We can re-use them in every context our application can be used (HTTP, CLI, API, AMQP or queue messaging, etc)! Additionally, we've firmly set up a boundary between a framework and our application. The application can, potentially, be used separately from our framework.

That being said, we still might use a framework to implement some application level needs, such as validation, event dispatching, database access, email drivers and many other things our frameworks can

do for us! The Use-Case application boundary is just one aspect of Hexagonal Architecture.

Use Case/Command's main benefit is keeping code DRY - we can re-use the same use case code in multiple contexts (web, API, CLI, etc).

# All the Contexts

- Web
- API
- CLI
- Queue
- Event Handler



Use Cases also serve to further decouple your application from the framework. This gives some protection from framework changes (upgrades, etc) and also makes testing easier.

Taken to an extreme, you can potentially switch frameworks without re-coding our application. However, I consider this the **edgiest edge case to ever case edges**. It's neither a realistic nor a worthy goal. We want our applications to be easy to work with, not fulfill some arbitrary metric or rare use case.

## Example Command, Command Bus and Handler

First, our application knows it needs Commands. It also knows it needs a Bus to execute the commands. Finally, we need a Handler to orchestrate the execution of the command.

Commands are, in a sense, arbitrary. Their purpose is simply to exist. Their mere existence fulfills the role of defining how an application should be used. The data they demand tells us what data is needed to fulfill the command. So, we don't really need to interface a Command. They are simply a name (a description) and a DTO (data transfer object).

```
class RegisterUserCommand {  
  
    public function __construct(username, email, password)  
    {  
        // set data here  
    }  
  
    // define getters here  
}
```

So, our Command used to register a new user is quite simple. All at once, we provide an explicit definition of one way our application can be used, and what data should accompany that command.

Our Handlers are a bit more complex. They are coupled to a Command in that they expect the data from a Command to be available. This is a spot of tight coupling. Changing some business logic may result in changing the Handler, which may result in changing the Command. As these are all concerns of the all-important business logic, this tight coupling within Domain concerns is deemed "OK".

While Commands are simple DTO's (containing various data), Handlers have behavior, which the Command Bus makes use of. The handlers, being in the Application Layer, orchestrate the use of Domain entities to fulfill a Command.

The CommandBus used to execute a command must be able to execute all Handlers, and so we'll define a Handler interface to ensure the Command Bus always has something it can work with.

```
interface Handler {  
  
    public function handle($command);  
}
```

Handlers then must have a `handle` method, but are free to handle the fulfillment of the Command in any way it needs. For our `RegisterUserCommand`, let's take a look at what its Handler might look like:

```
class RegisterUserHandler {  
  
    public function handle($command)  
    {  
        $user = new User;  
        $user->username = $command->username;  
        $user->email = $command->email;  
        $user->password = $this->auth->hash($command->password);  
  
        $user->save();  
  
        $this->dispatcher->dispatch( $user->flushEvents() );  
  
        // Consider also returning a DTO rather than a class with behavior  
        // So our "view" layers (in whatever context) can't accidentally affect  
        // our application - it can just read the results  
        return $user->toArray();  
    }  
}
```

We can see that our handler orchestrates the use of some Domain Entities, including assigning data, saving it and dispatching any raised events (if our entities happen to raise events).

Similar to our interface example where we used a Decorator to add some extra behavior to our notifier, consider what behaviors might be useful to add to a CommandBus or Handler.

Lastly, we'll discuss the most interesting of our three actors - the Command Bus.

The Command Bus can have multiple implementations. For example, we can use a synchronous Command Bus (running commands as they are received) or perhaps we can create a queue Command Bus, which runs all queued commands only when the queue is flushed. Or perhaps we choose to create an asynchronous Command Bus, which fires jobs into a worker queue, to be worked on as the jobs are received, out of band of the current user's request.

Since we have multiple possible implementations, we'll interface the Command Bus:

```
interface CommandBus {
```



```
public function execute($command);  
}
```

We've seen a simple implementation of this already. Let's see it a bit more fleshed out:

```
class SimpleCommandBus implements CommandBus {  
  
    public function __construct(Container $container, CommandInflector $inflector)  
    {  
        $this->container = $container;  
        $this->inflector = $inflector;  
    }  
  
    public function execute($command)  
    {  
        return $this->resolveHandler($command)->handle($command);  
    }  
  
    public function resolveHandler($command)  
    {  
        return $this->container->make( $this->inflector->getHandlerClass($command) );  
    }  
}
```

The `CommandInflector` can use any strategy to get a Handler from a Command class. For example, `return str_replace('Command', 'Handler', get_class($command));` is effective. It's simple, and only requires you keep a certain directory structure for Handlers and Commands (assuming PSR-style autoloading). How you accomplish this is up to your and your project needs.

What else might we use besides a "Simple" Command Bus? Well we might instead call it a "SynchronousCommandBus", as it's processing commands as they come - synchronously. This infers that we might also consider creating an `AsynchronousCommandBus`. Instead of processing the commands directly, it might pass them into a worker queue to get processed whenever the job is reached, out of band of the current request.

In addition to different implementations of the Command Bus, we can also add onto our existing ones with more Decorators. For example, I find it useful to wrap some validation around a Command Bus, so that it attempts to validate the Command data before processing it.

```
class ValidationCommandBus implements CommandBus {

    public function __construct(CommandBus $bus, Container $container, CommandInflector $inflector)
    {
        $this->validate($command);
        return $this->bus->execute($command);
    }

    public function validate($command)
    {
        $validator = $this->container->make($this->inflector->getValidatorClass($command))
        $validator->validate($command); // Throws exception if invalid
    }
}
```

The `ValidationCommandBus` is a decorator - it does the validation, and then passes the Command off to another Command Bus to execute it. The next Command Bus might be another decorator (perhaps a logger?) or might be the actual Bus doing the processing.

So, combined with different types of Command Buses and possible behaviors we can add on top of our Command Buses, we have a pretty powerful way to handle our application Commands (Use Cases) being called!

And these ways are insulated as much as possible from layers outside of the Application. The Framework Layer (and beyond) does not dictate how the application is used - the application itself dictates its usage.

## Dependencies

Not explicitly mentioned was the notion of dependencies. Hexagonal Architecture espouses a one-way flow of dependencies: From the outside, in. The Domain Layer (the inner-most layer) should not depend on layers outside of it. The Application Layer should depend on the Domain Layer, but not on the Framework Layer. The Framework Layer should depend on the Application Layer, but not on externalities.

We talked about interfaces as the primary means to encapsulate change. These let us define how

communication between layers was accomplished within our application without coupling layers together. Thinking about dependencies is another way of saying the same thing. Let's see how.

## Dependencies: Moving In

When our data/logic is flowing "in", dependencies are easier to visualize. If an HTTP request reaches our server, we need code to handle it, otherwise nothing happens. External HTTP requests require our Framework Layer to interpret a request to code. If our Framework interprets a request and routes it to a controller, the controller needs something to act upon. Without our Application Layer, it has nothing to do. The Framework Layer depends on the Application Layer. Our Application Layer needs the Domain Layer in order to have something to orchestrate - it depends on the Domain Layer in order to fulfill a request. The Domain Layer (for the most part) depends on the behavior and constraints found within itself.

When we talk about a request starting from the outside, and the flow of code for handling a request moving inward, dependencies are fairly easy to spot. The outside layers depend on the inside layers, but they can also be ignorant of what the inner layers are doing - they just need to know the methods to call and the data to pass. Implementation details are safely encapsulated away in their proper place. Our use of interfaces between layers has seen to that.

## Dependencies: Going Out (IoC)

Dependencies going out are a little more complex. This describes what our application does in response to the request: Both in processing a request and responding to it when a request is processed. Let's look at some examples:

Our Domain Layer will likely need database access to create some domain entities. This means our application "depends" on some sort of data storage.

Our Application Layer may need to send a notification when it finishes a task. If this is implemented as an email notification (for example, if we're using SES), then our Application Layer can be said to depend on SES for sending a notification email.

We can see here that conceptually, our inner layers are depending on things found in layers outside of it! How do invert this?

Of course I used the term "invert" on purpose. This is the point of "**Inversion of Control**", the "I" in SOLID. Here again, our interfaces serve us well.

We Invert Control by using Interfaces. These allow our layers to inform other layers how they will be

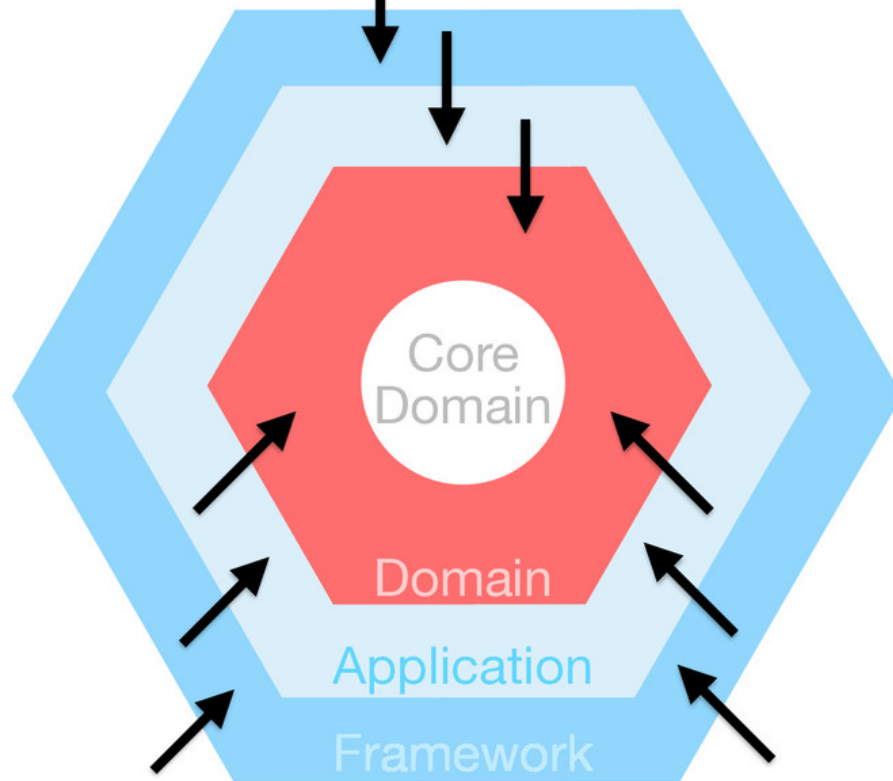
interacted with, and how they need to interact with other layers. It's up to the other layers to implement these interfaces. In this way, we are letting our inner layers dictate how they are used. This is Inversion of Control.

Our Domain Layer can define an interface for a repository class. This Repository Class will be implemented in another layer (Likely up in the Framework Layer with our Framework database classes). By using an interface, however, we are decoupling our Domain Layer from the specific persistence type used: We have the potential to change persistence models when we test, or if project needs dictate an actual technological change (lucky you, if you get to scale high).

It's a similar situation with the Notifier. Our Application Layer knows it needs to send out notifications. That's why we created the Notifier interface. Our Application doesn't need to know how the notifications are sent - it just knows it needs to define how it's to be interacted with. And so our Notifier interface, defined in the Application Layer, is implemented by an email (perhaps SES) in our Framework Layer. The Application Layer has inverted control by using an interface; it has told the outside layer how it's going to be used. The layers are decoupled as implementations can easily be switched.

So, when our logic is flowing from the "outside, in", we make use of our interfaces again. We employ the user of Inversion of Control so that dependencies keep flowing in one direction. We decouple our inner layers from outside layers, while still making using them!

# Dependencies



## Conclusion

This covers a lot of material! What you read here is the result of a lot of code architecture study. Instead of being very specific, it's a bit on the general side - we're dealing with concepts here, instead of concrete "do it this way" type rules.

Overall, Hexagonal Architecture is a description of "good" code practice. It's not a specific way to go about coding applications. Its concepts work for opinionated frameworks, as well as for the "no framework" crowd.

Hexagonal Architecture is another way to look at the same old rules we're reading about as we learn more about code architecture.