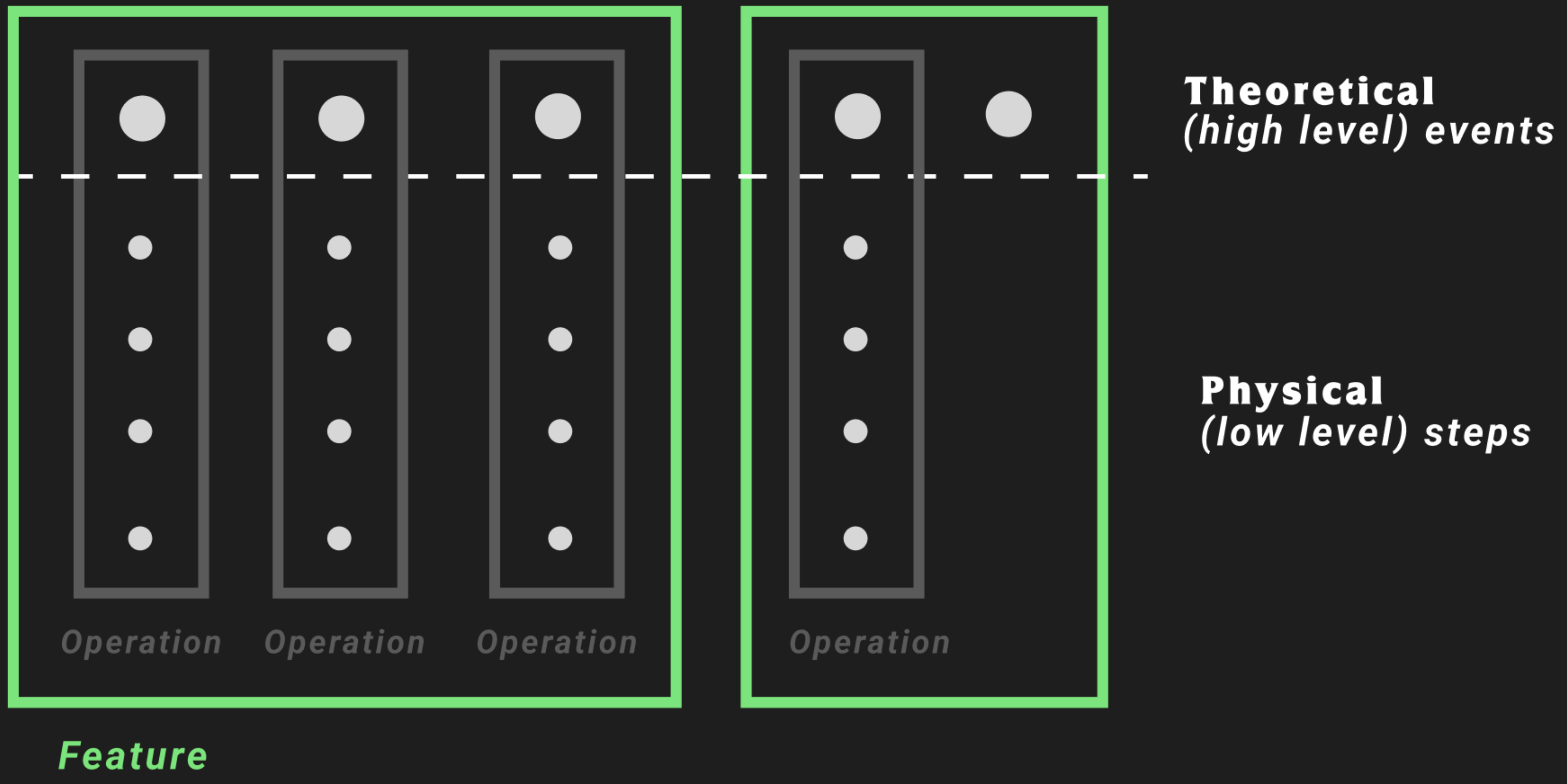
The SOFTWARE ESSENTIALIST

Physical Essential: Vertical Slicing



Physical Essential

Vertical Slicing

*If you can't *see your features* from the folder structure alone, there's a good chance maintainability is under pressure.*

Developers primarily work against Features.

Discovering, understanding, adding, changing, removing, testing, debugging features.

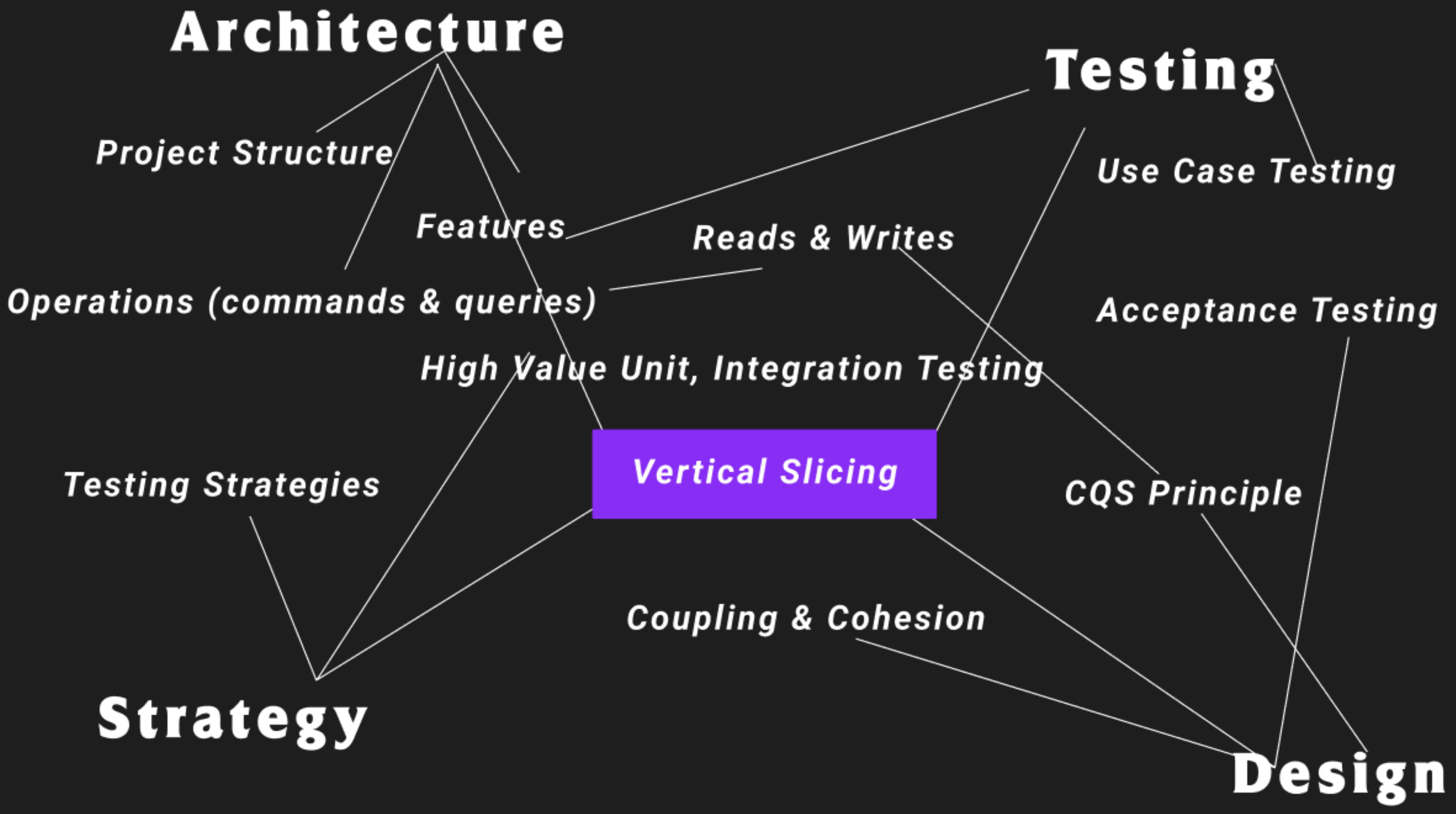
Because of this, you want to chunk your work, your modules, your features, your operations - into Vertical Slices of functionality.

There are two types of Slices: Features & Operations.

Features are high-level slices. These are valuable for the Customer. You build & Acceptance Test User Stories, Scenarios & Concrete Examples against these.

Operations are low-level slices. These are your commands & queries. Your API calls. They are what come together in sequence to make a feature.

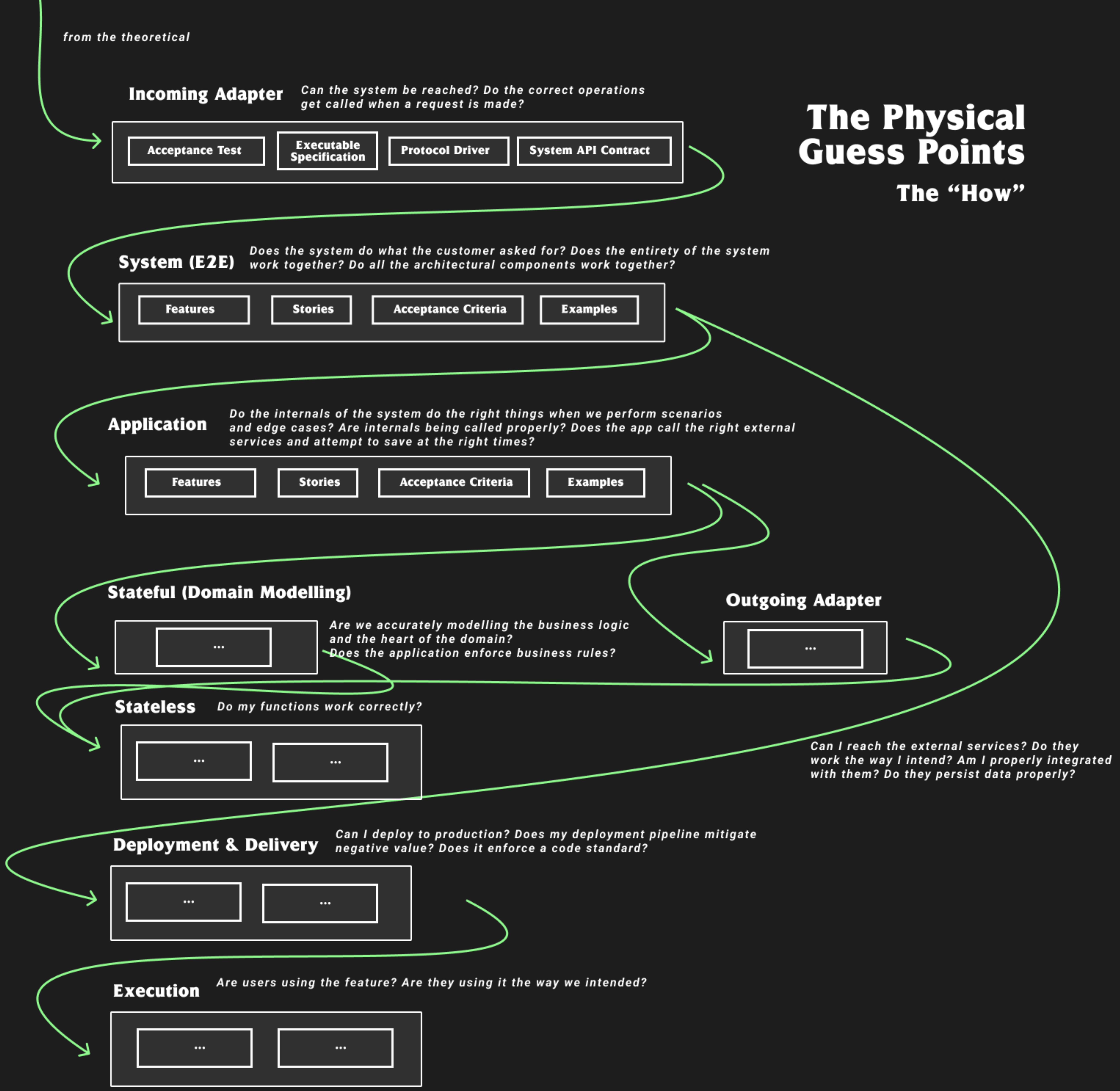
Focus on thinking, planning, organizing, structuring, testing & building vertical slices into your testing & architecture.



essentialist.dev

The Physical Guess Points

The "How"



Incoming Adapter *Can the system be reached? Do the correct operations get called when a request is made?*

System (E2E) *Does the system do what the customer asked for? Does the entirety of the system work together? Do all the architectural components work together?*

Application *Do the internals of the system do the right things when we perform scenarios and edge cases? Are internals being called properly? Does the app call the right external services and attempt to save at the right times?*

Stateful (Domain Modelling) *Are we accurately modelling the business logic and the heart of the domain? Does the application enforce business rules?*

Outgoing Adapter *Can I reach the external services? Do they work the way I intend? Am I properly integrated with them? Do they persist data properly?*

Stateless *Do my functions work correctly?*

Deployment & Delivery *Can I deploy to production? Does my deployment pipeline mitigate negative value? Does it enforce a code standard?*

Execution *Are users using the feature? Are they using it the way we intended?*

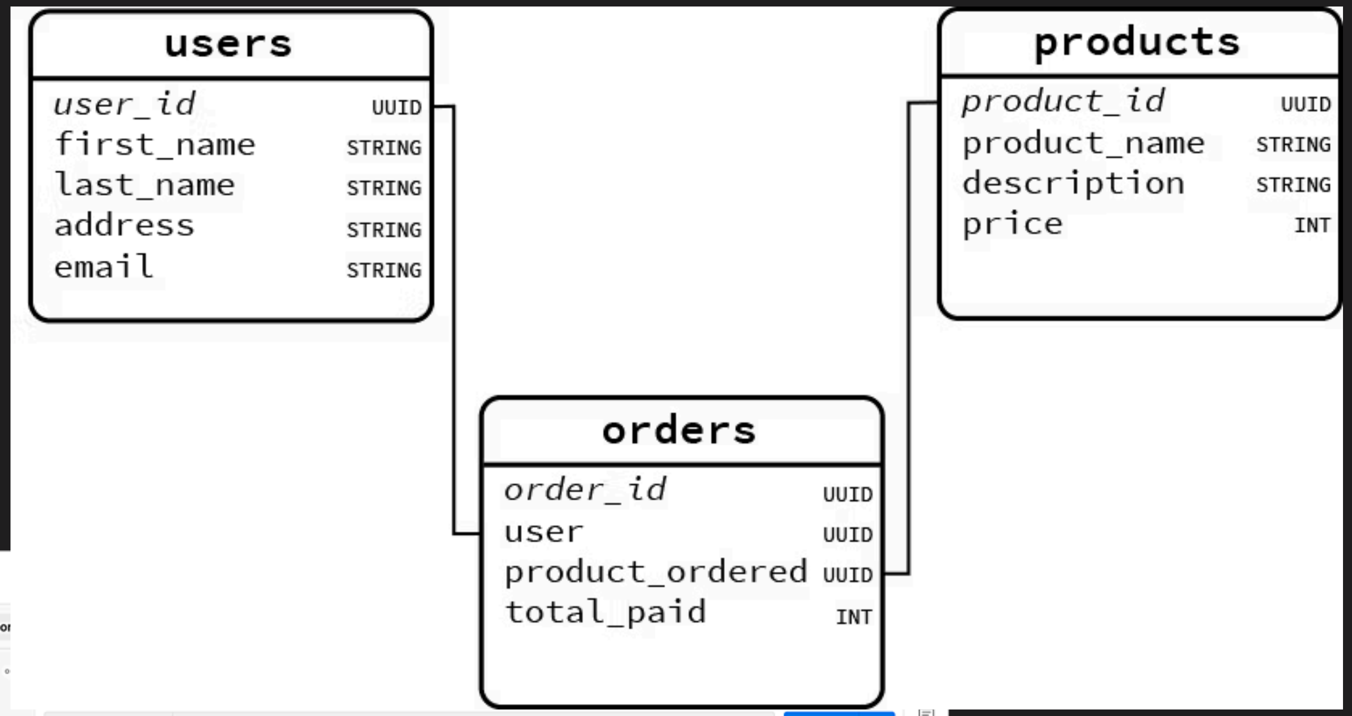
essentialist.dev

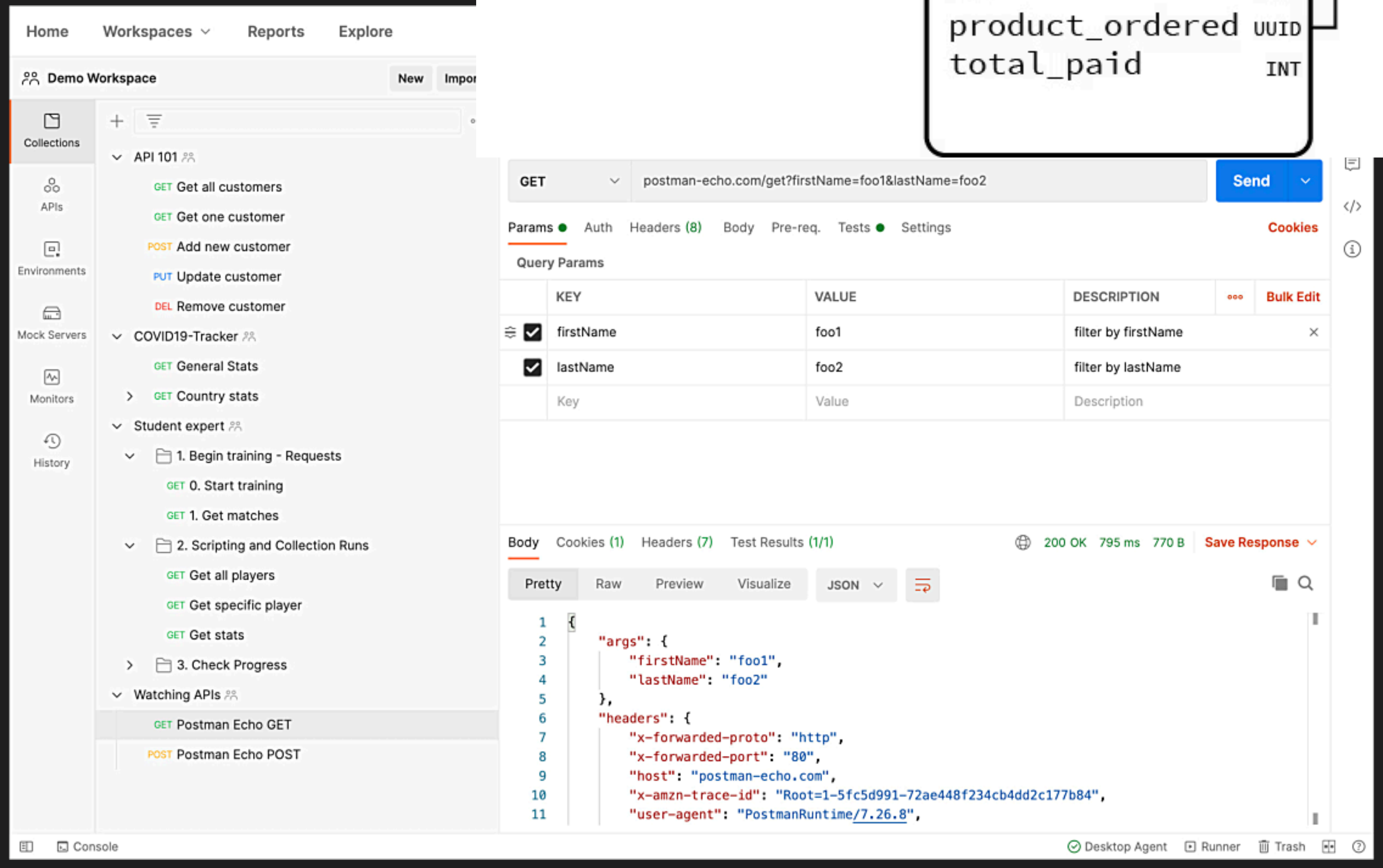
Where we’ll learn more

First, 3 common symptoms of a big maintainability problem

Symptom #1: Thinking In Code, Components, Low-Level Abstractions to Drive Features

- *Database-first*
- *React-component first*
- *(Even API-call first)*





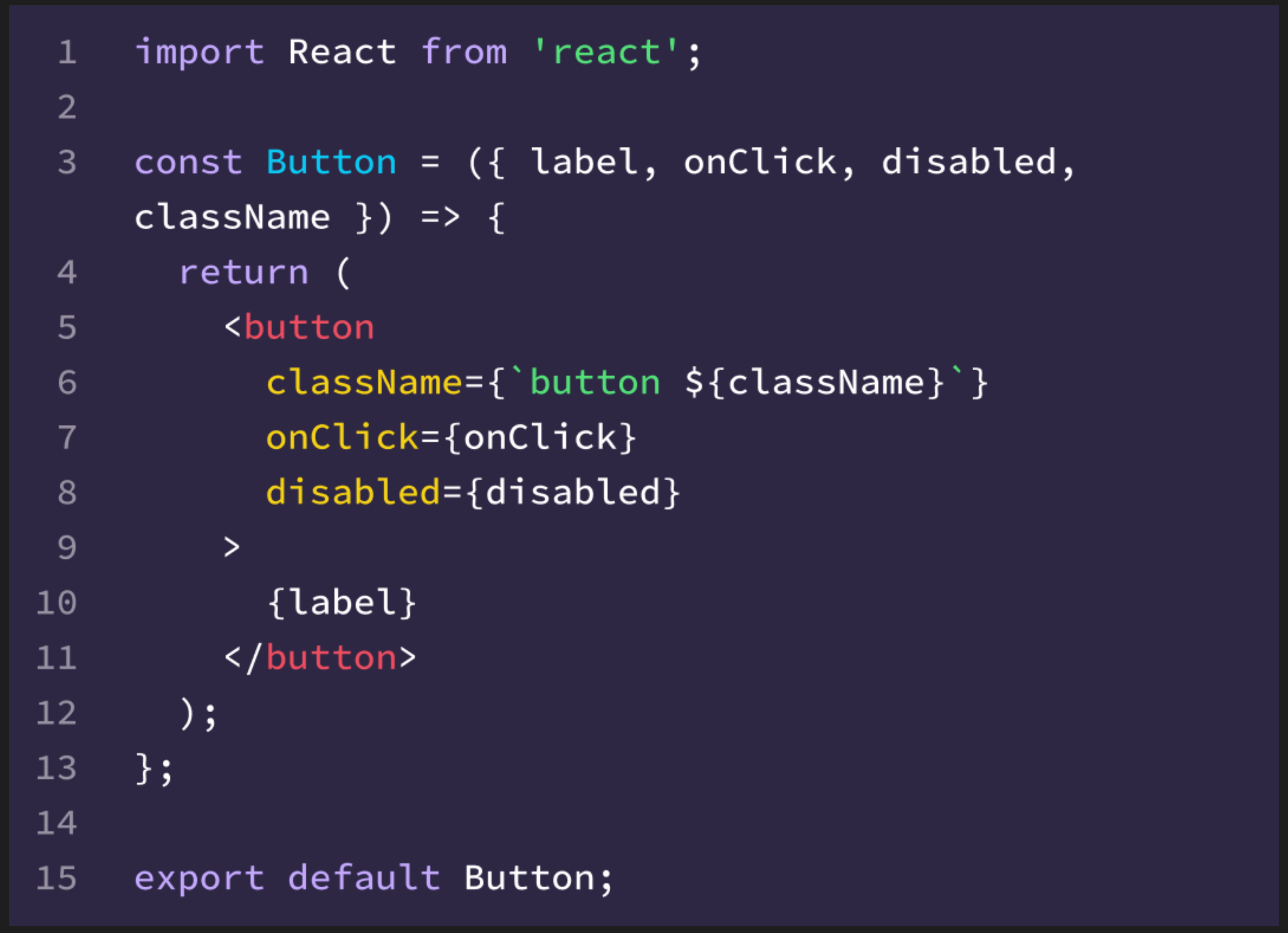
GET postman-echo.com/get?firstName=foo1&lastName=foo2

Query Params

	KEY	VALUE	DESCRIPTION
✓	firstName	foo1	filter by firstName
✓	lastName	foo2	filter by lastName
	Key	Value	Description

200 OK 795 ms 770 B

```json
{
    "args": {
        "firstName": "foo1",
        "lastName": "foo2"
    },
    "headers": {
        "x-forwarded-proto": "http",
        "x-forwarded-port": "80",
        "host": "postman-echo.com",
        "x-amzn-trace-id": "Root=1-5fc5d991-72ae448f234cb4dd2c177b84",
        "user-agent": "PostmanRuntime/7.26.8",
```



```jsx
import React from 'react';

const Button = ({ label, onClick, disabled,
className }) => {
  return (
    <button
      className={`button ${className}`}
      onClick={onClick}
      disabled={disabled}
    >
      {label}
    </button>
  );
};

export default Button;
```

essentialist.dev

Symptom #2: Packing By Infrastructure instead of Features

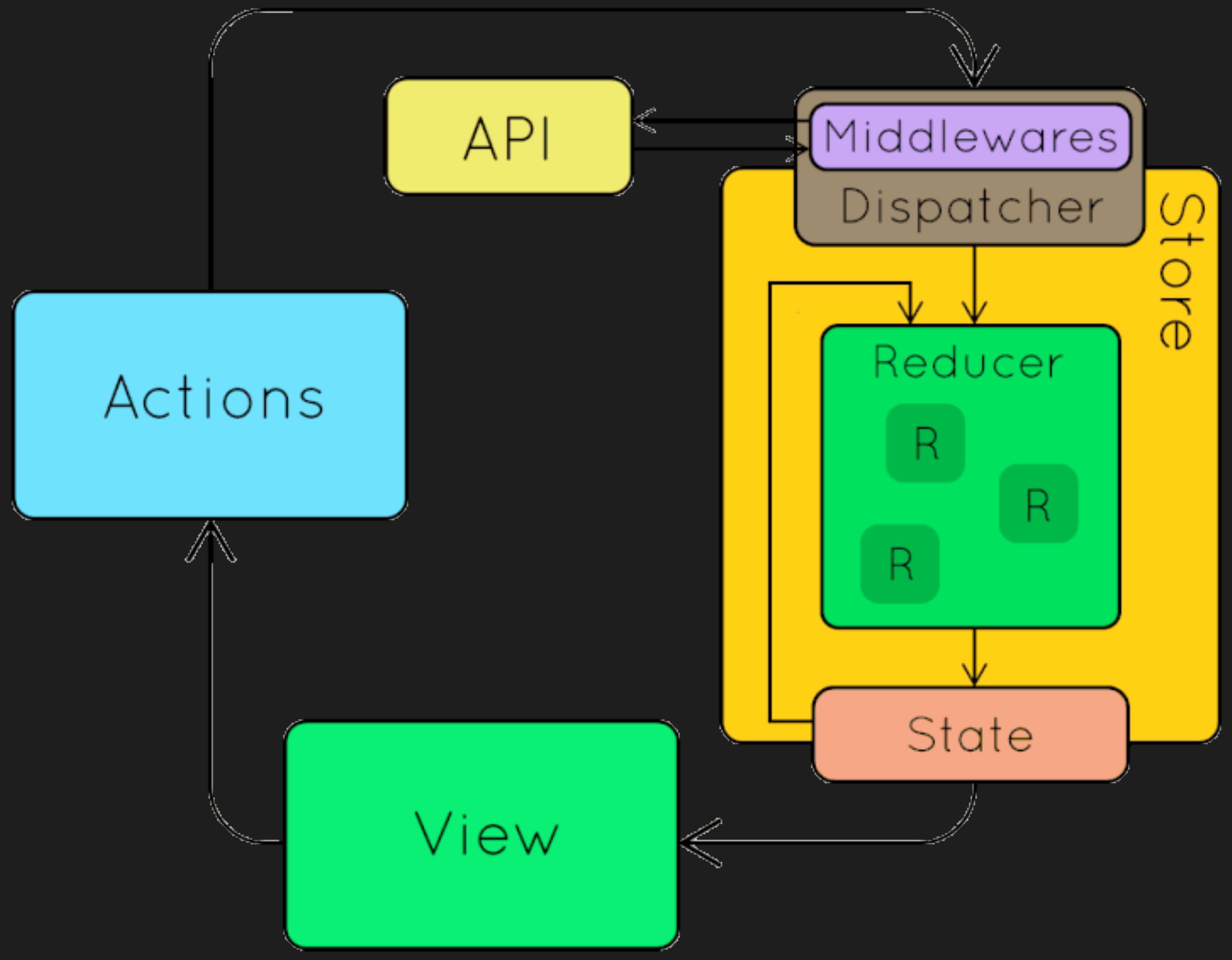
src/
redux/
thunks/
routes/

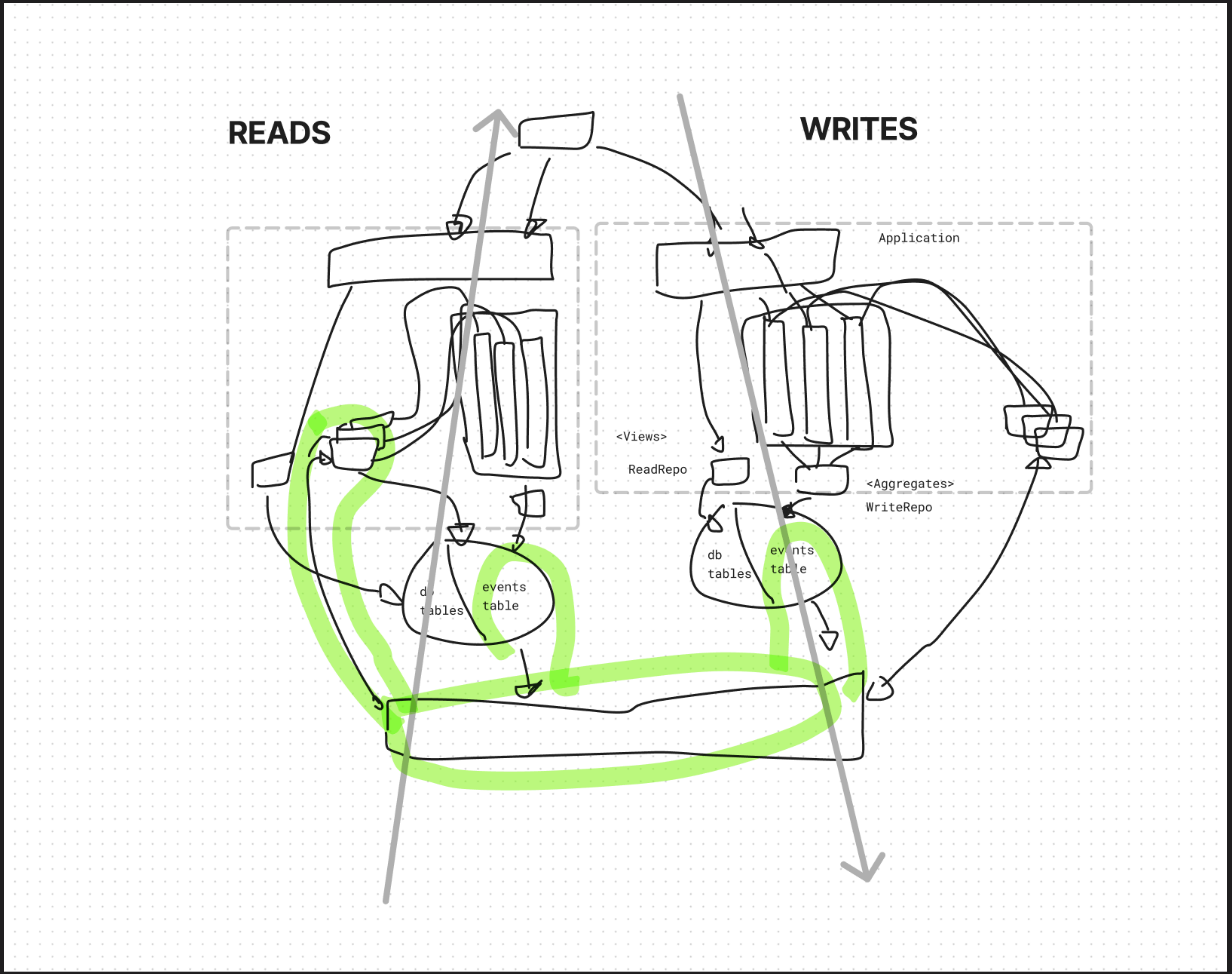
Vs

```
src/
|-- features/
|   |-- auth/
|   |   |-- components/
|   |   |   |-- LoginForm.tsx
|   |   |   |-- RegistrationForm.tsx
|   |   |   |-- UserProfile.tsx
|   |   |-- containers/
|   |   |   |-- AuthContainer.tsx
|   |   |   |-- ProfileContainer.tsx
|   |   |-- hooks/
|   |   |   |-- useAuth.ts
|   |   |-- actions/
|   |   |   |-- authActions.ts
|   |   |-- reducers/
|   |   |   |-- authReducer.ts
|   |   |-- sagas/
|   |   |   |-- authSagas.ts
|   |   |-- types/
|   |   |   |-- authTypes.ts
|   |   |-- index.ts
|   |-- dashboard/
|   |   |-- components/
|   |   |   |-- DashboardWidget.tsx
|   |   |   |-- DashboardSidebar.tsx
|   |   |-- containers/
|   |   |   |-- DashboardContainer.tsx
|   |   |-- actions/
|   |   |   |-- dashboardActions.ts
|   |   |-- reducers/
|   |   |   |-- dashboardReducer.ts
|   |   |-- sagas/
|   |   |   |-- dashboardSagas.ts
|   |   |-- types/
|   |   |   |-- dashboardTypes.ts
|   |   |-- index.ts
```

Symptom #3: Not Embracing a One-Directional Read/Write Flow (CQS Violations)

- *CQS = Command-Query Separation*
  - *API calls, commands, etc - they need to be either READS or WRITES*
  - *Ie: getOrCreateIfNotExists is a violation*
- *It's unclear that you have two code paths (one for reading, one for writing)*



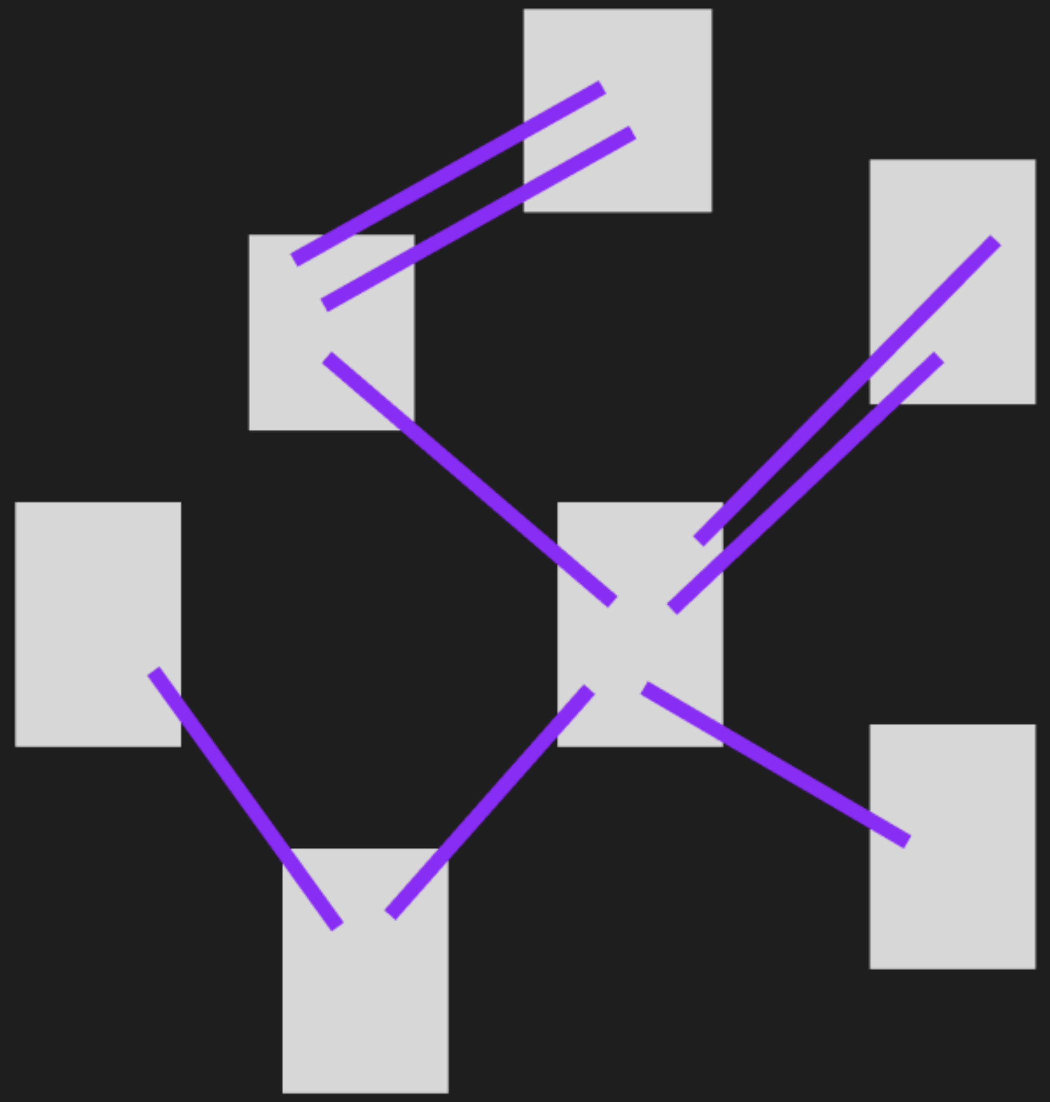


essentialist.dev

The underlying problem?

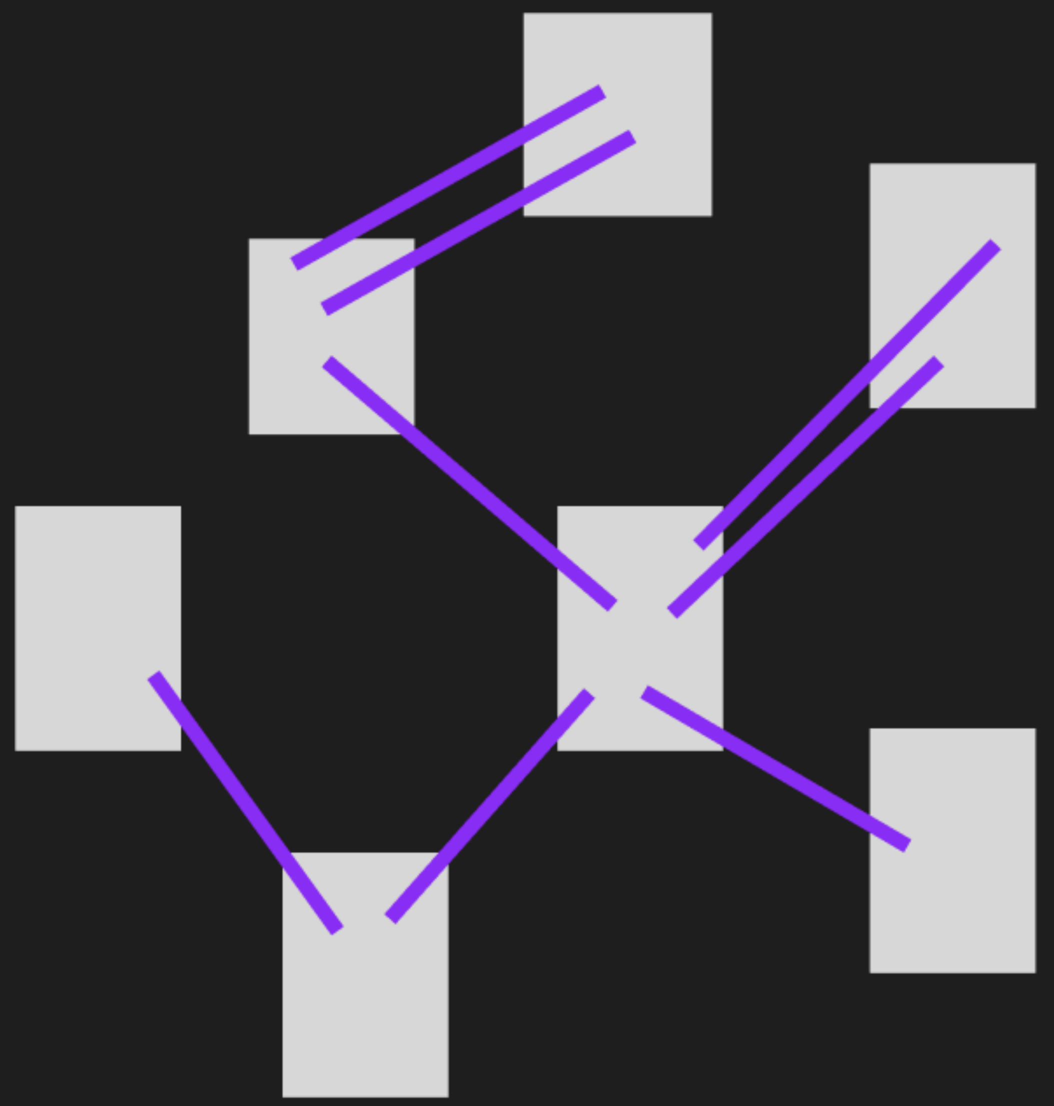
Coupling

The degree of interdependence between modules or components.



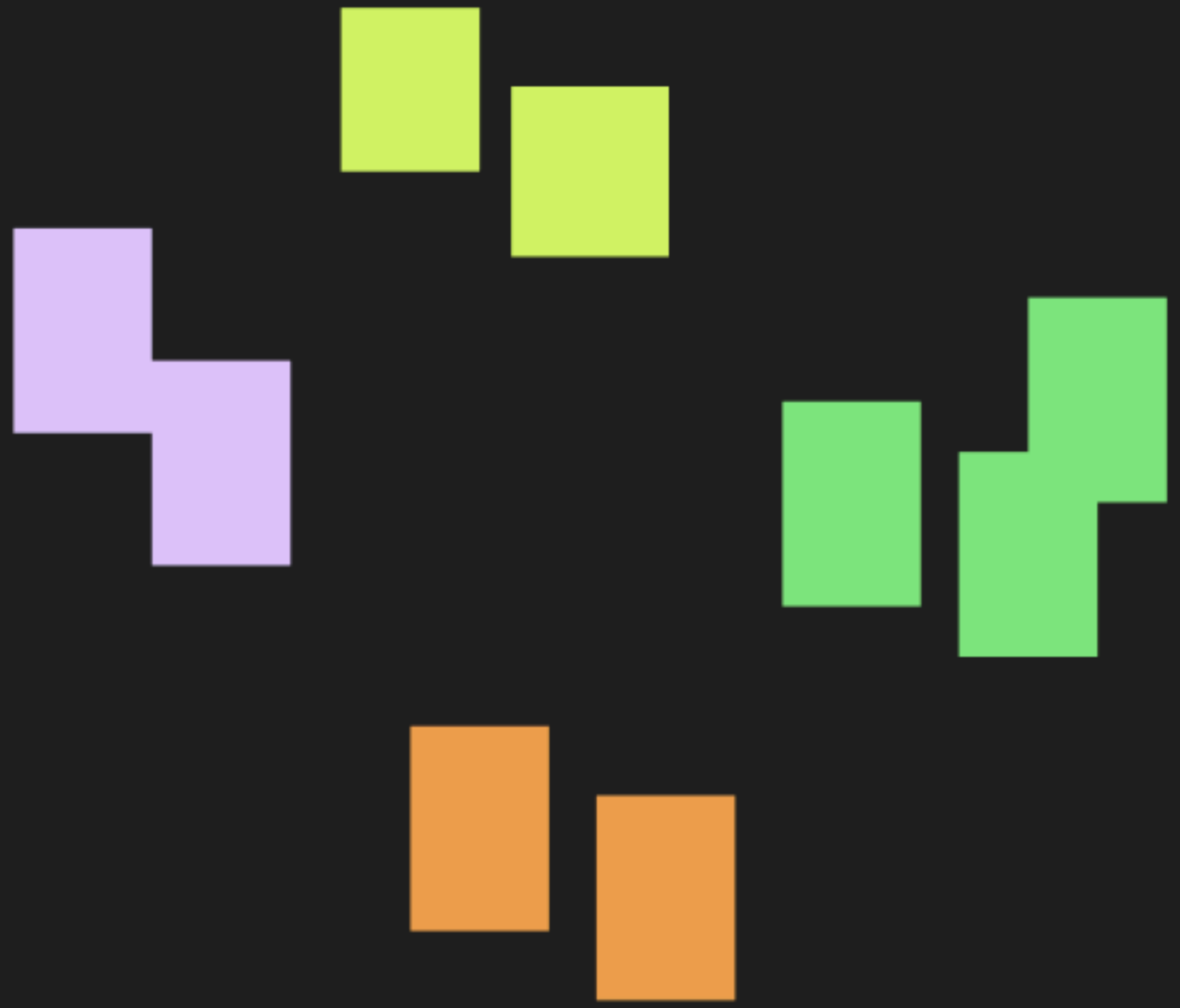
Coupling

The degree of interdependence between modules or components.



Cohesion

The degree of relatedness between modules or components.



essentialist.dev

Maintainability Problem

Tight Coupling

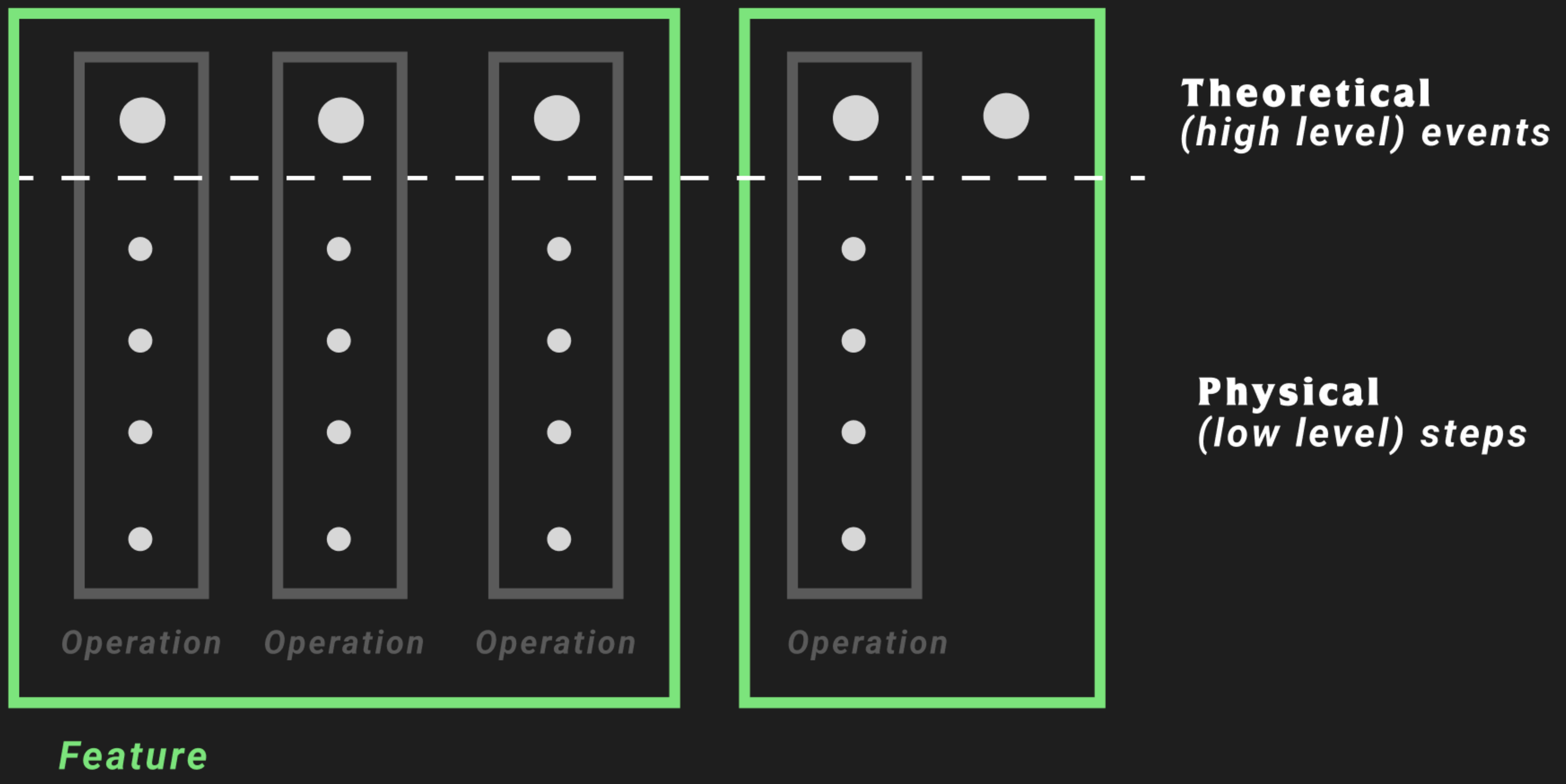
Example here

- Can’t tolerate change
- Afraid we’ll break something

Maintainability Problem
Low Cohesion

Example here

- **hard to understand**
- **Have to fling around codebase to do work, things don't make sense in a singular place**



Physical Essential

Vertical Slicing

essentialist.dev

What we'll cover

- *What is vertical slicing?*
- *How does it work?*
- *How does it help us write better code in the long term?*

I’ll make it simple

Software is all about inputs, outputs & state changes.

Slice type #1: Operations (commands & queries)

Show image of the abstraction by slicing by operation

Image that goes deeper to explain what an operation is

Code which is over here, shows a number ‘createUser’, ‘editUser’, etc hidden/closed

Actual code which shows this and then shows doing a new UserCreated() at the very end

Commands evoke state changes

Showing that it hits the database at the ver end and performs a state change

Queries return data (without changing state)

Show the sort of event modelling diagram here to depict that data goes in

An operation should be a command or a query but not both.

If you look closely, an information system really is just inputs and outputs

Show event modelling

Slice type #2: Features/modules

Show the abstraction prism again, but cut it

Organizing your codebase into slices (package by module/component/domain)

Modules - capabilities/domains

<capability>

    <features>

see the diagram that I wrote and also take a look at the Humans & Code stuff I wrote in the previous solidbook.

Alternative perspective

Data, Behaviour, Namespaces

Input, output

Command, query

Module, sub-module

Decoupling

(Independent slices)

Show an image this is how we decoupled the operation

Show an image - this is how we've decoupled the module & feature

Benefits of Vertical Slicing

- *Simple conceptual model*
- *Organize code by module/feature slices makes architecture much easier to understand and features easier to find*
- *Decoupling/independence (less "ripple")*