Refactoring with Clean Architecture

Matthew Knill
u6052043

A Report submitted as the assessment for an Independent Studies 12 units (COMP4560) in
The Research School of Computer Science
Australian National University
25/10/19
Acknowledgements

A great deal of time and energy was spent on this project and I would like to thank those who have helped and provided contributions to the project.

I would like to express my gratitude towards Dr Henry Gartner for his ongoing support over the course of this project. He has provided valuable advice and been highly encouraging throughout.

Finally, I would also like to express my gratitude towards my parents who have also provided me with encouragement and support.
Abstract
Clean Architecture, as proposed by Robert C. Martin, is a set of design principles that aims to keep the core business logic separate from the implementation details. [1]

This project aims to demonstrate how an existing web application can be refactored into Clean Architecture without having to start from scratch.

This report seeks to do the following:

- Identify some of the shortcomings of modern web development paradigms and motivate the use of Clean Architecture
- Describe what Clean Architecture is and why it’s useful
- Present a hypothetical case study of a customer who desires that more features be added to their existing web application
- In relation to this case study, provide an alternative approach which refactors the web application to adhere to Clean Architectural standards prior to adding the requested features
- Analyse the feasibility of this solution and its advantages and disadvantages
- Analyse the proposed solution’s applicability in refactoring other web applications to Clean Architecture

The results of this project prove that refactoring into Clean Architecture is possible though with some notable issues. The primary issues are that it can result in a high development cost for the initial refactoring and produce code that is complex. Nevertheless, for large projects, it is possible for these issues to be outweighed by code that is modular, flexible and testable. Furthermore, the results show that the refactoring process can be done iteratively and work alongside the original application whilst the refactoring is taking place. Therefore, refactoring could be favoured over starting again from scratch even if the respective development costs are similar.
# Table of Contents

Acknowledgements........................................................................................................... 2
Abstract.............................................................................................................................. 3

1 Introduction ...................................................................................................................... 6
   1.1 Web Frameworks ..................................................................................................... 6
   1.2 Adobe Flash .......................................................................................................... 6
   1.3 Other Frameworks .................................................................................................. 6
   1.4 Motivation ................................................................................................................. 6

2 Model View Controller ..................................................................................................... 7
   2.1 Model ..................................................................................................................... 7
   2.2 View ....................................................................................................................... 8
   2.3 Issues with MVC ..................................................................................................... 9

3 SOLID Principles ............................................................................................................ 9
   3.1 Single Responsibility Principle ........................................................................... 9
   3.2 Open/Closed Principle .......................................................................................... 9
   3.3 Liskov Substitution Principle .............................................................................. 10
   3.4 Interface Segregation Principle ......................................................................... 10
   3.5 Dependency Inversion Principle ........................................................................ 10

4 What is required? ............................................................................................................. 11

5 Clean Architecture .......................................................................................................... 12
   5.1 Clean Architecture Layers ................................................................................. 12
      5.1.1 Entities ........................................................................................................... 13
      5.1.2 Use Cases ...................................................................................................... 13
      5.1.3 Interface Adapters ....................................................................................... 13
      5.1.4 External Interfaces ....................................................................................... 13
      5.1.5 Other Layers ................................................................................................ 13
   5.2 Dependency Injection ............................................................................................ 13

6 Refactoring ...................................................................................................................... 15
   6.1 Case Study ............................................................................................................ 15
   6.2 Analysis ................................................................................................................... 16
      6.2.1 Replicate ...................................................................................................... 17
      6.2.2 Refactor the Controllers ......................................................................... 17
      6.2.3 Alter the Views ............................................................................................ 17
6.2.4 Test ............................................................................................................................... 17
6.3 Feasibility ........................................................................................................................................ 17
6.4 Issues with using this approach ........................................................................................................... 18
   6.4.1 Predominantly focussed on the backend ......................................................................................... 18
   6.4.2 Increased complexity .................................................................................................................... 18
   6.4.3 No longer follows conventional design patterns ............................................................................... 18
6.5 Results ............................................................................................................................................ 19
7 Conclusion ............................................................................................................................................ 19
8 Future Development ............................................................................................................................. 20
9 Appendix ............................................................................................................................................. 21
  9.1 Original MVC Web Application ........................................................................................................ 21
     9.1.1 Database .................................................................................................................................... 21
  9.2 Clean Architecture Entities Diagram ................................................................................................. 21
  9.3 Clean Architecture Use Cases Diagram ............................................................................................ 23
  9.4 Folder Structure ............................................................................................................................... 24
     9.4.1 Original Folder Structure ............................................................................................................ 24
     9.4.2 Clean Architecture Folder Structure .......................................................................................... 25
  9.5 Source Code .................................................................................................................................... 26
  9.6 Cryptocurrency Bot in a weakly typed language ............................................................................... 27
     9.6.1 Cryptocurrency Bot class diagram .............................................................................................. 27
Bibliography ............................................................................................................................................ 29
1 Introduction

Technology undoubtedly moves at an alarming rate. According to Moore’s Law, the number of transistors per silicon chip doubles every year (this was later revised to 18 months as more evidence arose). [2] While Moore’s law may be less relevant today, [3] it does show the immense amount of growth in the IT industry over the last 50 years – moving from simple calculations to increasingly complex software systems. However, due to this immense rate of growth, there has been little time to develop concrete strategies to build these systems.

1.1 Web Frameworks

Web frameworks, and in fact all software frameworks, are abstracted pieces of code that can rapidly speed up development by providing general functionality to common tasks. [4] New frameworks and data storage options constantly appear, boasting innovative advantages that make systems easier and faster to develop and run. Often the developers of these frameworks assume full compliance with the framework itself. Serious issues can arise though when integrating with other frameworks or swapping out deprecated frameworks.

1.2 Adobe Flash

As a prominent example, Adobe Flash was once the dominant web framework. Flash enabled websites to be highly interactive and graphical – a strong addition to the web at the time of its release in 1996. [5] With the introduction of HTML5/CSS3 and an array of security challenges, Flash’s usage rapidly decreased in 2018 and in 2020 it will be fully deprecated. [6] With that steady decline, developers often had to work tirelessly to rewrite a considerable amount of code to adhere to new and improved web standards. Often, the reworked web applications and websites had similar functionality to the previous. As such, this would cost companies huge sums of money and offer no real benefit other than keeping up with the current standards.

1.3 Other Frameworks

The Adobe Flash experience is by no means isolated. Other web frameworks such as JavaScript UI frameworks, have had a lifespan that was far less than that of Adobe Flash. The situation is described in a post on Stack Overflow (a developer sharing site) by Ian Allen as follows:

“Every six months or so, a new one pops up, claiming that it has revolutionized UI development. Thousands of developers adopt it into their new projects, blog posts are written, Stack Overflow questions are asked and answered, and then a newer (and even more revolutionary) framework pops up to usurp the throne.” [7]

1.4 Motivation

In comparison to common design patterns such as the Model View Controller (MVC), this project aims to propose an alternative structure using Robert C. Martin’s Clean Architecture that attempts to solve some these issues.

Oddly, Robert C. Martin in his book on Clean Architecture, does not provide many methods of improving the current architecture used in existing systems (and it is a point of criticism). [8] While having the potential of being time consuming and difficult, the following is a possible method of transforming a web application into Clean Architecture.
2 Model View Controller

The Model View Controller (MVC) is a simple and widely used design pattern that aims to separate the code into three different layers of which code can be organised into.

Note: For the purposes of this project, an initial inventory application was built using PHP and Laravel (MVC web application framework). This was used as a basis for the case study below so that it could later be refactored into Clean Architecture. Please see the case study (section 7.1) and the Appendix (section 8.2) for more information.

The following are descriptions of each of the three MVC layers:

2.1 Model

The model is responsible for the raw data of the application. This is often a database or some other storage mechanism along with some sort of abstraction to create, update and delete data (often referred to as CRUD operations).

For instance, in a very simple web application, users may be stored in a Structured Query Language (SQL) database like this:

<table>
<thead>
<tr>
<th>Name</th>
<th>Email</th>
<th>DoB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td><a href="mailto:bob@example.com">bob@example.com</a></td>
<td>23/05/95</td>
</tr>
<tr>
<td>Bill</td>
<td><a href="mailto:bill@example.com">bill@example.com</a></td>
<td>15/02/92</td>
</tr>
</tbody>
</table>

The model could then retrieve this data using a simple SQL query. Finally, the model could then represent a user in object form, if desirable, as follows:
Note: *this could be done manually or by using an Object Relational Mapper (ORM)*

### 2.2 View

The view is responsible for what the user sees and how they interact with the web application. Often, the view is written in HTML, CSS and Javascript to be view on a browser. Many MVC frameworks support a templating engine for the view. The following is an example of a view which uses Laravel's template engine (called Blade):

```html
<!DOCTYPE html>
<html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
<head>
    @include("includes.head")
</head>
<body>
    <div id="app">
        <main class="py-4">
            @foreach ($customers as $cust)
                <a href="#">{{ $cust->custName }}</a>
            @endforeach
        </main>
    </div>
</body>
</html>
```

In the above example, no back-end logic is done (i.e. no variables added/edited etc.). Variables are simply passed into the view and printed out.

The **controller** is essentially the “liaison” between the view and model. It is used to connect the user to the model and vice versa. The following is a very simple controller which gets some customers from the database and returns the *listCustomers* view along with the users retrieved from the database:

```php
class PagesController extends Controller
{
    public function customers()
    {
        $users = DB::table('customers')->get(); // get all users from database
        return view('listCustomers', ["customers" => $customers]);
    }
}
```
The final aspect of many MVC frameworks are routes which route a page URI to a respective controller and view action method. [9] The following would be a route to the customers controller above in Laravel:

Route::get('customers', 'PagesController@customers');

2.3 Issues with MVC

MVC is simple and with one of the many MVC frameworks allows for rapid development. However, for larger web applications, especially those that require many external agents, the few layers and numerous cross dependencies are simply not adequate. While the view layer of MVC is quite well defined, often the model or controller layer is bloated, messy and difficult to change. More often than not, the model is the layer that ends up bloated and usually results in tight coupling with the Database.

If we used Laravel’s Elequent Object Relational Mapper (ORM) to facilitate the database operations, we might have the luxury being able to easily switch between different persistent storage solutions. However, not only does the Elequent ORM only support certain SQL databases, if it we were to progressively move away from Laravel, all that code would be highly redundant (in fact we’d probably be better off just using in-text SQL rather than the ORM). This is because, the code would be highly tailored to the Elequent ORM which is tightly bound to the Laravel structure.

3 SOLID Principles

The SOLID Principles, as coined by Robert C. Martin, are guidelines to drive all forms of software development including web development. Appropriate application of these principles can result in code that is easy to read, test and maintain. [10]

The SOLID principles are as follows:

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

The following are descriptions of each principle:

3.1 Single Responsibility Principle

“Each Class should be responsible for a single part of the software”

Essentially this is stating that each class should only have one reason to change. For instance, we might decide to update our cost structure for an online store. In this case we don’t want to have to meticulously traverse through a myriad of different classes to simply change how the items are charged. We would rather have a Cost class that exclusively handles how items are charged. It not only is elegant, changing code in this class will likely break other parts of the code and vice-versa. [11]

3.2 Open/Closed Principle

“Classes should be open for extension, but closed for modification”
Suppose that a junior developer is tasked with adding a subscription-based cost structure to the online store in the example above. Naturally, he goes in and changes our well tested Cost class around to support the new cost structure. Unfortunately, this breaks not only the Cost class but all the code that depends on it.

This principle helps prevent errors in existing code when new features are added. By opening up the Cost class for extension, the junior developer can create a new class, say SubscriptionCost, and then implement the required subscription functionality. [10]

### 3.3 Liskov Substitution Principle

*“When extending a class, the subclass should be compatible with the superclass”*

In our online store example, the Cost class may have a totalCost method which returns a float representing the total cost of the purchase. The junior developer decides, when extending, to return a TotalSubscriptionCost object which also specifies the amount of days paid for in the subscription model. While seemingly logical in practice, it violates the Liskov Substitution Principle meaning other code that use the Cost class will not be compatible with the SubscriptionCost class. [10]

### 3.4 Interface Segregation Principle

*“Clients should not be forced to depend on methods that they do not use”*

For instance, if for our online store example, upon initially writing the Cost class we, in fact, have prior knowledge of the need for subscription-based pricing. As such, we decide to use a Cost interface instead and have two classes (i.e. a FixedCost class and SubscriptionCost class) implement this interface. If our Cost interface had a Subscription method, it would violate the Interface Segregation Principle as the FixedCost would not need it. [10]

### 3.5 Dependency Inversion Principle

*“Higher-level classes should not depend on low-level classes. Both should depend on abstractions. Abstractions shouldn’t depend on details. Details should depend on abstractions.”*

- **Low-level** classes refer to operations such as writing to a database or interfacing with Apache to deliver a page.
- **High-level** classes refer to complex business logic that interfaces with the low-level classes to achieve something useful

The Dependency Inversion Principle is a key principle in Clean Architecture. Essentially, it is undesirable to have a web application that cannot take advantage of other technologies in the future. High-level classes should be detailed through abstractions on the Low-level classes. Such abstractions can be implemented for other technologies to allow the High-level classes to use them.

This following is representation of how this could be implemented for a BudgetReport class:
In the diagram above, the *BudgetReport* class does not depend on which database is implemented. We could change from MySQL to MongoDB or vice-versa without affecting any of the other code.

### 4 What is required?

The SOLID principles are a general guide to drive application development; however, they are just that – guides, not architectures. So, in conjunction with the SOLID principles, the following outcomes are required from our application (according to Robert C. Martin):

- To be **independent of frameworks**. Wonderful frameworks come and go constantly. Exclusively using a do-it-all framework like Laravel means that if it were to ever be deprecated or you decide to move to a different framework, a large percentage of your codebase could be completely redundant.
- To be **testable**. Testing is a huge part of application development. Each component should be individually testable without the need to User Interface (UI), database, web server etc.
- To be **independent of UI**. The UI often changes frequently and therefore there it should be easy to do so without affecting the business logic.
- To be **independent of Persistent Storage solution**. There are many persistent storage solutions available (MySQL, PostgreSQL, etc.). It should be possible to switch them out without affecting the business logic.
- To be **independent of any external agency**. The application will likely have to interface with the outside world; however, it should be completely independent of them.
5 Clean Architecture

Clean Architecture, as coined by Robert C. Martin, is an architectural standard for structuring and organising code to adhere to the desirable outcomes that were outlined in section 4.

One of the core components of Clean Architecture is that all dependencies point inwards. The following common representation of how Clean Architectural software should be structured:

![The Clean Architecture Diagram](sourced_from_[12])

As shown, the **Business Logic** (or **Business Rules**) is at the centre and contains both **Entities** and **Use Cases** (discussed later). Clean Architecture ensures that components on the outer layers can act as plugins to the business logic rather than the as the business logic itself.

Creating adapters for the database, UI, etc. allows us to push them to the outer layer and thus make them interchangeable.

5.1 Clean Architecture Layers

The following sections discuss the different layers of Clean Architecture:
5.1.1 Entities
Entities are the Enterprise business rules within an organization. They are notably the least likely to change and can span across multiple applications. For example, a medical organisation, may have the Entities *Doctor* and *Patient*. There could be many separate applications to handle referrals, billing and appointments however the *Doctor* and *Patient* Entities stay the same regardless.

5.1.2 Use Cases
Use cases are application specific core business rules. They must not have any dependencies on external agents and are intended to explicitly define what the application does.

5.1.3 Interface Adapters
In this layer, the Adapter design pattern is used as a one-way conversion from the format within the use cases to the format used in the external agents. This layer could entirely contain an MVC GUI including the views, controllers etc. This would also, for example, be the layer whereby SQL queries would be put for a SQL Database.

5.1.4 External Interfaces
These are the interfaces that are external to the application but still necessary. These might be, for instance, the database itself, an accounting package interfaced with through an API etc. [12]

5.1.5 Other Layers
The above layers are mostly just guide layers and the actual application maybe require more if it is quite large.

5.2 Dependency Injection
One component that is often helpful in Clean Architecture is the notion of Dependency Injection. Dependency Injection is a method of, as the name suggests, injecting dependencies into the inner business rules of the application. These could be passed via a constructor or an invoked function.

For instance, the core business rules might need to store something (i.e. a user). So, we could have a *PersistentStorage* interface like the following:

```php
interface PersistentStorage {

    /**
     * Return all objects
     * @param mixed $obj the type of object to return (empty instance)
     * @return mixed
     * @throws \Exception
     */
    function getCustomer($obj);

    //... any other PersistentStorage functions
}
```

Then to store a customer we might have a *Customer* class like this which takes a PersistenStorage object as an argument (this still adheres to Clean Architecture as both the *PersistentStorage* interface and *Customer* class are in the same layer):
Then we implement the Persistent storage for say a MySQL persistent storage solution:

class MySQL implements PersistentStorage {
    /**
     * @var MySQL
     */
    public $db;

    //... all the required functions in the PersistentStorage interface
}

And finally, from the outer layers, we can inject the MySQL persistent storage and insert a customer:
This means that we can use store to a persistent storage on the inner business rules without coupling to any specific type.

## 6 Refactoring

Clean Architecture should be considered for any new project that would benefit from it. However, a huge number of projects already exist that are in need of a Clean Architectural alternative. Superficially, it would seem that these projects require a complete rewrite. Often though, this is not feasible due to the cost and time implications of doing so.

This section will demonstrate that it is possible for an existing project to be refactored to Clean Architecture as long as some of the strict rules with Clean Architecture are slightly relaxed. The methods used are mostly applicable to web-based projects and especially those that use a Model View Controller design pattern.

### 6.1 Case Study

The following web project request will be used to reflect various design patterns and potentially help inspire the use of Clean Architecture:

*Sheryll, the CEO of an online store requires the use of an inventory management system to manage the store’s stock. However, the inventory management system (IMS) she uses currently cannot integrate with the external packages she uses (such as her accounting package and employee portal). Her current IMS is written in PHP using the Laravel framework along with a MySQL database. The codebase is not elegant but works according the requirements prior to the current development request. She is has also mentioned that she cannot afford to completely rewrite the IMS, though is open to any solution.*

*It was suggested by all relevant stakeholders to simply write the import modules directly into Laravel but Sheryll is wondering if this would be the best solution.*
6.2 Analysis

Assuming that Sheryll takes the advice of her stakeholders and instructs us to write the module directly into the Laravel framework. As described in section 2.3, a typical scenario is that the model ends up taking the bulk of the codebase. So, what happens when, for example, Sheryll mentions that the system is not fast enough and wants to swap out the SQL database with a NoSQL solution. A large quantity of the codebase will have to be rewritten including import/export modules for external agents such as Sheryll’s accounting package.

Given that the codebase is “not elegant,” ideally a complete rewrite would be an adequate solution. However, we are told by Sheryll that a complete rewrite is not feasible.

An alternative solution is to iteratively transform Sheryll’s current MVC web application into Clean Architecture. Essentially, this is done by packaging up the entire Laravel folder structure pushing it out into the external interface layer of a new Clean Architecture project that is indicative of original project. The goal is then to make the model obsolete with all the controllers pointing inward to the Clean Architecture business rules. Also, while we do save some time in not having to rewrite everything, a large amount of code will need to be either rewritten or refactored.

Below is a model of how an MVC project that has been refactored into Clean Architecture may look:

![Clean Architecture MVC integration diagram](image)

*Figure 4 Clean Architecture MVC integration diagram*

Here are the primary steps involved (each will be discussed in detail below):
1. **Replicate the model** ‘cleanly’ in Clean Architecture acknowledging the structure of the data and the core Business/Enterprise rules
2. **Refactor the Controllers** to use the new Clean Architecture codebase
3. **Alter the Views** to adhere to the newly formatted Controllers
4. **Test** to ensure the functionality is equal what was previously provided or improved

### 6.2.1 Replicate

This will be the most time consuming especially if the Sheryll’s MVC web application follows the normal convention of a fat model and skinny controller. However, even though the code is not following a Clean Architectural convention, there can, and should be, code that is reused. For instance, if upon careful consideration, we find a function that has no dependencies, we can then safely place it into the core business rules if appropriate.

If the function does depend on another class, we may be able to create a more generic function that can have its dependency injected (i.e. through dependency injection).

### 6.2.2 Refactor the Controllers

This is also quite time consuming as the controller may be entirely incompatible with the new Clean Architectural business rules. However, what is possible is that we don’t deprecate the old model as soon as the business rules are created. And from that we can temporarily leave some controllers using the old model until the need arises to move them over. While this is not elegant, it does allow for an interim step in the move to Clean Architecture.

### 6.2.3 Alter the Views

Since the views are already well defined in most MVC implementations, it will likely be a simple task to alter the view to adhere to the refactored controllers.

### 6.2.4 Test

The goal is, at least initially, to replicate the old system in Clean Architecture. This requires potentially a large test suite to ensure that it works correctly.

### 6.3 Feasibility

A feasibility analysis for web development can be a difficult task. Time and cost, in particular, are core components in the feasibility of refactoring as opposed to starting again from scratch. Having a refactoring cost that even remotely resembles building from the ground up is simply not acceptable. Unfortunately, the development involved in refactoring to Clean Architecture is quite substantial. Almost certainly for small projects, starting again would be more logical and potentially even more cost efficient.

For Sheryll’s project in particular, we would need to go through the laborious task of removing the model and replicating its functions in Clean Architecture then connecting the controllers and finally altering views and various other components to make them compatible. It would seem, in this case, that it would be better working the other direction – starting again using Clean Architecture and only taking components from the original project that would adhere to a Clean Architectural approach.

Nevertheless, while sometimes the costs may only be marginally decreased (or even increased) with refactoring to Clean Architecture as opposed to starting from scratch, the key advantage is that it can be
done iteratively. For instance, many customers who have a working product, like Sheryll, require that that product is functional whilst development is occurring. Moreover, they may desire new features for the old project before the new one is complete.

Also, upon finally completing the refactoring process, the cost of future development can often be drastically reduced.

6.4 Issues with using this approach

6.4.1 Predominantly focussed on the backend

While this refactoring process attempts to structure the server side of a web application (backend) to adhere to Clean Architecture, it makes little to no attempt to do the same on the client side (frontend). Essentially, the frontend is only modified to support the new backend. This would especially be problematic if complex JavaScript code is used in front end web applications. Packaging up the frontend in Laravel means that any complex JavaScript code would also remain in Laravel and thus not ascertain the benefit of refactoring to Clean Architecture.

6.4.2 Increased complexity

A possible issue with Clean Architecture is that it can add more complexity. For instance, there may need to be a myriad of use cases that are necessary to perform what could be done using simple database queries.

While it’s favourable having the ability to swap out the persistent storage and/or frameworks for others, if it is never done, the extra work required to decouple from them is useless.

For example, what if Sheryll mentioned that she intends to deprecate the current self-hosted inventory system after a year in favour of an already implemented externally hosted inventory system. In this case, if she still wants to continue with the aforementioned project, it would likely be pointless trying to refactor it into Clean Architecture as it is unlikely that the database or any core frameworks will need to be changed. Furthermore, in the event that they do need to be changed, the risks could be mitigated against (i.e. she speeds up the move to the externally hosted inventory system).

While that is a special case, a more general scenario is that the flexibility of Clean Architecture can occasionally fail to justify the complexity it introduces. This can certainly be the case for simple web applications. Sheryll’s inventory system, while it has some complexity, may not be complex enough to warrant Clean Architecture even if the inventory system were to be used in the foreseeable future. For instance, if we decided to not use Clean Architecture for the project and just add to the current MVC framework, we could just deal with the risks of doing so. Furthermore, even if the entire web application needed to be rewritten in favour of another framework, it could still be cheaper than building a Clean Architecture approach. [13]

6.4.3 No longer follows conventional design patterns

Until the web application has fully been moved over to Clean Architecture, it will not follow conventional design patterns. This can mean it’s very difficult to get support from external developers or bring on new developers. The inventory system uses the Laravel framework which is highly supported and by moving away from it, this support is sacrificed. For Sheryll, this can be a significant consideration to the ongoing maintenance costs of supporting the inventorying management system as it will require developers that are trained on the unconventional design patterns used.
6.5 Results
As demonstrated in the resulting functional website and corresponding codebase (see Appendix), this approach does work in refactoring an existing web application to adhere to Clean Architecture.

After building the use cases and entities, I was then able to package up the entire Laravel framework and push it out to the Clean Architecture Interfaces layer.

Following this, I was able to instantiate the Clean Architecture use cases from the controllers within the Laravel framework. This was done with little to no outward dependencies.

However, the main caveats that I found were:

1. The resulting code suffers from the issues described in both section 6.4.2 and section 6.4.3 as it is now more complex and no longer follows conventional design patterns.
2. From a time development perspective, it would be difficult for the customer to justify this solution for a project of this size.
3. While not prevalent in this project, there will likely be rogue or cross dependencies somewhere in the codebase.

Below is an example of how the InventoryController class inside the Laravel project looks after being refactored to Clean Architecture:

```php
class InventoryController extends Controller
{
    /**
     * @var Inventory
     */
    private $inventory;

    /**
     * InventoryController constructor.
     */
    function __construct() {
        $this->inventory = new Inventory(new MySQL());
    }

    // ... required inventory methods
}
```

As can be seen, the InventoryController class creates an Inventory object and injects the MySQL dependency ensuring that there are no outward dependencies.

7 Conclusion
For many web application projects, Clean Architecture can be highly valuable. Following the principles of Clean Architecture results in a web application that is modular, flexible and testable. Clean architecture ultimately enables a web application to be future proof.
This project has demonstrated how Clean Architecture can be used for an existing project in the form of a case study. Specifically, this project describes how to refactor an existing codebase that uses the MVC design pattern to use Clean Architecture.

The following are steps that I found highly useful when refactoring an existing MVC web application into Clean Architecture:

1. **Replicate the model**
2. **Refactor the Controllers**
3. **Alter the Views**
4. **Test**

For the model in particular, I found numerous ways to decrease the development time such as using an already implemented ORM (i.e. Laravel’s *Eloquent* ORM) to facilitate the design of the entities.

While the refactoring is possible, I also demonstrated that this will certainly not be an adequate solution in all situations. From a developer’s perspective, Clean Architecture might be highly valuable and a necessary step, however from a customer’s perspective it might be a poor return on investment for projects that require little ongoing development.

Nevertheless, in some situations, existing complex web applications might benefit greatly from refactoring into Clean Architecture by being more flexible and testable. And coupled with the iterative development possible in refactoring as opposed to starting from scratch, it can be favourable solution to both the customer and the developers.

### 8 Future Development

This project mainly focusses on refactoring MVC to Clean Architecture. While some of the discussed methods may be useful in refactoring from other design patterns, the refactoring process will often be entirely different. For future development, it would therefore be beneficial to investigate the refactoring process for other design patterns with and without an associated framework. Furthermore, by investigating other design patterns, it is possible that more concrete strategies can be put in place for refactoring into Clean Architecture.
9 Appendix

9.1 Original MVC Web Application

This is the original web application used prior to refactoring with Clean Architecture. As mentioned, it was built in Laravel using the recommended folder structure and coding practices.

9.1.1 Database

The following is the original database design:

![Database Diagram]

While there is a large amount of work required to deprecate the model, the actual database structure mostly stayed the same when refactored. This is because, the MySQL class is an interface of the PersistantStorage class and can created with the original database in mind while still being independent from the business logic.

9.2 Clean Architecture Entities Diagram

The following is a class diagram of the Clean Architectural entities used for refactored inventory system:

![Class Diagram]
As can be seen it looks somewhat similar to the original database design.
9.3 Clean Architecture Use Cases Diagram

The following is a class diagram of the Clean Architectural use cases used for refactored inventory system. It presents the core functionality of the inventory system:

- **Authentication**
  - storage: PersistentStorage
  - authenticate(email, password): bool
  - createUser(email, name, passwordOne, passwordTwo)

- **Inventory**
  - storage: PersistentStorage
  - moveItem(item, location)
  - createUpdateItem(id, product, serial, mac, locations)

- **PersistentStorage**
  - getAll(obj)
  - getObj(obj, id)
  - storeObj(obj)

- **Purchase**
  - storage
  - addPurchaseOrder(order, product, count)
  - createUpdatePurchaseOrder(id, issueDate, supplier, products)

- **Purchase**
  - storage
  - addPurchaseOrder(order, product, count)
  - createUpdatePurchaseOrder(id, issueDate, supplier, products)
9.4 Folder Structure

9.4.1 Original Folder Structure

The original folder structure follows the default Laravel folder structure.

```markdown
- app
  - inventory
- bootstrap
- config
- database
- public
- resources
- routes
- storage
- tests
  - Tests
- vendor
  - .editorconfig
  - .env
  - .env.example
  - .gitattributes
  - .gitignore
  - .phpstorm.meta.php
  - styleci.yml
  - _ide_helper.php
  - _ide_helper_models.php
- artisan
- composer.json
- composer.lock
- Inventory EER Diagram.png
- Not Clean Project.iml
- npm-debug.log
- package.json
- phpunit.xml
- README.md
- server.php
- Server Config.md
- webpack.mix.js
```

The views are all stored in the `resources` directory.

Within the `app` directory, contains the controllers as well as the entities used for the Elequent ORM. These entities can provide useful information on the possible entities that could be used for the Clean Architecture implementation. The Elequent ORM was used in this implementation and these are the respective entities:
9.4.2 Clean Architecture Folder Structure

The following presents the folder structure used for the Clean Architecture implementation. As can be seen the entire, somewhat messy, Laravel framework is packaged up and pushed into the interfaces folder:
9.5 Source Code

The following are the respective links for the source code created for the purposes of this project. The first of these is the primary code base whereas the other two are secondary (i.e. used for refactoring and research purposes respectively)

- Original MVC Web Application (hosted on GitHub): [https://github.com/matthewknill/Laravel-Inventory-System](https://github.com/matthewknill/Laravel-Inventory-System)
9.6 Cryptocurrency Bot in a weakly typed language

I built a Cryptocurrency Bot in Python and some Python libraries to further develop my knowledge of Clean Architecture and promote its usage.

These were the requirements of the bot:

- Ability to trade cryptocurrency using an exchange
- Have “signals” of when to either buy or sell a given cryptocurrency. It should be easy to add delete or update signals.
- Have some kind of persistent storage to store price action, trades, settings, etc.
- Have a way to back-test with visual representation of the trades
- Notifications of trades and other statistics

I figured this was a perfect representation of the use for Clean Architecture as the core algorithm should have the ability to support many cryptocurrencies over many exchanges and be independent from the database.

For instance, one user of this bot might desire using it to trade Bitcoin (a type of cryptocurrency) on a certain exchange. Following that another user comes along wanting to use the bot to trade Ethereum (another type of cryptocurrency) on a completely different exchange.

The following diagram represents the structure of the Cryptocurrency Bot. The Exchange, PersistentStorage and Signals are all injected into the Algorithm class for it to use. Adding another Signal or another Exchange is just a matter of extending the respective class and implementing the required functions.

The interesting thing with Python, however, is that it does not have any support for interfaces or abstract classes. Being a weakly typed language, Python does allow you to not specify the variable type inject any type of object. This is dangerous though as an Exchange could be injected into the PersistentStorage variable. As such, more error checking is required to ensure this does not occur. In some ways this does show that Clean Architecture may be less useful in languages that are unable to be strongly typed.

The following is a class diagram of the Cryptocurrency Bot:
Core

Interfaces

Note: The `signal`, `exchange` and `persistent_storage` classes are strongly typed to make it easier to understand, however this will not work in where the variables will have to be left as generic
Bibliography


