Overview

[write summary here]

Concepts Overview

Level	description	example
Patterns	composition of the app based on the larger problem to be solved using design principles and practices. Should specify the kinds of interfaces needed and the relationships between them. Build patterns are like a class of pattern, where the various interfaces, elements, data infrastructure, and relationships are already somewhat defined.	Build: CRM, dashboard, form Design: OOP, dependency injection Data: SQL, NoSQL
Interfaces	collection of arranged elements that specifies the functionality each element should provide for a given user interface. It should specify the contract for a given element as well as the relationship between elements	page, app, input
Elements	elements are the basic building blocks of any application, elements are functions or objects that behave in specific or pre-determined ways	blocks, connections & requests, actions & events, operators,

Starting with a problem

For something even as simple as a web-form, there are a number of pages, interfaces, and elements that must be used. Over and above the technical components, there are also various design principles we can consult. To construct a form we would also need to make some decisions on what information we will capture, what checks or restrictions we might make on the data captured, and where we will send that data.

The initial spec we want is probably as simple as a series of tags that can be sorted. This sets the questions that will be answered by the patterns we decide to use. Already at this level we can start to see how we can make design decisions about the patterns.

```
# problem spec
## double hash indicates guidance / help text
---
problem: What are specialty coffee preferences of South Africans  ## frame this as a question
users:           ## who will be using this interface?
- cuppers     # people doing a formal tasting
- homebrewers # people making their own coffee at home
contributors:    # who will be maintaining, constructing, and making insights from this?
- developer    # who made this originally and can be consulted
- contributors # people who can build features
- support      # if something breaks these people can be consulted
documents:     ## what are the repeated collections of information?
- Brew recipe
- Coffee info
- User profile
- Blind tasting pairs
- Score sheet # maybe this is a more detailed review
- Homebrew notes # maybe this is simpler and can be customised
stories: ## sequences of actions, documents, and results for users
- blind tasting
- preset cupping
- homebrew
  - ?
---
```

for the `problem` there is already a gap between the information given and the rest of the construction. A simple survey with 3 input blocks might suffice this question if it's not properly understood and we might assume that all that's needed is name, location, and text for coffee selection, or maybe a multiple choice. However, this doesn't account for the subject and the field itself, where we need to specify what we mean by 'coffee'.

in the `users` section, we can see that there are two specific use cases for this application that are separate from the contributors. The patterning decision we must make should outline if this will take the form of 2 apps or one.

We could have one app for cupping settings that is a lot simpler and designed to accommodate for many entries in a short period of time, perhaps using a `List Block` element. However, for the Brew recipe, because this is about a repeated action over time, having different ways of capturing the information and ability to re-use previous information with a `_request` operator a could be really useful.

Construction of `documents` can start to tell us about data patterns that we can be using for this in later stages.

Our `stories` where slicing becomes important for the interface. Do we want to organise homebrews by method? if so, how do we represent goodness of fit between coffee type and method, or repeated recipes or new recipe construction. This is where we might refer back to the problem to set this definition.

> [!NOTE]
> Based on the above, you should be able to think about various patterns that would be good to use. These could be various:
>
> - build patterns (or repos you could copy over),
> - data patterns (this might depend on some existing data source that's easy to replicate or build out yourself), or
> - design patterns (such as the tension between abstraction and coupling you might like to use)

Mockups

Since you already have some rough ideas, you might like to compile some examples of data to use in the following phases. This might be manually created data, or some rough stories, or existing information that is close to what you need.

The goal of this is to make it practical without locking you into one particular style or way of doing things. The closer it can be to what the actual data looks like, the better. A nice way to generate some specific data could be to make a spreadsheet table or to make a small Apple shortcut or similar. An example can go a long way!

Meta-patterns

Branching & Iteration

Just like how things evolve over time, in these early phases it's very likely that the shape of what you're making will change depending on the size, scope, and nature of the app (or apps). As your app grows, it won't grow top to bottom, it'll develop in branching iterations.

For example:

1. As you add a new block on a page, so you will add new events and requests under the relevant headings
2. As you add new pages to your app, so you will add new components to your /shared folder to reduce repetition
3. As you add new apps to your project, so you will reconfigure plugins and services to account for these additions

This is the goal of the Lowdefy project, to be able to morph to what you need in that moment. Because of this, it's helpful to have things written out so that you can see even in the beginning stages how it morphs, changes, and ultimately evolves in a more iterative than linear way.

Abstraction and Coupling

Atomising your build for composability

Folder Structure

The folder structure isn't set, but general rule of thumb is, if $n > 2$, make a new structure to handle it. For example, when a component is used more than twice in an app, created a shared folder to reference so that it updates across all places. *(refer to abstraction and coupling section above)*

- /Project folder
  - /App

    `most examples start here because they are single apps, for multiple apps, this may also contain folders`
    - lowdefy.yaml

      `this is really all you need, everything could be configured from here, but probably shouldn't`
    - package.json `highest level or app level??`
    - menus.yaml `optional: if lowdefy.yaml is too big`
    - global_variables.yaml `optional: if lowdefy.yaml is too big`
    - connections.yaml `optional: if lowdefy.yaml is too big`
    - /pages
      - /page_1
        - page_1_module.yaml
        - page_1_view.yaml
      - /page_2
        - page_2_edit.yaml

        - page_2_new.yaml
    - /shared
        - shared_page.yaml
        - shared_component.js
        - shared_template.yaml.njk
        - shared_header_footer.yaml.njk
- /plugins
    - /plugin_1
    - /plugin_2
- pnpm-lock.yaml `this will sit in the highest level folder`
- pnpm-workspace.yaml `this will sit in the highest level folder`
- LICENSE `this will sit in the highest level folder`
- .swcrc `this will sit in the highest level folder`
- /docs `stored info like aggregations, to do lists, etc`

Interface

The interface is a loop for creating and iterating.

1. Your starting recipe
2. The ingredients you need to get, which projects and templates you can borrow from
3. Checks to see if you are on the right track
4. Checklist for adding elements

Instead of trying to account for everything in the beginning or trying to make it perfect, you can start with some very basic and simple ideas and grow it from there.

1. Start with the timeline

- [ ] Put detail into the story to create a time line, by user
- [ ] Attach documents and contingent documents to key nodes or decisions on the time line
- [ ] Map out the variations of this timeline that are less desirable or less likely
- [ ] Map out the alerts, warnings, and indicators that contributors would want to or need to see

Based on the above, you should be able to group key nodes into screens that will be needed. As you group the screens, you may also find that there are more screens needed than what you originally planned. For example, you might have the form screen, but you might also need a "your submission has been received" screen. While screens can be complex quickly, the surest way to make sure you don't miss anything important is by walking through it in your mind or on a page in the exact steps the user would take.

This should leave you with some rough wireframes, which we will abstract out in the next phase. In these wireframes it should be obvious what each element needs to do, or rather the most basic 'contract' each element will fulfill. The element selection isn't necessary at this stage, rather something as simple as: "display", or "display and edit", or "input response".

2. What are the core pages needed to facilitate this?

When you review your first rough drafts that account for the `documents`, `users`, and `stories`, you might already have some indication of what the pages should be. Now is your opportunity to review different examples of interfaces, draw some inspiration, and use a MECE [1] approach to the various pages, at least to get started. There's no right way.

Based on these pages, and how the documents are written and edited, you might start to see how the data patterns are emerging. You can then start creating whatever diagrams you need in order to facilitate this. However, there should already be some diagrams to begin with if you're creating from an existing build pattern.

3. Does this cover what you need it to cover

- ☐ Does this contain every document, every user, every story?
- ☐ Does this contain the necessary interfaces to manage, edit, and maintain the documents and users if needed?
- ☐ Is there any summary or overview information needed for any of the users that isn't covered by the existing stories?

These questions might lead you back up. It could even be a loop as you figure everything out.
The potential value of addressing these in a loop is that you can build out what you have with each successive loop, becoming more detailed on specific elements.

4. Elements Checklist

- ☐ What are the different events that happen? (start with timeline notes)
  - ☐ Are there appropriate actions set for each of these events?
- ☐ What are the core blocks that need to be on each page?
  - ☐ Are blocks appropriate for the kind of information (display or input)
- ☐ What function is being performed within this block? (`operators`)
  - ☐ Are inputs and outputs being appropriately handled?
- ☐ What information is being pulled into (`_request`) and written out of each block?
  - ☐ Is this request mapped in the `Request` action of the block?
  - ☐ Is this request set up as a connection in the app configuration?
  - ☐ What functions are being performed within this request? (`operators`)

Concept Details

events and actions

Events and Actions are the trigger and response like any kind of automation you might program. When `x` happens, do `y`, where the action is some system mandated action that either sits with the user (e.g. `CopyToClipboard`) or speaks to the system (`SetState`). These are the larger orchestration actions that can be taken in the world of the app—generally at an app or page level.

```
- id: example
  type: element
  - when: event
     do: action
      object: params
```

operators

Operators are functions that enable the developer to reason with computer. It includes a way of referring to specific items and objects and the various functions that can be performed with these objections. It is more detailed and more flexible than events and actions would often work more at an element level. Operators include everything from simple functions (e.g. `_and` , `_sum`) to more complex references to other files and variables set elsewhere (e.g. `_var` and `function`).

Operators have a higher level of complexity than events and actions because they facilitate much more flexibility of action between elements and methods. These are more comparable to formulas on a spreadsheet.

```
- id: example
  type: element
  - for: this input
    perform: function on app state change
      object: arguments
```

blocks

These are the catchment areas and spaces in which things happen and are seen. Blocks are state-less by nature, and change based on the rules written with operators and actions. They can collect and display information.

The four kinds of blocks are

1. Input blocks: collecting or taking in information from the user
2. Display blocks: anything visual on a page, from a table to an icon element.
3. Container blocks: the holding blocks, the larger construction of blocks with various interactive capabilities in which the input or display or list blocks sit.
4. List blocks: List category blocks render multiple `content areas` , based on data in the `state` object.

The basic composition of these elements, like the others includes an `id` and a `type` , these can get more and more complex by adding `properties` that are specific arguments specific to the type. Certain blocks, like display blocks can also use events to improve how well this can be interacted with.

```
# basic block
- id: example
  type: element
  properties:
    keys: values of the type

# lists, controllled, uncontrolled
- id: example
  type: element
  blocks:
    - id: example 4
      type: List block
      properties:
        list: things
```

Connections and requests

Connections are longstanding relationships that are arranged up front, usually by adding secrets through a third party provider or an auth provider. This is how the data usually gets into the app to the right user with the right permissions. While user roles and auth have more to do with the app pattern, the connections are about establishing how and where the information will come from and be fed to the app.

If the connections specifies the contract or relationship between your app and a particular source, the request is a standard order placed to the connection type (e.g. your database). Requests may be page or even specific and often work at the Interface level as an Action.

At the element level, the `_request` operator narrows down the order to a specific detail that needs to be used at the the element level. Either within a function or for a particular display, this operator allows us to use the data brought in with the `Request` action to this page that is mediated by the `Connection` we have set up.

1. Mutually exclusive, collectively exhaustive: it covers everything that needs to be covered without duplication ↩