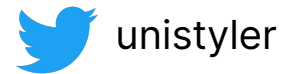


Frontend Architecture: How to Build a Zoo?

Thomas **gossi** Gossmann



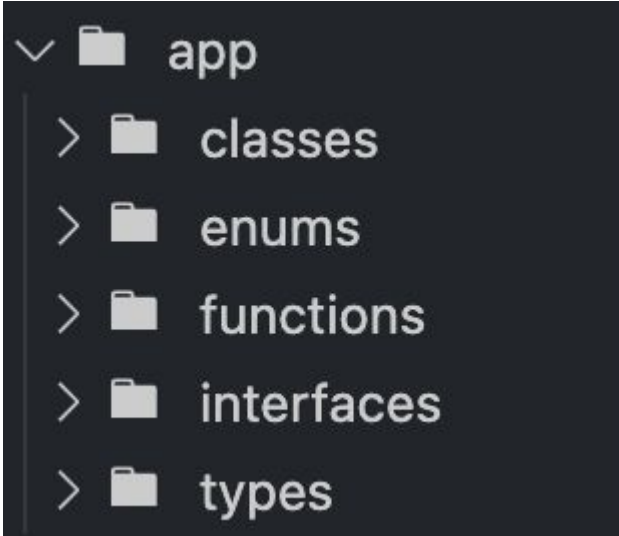
What is Software Architecture?

- Architecture is a skill, not (just) a role
- The ability to design a system, reason about tradeoffs and understand the design
- Flow of decisions (instead of upfront design work)
- “Software architecture is the stuff that’s expensive to change”

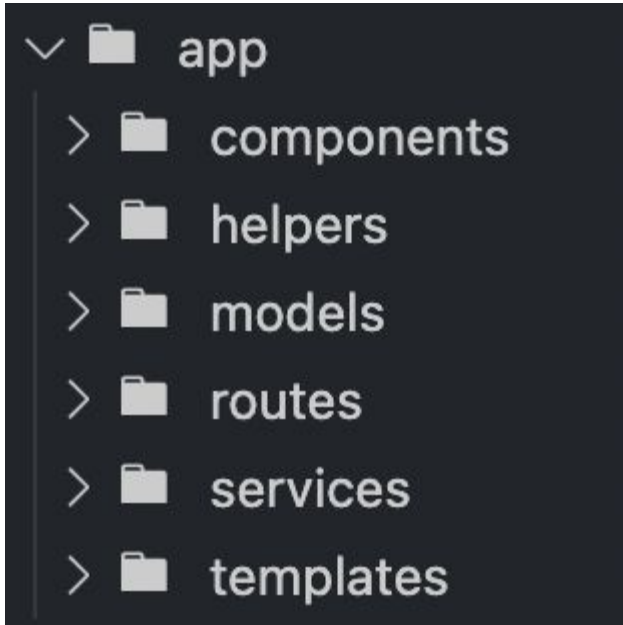
Examine this while building a zoo

Disclaimer: I’m not a domain expert for a zoo. This is all made up.







What does this product do? (1)



What does this product do? (2)



What does this product do? (3)

- ✓  app
 - >  atoms
 - >  molecules
 - >  organisms
 - >  pages
 - >  templates

Identify the Domain

- Why do People come to the Zoo?
 - Animals (Fauna) ?
 - Plants (Flora) ?
 - Info-/Entertainment ?
 - Food & Drinks ?

Identify the Domain

Core

Animals

Supporting

Plants

Info-/Entertainment

Generic

Food & Drinks

Restrooms

Identify the Domain

Core

Animals

Supporting

Plants

Info-/Entertainment

Generic

Food & Drinks

Restrooms

Distilling the Domain



-  Toiletten
-  Behinderten-WC (mit Euro-Schlüssel)
-  Wickelraum
-  Rollstuhlverleih
-  Restaurant
-  Kaffeehäuschen
-  Imbiss
-  Eis
-  Spielplatz
-  Servicecenter
-  Erste Hilfe
-  Defibrillator (AED)
-  Souvenir-Shop
-  Ausstellung (Berliner Zoo-Geschichte)

Bounded Contexts

- Multiple models can co-exist in a big project
- Contexts need explicit boundaries
- Model strictly consistent within the context
- Clear definitions communicate the purpose
- Context gives autonomy

Bounded Contexts

Option 1: Compounds



Picture from: <https://www.glmv.com/work/zoo-boise-master-plan/>

Reasons:

- They have natural boundaries
- Can react to outside events
- Autonomous inside

Option 2: Separate by Functional Staff

Medical Staff

- Anthropometry
- Blood samples
- Excrements

Zookeeper

- Food
- Water
- Clean compound

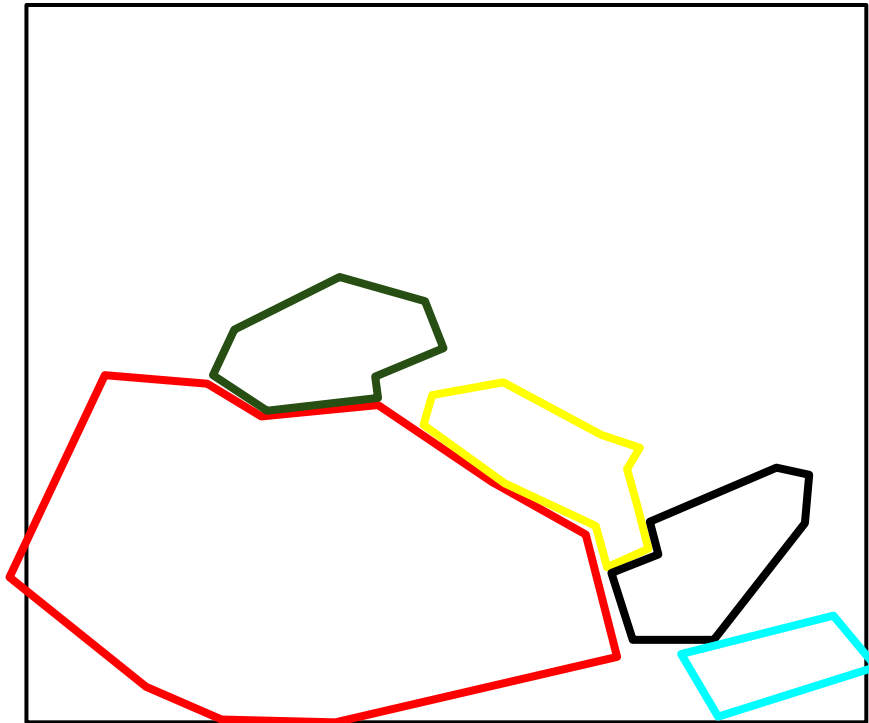
Gardener

- Which plants are growing?
- What are toxic plants to an animal?

Reasons:

- Functional teams
- Ubiquitous language by discipline

Option 3: Areas



Reasons:

- Cross-Functional teams per area
- Follows the team's communication structure (Conway's law)
- Keeps communication paths short

Bounded Contexts

Compounds



Picture from: <https://www.glmv.com/work/zoo-boise-master-plan/>

Reasons:

- Natural Boundaries are a great metaphor
- I'm not a domain expert for Zoo at all
- Use themed areas for hierarchical organization

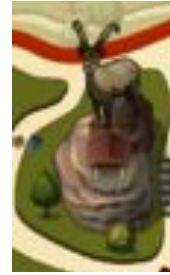
Zoom Levels



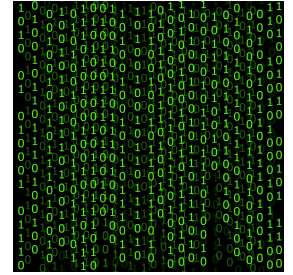
Level 1
Zoo



Level 2
Area



Level 3
Compound



Level 4
Code

C4 Model

model for visualising software
architecture

“maps of your code”

Vocabulary = Scary

Context Mapping

Generic Subdomain

Supporting Subdomain

Distilling the Domain

Bounded Context

De-composition

Core Domain

Ubiquitous Language

Zoom Levels



You
Level 1
Zoo

Level 2
Area

Level 3
Compound

Level 4
Code

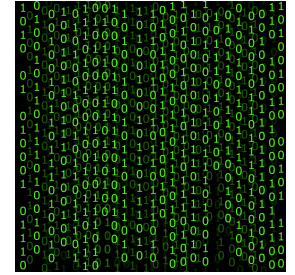
Business
Entire
Business

Business
Capabilities

Business
Services

Business
Rules

Zoom Levels



You Level 1
 Zoo

Level 2
Area

Level 3
Compound

Level 4
Code

Business Entire
 Business

Business
Capabilities

Business
Services

Business
Rules

C4 Context

Containers

Components

Code

Zoom Levels



You
Level 1
Zoo

Level 2
Area

Level 3
Compound

Level 4
Code

Business
Entire
Business

Business
Capabilities

Business
Services

Business
Rules

C4
Context

Containers

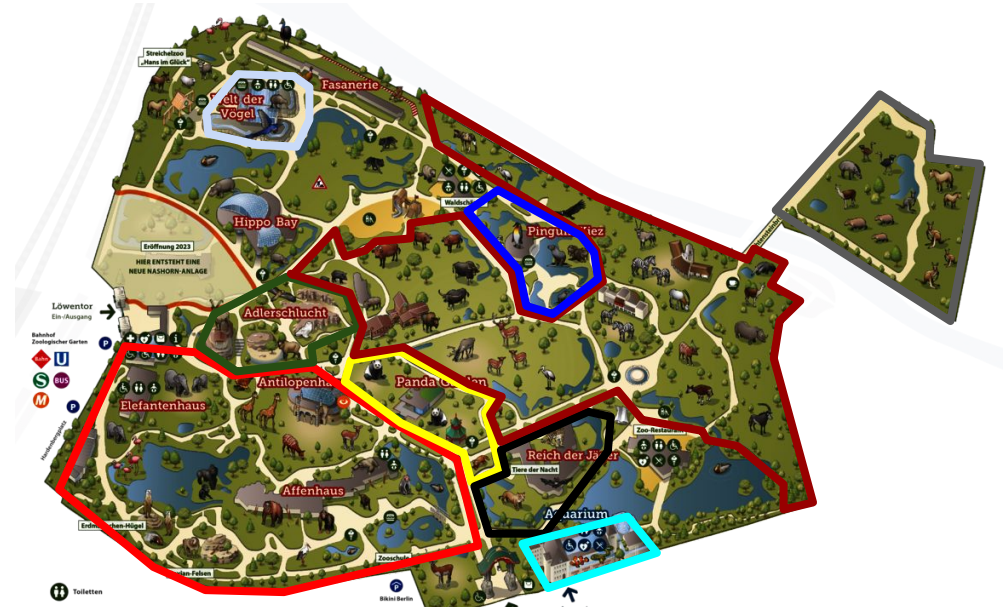
Components

Code

Zoo Folder Structure

Level 1
Zoo

- ✓ ZOO
 - > africa
 - > antarctica
 - > aquarium
 - > asia
 - > aviary
 - > mountains
 - > oceania
 - > predators
 - > savanna



Result

- ★ Understand the domain
- ★ Distilled the domain into manageable pieces
- ★ Carved out bounded context
- ★ **Established a language that works across disciplines**

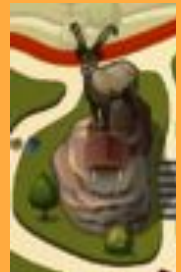
Part 2

Tactical Design

Level 2
Area



Level 3
Compound



Models Expressed in Software

Level 2
Area



Value Objects

Aggregates

Repositories

Entities

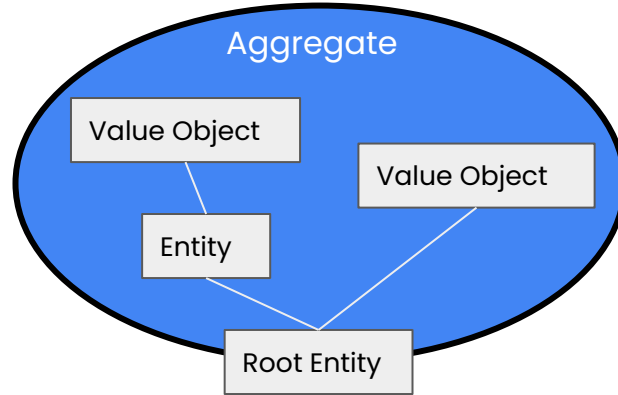
Events

Services

Factories

Aggregates

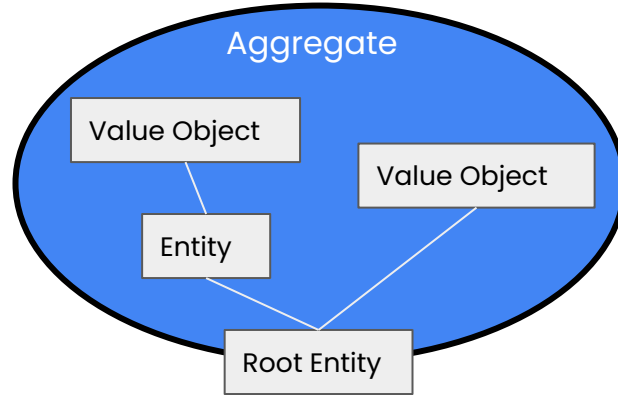
Backend



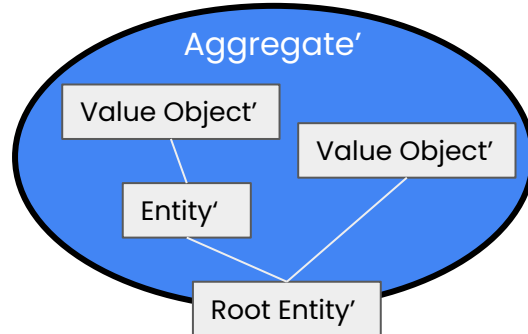
Frontend

Aggregates

Backend

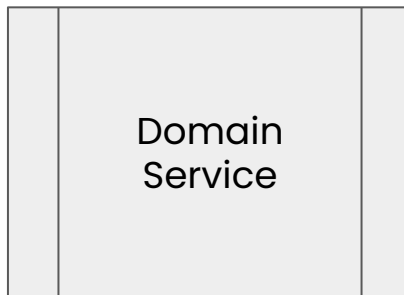


Frontend



Services

Backend



Business Process
(eg. feeding Animals)

Frontend

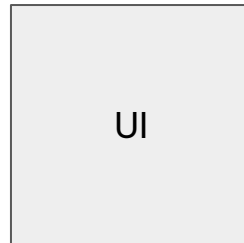
Services

Backend



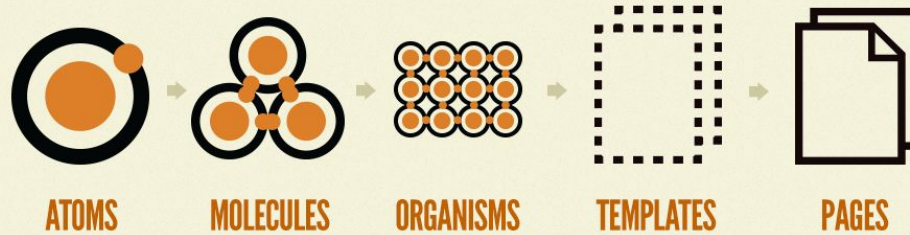
Business Process
(eg. feeding Animals)

Frontend

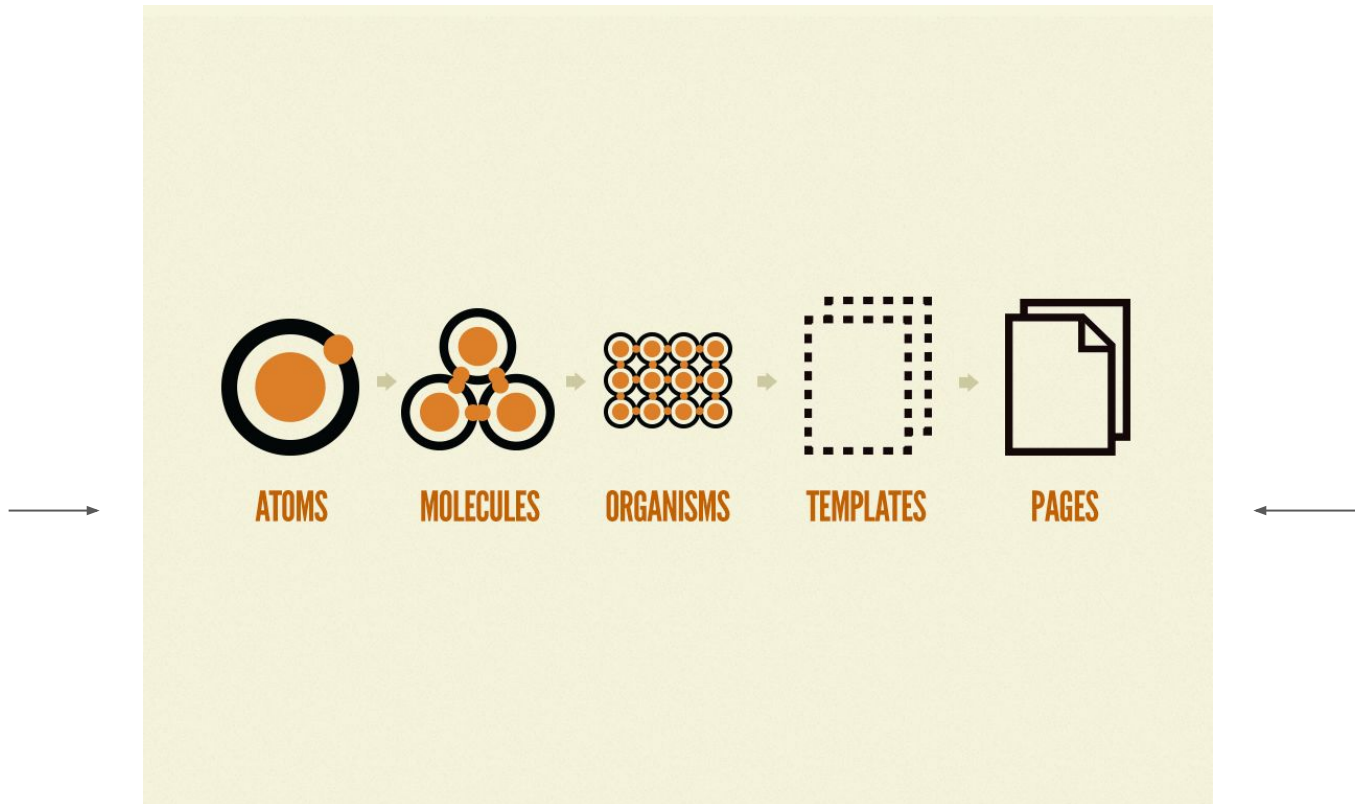


Screens
Forms
Status Elements

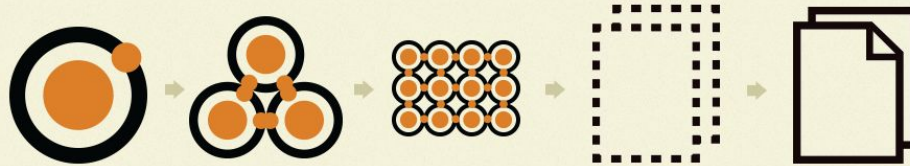
User Interfaces



User Interfaces



User Interfaces



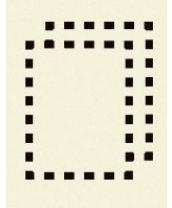
Component Classification



Areas

Use for housing multiple compounds

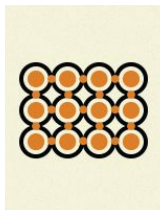
(Screens/Routes/Pages)



Pathways

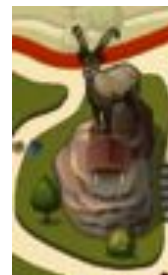
The same walking infrastructure for visitors of a zoo

Component Classification



Compounds

A component representing a compound



Interior

Custom tailored interior for *one* compound



Carryables

Elements that can be moved around (from one compound to another)

Models Expressed in Software

Level 2
Area



Value Objects

Aggregates

Repositories

Entities


Events

Services

Factories

Event Storming

Collaborative exploration of complex business domains

 Alberto Brandolini

eventstorming.com

Event

Command

Read Model

System

Policy

Actor

Models Expressed in Software

Level 2
Area



Value Objects

Aggregates

Repositories

Entities

Events

Commands

Services

Factories

OOUX

Object Oriented UX

Object-Oriented UX offers a better way to break up complexity, allowing us to work iteratively and holistically

👤 Sophia Prater

ooux.com

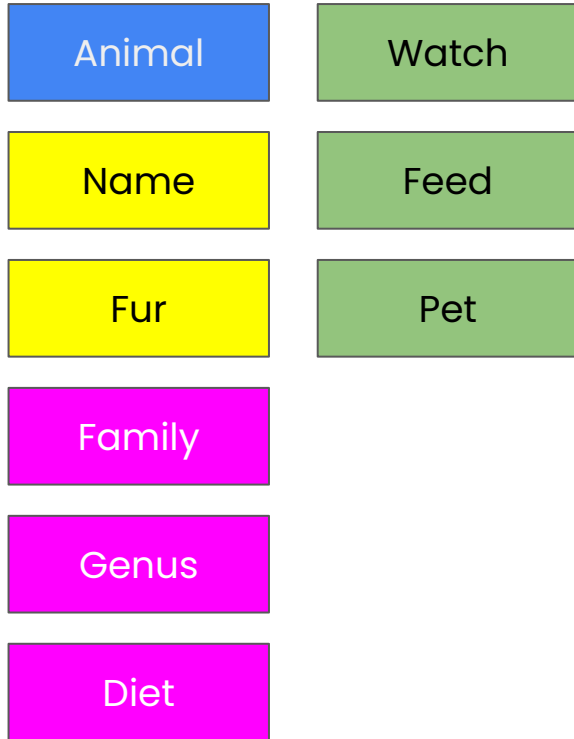
Object

Contents

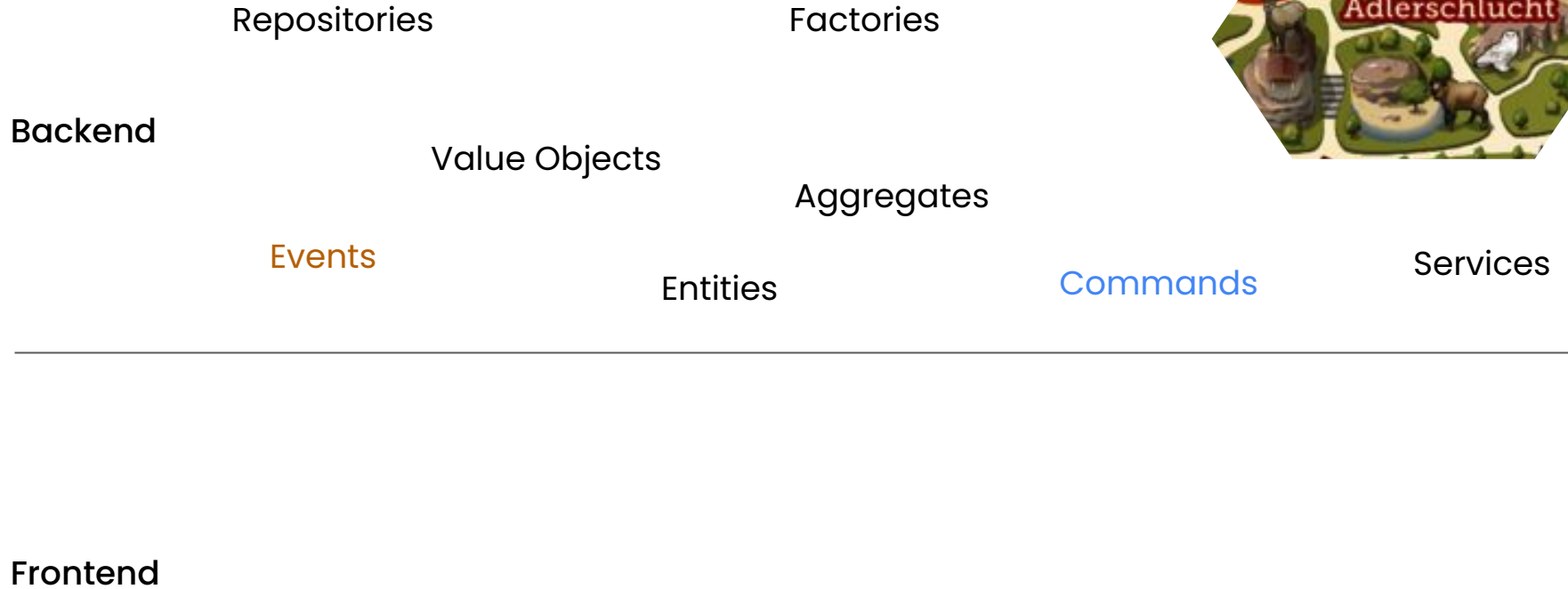
Actions

Metadata

An OOUX Animal

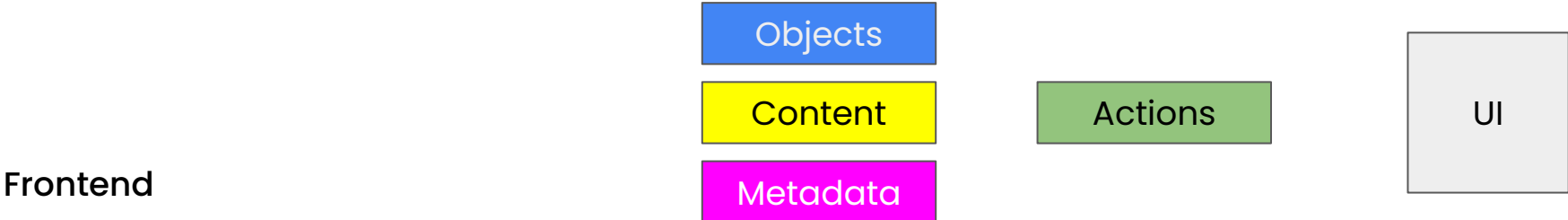


Backend + Frontend



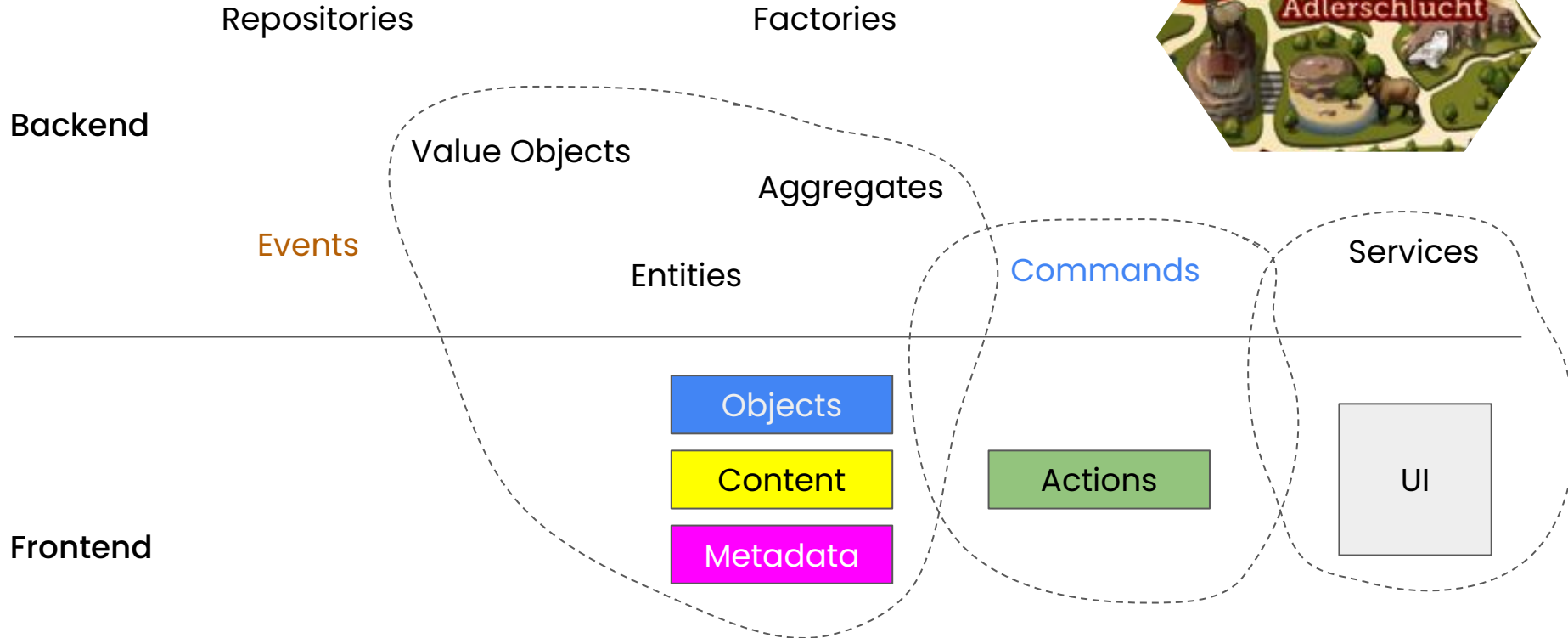
Backend + Frontend

Level 2
Area

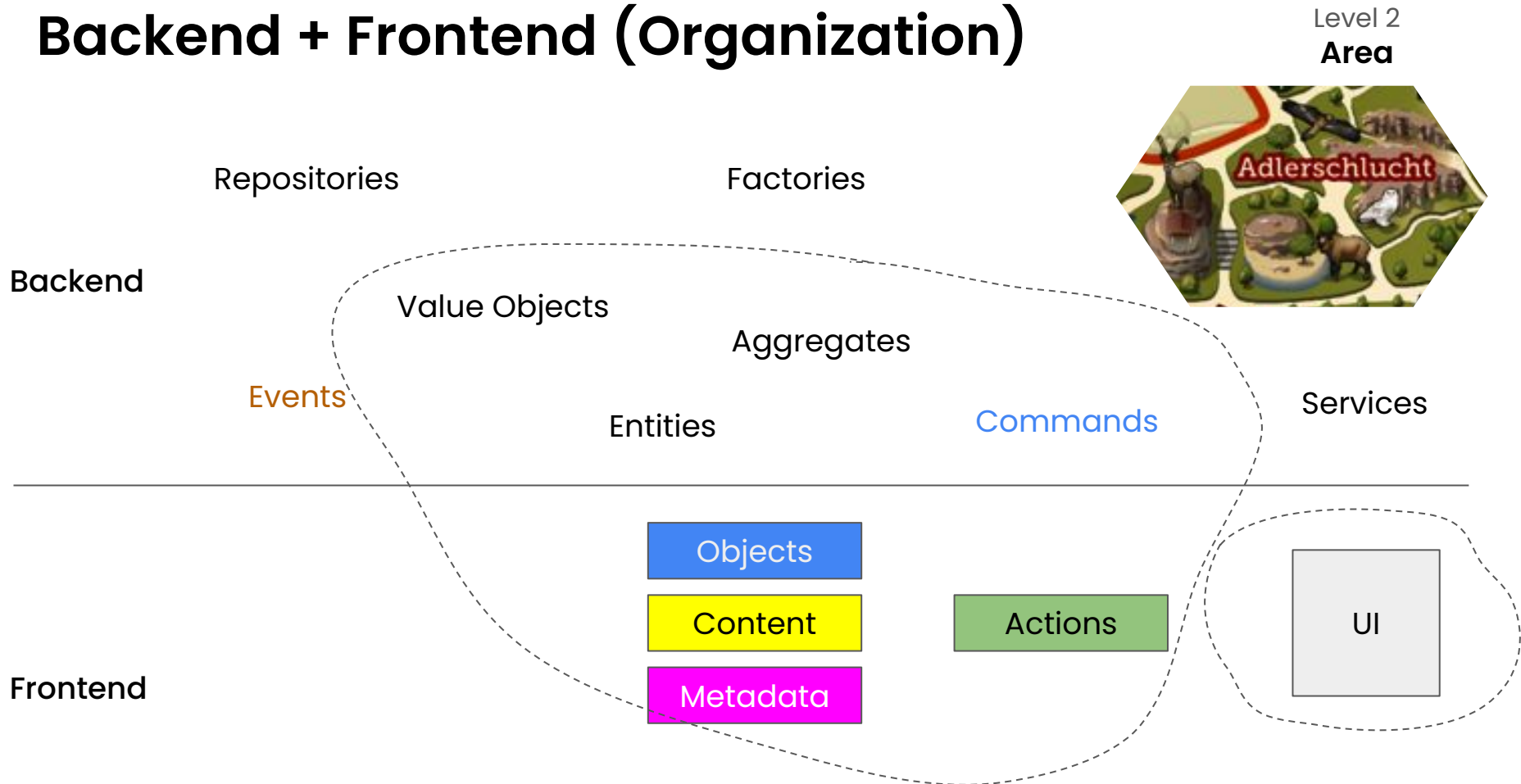


Backend + Frontend (Communication)

Level 2
Area



Backend + Frontend (Organization)



Separate UI and Business Logic

Business Logic

- Aggregates / Entities / Value Objects
(Nouns)
- Actions
(Verbs)
- Policies & Rules

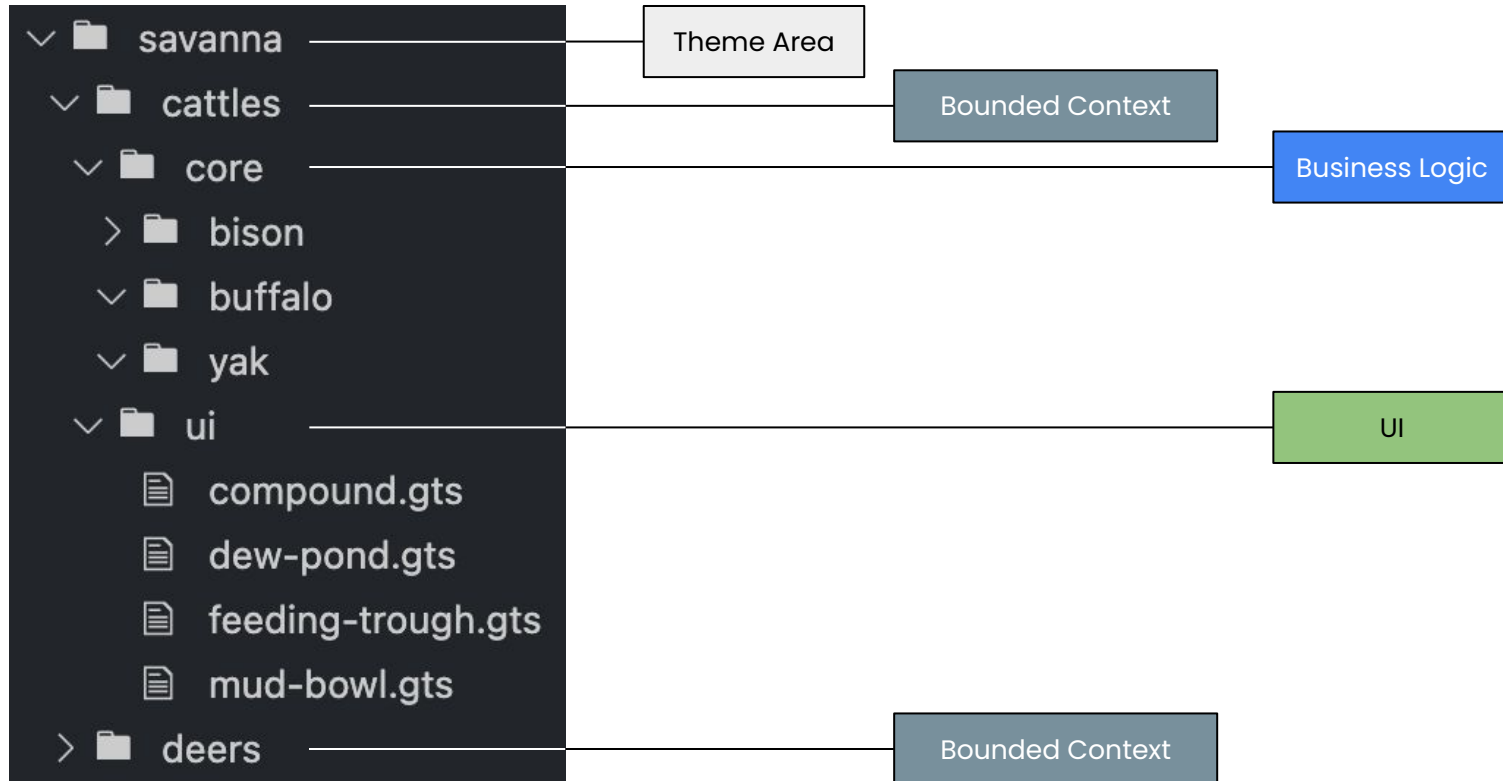
Visual Representation

- Communicate System State
- Offer Behavior

Benefits

- Maintainability
- Readability
- The feeling of “home”
- Fast feature development

Area Folder Structure



Tactical Design

Result

- ★ Frontend Modeling = Backend + Design
- ★ Separate UI and Business Logic
- ★ Component Classification that **follows the established language**

Part 3

Code

Level 4
Code



Aggregate / Reactivity

- Plain JS/TS
(Framework
Agnostic)
- Connect to your
Reactivity System
(UI / State
Management)

Aggregate / Reactivity

- Plain JS/TS
(Framework Agnostic)
- Connect to your
Reactivity System
(UI / State Management)

```
enum Diet {
  Herbivore,
  Carnivore,
  Nekrophage
}

class Animal {
  id: string;
  diet: Diet[];
  family: string;
  genus: string;
}

class Gibbon extends Animal {
  name: string;
  fur: string;

  eat() {}

  play() {}

  climb() {}
}
```

Classes

- (de)serialization
- (re)hydration

Functional

- DTO
- transforms

What's your choice?

```
enum Diet {
  Herbivore,
  Carnivore,
  Nekrophage
}

interface Animal {
  id: string;
  diet: Diet[];
  family: string;
  genus: string;
}

interface Gibbon extends Animal {
  name: string;
  fur: string;
}

function eat(monkey: Gibbon) {}

function play(monkey: Gibbon) {}

function climb(monkey: Gibbon) {}
```

CQS: Command & Query Separation

Methods to either be commands that perform an action or queries that respond data, but neither both!

Queries

- **Questions:** Ask facts about the system
- **Abilities/Authorization:** Control access

Actions & Commands

- **Command Design Pattern**
- **Commands:** Used in Backend to perform business logic
- **Actions:** Used in Frontend to connect vocabulary with designers and probably call commands in the backend

Queries

Questions:

Ask facts about the system?

```
function isHerbivore(animal: Animal) {  
  return animal.diet.includes(Diet.Herbivore);  
}  
  
function isCarnivore(animal: Animal) {  
  return (  
    animal.diet.includes(Diet.Carnivore)  
    || animal.diet.includes(Diet.Nekrophage)  
  );  
}
```

Abilities:

Control access

```
function canPet(animal: Animal) {  
  return !isCarnivore(animal)  
}
```

Expect to have
hundreds of these

Queries



Picture from: <https://www.glmv.com/work/zoo-boise-master-plan/>

Abilities:

Control access



```
export function canWatch(monkey: Gibbon, user: User) {  
  return true;  
}  
  
export function canFeed(monkey: Gibbon, user: User) {  
  return user.role === Role.Zookeeper;  
}
```

Queries

✗ DO NOT

Inline conditionals (in templates)



```
{#if user.role === Role.Zookeeper}
  <button on:click={() => feedGibbon(gibbon)}>
    Feed Gibbon
  </button>
{/if}
```

✓ DO


Extract into function



```
{#if canFeed(gibbon, user)}
  <button on:click={() => feedGibbon(gibbon)}>
    Feed Gibbon
  </button>
{/if}
```

Actions

Cause side-effects / Invoke commands on the backend



```
function pet(monkey: Monkey, client: ApiClient) {  
  client.post(`/monkey/${monkey.id}/pet`);  
}
```

Connect to your UI

```
import { pet as upstreamPet } from '../core/gibbon';
import { canPet as upstreamCanPet } from '../logic/gibbon';
import { ability, action } from 'framework';

const canPet = ability((monkey: Monkey, { services }) => {
  upstreamCanPet(monkey, services.user);
});

const pet = action((monkey: Monkey, { services }) => {
  upstreamPet(monkey, services.api);
});

<template>
  {{#if (canPet @animal)}}
    <button {{on "click" (fn pet @animal)}}>
      Pet {{@animal.name}}
    </button>
  {{/if}}
</template>
```

UI Code is reduced to its essentials

Connecting UI elements with your existing business logic

Deterministic Behavior and State Management

```
import { createMachine } from 'xstate';
import { canPet, pet } from './gibbon';

const monkeyMood = createMachine({
  id: 'toggle',
  initial: 'neutral',
  states: {
    neutral: { on: {
      PET: {
        target: 'happy',
        cond: 'canPet',
        actions: ['pet']
      }
    }
  },
  happy: { on: { STOP_PET: 'neutral' } }
}
});

actions: {
  pet: (context, event) => {
    pet(context.monkey, context.api);
  },
  guards: {
    canPet: (context, event) => {
      return canPet(context.monkey);
    }
  }
});
```

Example:

Using a Statechart to control the mood of monkey

Deterministic behavior

Use existing business logic

Pure Core

- Extract core into plain JS/TS
 - Aggregates / Entities / Value Objects
 - Actions
 - Abilities & Questions
- Framework agnostic
- Control your app through **core**
- High and excellent test coverage
- Use folders as **signposts**

Use folders as signposts

Three examples for `core/`

```

  ✓ monkey house
  ✓ core
  ✓ aggregates
    TS gibbon.ts
  ✓ logic
    TS gibbon-mood.ts
    TS gibbon.ts

```

```

  ✓ monkey house
  ✓ core
  ✓ gibbon
    TS actions.ts
    TS aggregate.ts
    TS mood.ts
    TS queries.ts

```

```

  ✓ monkey house
  ✓ core
  ✓ aggregates
    > entities
    > value-objects
    TS gibbon.ts
  ✓ logic
  ✓ machines
    TS gibbon-mood.ts
    TS gibbon.ts

```

Files + Folder give a *home*
Choose based on your needs

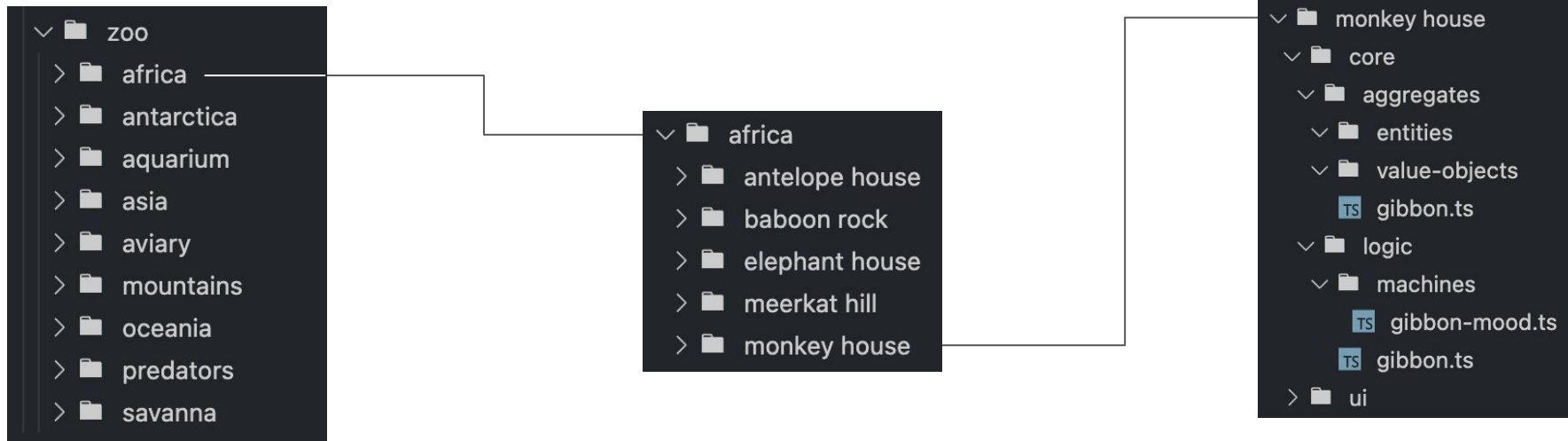
Code

Result

- ★ Core layer in plain JS/TS
- ★ Unit testing for your business logic
- ★ Connect them to your reactivity system
- ★ Connect core to UI, machines and other parts
- ★ Use CQS design pattern
- ★ Use folders as signposts

Summary

- ★ Use available methodologies (EventStorming, OOUX, Atomic Design)
- ★ Extract and apply the essence
- ★ **Establish your own language bound to your product**
- ★ Let the **code speak your product**



Thank you

Thomas **gossi** Gossmann

