Abstractions & Errors

Ground Work

What is an Abstraction?

macOS Dictionary says

> the process of considering something independently of its associations, attributes, or concrete accompaniments

Wikipedia says

> the creation of abstract concept-objects by mirroring common features or attributes of various non-abstract objects or systems of study[3] – the result of the process of abstraction.

the act of creating a new concept & hiding details or attributes away

Abstractions & Tools for creating Abstractions

- programming languages
- functions
- classes
- methods
- modules
- structs

The term & concept of Abstractions is overloaded!

Control Abstraction

Also known as *Procedural Abstractions*.

modules, **functions**, or **methods** are the common tools used for implementing this class of abstractions.

Structured Programming in general an example of this.

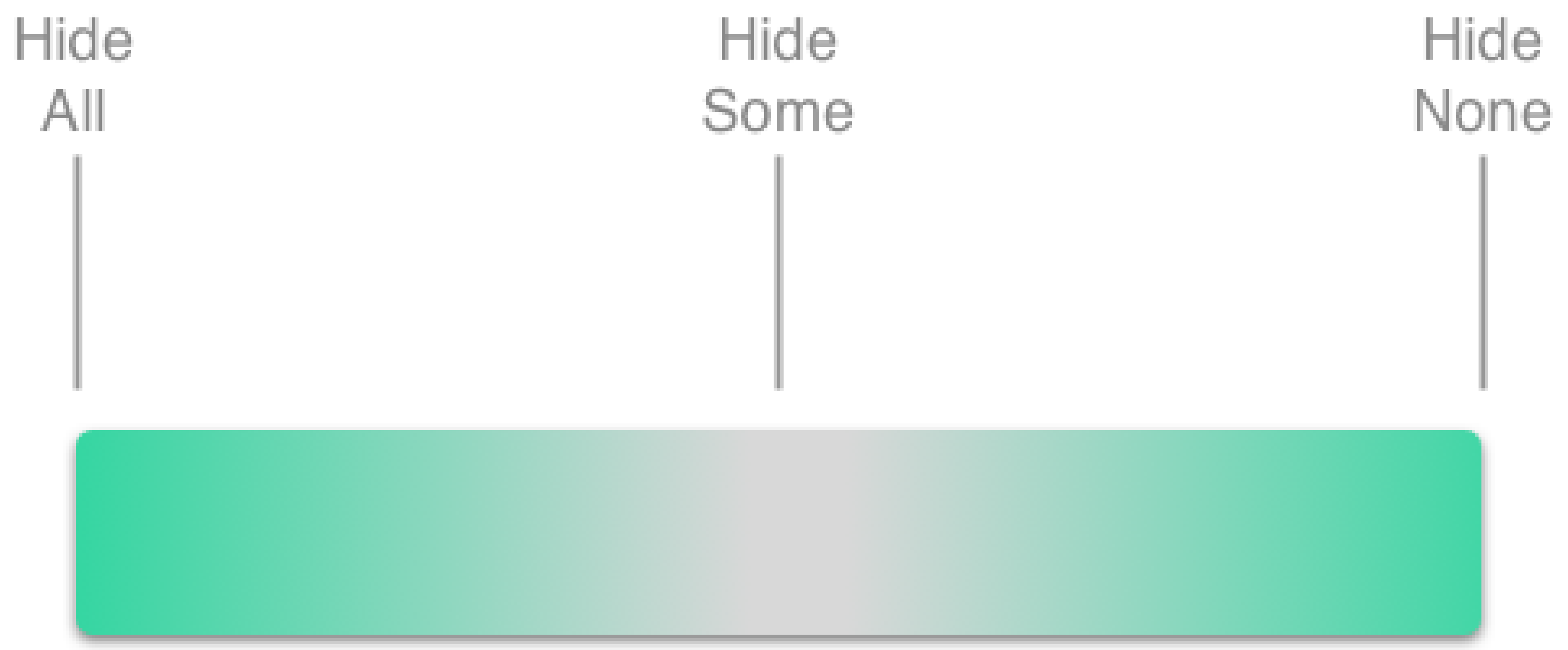
```
a := (1 + 2) * 5
```

The above uses a number of procedural abstractions

- `` `+` `` addition operator
- `` `*` `` multiplication operator
- `` `()` `` precedence operator
- `` `:=` `` assignment operator.

Data Abstraction

structs and **classes** are the common tools used for implemeting this class of abstractions.



Data abstractions that **don’t hide attributes** are still valuable because they still **introduce a concept**.

Abstractions & Programming Paradigms

Functional Programming

- Data Abstractions
- Operation Abstractions

Object Oriented Programming

- Merge of Data & Operation Abstractions

So really what is the abstraction then?

Anatomy of Abstraction

A contrived example

```
function add(v1: number, v2: number): number {
    return v1 + v2
}
```

Anatomy of Abstraction

A contrived example

```
function add(v1: number, v2: number): number {
    return v1 + v2
}
```

- **name** - ``add``
- **interface** - ``add(number, number): number``
  - inputs - ``(number, number)``
  - return - ``number``
- **internals** (hidden away) - ``v1 + v2``

Anatomy of Abstraction

- **name**
- **interface**
  - inputs
  - return
- **internals** (hidden away)

So again what is the abstraction then?

The Abstraction is two things

- **Concept**
- **Interface**

From the previous example we would have the following:

- **Concept** - `` `adding` ``
- **Interface** - `` `add(number, number): number` ``

In many languages **protocols** / **interfaces** can be defined to help formalize the concept of interfaces. It is imporant to understand that they exist irrespective of being formalized.

Many languages also provide **generics** to facilitate implemnting even more abstract/generic functions.

```
function add<T>(v1: T, v2: T): T {
    return v1 + v2
}
```

Another Example - What could go wrong?

```
function getUserProfile(userId: string): UserProfile {
    const userProfile = restClient.get('/user/' + userId);
    return userProfile
}
```

Another Example - What could go wrong?

```
function getUserProfile(userId: string): UserProfile {
    const userProfile = restClient.get('/user/' + userId);
    return userProfile
}
```

- `` `restClient.get` `` can throw a `` `RestClientError` `` exception

Another Example - Abstraction Anatomy?

```
function getUserProfile(userId: string): UserProfile {
    const userProfile = restClient.get('/user/' + userId);
    return userProfile
}
```

- **name** -
- **interface** -
  - inputs -
  - return -
- **internals** (hidden away) -

Another Example - Anything else wrong?

```
function getUserProfile(userId: string): UserProfile {
    const userProfile = restClient.get('/user/' + userId);
    return userProfile
}
```

- **name** - `` `getUserProfile` ``
- **interface** - `` `getUserProfile(string): UserProfile` ``
  - inputs - `` `(string)` ``
  - return - `` `UserProfile` ``
- **internals** (hidden away) - `` `restClient.get(...)` `` - the how of getting the user profile

Another Example - Implicit Exception Missing

```
function getUserProfile(userId: string): UserProfile {
    const userProfile = restClient.get('/user/' + userId);
    return userProfile
}
```

- name - `getUserProfile`
- interface - `getUserProfile(string): UserProfile`
  - inputs - `(string)`
  - return - `UserProfile`
  - **exceptions** - `RestClientError`
- internals (hidden away) - `restClient.get(...)` - the how of getting the user profile

Leaky Abstraction

A.K.A. a **non "Holding an Abstraction"**

This basically means that some detail of the **internals** that is supposed to be hidden away is being exposed.

In the above example the `` `RestClientError` `` was a lower level implementation details that was being leaked out.

Exceptions - What did we learn?

- implicit
- they promote **leaky abstractions**

What is the alternative?

Result Type

A.K.A - Either

an abstraction that allows us to formalize the concept of success/failure in an explicit manner

`` `Result` `` is similar to the concept of `` `Optional` `` or `` `Maybe` `` which represent two states, the state of having a value vs not having a value. But in the case of `` `Result` `` it represents something being successful or not successful.

`` `Result` `` can conceptually be thought of as an `` `enum` `` with associated values. In fact in many languages it is implemented this way.

```
Ok(value)
Err(error)
```

Result Example

Before

with exceptions we had to magically know it could throw a ``RestClientError`` exception.

```
restClient.get(url: string): Response
```

Now

with ``Result``

- from the signature we **explicitly** know what type of errors are possible, ``RestClientError``
- the type system helps us make sure we handle both the success and failure cases

```
restClient.get(url: string): Result<Response, RestClientError>
```

Result - learnings

- solves implicitness of exceptions
- makes it possible to use the type system to make sure we are handling the error cases
- promotes **"Held Abstractions"**

Held Abstractions

But if we make our abstractions **held/true** we will have to create a bunch of error types and do a bunch of mapping between them right?

Yes, and you should have been already doing this.

By **not** doing this you have been propagating internals of abstractions through many if not all layers of your application. Effectively making all your abstractions **leaky abstractions**.

It doesn't have to be that hard though

Classifying Errors

Way to reduce number of error types you have to create.

Create an error type to represent a class of errors and then use that error type as part of multiple abstraction interfaces.

`` `IOError` `` as an example of this.

You could in theory have a bunch of **procedural abstractions** that abstract away interacting with the file system. All of these could use the `` `IOError` `` type.

Warning

Classifying errors or even using classified error types takes a lot of thought and is a tricky thing to do. You can easily end up in scenarios where a subset of a classified error type makes sense for your abstraction but the other cases don’t.

Abstraction Layering/Building

Mapping between layers of abstraction

We can start to see this with the example we had before.

```
function getUserProfile(userId: string): UserProfile {
    const userProfile = restClient.get('/user/' + userId);
    return userProfile
}
```

We have the `` `getUserProfile` `` abstraction which is abstracting away the how of getting the user profile.

The **internals** of that abstraction are using the `` `restClient.get()` `` abstraction.

Leaving us with an abstraction layering of `` `getUserProfile() -> restClient.get()` ``.

Abstraction Layering/Building

Mapping between layers of abstraction

If we had the exception version of `restClient.get()` we would have to catch the exception and map it to a `Result` with an error type matching the layer of our `getUserProfile` abstraction, e.g. `GetUserProfileError`.

Otherwise we would have a **leaky abstraction**.

```
function createGetUserProfileError(error: RestClientError): GetUserProfileError {
    ...
}

function getUserProfile(userId: string): Result<UserProfile, GetUserProfileError> {
    try {
        const userProfile = restClient.get('/user/' + userId);
        return Ok(userProfile);
    } catch (error) {
        return Err(createGetUserProfileError(error));
    }
}
```

Abstraction Layering/Building

Mapping between layers of abstraction

If we had the `Result` version of `restClient.get()` we would still have to map the error.

```
function createGetUserProfileError(error: RestClientError): GetUserProfileError {
    ...
}

function getUserProfile(userId: string): Result<UserProfile, GetUserProfileError> {
    const result = restClient.get('/user/' + userId);
    if (result.ok) {
        return Ok(result.val);
    } else {
        return Err(createGetUserProfileError(result.val));
    }
}
```

Those are basically the same though!

Abstraction Layering/Building - Result helps

Mapping between layers of abstraction

`Result` types provides us a `mapErr` method to help us specifically with this case.

```
function createGetUserProfileError(error: RestClientError): GetUserProfileError {
    ...
}

function getUserProfile(userId: string): Result<UserProfile, GetUserProfileError> {
    return restClient.get('/user/' + userId)
        .mapErr(createGetUserProfileError);
}
```

The `mapErr` method collapses the conditional away and applys the `createGetUserProfileError` function in the case where the result is unsuccessful.

Abstraction Layering/Building

Mapping between layers of abstraction

There is more to it though right?

We probably want to do other things with errors in certain cases.

There are only so many options

Abstraction Layering/Buliding - Error Handling

What do we do with errors?

- **consume** the error if we can programmatically handle it
- **translate** error into a higher level abstraction error
  - nest error into a higher level abstraction error
- **propagate** error through without nesting (*probably creating a leaky abstraction*)

Error Handling - Consume

Consuming and handling errors so consumers of your abstraction don’t have to.

`Result` provides `unwrapOr()` to facilitate this.

```
function createDefaultUserProfile(): UserProfile {
    ...
}

function getUserProfile(userId: string): UserProfile {
    const defaultProfile = createDefaultUserProfile();
    return restClient.get('/user/' + userId)
        .unwrapOr(defaultProfile);
}
```

Error Handling - Translate/Nest

Translating/Nesting lower level errors into error types fitting to your abstraction layer so consumers are working at the same layer of abstraction.

`Result` provides `mapErr` to facilitate this.

```
function createGetUserProfileError(error: RestClientError): GetUserProfileError {
    // only difference is the logic here for Translate vs. Nest
    ...
}

function getUserProfile(userId: string): Result<UserProfile, GetUserProfileError> {
    return restClient.get('/user/' + userId)
        .mapErr(createGetUserProfileError);
}
```

Error Handling - Propagate

Propagating a classified error fitting for the traversed layers of abstraction so that consumers of the higher level abstraction can handle the error.

The following as an example of how to do it. But it is a **leaky abstraction** so it is an example of what you shouldn't do.

```
function getUserProfile(userId: string): Result<UserProfile, RestClientError> {
    return restClient.get('/user/' + userId);
}
```

Warning

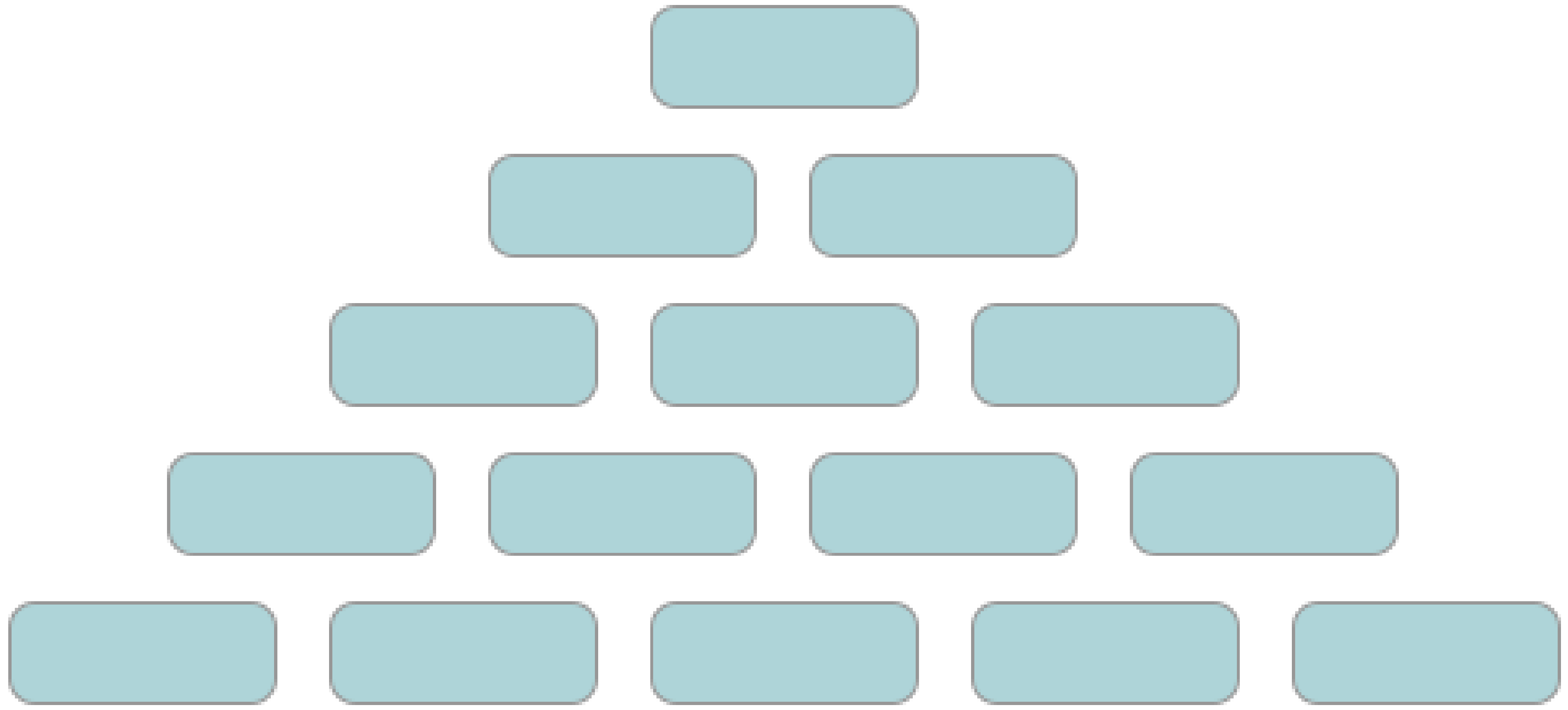
Be careful when doing this as you are likely creating a **leaky abstraction**. You should **only** use this when you are propagating a **classified** error that can safely span abstraction layers without causing a **leaky abstraction**.

Sprawl?

You might be thinking. Doesn’t this sprawl out of control?

Actually it is a natural collapsing pattern because the inner most layers of your application architecture that have possible errors get aggregated/grouped into errors at higher levels of abstraction.

You can look it at like a pyramid with the error originators at the base and the highest level error abstractions at the top.



Questions?