Game State

Presented by Haashim-Ali Hussain

Imperial College London Game Development

What?

State is Data

that describes the game at a fixed moment in time

```lua
-- initialise variable to 0;
local i = 0

-- iterate through the values 1 to 10
while i < 10 do

  -- mutate the variable i
 i = i + 1

  -- mutate the output
 print( tostring( i ) .. "/10" )
end
```

```
class PersonWhoDoesntLikeTheirAge {
  // Creating internal state
  // This is mutable but is never mutated
  constructor (private age : number) {}

  // Externalising state
  getAge() {
    // External state is completely distinct
    return 21;
  }
}
```

```c
int global_state = 0;

void mutate_global_state() {
  global_state = global_state + 1;
}

void state_monitor() {
    while (true) {
        printf("global_state = %d \n", global_state);
        sleep(1);
    }
}

void main () {
    // spawn the monitor thread
    pthread_t monitor_thread;
    pthread_create(&monitor_thread, NULL, (void *) state_monitor, NULL);
    while (true) {
        // spawn a thread to mutate the global state asynchronously
        pthread_t thread;
        pthread_create(&thread, NULL, mutate_global_state, NULL);
    }
}
```

Stateful Reflection

So Many Types…

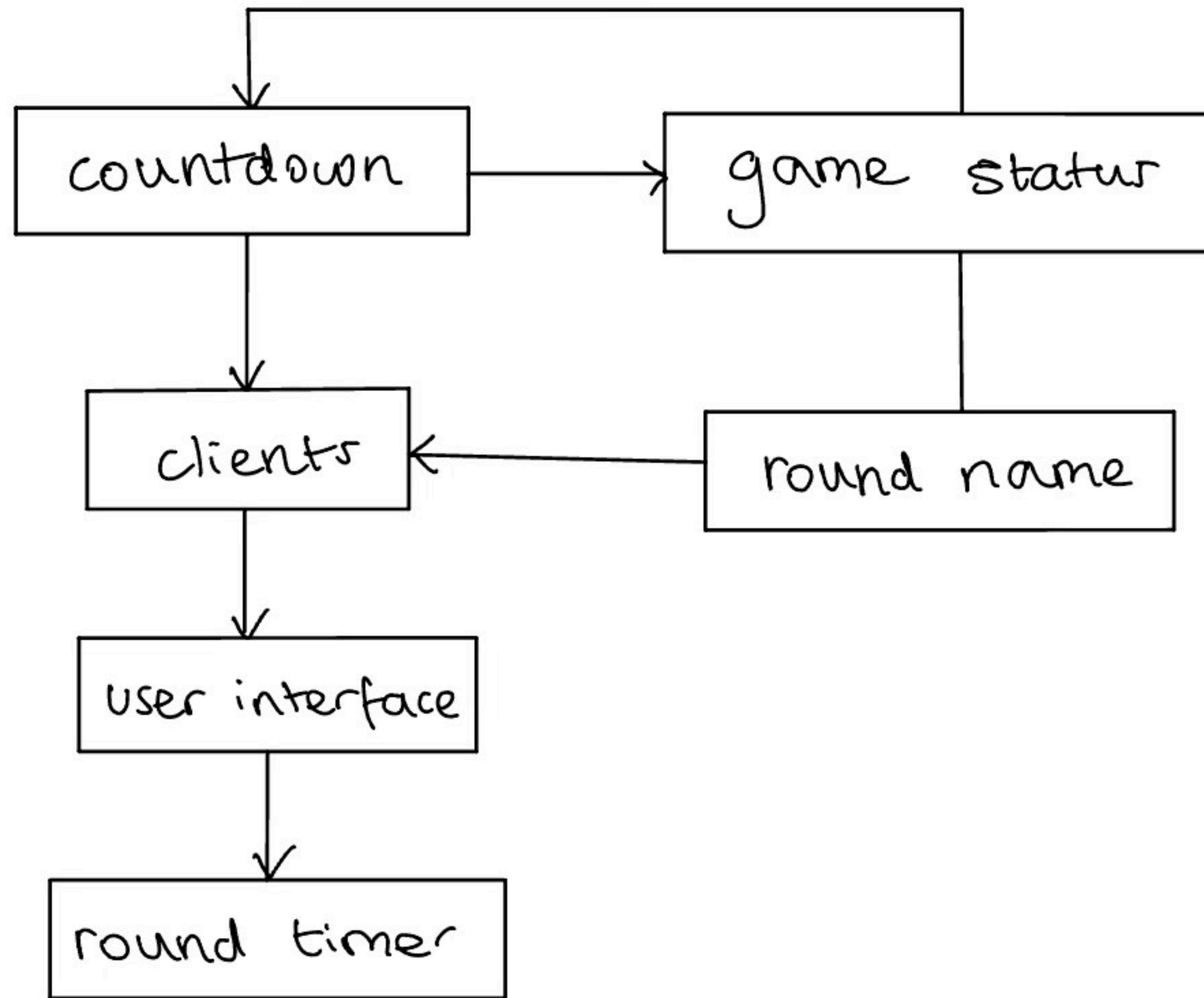
Let's explain

- Internal State
- Mutable State
- Immutable State
- Shared/Global State
- We are ignoring Networking for now

Game State is Confusing

Let’s disambiguate

- Game State can refer to many things.
- The entire state of the game at any point in time is one interpretation.
- We strive for a minimal game state from which all other substates can be inferred.
- This can be thought of as a reduction of the entire Game State into a core state machine.
- And all dependent states on this core can be thought of as effects.
- Thus simplifying Game State state into this idea of a “core” state.


countdown
game status
clients
round name
user interface
round timer

Why is Game State so Complex?

A brief dive into the intricacies.

- Networking adds a layer of inconsistency to Game State.
- Not all Game State needs to be:
  - Replicated
  - Consistent
  - Public
  - Server-Authoritative
  - Static

Paradigm Time

Why are Paradigms Important?

Like fr?

- Paradigms shape your code
  - This affects both readability and extensibility
- They influence the level of coupling in your codebase
  - This has knock-on effects on maintainability
- They affect performance of code
  - And not just as a micro-optimisation

Which Paradigms are Good Tools?

And which aren't?

- Object Oriented Programming is good at protecting its Internal State.
  - But large inheritance trees lead to tightly coupled code.
  - Internal State has the tendency to compound and become hard to manage.
  - Extensibility is hard
- Data Oriented Programming is ideal
  - Entity Component Systems bypass the above problems architecturally
  - And allow for optimisations abusing cache-locality

Reactive Programming

Is great for State

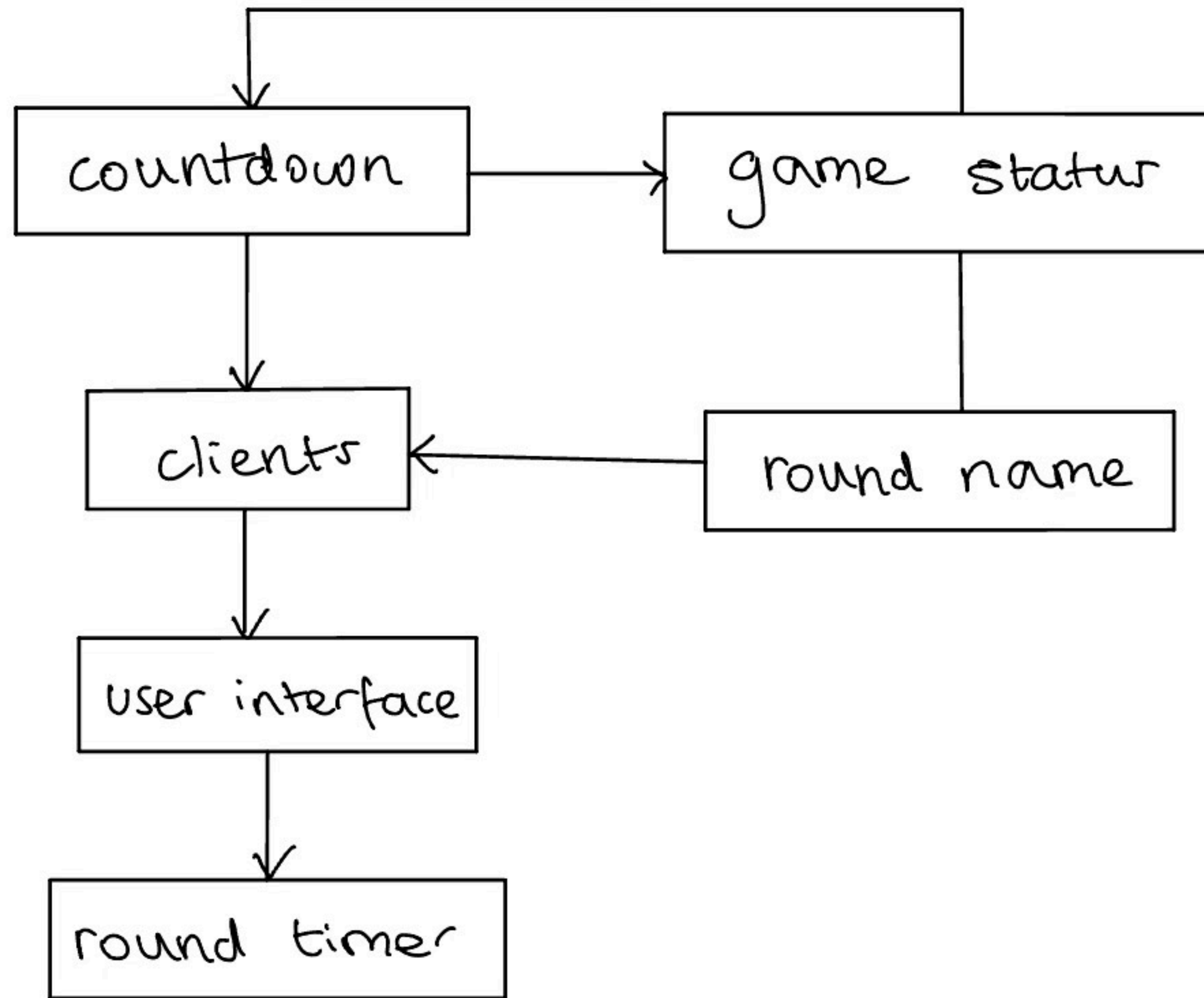
- Reactive programming is a subset of declarative programming
  - Used in UI frameworks (React is not pure Reactive though!)
  - You have encountered another subset of declarative before if you've programmed functionally (Haskell?)
  - Ties into the game state machine discussed earlier
- Lets you declare your state
  - And it's interactions

Reactive Programming

Is great for State

- It allows for asynchronous programming
  - You have met this paradigm before if you’ve used the Promise pattern
  - Asynchronous code is great for concurrency

```typescript
new Promise<number>((resolve, reject) => {
  // Do some asynchronous work
  // I.e. fetching data
  let data: number = 10;
  resolve(data);
})
  .then((data: number) => {
    console.log(data);
    return data.toString();
  })
  .then((data: string) => {
    console.log(data);
    return data.length;
  })
  .catch((error: any) => {
    console.log(error);
  });
```


countdown
game status
clients
round name
user interface
round timer

Reactive Programming

Isn’t all good!

- You have to change the way you debug
  - It does not get easier
  - You now have debug a state graph… somehow
- It’s a whole Paradigm Shift
  - It is sometimes unintuitive
  - There is a big learning curve

Implementation Station

Observable State Tree

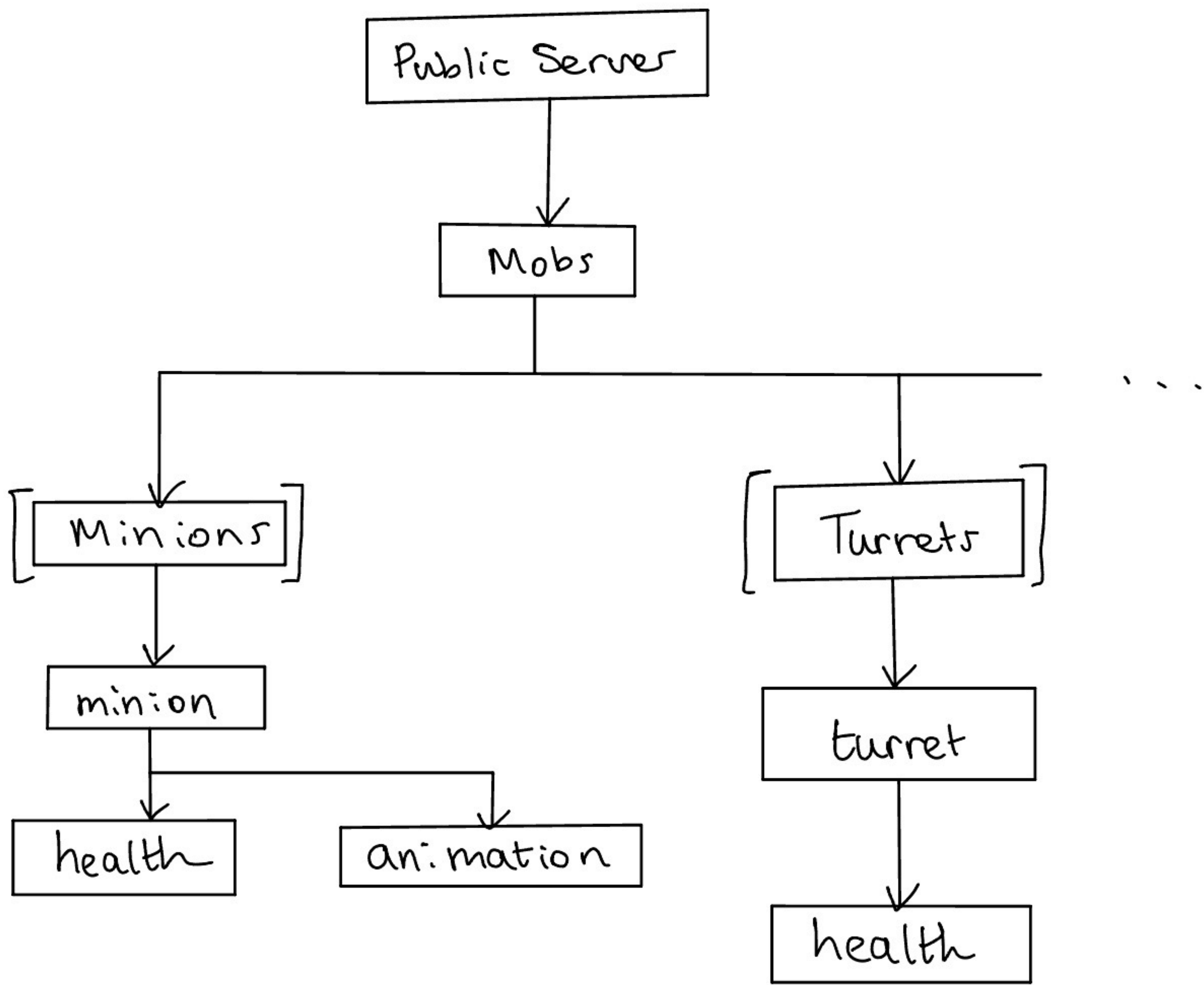
Simplifies the State Graph

- Observer Pattern
  - Nodes represent an Observable State
- Children represent Logical Substates
  - Changes are propagated up the Ancestor Chain
  - Intuitive design pattern
- Very much possible to have a directed Observable State Graph instead.
  - Introduces cyclic state dependencies

Replication Situation

Network Communication Implementation

- Replication Baskets
  - ClientPrivate, ClientPublic, ServerPublic, ServerPrivate
- Data Ownership
- Client/Server Authority
- Client/Server Read/Write Access
- Consistent/Inconsistent State



Public Server
Mobs
Minions
minion
health
animation
Turrets
turret
health
. . .

```
const STATES = ROAST.CreateDefinitions({
  Public: Nodes.PublicServer({
    Mobs: Nodes.Branch({
      Minions: Nodes.Vine((mobData: MobData) => {
        return {
          Health: Nodes.Leaf(100),
          MobID: Nodes.Leaf(mobData),
          Animation: Nodes.Leaf(AnimationState.IDLE),
        };
      ),
      Turrets: Nodes.Vine((turret: TurretSize) => {
        return {
          Health: Nodes.Leaf(
            turret === TurretSize.SMALL
            ? 500
            : turret === TurretSize.LARGE
            ? 1000
            : 2500,
          ),
        };
      }),
      Jungle: Nodes.Vine((mobData: MobData) => {
        return {
          Health: Nodes.Leaf<number>(),
        };
      }),
    }),
  }),
  Client: Nodes.PublicClient((plr) => {
    return {
      Health: Nodes.Leaf<number>(),
    };
  }),
  Server: Nodes.PrivateServer({}),
  Private: Nodes.PrivateClient({}),
});
```

Thanks for listening!