Step Driven Development

Status

`draft`

Context

Clean architecture promotes a clean separation of layers, but they are split based on responsibility.

Things have changed, and the way we develop is geared towards delivering features. Every addition or change to a feature will involve code changes across different layers, aka vertical slicing.

Layers

These layers tend to form a *gradient*, where some responsibility may overlap too. Layers are rarely independent as one would think. In clean architecture for example, each layer will depend on the layers below it. For example, in golang, if we have a usecase package that calls the repository package, one would think they could be clearly substituted by just declaring the interface at the usecase layer. However, there will be some types that might need to be imported from the repository layer, hence coupling them together.

Regardless of how we layer our systems, one fact remain: layers are just human concerns. They do not in any way impact the way machine works.

Function as a whole

Let's say you have a large usecase. No matter how you break it into smaller functions, or moving them into different layers, the size of that usecase do not actually change.

When we have a large usecase, it can become hard to understand the logic behind it. As human, we try to understand how logic works step by step. But when the number of steps exceeds a certain threshold, we just lose context of what the last n step is doing, and how it relates to the current step, and the usecase. Coupled with conditional logic, things gets messy fast.

If we break them up into smaller methods, we might lose context of the overall logic, because the logic now becomes fragmented across different methods or files.

As a developer, we need to find a sweet spot for the right size of a function.

Layers are not the same

Clean architecture helps to (probably) promote cleaner separation between layers. However, what it does not take into account (as with other architecture) is the rate of change and growth of the layers.

Lets take for example the presentation layer, specifically API controllers. Whenever we need to expose a new API, we create a new controller method to do so. The role of the controller is pretty much fixed too, it should be void of business logic and mostly does serialisation and deserialization of objects to json.

All the layers adhere to Single Responsibility Principle.

But usecase layer grows over time. We can imagine each layer as a ping pong ball in the beginning. after countless iteration and new features added, the usecall layer now becomes the size of a basketball. A big ball of mud.

Within the layer, there is not much guidance on how to further break it down into maintainable puzzle pieces. A method with 1000 lines will strike fear in every developer.

Step driven development

So how to divide and conquer our ever-growing usecase layer?

Simple, we break them into steps.

Abstractions are important too. The usecase layer has always been the dumping ground for all kinds of logic without abstraction.

Code as Document

One of the advantage of step driven development is that now it can read as a pseudo code. We can even add tools to log each steps as a snapshot together with the request response and also generate sequence diagram with it.

Functional Steps

Pure functions steps can be used directly, assuming that we have control over the input of the previous steps.

Steps with dependencies

For steps with dependencies, we can inject it after building it. Usually it is preferable to pass an interface too since we don't really care about the implementation.

Composition

Steps can be composed, similar to how pipe in linux works.

A step can have smaller substeps, but the idea is the same.

At the highest level, we have a step with an input and output, which can be optional.

As long as the output of a step matches the input of the previous step, we can chain them.

The ideal scenario is that there are no dependencies between steps, and the order of execution does not matter.

Steps from user story/usecase

Can we derive steps from the user story and/or usecase. Partially. It would be more accurate to derive it from the system flow, which is also explained in one of the ADR.

Decisions

It is hard to deal with changing requirements too, especially when you need to touch unrelated code in different layers. Hence, we want to introduce the concept of step-driven-development.

Instead of this:

```go
package main

type AuthUsecase struct {
    userRepo UserRepository
}

func (uc *AuthUsecase) Login(ctx context.Context) error {
    // Fetch user from repo ...
    // Check password match ...
}
```

We have dependencies such as repository declared etc.

```go
package main

type AuthUsecase struct {
    loginStep loginStep
}

func (uc *AuthUsecase) Login(ctx context.Context) error {
    // Do login
    // uc.loginStep
    //
}
```

If the requirement wants to add another step, such as sending email to indicate you are logged in:

```go
package main

type AuthUsecase struct {
    step0 loginStep
    step1 sendEmailOnLoginStep
}

func (uc *AuthUsecase) Login(ctx context.Context) error {
    // Do login
    // uc.step0
    // uc.step1
}
```

One advantage is we abstracted the steps so that we don't need to concern ourselves with implementation details. During testing, we can also test each steps independently.

Examples

- reusability of each steps
- testing each step independently
- bringing steps together
- logging steps
- swapping steps
- alternative flows

Testing

Testing steps/layers can be challenging.

We want to be able to test them independently, yet assert that they can chain together to run end to end.

One strategy is to use snapshots.

For example, your usecase may be composed of several long steps. This can be logically decomposed into say two clear steps boundary. We can test each step boundary, and snapshot the previous step boundary to pass to the next step boundary.

This form of delegation ensures that both steps are *continuous*.

One side effect is that they may be creating a dependency.

Mocking

Why do we need mocking? the reason is simple, we do not want to execute the side effects (e.g database queries) or want to avoid making API calls.

Most people mock the response in order to test control flow. This is wrong, and doesn't add much value. It also couples two steps that otherwise could have been tested independently.

To elaborate further, take the example of the step

1. create account(email): Account
2. send welcome email(Account): error

If we mock the first step, we are essentially

- discarding the validation for the create account request
- mocking the response for the response Account
- coupling the response with the step send welcome email (which could have different behavior depending on the account returned)

For the last point, we could have just tested the send welcome email step independently.

In short, if all the steps are mockable, and there are no inlined steps, there is no value in mocking the steps for testing.

Data pipelines

Basically, we want to treat each step as a black box, where only the input and output matters, and can be pipe to the next step.

This is similar to how unix works.

Basically every step we execute is just a series of data transformation. What is important is validating the request and response for each step.

Clean Usecase

Most usecase suffers because they have inlined logic. A basic example is calculation.

```
orders = repo.GetOrder()
total = do some order calculation
```

The problem with this is, we now have to execute the whole usecase in order to test this calculation.

In short, don't in-line logic. That includes separating them into a method or a function call. Instead, shift the logic into a step and test the step separately.

Substeps

Once we have extracted all the steps from the usecase, we can focus on each step individually.

Within each step, we can also have more steps to execute.

They are usually

- constructing a new request
- mapping request
- business logic
- side effects

Repository

Instead of using steps, we can stick with the default approach of using repository.

There should only be one repository per usecase.

All external data will be handled in the repository, which will only accept repository types and return domain types.

Testing

We only want to test the success scenario for the whole business unit. The failure scenario for each step can be tested separately.

Mocking a step to fail and testing the whole business unit will lead to O(N)^2 complexity.

Use single struct per write usecase

For reads, we can have a struct with many methods, since they usually involves querying the data store and does not have complicated substeps.

For writes, it is recommended to use a single struct per business units.

Writes usually has complex steps involved and we want to be able to test each steps individually.

```go
type UserReader interface {
  Find(ctx context.Context, id string) (*User, error)
  List(ctx context.Context) ([]User, error)
  // ...
}
```

The business specific writer will have the main method called `Exec`, while the steps are private methods.

```go
type AuthenticateUseCase struct {
  repo authenticateRepository
}

func (uc *AuthenticateRepository) Exec(ctx context.Context, req
AuthenticateRequest) (string, error) {
  // A series of steps that are private methods.
  user, err := uc.repo.FindUserByEmail(ctx, req.Email)
  if err != nil {
    return "", err
  }
  // uc.encrypyPassword()
}
```