



TS

ENTERPRISE PATTERNS

Course Introduction

We will "hang out," and I am going to show you the **patterns** and, more importantly, the **mental models** I use to program enterprise applications.



Agenda

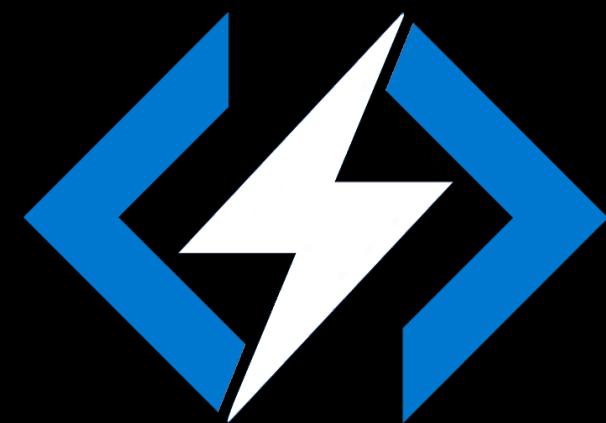
- Managing Complexity
- Local Complexity
- The Axis of Evil™
- Feature Complexity
- The Four Elements of Programming
- Application Complexity
- The Fifth Element of Programming
- Distributed Complexity



<https://github.com/onehungrymind/fem-enterprise-patterns>



<https://stackblitz.com/edit/micro-refactor-example>



<https://stackblitz.com/edit/angular-rxjs-quickstart>

<https://stackblitz.com/edit/angular-rxjs-quickstart-challenges>

The online code editor for web x +

StackBlitz Docs GitHub Discord Blog Sign in

Now in technology preview! [Learn more.](#)

The online IDE for web applications. Powered by Visual Studio Code.

Create, share & embed live projects – in just one click.

START A NEW PROJECT

 Angular
TypeScript

 React
Javascript

 Ionic
TypeScript

 TypeScript
Blank project

 RxJS
TypeScript

 Svelte
Javascript

A photograph of two men looking at a computer monitor. On the left, a young man with short brown hair, wearing a white and blue plaid shirt, is leaning forward with his hands on the desk, looking intently at the screen. On the right, an older man with grey hair, wearing a light-colored button-down shirt, is also looking at the screen. The computer monitor is positioned in the center of the frame, displaying some text or graphics that are not clearly legible.

TEACH ME TO PROGRAM

A man with a white beard and mustache, wearing a light-colored button-down shirt, is painting a vertical wooden fence. He is holding a paintbrush in his right hand and a paint bucket in his left. A younger man with dark hair, wearing a blue and white polo shirt and a white headband, stands behind him, watching. They are outdoors, with green trees and bushes in the background.

LET'S START HERE



AND DO THIS FOR A WHILE



THIS IS GOING TO GET WEIRD

A middle-aged man with a grey beard and mustache is looking upwards and to his left with a neutral expression. He is wearing a light-colored, short-sleeved button-down shirt. In his right hand, he holds a small, thin object, possibly a cigarette or a piece of food, between his fingers. His left hand is partially visible, showing a ring on his ring finger. The background is slightly blurred, showing what appears to be an indoor setting with warm lighting.

REALLY WEIRD!

BUT WE DO THIS...



...FOR MOMENTS LIKE THIS



Managing Complexity

The biggest problem in the development and maintenance of large-scale software systems is complexity – large systems are hard to understand.

Out of the Tarpit - Ben Moseley Peter Marks

We believe that the major contributor to this complexity in many systems is the handling of state and the burden that this adds when trying to analyse and reason about the system. Other closely related contributors are code volume, and explicit concern with the flow of control through the system.

Out of the Tarpit - Ben Moseley Peter Marks

Complexity
and purgatory

```
it('should recalculate total cost', () => {
  const mockCourse = {
    id: '1',
    title: 'Mock Course',
    price: 8.00
  };
  component.mode = 'update';
  const total = 28;

  const r1 = component.updateCoursesAndRecalculateCost(mockCourse);
  expect(r1).toBe(total);

  const r2 = component.updateCoursesAndRecalculateCost(mockCourse);
  expect(r2).toBe(total);

  const r3 = component.updateCoursesAndRecalculateCost(mockCourse);
  expect(r3).toBe(total);
});
```

State Management

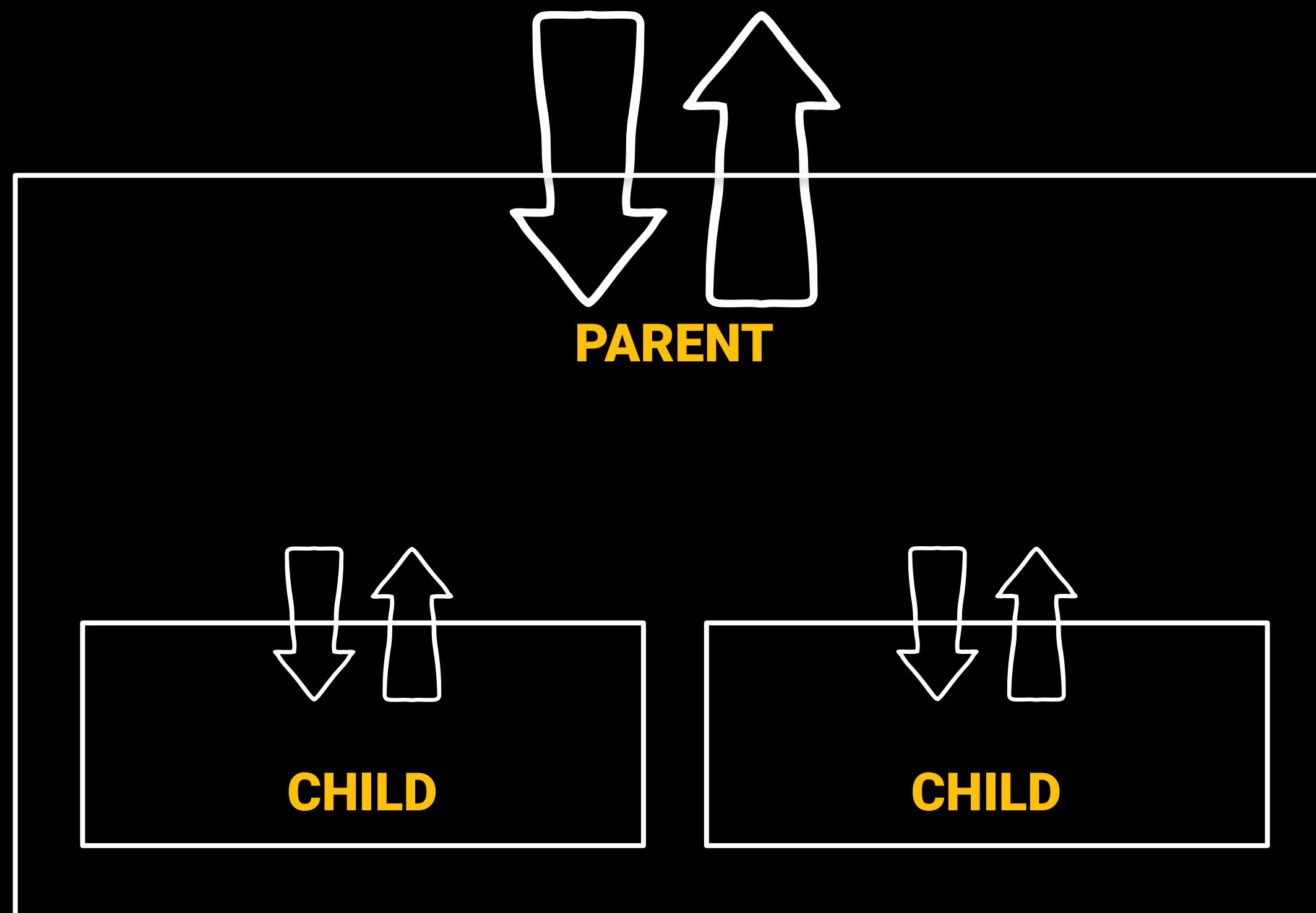
```
it('should recalculate total cost', () => {
  const mockCourse = { id: '1', title: 'Mock Course', price: 8.00 };
  component.mode = 'update';
  const total = 28;

  const r1 = component.updateCoursesAndRecalculateCost(mockCourse);
  expect(r1).toBe(total);

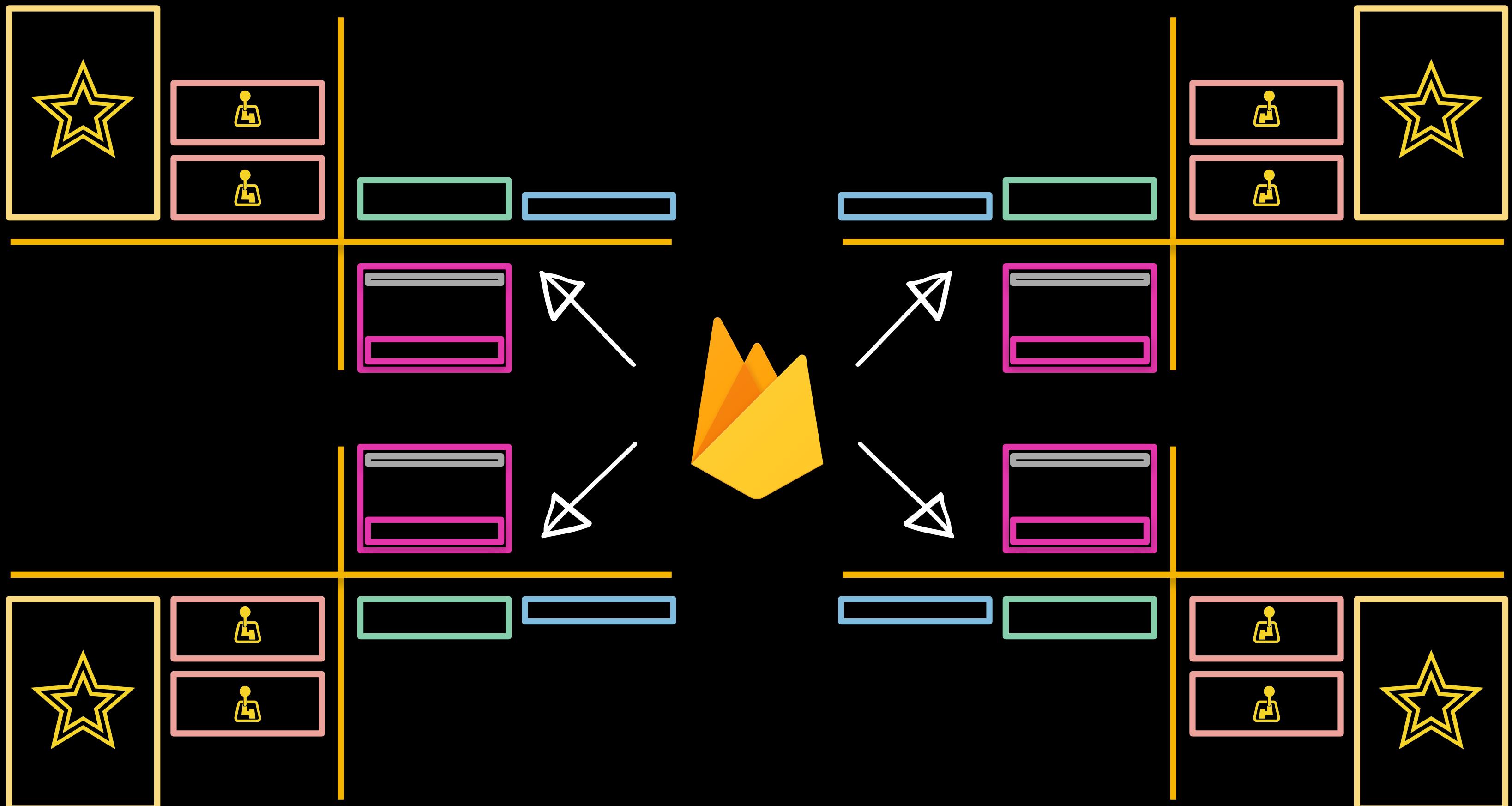
  component.mode = 'create';
  const r2 = component.updateCoursesAndRecalculateCost(mockCourse);
  expect(r2).toBe(total); // WRONG!

  component.mode = 'delete';
  const r3 = component.updateCoursesAndRecalculateCost(mockCourse);
  expect(r3).toBe(total); // WRONG!
});
```

State Management



Control Flow



Control Flow

```
updateCoursesAndRecalculateCost(course: Course, mode: string) {
  switch (mode) {
    case 'create':
      const newCourse = Object.assign({}, course, { id: uuidv4() });
      this.courses = [...this.courses, newCourse];
      break;
    case 'update':
      this.courses = this.courses.map(_course => course.id === _course.id
        ? Object.assign({}, course) : _course);
      break;
    case 'delete':
      this.courses = this.courses.filter(_course => course.id !== _course.id);
      break;
    default:
      break;
  }

  return this.courses.reduce((acc, curr) => acc + curr.price, 0);
}
```

Code Volume

Shared Mutable State

```
class Inventory {
  ledger = { total: 1200 };
}

class ItemsComponent {
  ledger: any;
  constructor(private inventory:Inventory) {
    this.ledger = inventory.ledger;
  }
  add(x) { this.ledger.total += x; }
}

class WidgetsComponent {
  ledger: any;
  constructor(private inventory:Inventory) {
    this.ledger = inventory.ledger;
  }
  add(x) { this.ledger.total += x; }
}
```

```
class Inventory {
  ledger = { total: 1200 };
}

class ItemsComponent {
  ledger: any;
  constructor(private inventory:Inventory) {
    this.ledger = inventory.ledger;
  }
  add(x) { this.ledger.total += x; }
}

class WidgetsComponent {
  ledger: any;
  constructor(private inventory:Inventory) {
    this.ledger = inventory.ledger;
  }
  add(x) { this.ledger.total += x; }
}
```

Local Complexity

Can I know the result of this
code at all times?

Can I reuse this code?

Can I test this code?

The Axis of Evil™

It is *impossible* to write
good tests for bad
code

If you are committed
to writing tests...

Then you must
commit to writing
testable code.

The Axis of Evil™

```
price;
mode;
widgets: Widget[];

recalculateTotal(widget: Widget) {
  switch (this.mode) {
    case 'create':
      const newWidget = Object.assign({}, widget, {id: uuidv4()});
      this.widgets = [...this.widgets, newWidget];
      break;
    case 'update':
      this.widgets = this.widgets.map(wdgt =>
        (widget.id === wdgt.id) ? Object.assign({}, widget) : wdgt);
      break;
    case 'delete':
      this.widgets = this.widgets.filter(wdgt => widget.id !== wdgt.id);
      break;
    default:
      break;
  }

  this.price = this.widgets.reduce((acc, curr) => acc + curr.price, 0);
}
```

```
const testService = new RefactorService();
const testWidget = { id: 100, name: '', price: 100, description: ''};
const testWidgets = [{ id: 100, name: '', price: 200, description: ''}];
testService.widgets = testWidgets;

testService.mode = 'create';
testService.recalculateTotal(testWidget);

testService.mode = 'update';
testService.recalculateTotal(testWidget);

testService.mode = 'delete';
testService.recalculateTotal(testWidget);
```

```
const testService = new RefactorService();
const testWidget = { id: 100, name: '', price: 100, description: ''};
const testWidgets = [{ id: 100, name: '', price: 200, description: ''}];
testService.widgets = testWidgets;

testService.mode = 'create';
testService.recalculateTotal(testWidget);

testService.mode = 'update';
testService.recalculateTotal(testWidget);

testService.mode = 'delete';
testService.recalculateTotal(testWidget);
```

Hidden State

Violating the SRP

Nested Logic

```
price;  
mode;  
widgets: Widget[];  
  
recalculateTotal(widget: Widget) {  
  
    switch (this.mode) {  
        case 'create':  
            const newWidget = object.assign({}, widget, {id: uuidv4()});  
            this.widgets = [...this.widgets, newWidget];  
            break;  
        case 'update':  
            this.widgets = this.widgets.map(wdgt =>  
                (widget.id === wdgt.id) ? object.assign({}, widget) : wdgt);  
            break;  
        case 'delete':  
            this.widgets = this.widgets.filter(wdgt => widget.id !== wdgt.id);  
            break;  
        default:  
            break;  
    }  
  
    this.price = this.widgets.reduce((acc, curr) => acc + curr.price, 0);  
}
```

Hidden State
Violating the SRP
Nested Logic

```
price;
mode;
widgets: Widget[];

recalculateTotal(widget: Widget) {

  switch (this.mode) {
    case 'create':
      const newWidget = object.assign({}, widget, {id: uuidv4()});
      this.widgets = [...this.widgets, newWidget];
      break;
    case 'update':
      this.widgets = this.widgets.map(wdgt =>
        (widget.id === wdgt.id) ? object.assign({}, widget) : wdgt);
      break;
    case 'delete':
      this.widgets = this.widgets.filter(wdgt => widget.id !== wdgt.id);
      break;
    default:
      break;
  }

  this.price = this.widgets.reduce((acc, curr) => acc + curr.price, 0);
}
```

Hidden State
Violating the SRP
Nested Logic

```
price;
mode;
widgets: Widget[];

recalculateTotal(widget: Widget) {

  switch (this.mode) {
    case 'create':
      const newWidget = object.assign({}, widget, {id: uuidv4()});
      this.widgets = [...this.widgets, newWidget];
      break;
    case 'update':
      this.widgets = this.widgets.map(wdgt =>
        (widget.id === wdgt.id) ? object.assign({}, widget) : wdgt);
      break;
    case 'delete':
      this.widgets = this.widgets.filter(wdgt => widget.id !== wdgt.id);
      break;
    default:
      break;
  }

  this.price = this.widgets.reduce((acc, curr) => acc + curr.price, 0);
}
```

Ironically, these are the
easiest problems to fix.

Dependency Injection

Extract to Method

```
courses: Course[];  
totalCost;  
mode;  
  
updateCoursesAndRecalculateCost(course: Course) {  
    switch (this.mode) {  
        case 'create':  
            const newCourse = Object.assign({}, course, { id: uuidv4() });  
            this.courses = [...this.courses, newCourse];  
            break;  
        case 'update':  
            this.courses = this.courses.map(_course => course.id === _course.id  
                ? Object.assign({}, course) : _course);  
            break;  
        case 'delete':  
            this.courses = this.courses.filter(_course => course.id !== _course.id);  
            break;  
        default:  
            break;  
    }  
  
    return this.courses.reduce((acc, curr) => acc + curr.price, 0);  
}
```

```
it('should recalculate total cost', () => {
  const mockCourse = { id: '1', title: 'Mock Course', price: 8.00 };
  component.mode = 'update';
  const total = 28;

  const r1 = component.updateCoursesAndRecalculateCost(mockCourse);
  expect(r1).toBe(total);

  const r2 = component.updateCoursesAndRecalculateCost(mockCourse);
  expect(r2).toBe(total);

  const r3 = component.updateCoursesAndRecalculateCost(mockCourse);
  expect(r3).toBe(total);
});
```

```
it('should recalculate total cost', () => {
  const mockCourse = { id: '1', title: 'Mock Course', price: 8.00 };
  component.mode = 'update';
  const total = 28;

  const r1 = component.updateCoursesAndRecalculateCost(mockCourse);
  expect(r1).toBe(total);

  const r2 = component.updateCoursesAndRecalculateCost(mockCourse);
  expect(r2).toBe(total);

  const r3 = component.updateCoursesAndRecalculateCost(mockCourse);
  expect(r3).toBe(total);
});
```

```
it('should recalculate total cost', () => {
  const mockCourse = { id: '1', title: 'Mock Course', price: 8.00 };
  component.mode = 'update';
  const total = 28;

  const r1 = component.updateCoursesAndRecalculateCost(mockCourse);
  expect(r1).toBe(total);

  component.mode = 'create';
  const r2 = component.updateCoursesAndRecalculateCost(mockCourse);
  expect(r2).toBe(total); // WRONG!

  component.mode = 'delete';
  const r3 = component.updateCoursesAndRecalculateCost(mockCourse);
  expect(r3).toBe(total); // WRONG!
});
```

```
courses: Course[];  
totalCost;  
mode;  
  
updateCoursesAndRecalculateCost(course: Course) {  
    switch (this.mode) {  
        case 'create':  
            const newCourse = Object.assign({}, course, { id: uuidv4() });  
            this.courses = [...this.courses, newCourse];  
            break;  
        case 'update':  
            this.courses = this.courses.map(_course => course.id === _course.id  
                ? Object.assign({}, course) : _course);  
            break;  
        case 'delete':  
            this.courses = this.courses.filter(_course => course.id !== _course.id);  
            break;  
        default:  
            break;  
    }  
  
    return this.courses.reduce((acc, curr) => acc + curr.price, 0);  
}
```

```
updateCoursesAndRecalculateCost(course: Course, mode: string) {
  switch (mode) {
    case 'create':
      const newCourse = Object.assign({}, course, { id: uuidv4() });
      this.courses = [...this.courses, newCourse];
      break;
    case 'update':
      this.courses = this.courses.map(_course => course.id === _course.id
        ? Object.assign({}, course) : _course);
      break;
    case 'delete':
      this.courses = this.courses.filter(_course => course.id !== _course.id);
      break;
    default:
      break;
  }

  return this.courses.reduce((acc, curr) => acc + curr.price, 0);
}
```

Dependency Injection

Extract to Method

```
courses: Course[];  
totalCost;  
mode;  
  
updateCoursesAndRecalculateCost(course: Course, mode: string) {  
  switch (mode) {  
    case 'create':  
      const newCourse = Object.assign({}, course, { id: uuidv4() });  
      this.courses = [...this.courses, newCourse];  
      break;  
    case 'update':  
      this.courses = this.courses.map(_course => course.id === _course.id  
        ? Object.assign({}, course) : _course);  
      break;  
    case 'delete':  
      this.courses = this.courses.filter(_course => course.id !== _course.id);  
      break;  
    default:  
      break;  
  }  
  
  return this.courses.reduce((acc, curr) => acc + curr.price, 0);  
}
```

```
updateCoursesAndRecalculateCost(course: Course, mode: string) {
  switch (mode) {
    case 'create':
      this.courses = this.createCourse(this.courses, course);
      break;
    case 'update':
      this.courses = this.updateCourse(this.courses, course);
      break;
    case 'delete':
      this.courses = this.deleteCourse(this.courses, course);
      break;
    default:
      break;
  }
  return this.courses.reduce((acc, curr) => acc + curr.price, 0);
}

createCourse(courses, course) {
  return [...courses, Object.assign({}, course, { id: uuidv4() })];
}

updateCourse(courses, course) {
  return courses.map(_course => course.id === _course.id ? Object.assign({}, course) : _course);
}

deleteCourse(courses, course) {
  return courses.filter(_course => course.id !== _course.id);
}
```

```
updateCoursesAndRecalculateCost(course: Course, mode: string) {
    switch (mode) {
        case 'create':
            this.courses = this.createCourse(this.courses, course);
            break;
        case 'update':
            this.courses = this.updateCourse(this.courses, course);
            break;
        case 'delete':
            this.courses = this.deleteCourse(this.courses, course);
            break;
        default:
            break;
    }
    return this.courses.reduce((acc, curr) => acc + curr.price, 0);
}
```

```
onCourseUpdated(course, mode) {
  this.courses = this.updateCourses(this.courses, course, mode);
  this.totalCost = this.updateTotalCost(this.courses);
}
```

```
updateCourses(courses, course, mode) {
  switch (mode) {
    case 'create':
      return this.createCourse(courses, course);
    case 'update':
      return this.updateCourse(courses, course);
    case 'delete':
      return this.deleteCourse(courses, course);
    default:
      return courses;
  }
}
```

```
updateTotalCost(courses) { return courses.reduce((acc, curr) => acc + curr.price, 0); }
```

```
createCourse(courses, course) { return [...courses, Object.assign({}, course, { id: uuidv4() })]; }
```

```
updateCourse(courses, course) {
  return courses.map(_course => course.id === _course.id
    ? Object.assign({}, course) : _course);
}
```

```
deleteCourse(courses, course) { return courses.filter(_course => course.id !== _course.id); }
```

Fine-grained code vs
Coarse-grained code

```
courses: Course[];  
totalCost;  
mode;  
  
updateCoursesAndRecalculateCost(course: Course) {  
    switch (this.mode) {  
        case 'create':  
            const newCourse = Object.assign({}, course, { id: uuidv4() });  
            this.courses = [...this.courses, newCourse];  
            break;  
        case 'update':  
            this.courses = this.courses.map(_course => course.id === _course.id  
                ? Object.assign({}, course) : _course);  
            break;  
        case 'delete':  
            this.courses = this.courses.filter(_course => course.id !== _course.id);  
            break;  
        default:  
            break;  
    }  
  
    return this.courses.reduce((acc, curr) => acc + curr.price, 0);  
}
```

```
onCourseUpdated(course, mode) {
  this.courses = this.updateCourses(this.courses, course, mode);
  this.totalCost = this.updateTotalCost(this.courses);
}

updateCourses(courses, course, mode) {
  switch (mode) {
    case 'create':
      return this.createCourse(courses, course);
    case 'update':
      return this.updateCourse(courses, course);
    case 'delete':
      return this.deleteCourse(courses, course);
    default:
      return courses;
  }
}

updateTotalCost(courses) { return courses.reduce((acc, curr) => acc + curr.price, 0); }

createCourse(courses, course) { return [...courses, Object.assign({}, course, { id: uuidv4() })]; }

updateCourse(courses, course) {
  return courses.map(_course => course.id === _course.id
    ? Object.assign({}, course) : _course);
}

deleteCourse(courses, course) { return courses.filter(_course => course.id !== _course.id); }
```

```
export class CoursesFacade {
    loaded$ = this.store.pipe(select(coursesSelectors.getCoursesLoaded));
    allCourses$ = this.store.pipe(select(coursesSelectors.getAllCourses));
    selectedCourse$ = this.store.pipe(select(coursesSelectors.getSelectedCourse));

    constructor(
        private store: Store<fromCourses.CoursesPartialState>,
        private actions$: ActionsSubject
    ) {}

    loadCourses() {
        this.dispatch(coursesActions.loadCourses());
    }

    createCourse(course: Course) {
        this.dispatch(coursesActions.createCourse({ course }));
    }

    updateCourse(course: Course) {
        this.dispatch(coursesActions.updateCourse({ course }));
    }

    deleteCourse(contentType: ContentType.Course, course: ItemType) {
        this.dispatch(coursesActions.deleteCourse({ contentType, course }));
    }

    dispatch(action: Action) {
        this.store.dispatch(action);
    }
}
```

Refactoring Exercise

- The first step to writing good tests is writing great code
- In the **challenge** branch, open **apps/examples/src/app/examples/01-refactor/refactor.service.ts**
- Refactor the service so that the tests in **apps/examples/src/app/examples/01-refactor/refactor.service.spec.ts** pass
- You can validate this by running **ng test --test-file=refactor.service**

Feature Complexity

Four Elements of Programming

A dramatic, low-key lighting photograph of two men. One man, older with a shaved head, is in profile, looking towards the left. The other man, younger with dark hair, is positioned behind him, holding him from the waist. The background is dark with some blue and orange highlights.

WHEN I LOOKED BACK...

I realized that I was essentially
doing the same four things over
and over.

Describing Things

Performing Actions

Making Decisions

Repeating via Iteration

Data Structures
Performing Actions
Making Decisions
Repeating via Iteration

Data Structures
Functions
Making Decisions
Repeating via Iteration

Data Structures

Functions

Conditionals

Repeating via Iteration

Data Structures

Functions

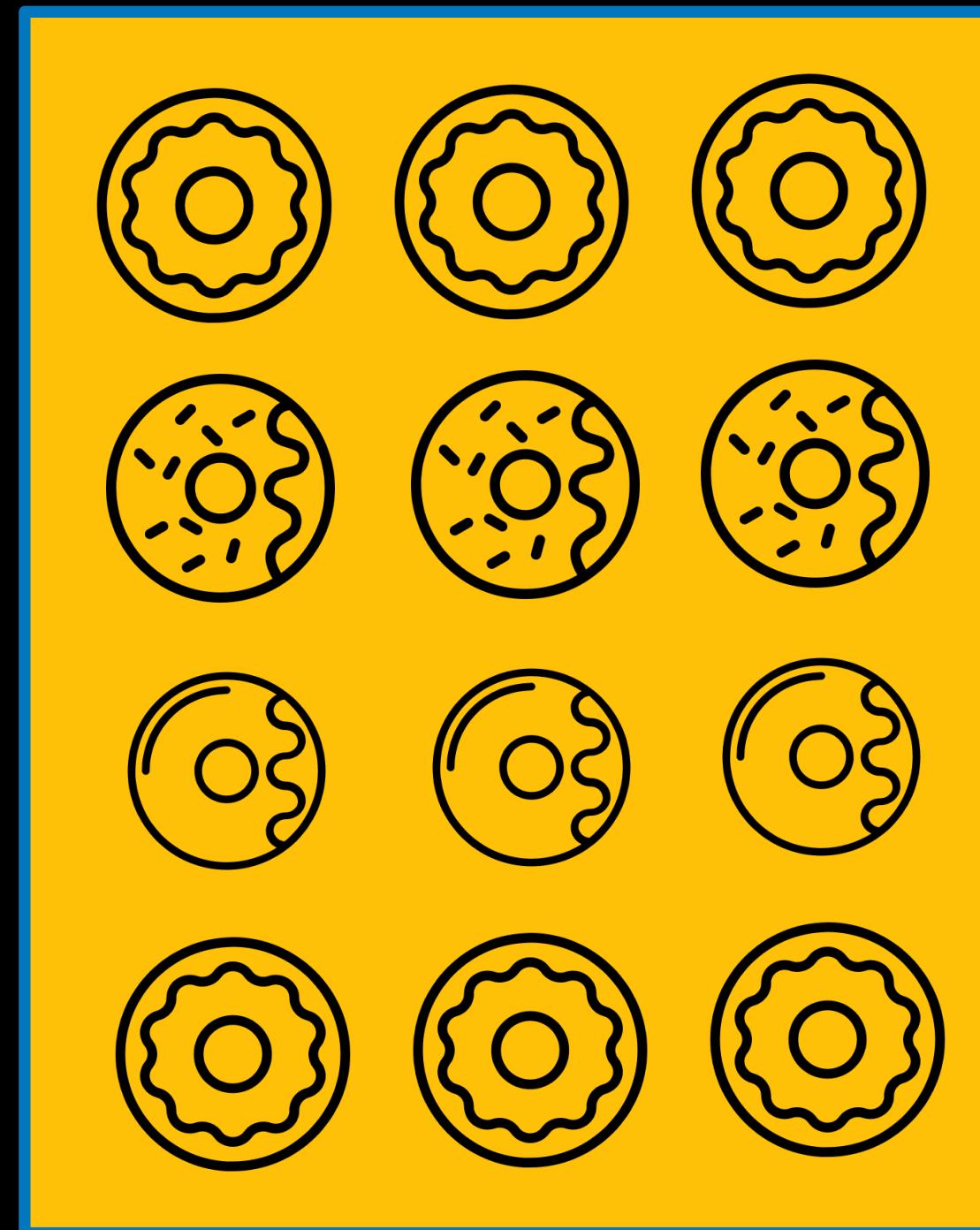
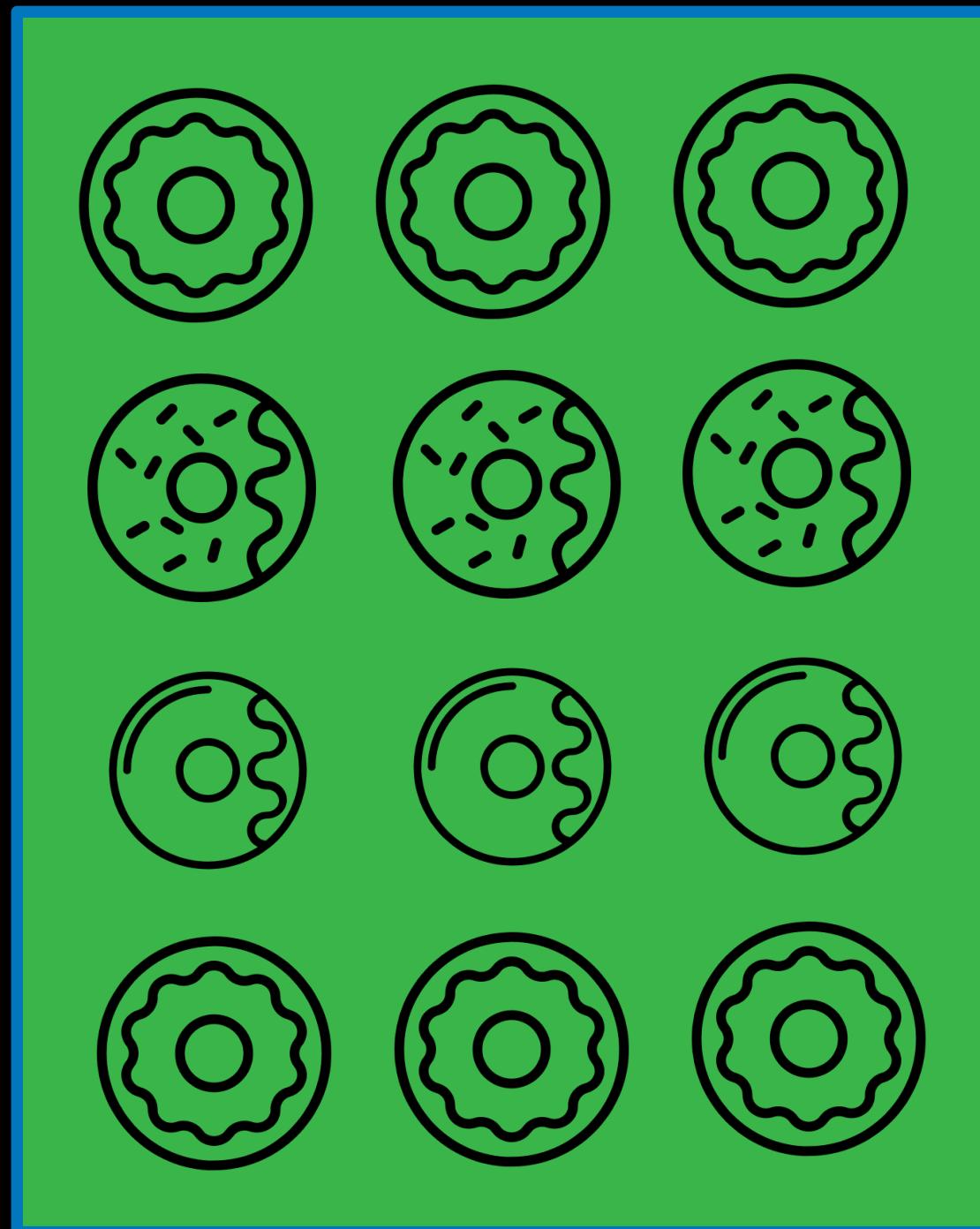
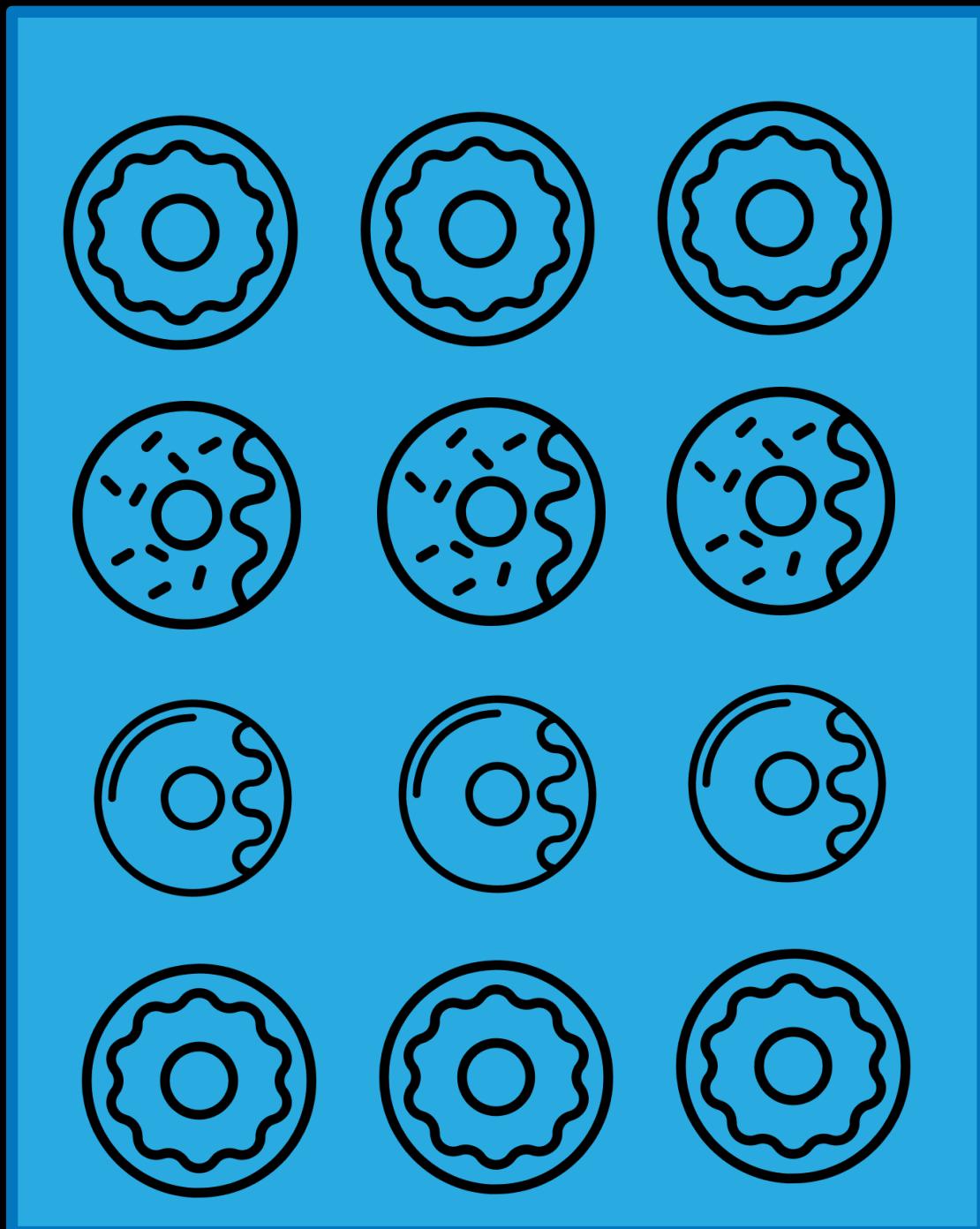
Conditionals

Iterators

I realized that I had been doing
these things since I was a **very**
small child.

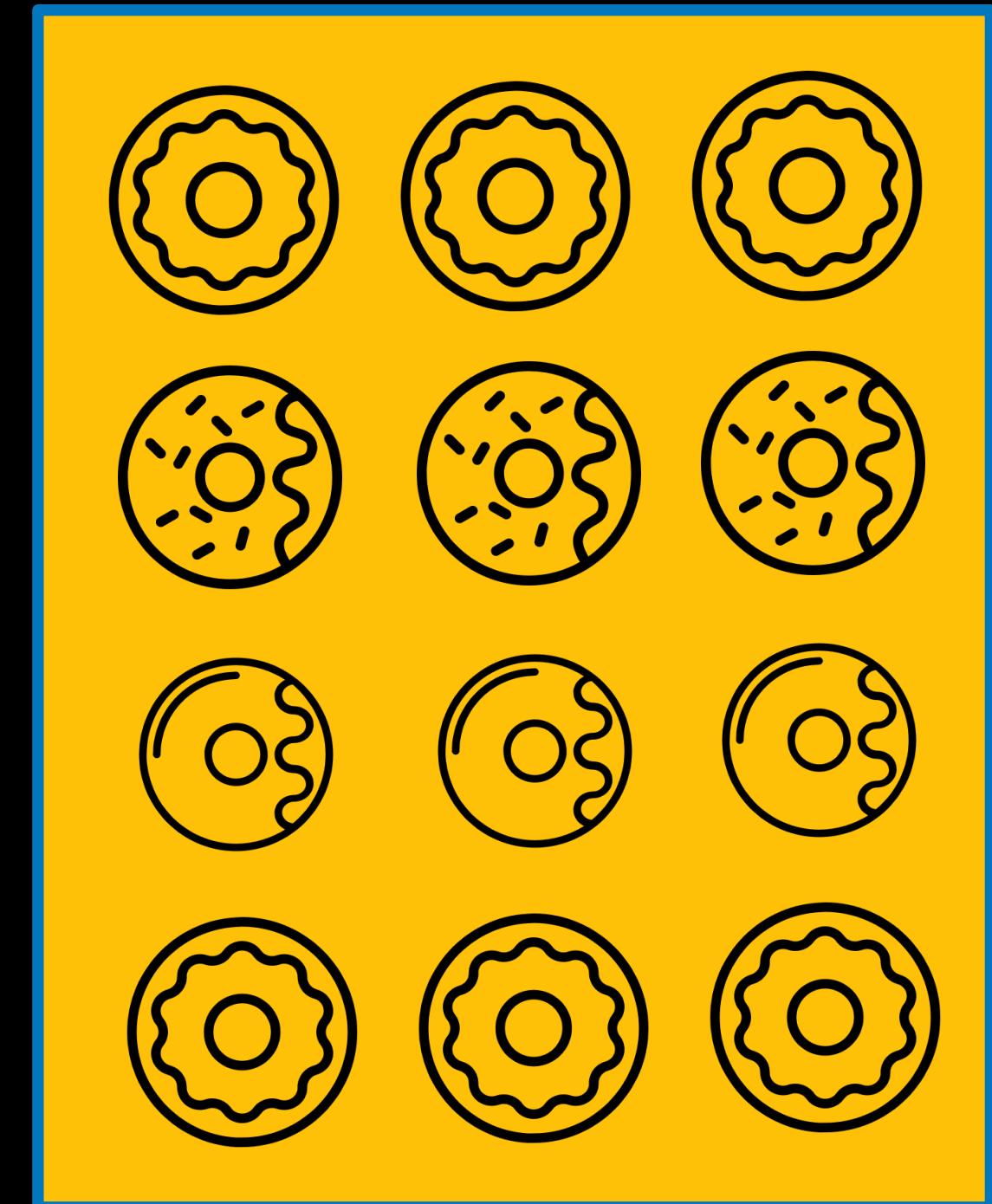
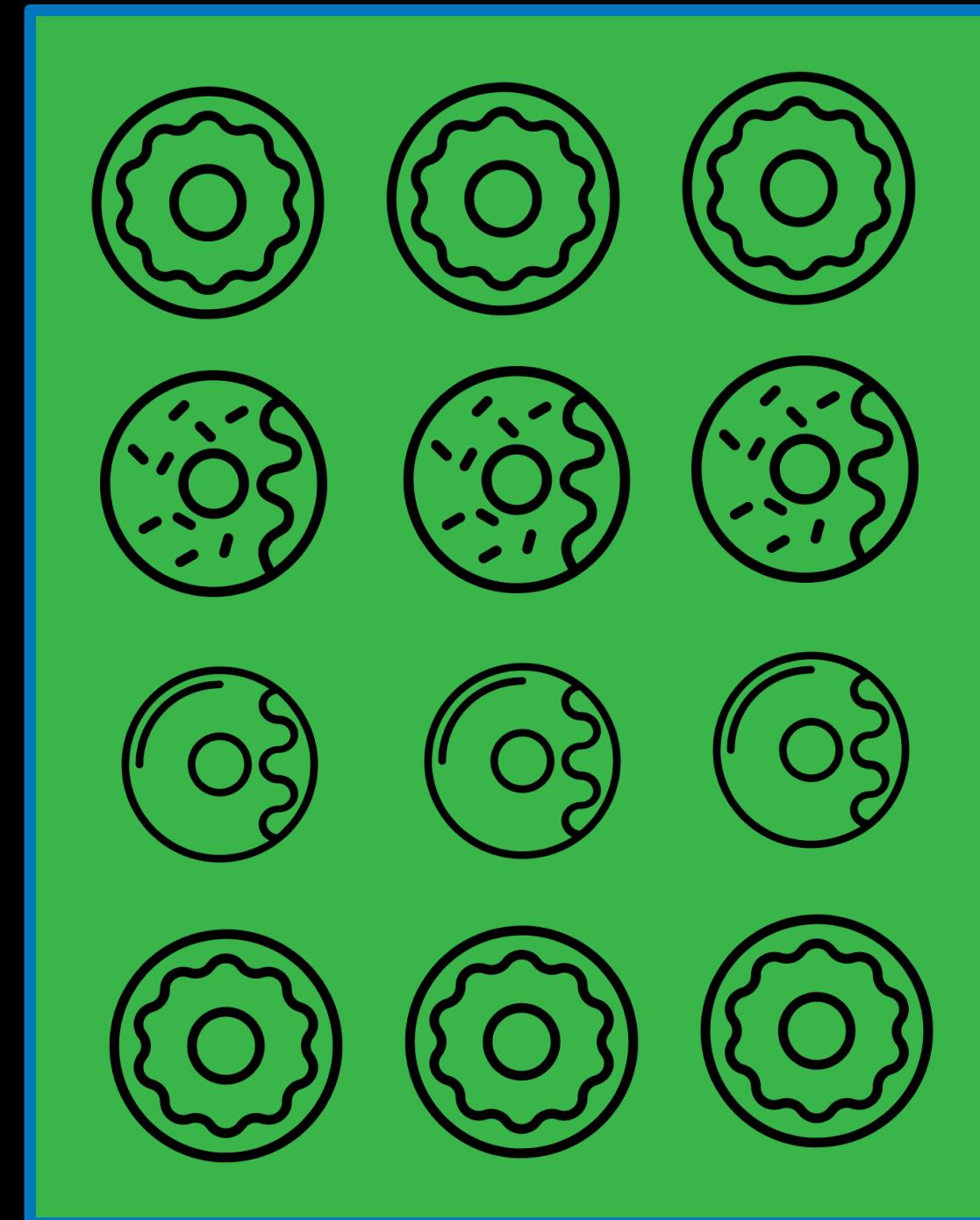
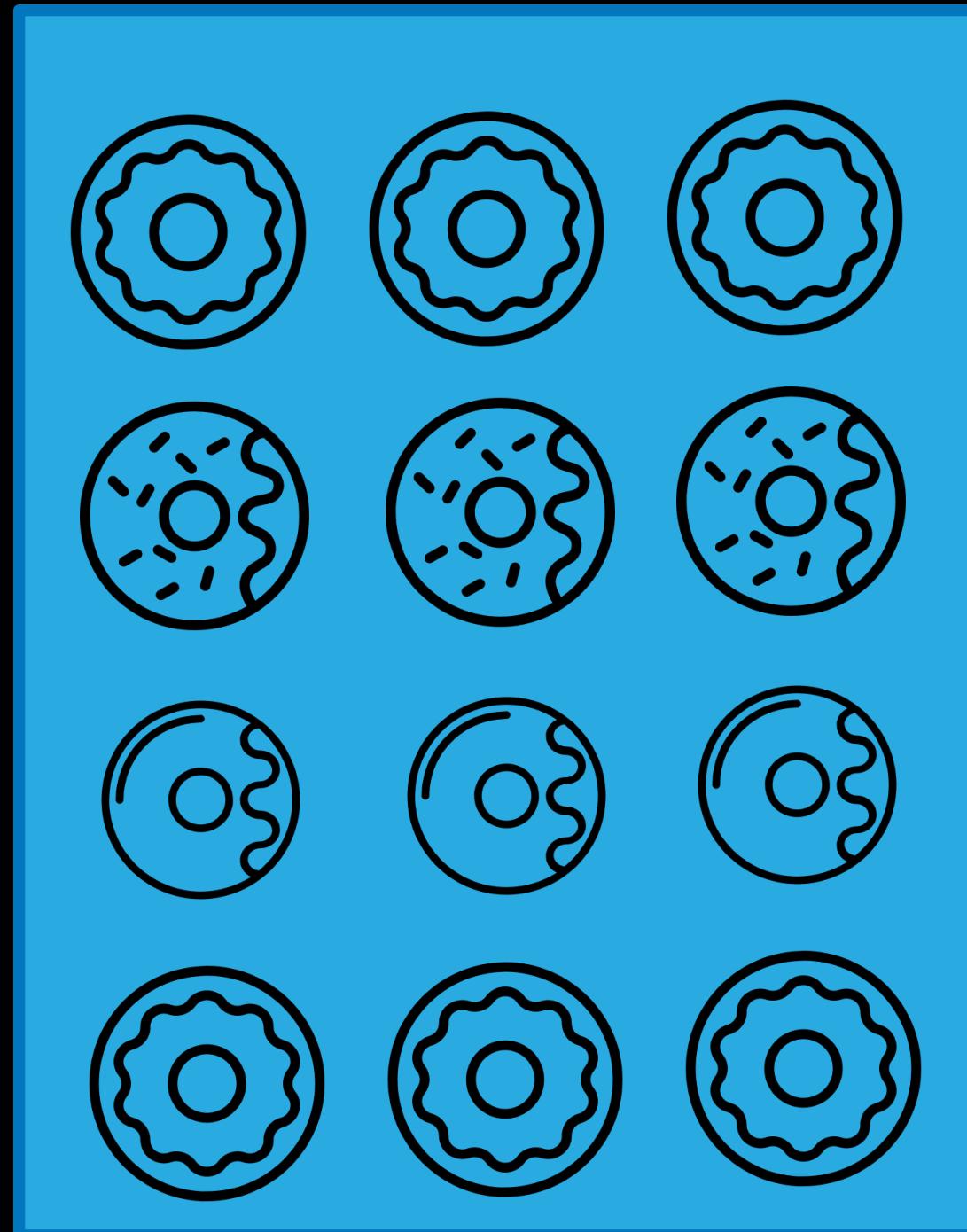
BONUS!

Thought Exercise



Do you see those trays of
donuts?

Do you see the green tray?
On that tray, find all the donuts
that have sprinkles on them.
Bring them back to me.



```
const selectedTray = trays.find(tray => tray.color === 'green');
const sprinkleDonuts = selectedTray.donuts.filter(donut => donut.sprinkles)
child.fetch(sprinkleDonuts);
```

TypeScript

Interlude

What do we mean when we talk
about "plain old JavaScript"?

ECMAScript 5

```
function Shape(id, x, y) {  
    this.id = id;  
    this.setLocation(x, y);  
}  
  
Shape.prototype.setLocation = function(x, y) {  
    this.x = x;  
    this.y = y;  
};  
  
Shape.prototype.getLocation = function() {  
    return {  
        x: this.x,  
        y: this.y  
    };  
};
```

ECMAScript 6*

```
class Shape {  
    constructor(id, x, y) {  
        this.id = id;  
        this.setLocation(x, y);  
    }  
  
    setLocation(x, y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    getLocation() {  
        return {  
            x: this.x,  
            y: this.y  
        };  
    }  
}
```

TypeScript

We are going to use TypeScript
because it is better at
communicating intention

```
class Shape {  
    x: number;  
    y: number;  
  
    constructor(private id: string, x: number, y: number) {  
        this.setLocation(x, y);  
    }  
  
    setLocation(x: number, y: number): void {  
        this.x = x;  
        this.y = y;  
    }  
  
    getLocation(): {  
        x: number;  
        y: number  
    } {  
        return {  
            x: this.x,  
            y: this.y  
        };  
    }  
}
```

```
interface Location {  
    x: number;  
    y: number;  
}  
  
class Shape {  
    location: Location;  
  
    constructor(private id: string, x: number, y: number) {  
        this.setLocation(x, y);  
    }  
  
    setLocation(x: number, y: number): void {  
        this.location.x = x;  
        this.location.y = y;  
    }  
  
    getLocation(): Location {  
        return this.location;  
    }  
}
```

But... it is JUST JavaScript

Objects as Nouns



```
const client = {  
  id: '1',  
  firstName: 'John',  
  lastName: 'Doe',  
  company: 'Acme, Inc'  
}
```

Object

```
const newClient = { id: null, firstName: '', lastName: '', company: '' };
```

Object

```
const clients = [
  {
    id: '1',
    firstName: 'John',
    lastName: 'Doe',
    company: 'Acme, Inc'
  },
  {
    id: '2',
    firstName: 'Jane',
    lastName: 'Smith',
    company: 'Super, Inc'
  }
];
```

Collections

```
interface Client {  
  id?: string;  
  firstName: string;  
  lastName: string;  
  company: string;  
}
```

Interface

```
const clients: Client[] = [
  {
    id: '1',
    firstName: 'John',
    lastName: 'Doe',
    company: 'Acme, Inc'
  },
  {
    id: '2',
    firstName: 'Jane',
    lastName: 'Smith',
    company: 'Super, Inc'
  }
];
```

Strong Collections

```
class VipClient implements Client {  
    firstName;  
    lastName;  
    company;  
  
    constructor(firstName, lastName, company) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.company = company;  
    }  
}
```

Classes

```
const ironMan = new VipClient('Tony', 'Stark', 'Stark Industries');
console.log(ironMan.firstName); // Tony
console.log(ironMan.lastName); // Stark
console.log(ironMan.company); // Stark Industries
```

Concrete Instances



```
interface ClientState {
  clients: Client[];
  currentClient: Client;
}
```

```
const newClient: Client = { id: null, firstName: '', lastName: '', company: '' };

const initialState: ClientState = {
  clients,
  currentClient: newClient
};
```

Challenges

- Create an **object** that represents a client **project**
- Add some appropriate **properties** to your new **project** object
- Create an **interface** to represent your **project** model
- Create an **array** of **project** objects
- Create an **interface** to represent **project state**
- Create an **initialState** object that implements the **ProjectsState** interface

Methods as Verbs



```
const selectClient = function(clientState, client) {  
  return {  
    clients: clientState.clients,  
    currentClient: client  
  };  
};
```

Function

```
const selectClient = (clientState, client) => {
  return {
    clients: clientState.clients,
    currentClient: client
  };
};
```

Fat Arrow Function

```
const selectClient = (clientState, client) => {
  return {
    clients: clientState.clients,
    currentClient: client
  };
};

const spiderMan = {
  id: '1000', firstName: 'Peter', lastName: 'Parker', company: 'Student'
};

// Call the method
selectClient(initialState, spiderMan);

// Store the results
const clientsState = selectClient(initialState, spiderMan);
```

```
class ClientStore {  
    clients: Client[];  
    currentClient: Client;  
  
    load(clients) {  
        this.clients = clients;  
    }  
  
    read() {  
        return this.clients;  
    }  
  
    select(client) {  
        this.currentClient = client;  
    }  
  
    create(client) {  
        this.clients.push(client); // this is bad but hang on  
    }  
}
```

```
const clientStore = new ClientStore();
clientStore.load(clients);
clientStore.select(clients[0]);

const spiderMan = {
  id: '1000',
  firstName: 'Peter',
  lastName: 'Parker',
  company: 'Student'
};
clientStore.create(spiderMan);
```



```
class Store {  
    state;  
  
    constructor(state) {  
        //...  
    }  
  
    getState() {  
        //...  
    }  
  
    select(key) {  
        //...  
    }  
}
```

```
class Store {  
    state;  
  
    constructor(state) {  
        this.state = state;  
    }  
  
    getState() {  
        return this.state;  
    }  
  
    select(key) {  
        return this.state[key];  
    }  
}
```

Challenges

- Create a **ProjectStore** class
- Add a **state** property to the class
- Add a **constructor** to the class that accepts a **state** parameter
- Add a **getState** method
- Add a **select** method
- Instantiate the **ProjectStore** class with the **initialState** object
- Call **select** on the class to get the **projects** collection

Decisions and Conditionals



```
true === true  
batman.secretIdentity === 'Bruce Wayne'  
client.id !== payload.id
```

Boolean Expressions

```
saveClient(client) {  
    if (!client.id) {  
        this.createClient(client);  
    } else {  
        this.updateClient(client);  
    }  
}
```

If Else

```
state.clients.map(client => {
  return (client.id === payload.id) ? Object.assign({}, client, payload) : client;
})
```

Ternary Operator

```
state.clients.map(client => {
  return (client.id === payload.id) ? Object.assign({}, client, payload) : client;
})
```

Ternary Operator

```
const calculate = (a, b, operation) => {
  switch(operation) {
    case 'add':
      return a + b;
    case 'subtract':
      return a - b;
    case 'multiply':
      return a * b;
    case 'divide':
      return a / b;
    default:
      return 42; // DUH!
  }
}
```

Switch Case



```
const reducer = (state = initialState, action) => {
  switch (action.type) {
    case 'load':
      return loadClients(state, action.payload);
    case 'select':
      return selectClient(state, action.payload);
    case 'create':
      return createClient(state, action.payload);
    case 'update':
      return updateClient(state, action.payload);
    case 'delete':
      return deleteClient(state, action.payload);
    case 'clear':
      return clearClient(state, action.payload);
    default:
      return state;
};
```

```
interface Action {  
  type: string;  
  payload?: any;  
}
```

```
const CLIENT_LOAD      = '[Client] Load';
const CLIENT_CREATE    = '[Client] Create';
const CLIENT_UPDATE    = '[Client] Update';
const CLIENT_DELETE    = '[Client] Delete';
const CLIENT_SELECT    = '[Client] Select';
const CLIENT_CLEAR     = '[Client] Clear';
```

```
const reducer = (state = initialState, action: Action) => {
  switch (action.type) {
    case CLIENT_LOAD:
      return loadClients(state, action.payload);
    case CLIENT_SELECT:
      return selectClient(state, action.payload);
    case CLIENT_CREATE:
      return createClient(state, action.payload);
    case CLIENT_UPDATE:
      return updateClient(state, action.payload);
    case CLIENT_DELETE:
      return deleteClient(state, action.payload);
    case CLIENT_CLEAR:
      return clearClient(state, action.payload);
    default:
      return state;
};
```

Challenges

- Create a **reducer** function that accepts **state** and an **action** parameter
- Add a **switch** statement that evaluates the **action type**
- Add a condition for **load**, **read**, **create**, **update**, and **delete**
- Create an appropriate method for each condition that accepts **state** an **action.payload** // The methods can just return state for now
- **BONUS!** Create an Action interface and action type constants

Collections and Iterators



```
for(let i = 0, len = clients.length; i < len; ++i) {  
  console.log(clients[i]);  
}
```

Class For Loop

```
clients.forEach(client => console.log(client));
```

Array.forEach

```
// Filter
const inactiveClientId = '1';
const filteredClients = clients.filter(client => client.id !== inactiveClientId);

// Map
const poachClients = clients.map(client => client.company = 'MINE! MWAHAHAHAH!');

// Reduce
const totalLTV = clients.reduce((acc, curr) => acc + curr.spend, 0);
```

Higher Order Functions

BONUS!

Immutable Operations

```
case CREATE_WIDGET:  
  state.push(action.payload);  
  return state;
```

Mutable

```
case UPDATE_WIDGET:  
  state.forEach((widget, index) => {  
    if (widget[comparator] === action.payload[comparator]) {  
      state.splice(index, 1, action.payload);  
    }  
  });  
  return state;
```

Mutable!

```
case DELETE_WIDGET:  
  state.forEach((widget, index) => {  
    if (widget[comparator] === action.payload[comparator]) {  
      state.splice(index, 1);  
    }  
  });  
  return state;
```

Mutable!

```
case CREATE_ITEM:  
  return [...state, action.payload];
```

```
case CREATE_ITEM:  
  return state.concat(action.payload);
```

The `concat()` method is used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array. - MDN

Immutable!

```
case UPDATE_ITEM:  
  return state.map(item => {  
    return item[comparator] === action.payload[comparator]  
      ? Object.assign({}, item, action.payload) : item;  
  });
```

The `map()` method creates a new array with the results of calling a provided function on every element in this array.

The `Object.assign()` method is used to copy the values of all enumerable own properties from one or more source objects to a target object. It will return the target object.

Immutable!

```
case DELETE_ITEM:  
  return state.filter(item => {  
    return item[comparator] !== action.payload[comparator];  
});
```

The `filter()` method creates a new array with all elements that pass the test implemented by the provided function.

Immutable!

```
case CREATE_WIDGET:  
  Object.freeze(state);  
  state.push(action.payload);  
  return state;
```

The `Object.freeze()` method freezes an object: that is, prevents new properties from being added to it; prevents existing properties from being removed; and prevents existing properties, or their enumerability, configurability, or writability, from being changed. **In essence the object is made effectively immutable. The method returns the object being frozen.**

Object.freeze



```
const createClient = (state, payload) => {
  const newClient = Object.assign({}, payload, {id: uuidv4()});
  return {
    clients: [...state.clients, newClient],
    currentClient: state.currentClient
  };
};
```

```
const createClient = (state, payload) => {
  const newClient = Object.assign({}, payload, {id: uuidv4()});
  return {
    clients: [...state.clients, newClient],
    currentClient: state.currentClient
  };
};
```

```
const updateClient = (state, payload) => {
  return {
    clients: state.clients.map(client => {
      return client.id === payload.id ? Object.assign({}, client, payload) : client;
    }),
    currentClient: state.currentClient
  };
};
```

```
const updateClient = (state, payload) => {
  return {
    clients: state.clients.map(client => {
      return client.id === payload.id ? Object.assign({}, client, payload) : client;
    }),
    currentClient: state.currentClient
  };
};
```

```
const deleteClient = (state, payload) => {
  return {
    clients: state.clients.filter(client => client.id !== payload.id),
    currentClient: state.currentClient
  };
};
```

```
const deleteClient = (state, payload) => {
  return {
    clients: state.clients.filter(client => client.id !== payload.id),
    currentClient: state.currentClient
  };
};
```

Challenges

- Update your reducer methods to properly modify state using immutable operations
- Use **array concatenation** for creating a project
- Use **array.map** and **Object.assign** to update an existing project
- Use **array.filter** to delete an existing project

The Fifth Element of Programming

Observable streams

Observables give us a powerful way to **encapsulate**, **transport** and **transform** data from user interactions to create powerful and immersive experiences.

Encapsulate

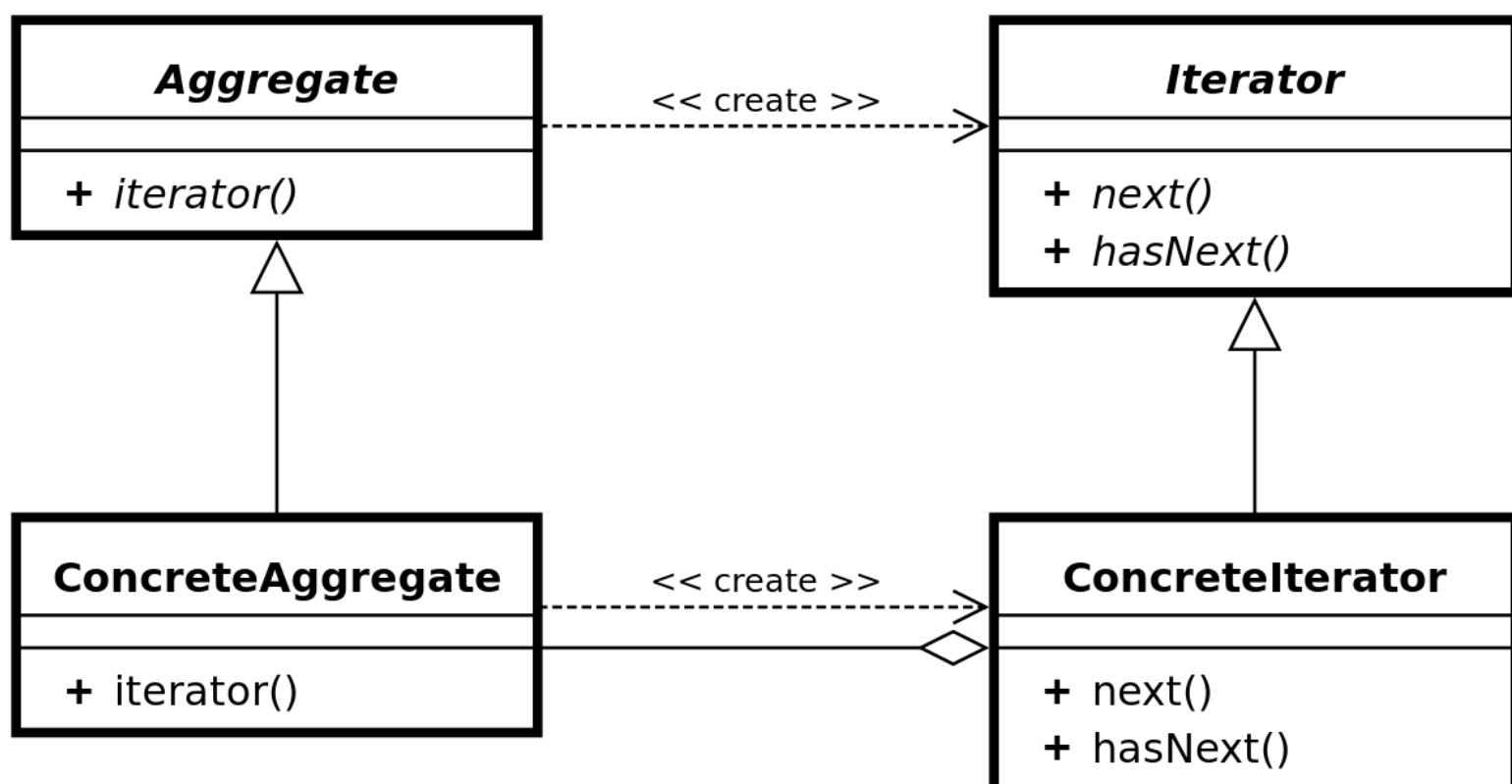
Transport

Transform

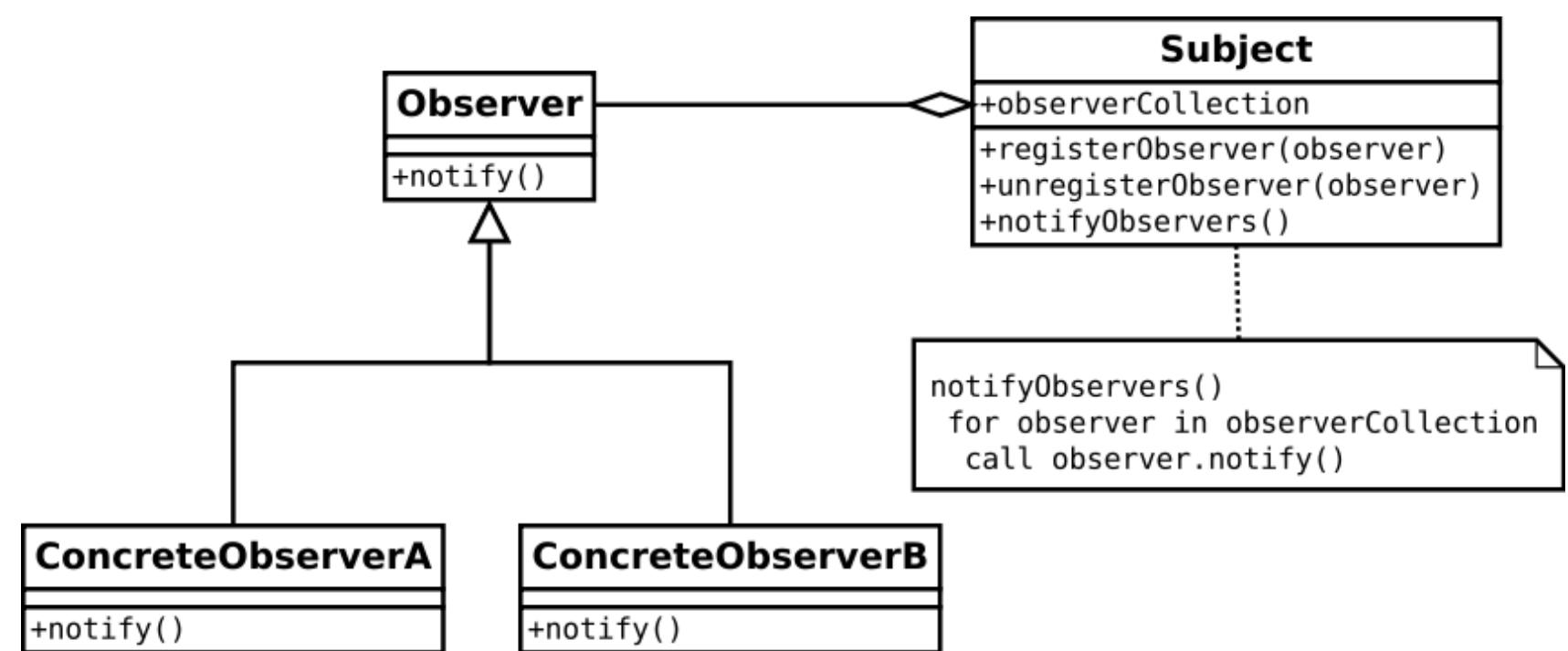
Encapsulate
Transport
Transform

Encapsulate
Transport
Transform

Iterator Pattern



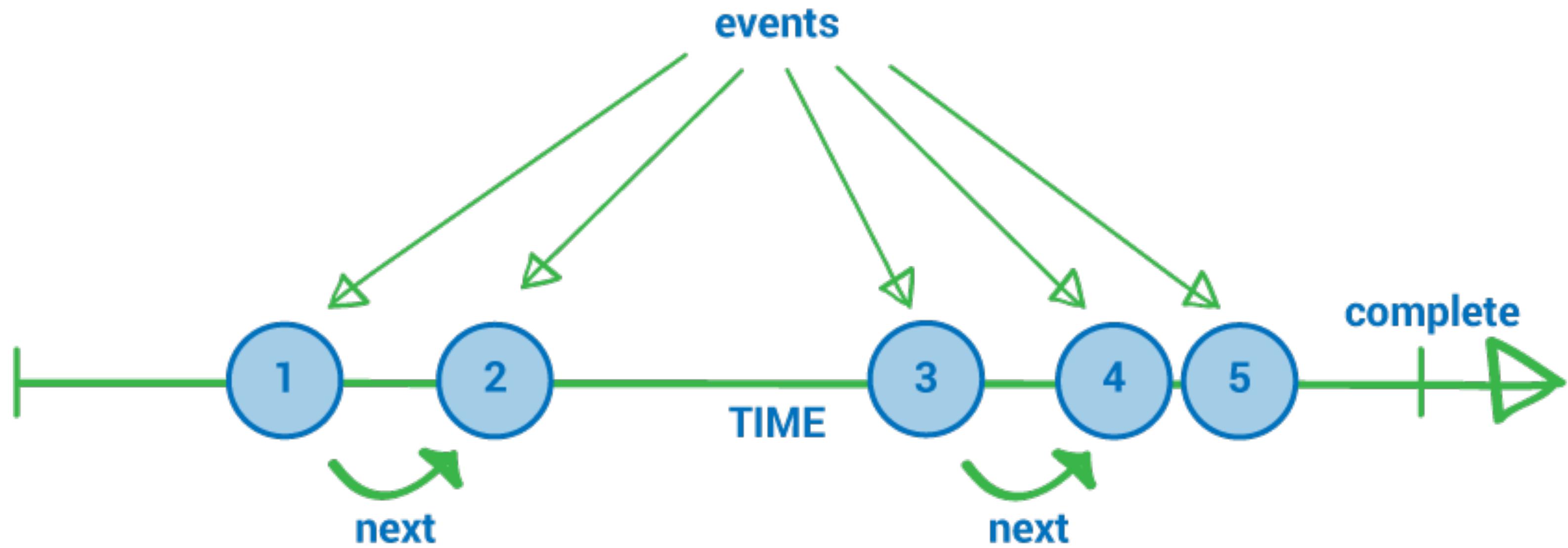
Observer Pattern



State

Communication

Communicate
state over time



Observable stream

	SINGLE	MULTIPLE
SYNCHRONOUS	Function	Enumerable
ASYNCHRONOUS	Promise	Observable

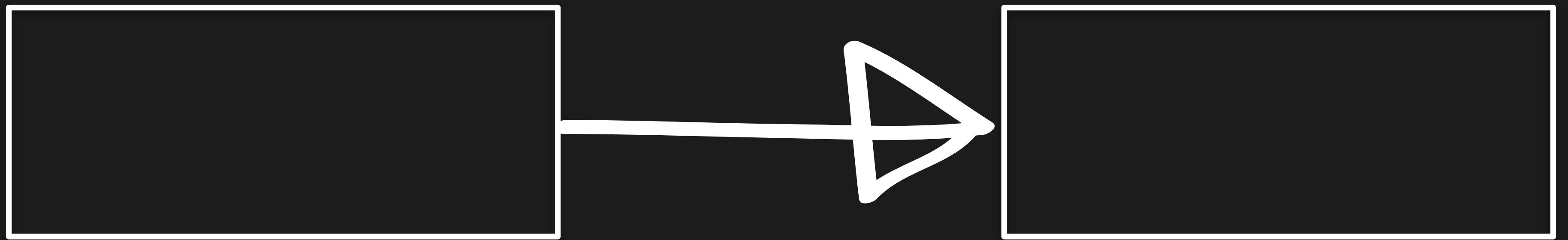
Values over time

Value consumption

	SINGLE	MULTIPLE
PULL	Function	Enumerable
PUSH	Promise	Observable

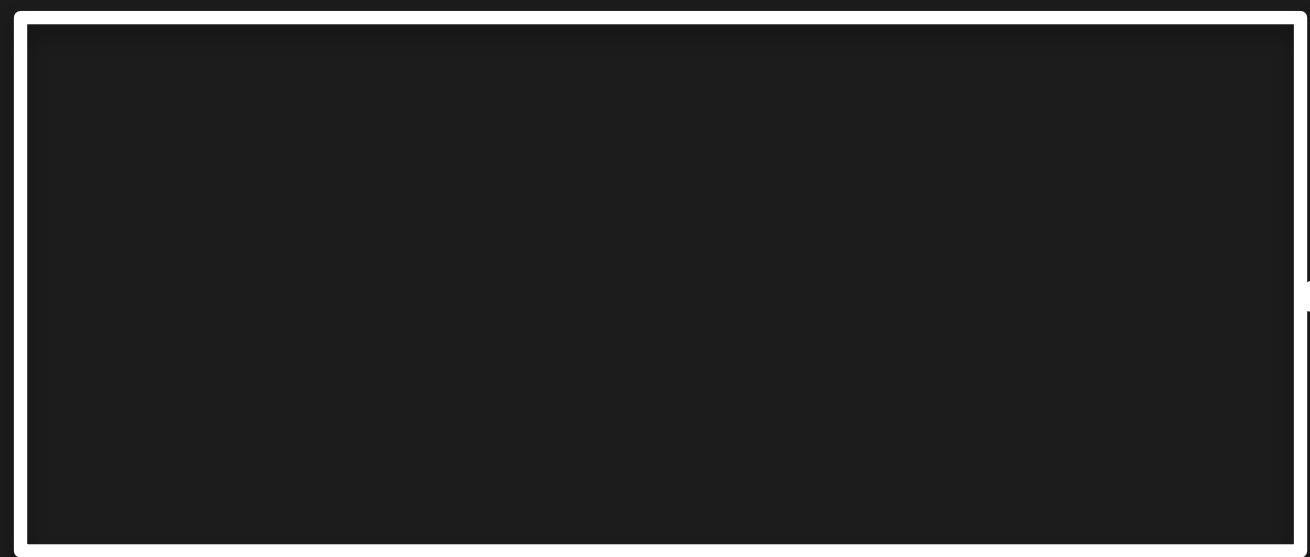
But
observables
are hard!!!

The
Observable
Stream

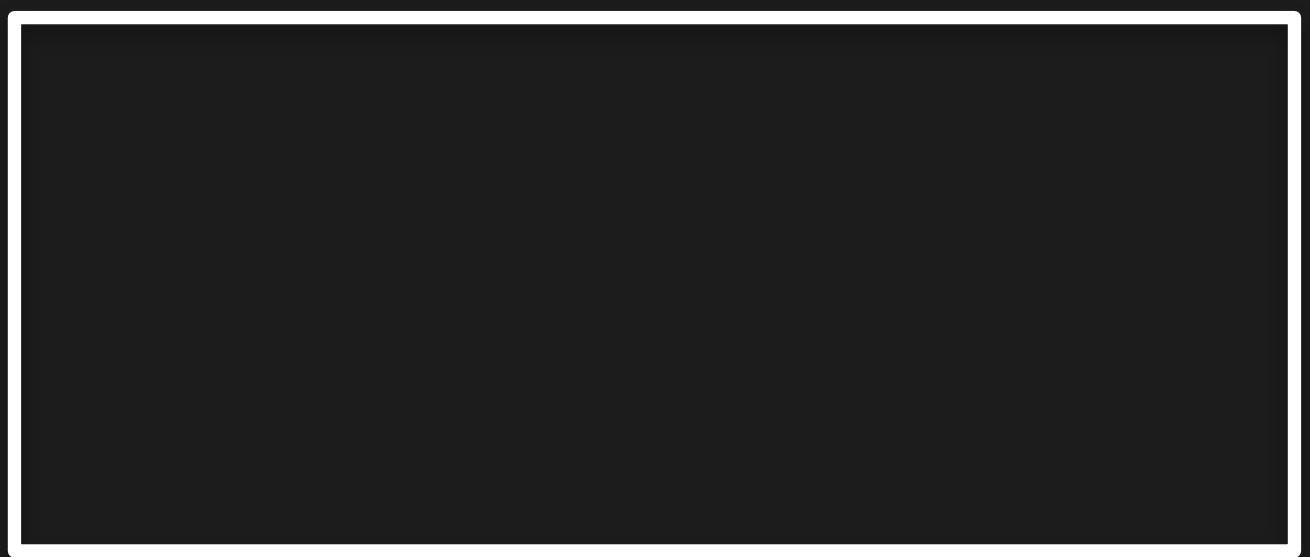
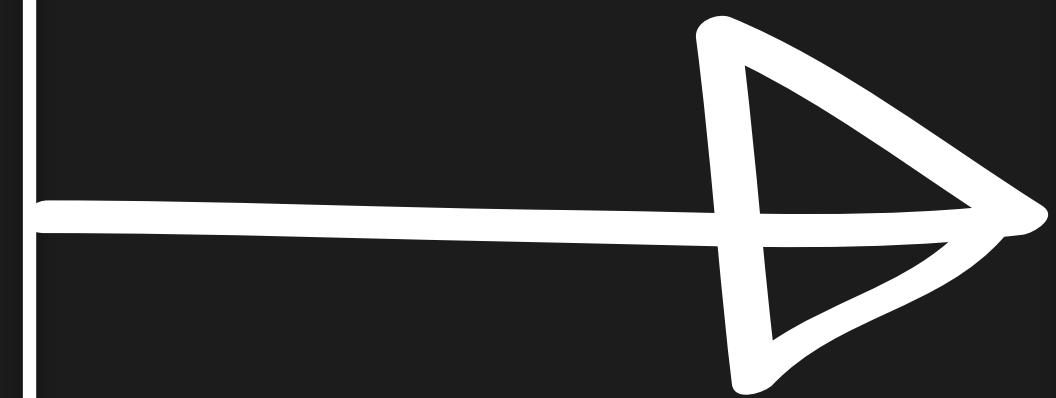


input

output



output



input

The
Basic
Sequence

initial output

magic

final input



event

operators

subscribe



```
export class BasicSequenceComponent implements OnInit {
  @ViewChild('btn') btn;
  message: string;

  ngOnInit() {
    fromEvent(this.getNativeElement(this.btn), 'click')
      .subscribe(result => this.message = 'Beast Mode Activated!');
  }

  getNativeElement(element) {
    return element._elementRef.nativeElement;
  }
}
```

```
export class BasicSequenceComponent implements OnInit {
  @ViewChild('btn') btn;
  message: string;

  ngOnInit() {
    fromEvent(this.getNativeElement(this.btn), 'click')
      .subscribe(result => this.message = 'Beast Mode Activated!');
  }

  getNativeElement(element) {
    return element._elementRef.nativeElement;
  }
}
```

Initial output

```
export class BasicSequenceComponent implements OnInit {
  @ViewChild('btn') btn;
  message: string;

  ngOnInit() {
    fromEvent(this.getNativeElement(this.btn), 'click')
      .subscribe(result => this.message = 'Beast Mode Activated!');
  }

  getNativeElement(element) {
    return element._elementRef.nativeElement;
  }
}
```

Final input

```
export class BasicSequenceComponent implements OnInit {
  @ViewChild('btn') btn;
  message: string;

  ngOnInit() {
    fromEvent(this.getNativeElement(this.btn), 'click')
      .pipe(
        map(event => 'Beast Mode Activated!')
      )
      .subscribe(result => this.message = result);
  }

  getNativeElement(element) {
    return element._elementRef.nativeElement;
  }
}
```

Everything in between

```
export class BasicSequenceComponent implements OnInit {
  @ViewChild('btn') btn;
  message: string;

  ngOnInit() {
    fromEvent(this.getNativeElement(this.btn), 'click')
      .pipe(
        filter((event: KeyboardEvent) => event.shiftKey),
        map(event => 'Beast Mode Activated!')
      )
      .subscribe(result => this.message = result);
  }

  getNativeElement(element) {
    return element._elementRef.nativeElement;
  }
}
```

Everything in between

```
export class BasicSequenceComponent implements OnInit {
  @ViewChild('btn') btn;
  message: string;

  ngOnInit() {
    fromEvent(this.getNativeElement(this.btn), 'click')
      .pipe(
        filter((event: KeyboardEvent) => event.shiftKey),
        map(event => 'Beast Mode Activated!')
      )
      .subscribe(result => this.message = result);
  }

  getNativeElement(element) {
    return element._elementRef.nativeElement;
  }
}
```

How do we
preserve state
in a stream?

```
<button #right mat-raised-button color="accent">Move Right</button>
<div class="container">
  <div #ball class="ball"
    [style.left]="position.x + 'px'"
    [style.top]="position.y + 'px'">
  </div>
</div>
```

```
@ViewChild('right') right;
position: any;

ngOnInit() {
  fromEvent(this.getNativeElement(this.right), 'click')
    .pipe(
      map(event => 10),
      startWith({x: 100, y: 150}),
      scan((acc: Coordinate, curr: number) => Object.assign({}, acc, {x: acc.x + curr}))
    )
    .subscribe(position => this.position = position);
}
```

```
@ViewChild('right') right;
position: any;

ngOnInit() {
  fromEvent(this.getNativeElement(this.right), 'click')
    .pipe(
      map(event => 10),
      startWith({x: 100, y: 150}),
      scan((acc: Coordinate, curr: number) => Object.assign({}, acc, {x: acc.x + curr}))
    )
    .subscribe(position => this.position = position);
}
```

```
@ViewChild('right') right;
position: any;

ngOnInit() {
  fromEvent(this.getNativeElement(this.right), 'click')
    .pipe(
      map(event => 10),
      startWith({x: 100, y: 150}),
      scan((acc: Coordinate, curr: number) => Object.assign({}, acc, {x: acc.x + curr}))
    )
    .subscribe(position => this.position = position);
}
```

```
@ViewChild('right') right;
position: any;

ngOnInit() {
  fromEvent(this.getNativeElement(this.right), 'click')
    .pipe(
      map(event => 10),
      startWith({x: 100, y: 150}),
      scan((acc: Coordinate, curr: number) => Object.assign({}, acc, {x: acc.x + curr}))
    )
    .subscribe(position => this.position = position);
}
```

What if we have
more than one
stream?

```
@ViewChild('left') left;
@ViewChild('right') right;
position: any;

ngOnInit() {
  const left$ = fromEvent(this.nativeElement(this.left), 'click')
    .pipe(map(event => -10));

  const right$ = fromEvent(this.nativeElement(this.right), 'click')
    .pipe(map(event => 10));

  merge(left$, right$)
    .pipe(
      startWith({x: 200, y: 200}),
      scan((acc: Coordinate, curr: number) => Object.assign({}, acc, {x: acc.x + curr}))
    )
    .subscribe(position => this.position = position);
}
```

```
@ViewChild('left') left;
@ViewChild('right') right;
position: any;

ngOnInit() {
  const left$ = fromEvent(this.nativeElement(this.left), 'click')
    .pipe(map(event => -10));

  const right$ = fromEvent(this.nativeElement(this.right), 'click')
    .pipe(map(event => 10));

  merge(left$, right$)
    .pipe(
      startWith({x: 200, y: 200}),
      scan((acc: Coordinate, curr: number) => Object.assign({}, acc, {x: acc.x + curr}))
    )
    .subscribe(position => this.position = position);
}
```

What can we
put in a stream?

```
ngOnInit() {
  const leftArrow$ = fromEvent(document, 'keydown')
    .pipe(
      filter((event: KeyboardEvent) => event.key === 'ArrowLeft'),
      mapTo(position => this.decrement(position, 'x', 10))
    );
  const rightArrow$ = fromEvent(document, 'keydown')
    .pipe(
      filter((event: KeyboardEvent) => event.key === 'ArrowRight'),
      mapTo(position => this.increment(position, 'x', 10))
    );
  merge(leftArrow$, rightArrow$, upArrow$, downArrow$)
    .pipe(
      startWith({x: 100, y: 100}),
      scan((acc, curr: Function) => curr(acc))
    )
    .subscribe(position => this.position = position);
}

increment(obj, prop, value) {
  return Object.assign({}, obj, {[prop]: obj[prop] + value});
}

decrement(obj, prop, value) {
  return Object.assign({}, obj, {[prop]: obj[prop] - value});
}
```

```
ngOnInit() {
  const leftArrow$ = fromEvent(document, 'keydown')
    .pipe(
      filter((event: KeyboardEvent) => event.key === 'ArrowLeft'),
      mapTo(position => this.decrement(position, 'x', 10))
    );
  const rightArrow$ = fromEvent(document, 'keydown')
    .pipe(
      filter((event: KeyboardEvent) => event.key === 'ArrowRight'),
      mapTo(position => this.increment(position, 'x', 10))
    );
  merge(leftArrow$, rightArrow$)
    .pipe(
      startWith({x: 100, y: 100}),
      scan((acc, curr: Function) => curr(acc))
    )
    .subscribe(position => this.position = position);
}

increment(obj, prop, value) {
  return Object.assign({}, obj, {[prop]: obj[prop] + value});
}

decrement(obj, prop, value) {
  return Object.assign({}, obj, {[prop]: obj[prop] - value});
}
```

```
ngOnInit() {
  const leftArrow$ = fromEvent(document, 'keydown')
    .pipe(
      filter((event: KeyboardEvent) => event.key === 'ArrowLeft'),
      mapTo(position => this.decrement(position, 'x', 10))
    );
  const rightArrow$ = fromEvent(document, 'keydown')
    .pipe(
      filter((event: KeyboardEvent) => event.key === 'ArrowRight'),
      mapTo(position => this.increment(position, 'x', 10))
    );
  merge(leftArrow$, rightArrow$)
    .pipe(
      startWith({x: 100, y: 100}),
      scan((acc, curr: Function) => curr(acc))
    )
    .subscribe(position => this.position = position);
}

increment(obj, prop, value) {
  return Object.assign({}, obj, {[prop]: obj[prop] + value});
}

decrement(obj, prop, value) {
  return Object.assign({}, obj, {[prop]: obj[prop] - value});
}
```

```
ngOnInit() {
  const leftArrow$ = fromEvent(document, 'keydown')
    .pipe(
      filter((event: KeyboardEvent) => event.key === 'ArrowLeft'),
      mapTo(position => this.decrement(position, 'x', 10))
    );
  const rightArrow$ = fromEvent(document, 'keydown')
    .pipe(
      filter((event: KeyboardEvent) => event.key === 'ArrowRight'),
      mapTo(position => this.increment(position, 'x', 10))
    );
  merge(leftArrow$, rightArrow$)
    .pipe(
      startWith({x: 100, y: 100}),
      scan((acc, curr: Function) => curr(acc))
    )
    .subscribe(position => this.position = position);
}

increment(obj, prop, value) {
  return Object.assign({}, obj, {[prop]: obj[prop] + value});
}

decrement(obj, prop, value) {
  return Object.assign({}, obj, {[prop]: obj[prop] - value});
}
```

How can we
sequence a stream?

```
@ViewChild('ball') ball;  
position: any;  
  
ngOnInit() {  
  const BALL_OFFSET = 50;  
  
  const move$ = fromEvent(document, 'mousemove')  
    .pipe(  
      map((event: MouseEvent) => {  
        const offset = $(event.target).offset();  
        return {  
          x: event.clientX - offset.left - BALL_OFFSET,  
          y: event.pageY - BALL_OFFSET  
        };  
      })  
    );  
  
  const down$ = fromEvent(this.ball.nativeElement, 'mousedown');  
  const up$ = fromEvent(document, 'mouseup');  
  
  down$.pipe(  
    switchMap(event => move$.pipe(takeUntil(up$))),  
    startWith({x: 100, y: 100})  
  )  
    .subscribe(position => this.position = position);  
}
```

```
@ViewChild('ball') ball;
position: any;

ngOnInit() {
  const BALL_OFFSET = 50;
  const move$ = fromEvent(document, 'mousemove')
    .pipe(
      map((event: MouseEvent) => {
        const offset = $(event.target).offset();
        return {
          x: event.clientX - offset.left - BALL_OFFSET,
          y: event.pageY - BALL_OFFSET
        };
      })
    );
  const down$ = fromEvent(this.ball.nativeElement, 'mousedown');

  down$.pipe(
    switchMap(event => move$),
    startWith({x: 100, y: 100})
  )
  .subscribe(position => this.position = position);
}
```

```
@ViewChild('ball') ball;  
position: any;  
  
ngOnInit() {  
  const BALL_OFFSET = 50;  
  
  const move$ = fromEvent(document, 'mousemove')  
    .pipe(  
      map((event: MouseEvent) => {  
        const offset = $(event.target).offset();  
        return {  
          x: event.clientX - offset.left - BALL_OFFSET,  
          y: event.pageY - BALL_OFFSET  
        };  
      })  
    );  
  
  const down$ = fromEvent(this.ball.nativeElement, 'mousedown');  
  
  down$.pipe(  
    switchMap(event => move$),  
    startWith({x: 100, y: 100})  
  ).subscribe(position => this.position = position);  
}
```

```
@ViewChild('ball') ball;  
position: any;  
  
ngOnInit() {  
  const BALL_OFFSET = 50;  
  
  const move$ = fromEvent(document, 'mousemove')  
    .pipe(  
      map((event: MouseEvent) => {  
        const offset = $(event.target).offset();  
        return {  
          x: event.clientX - offset.left - BALL_OFFSET,  
          y: event.pageY - BALL_OFFSET  
        };  
      })  
    );  
  
  const down$ = fromEvent(this.ball.nativeElement, 'mousedown');  
  
  down$.pipe(  
    switchMap(event => move$),  
  
    startWith({x: 100, y: 100})  
  )  
    .subscribe(position => this.position = position);  
}
```

```
@ViewChild('ball') ball;  
position: any;  
  
ngOnInit() {  
  const BALL_OFFSET = 50;  
  
  const move$ = fromEvent(document, 'mousemove')  
    .pipe(  
      map((event: MouseEvent) => {  
        const offset = $(event.target).offset();  
        return {  
          x: event.clientX - offset.left - BALL_OFFSET,  
          y: event.pageY - BALL_OFFSET  
        };  
      })  
    );  
  
  const down$ = fromEvent(this.ball.nativeElement, 'mousedown');  
  const up$ = fromEvent(document, 'mouseup');  
  
  down$.pipe(  
    switchMap(event => move$.pipe(takeUntil(up$))),  
    startWith({x: 100, y: 100})  
  )  
    .subscribe(position => this.position = position);  
}
```

What effect does the
origin of the stream
have on the output?

```
<app-line  
  *ngFor="let line of lines" [line]="line">  
</app-line>
```

```
<svg style="position: absolute" width="100%" height="100%>
<line [attr.x1]="line.x1" [attr.y1]="line.y1"
[attr.x2]="line.x2" [attr.y2]="line.y2"
style="stroke:rgb(255,0,0);stroke-width:2px"/>
</svg>
```

```
lines: any[] = [];
ngOnInit() {
  fromEvent(document, 'click')
    .pipe(
      map((event: MouseEvent) => {
        const offset = $(event.target).offset();
        return {
          x: event.clientX - offset.left,
          y: event.pageY - offset.top
        };
      }),
      pairwise(),
      map(positions => {
        const p1 = positions[0];
        const p2 = positions[1];
        return { x1: p1.x, y1: p1.y, x2: p2.x, y2: p2.y };
      }),
    )
    .subscribe(line => this.lines = [...this.lines, line]);
}
```

```
lines: any[] = [];
ngOnInit() {
  fromEvent(document, 'mousemove')
    .pipe(
      map((event: MouseEvent) => {
        const offset = $(event.target).offset();
        return {
          x: event.clientX - offset.left,
          y: event.pageY - offset.top
        };
      }),
      pairwise(),
      map(positions => {
        const p1 = positions[0];
        const p2 = positions[1];
        return { x1: p1.x, y1: p1.y, x2: p2.x, y2: p2.y };
      }),
    )
    .subscribe(line => this.lines = [...this.lines, line]);
}
```

How do we control an
observable stream?

```
const observable = new Observable(subscriber => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  subscriber.complete();
});
```

```
const observer: Observer<any> = {
  next(x) { console.log(x) },
  error(err) { console.log(err) },
  complete() { console.log('complete') }
}
```

```
const subscription = observable.subscribe(observer);
subscription.unsubscribe();
```

```
export class NotificationService {  
    private subject = new Subject();  
    notifications$ = this.subject.asObservable();  
  
    dispatch(notification) {  
        this.subject.next(notification);  
    }  
}
```

```
export class NotificationService {  
    private subject = new Subject();  
    notifications$ = this.subject.asObservable();  
  
    dispatch(notification) {  
        this.subject.next(notification);  
    }  
}
```

```
export class AppComponent implements OnInit {
  constructor(private ns: NotificationService) {}

  ngOnInit() {
    this.ns.notifications$.subscribe((notification) =>
      this.showNotification(notification)
    );
  }

  showNotification(notification) {
    // ...
  }
}
```

```
export class NotificationService {  
    private subject = new Subject();  
    notifications$ = this.subject.asObservable();  
  
    dispatch(notification) {  
        this.subject.next(notification);  
    }  
}
```

```
export class NotificationService {  
    private subject = new Subject();  
    notifications$ = this.subject.asObservable();  
  
    dispatch(notification) {  
        this.subject.next(notification);  
    }  
}
```

```
export class EventCommunicationComponent {  
  searchControl: FormControl = new FormControl('');  
  
  constructor(private notifications: NotificationService) {}  
  
  notify(message) {  
    this.notifications.dispatch(message);  
  }  
}
```

Better subscriptions
with the `async pipe`

```
const valueStream$ = this.myForm.valueChanges
  .pipe(
    map(val => this.parseRange(val)),
    pairwise(),
    filter(([oldVal, newVal]) => this.filterMinMaxValues(oldVal, newVal)),
    map(([oldVal, newVal]) => newVal),
    tap(range => this.salesNumbers.updateRange(range))
);
```

```
this.minValue$ = valueStream$
  .pipe(
    map(vals => vals.min),
    startWith(this.startMin)
);
```

```
this.maxValue$ = valueStream$
  .pipe(
    map(vals => vals.max),
    startWith(this.startMax)
);
```

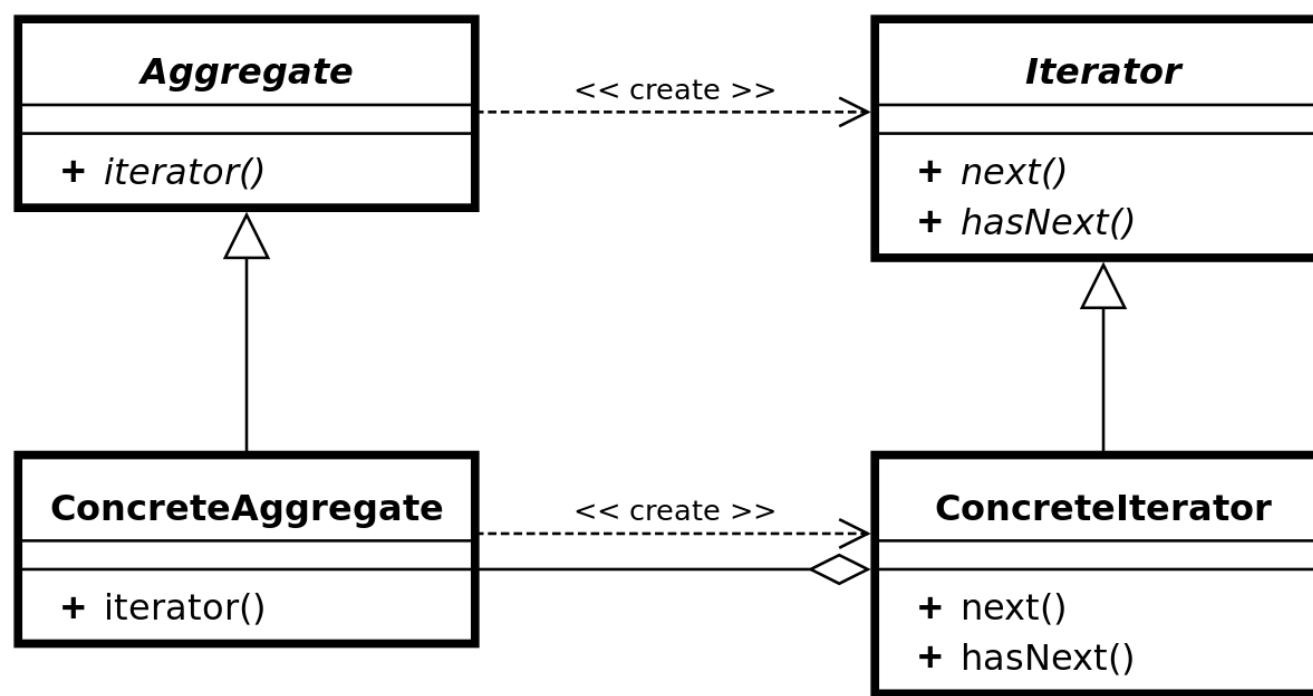
```
<form [formGroup]="myForm">
  <input
    type="range"
    formControlName="min"
    [min]="min"
    [max]="max"
    [step]="step" />
  <p>Buy at {{ minValue$ | async | currency }}</p>
  <input
    type="range"
    formControlName="max"
    [min]="min"
    [max]="max"
    [step]="step" />
  <p>Sell at {{ maxValue$ | async | currency }}</p>
</form>
```

```
<form [formGroup]="myForm">
  <input
    type="range"
    formControlName="min"
    [min]="min"
    [max]="max"
    [step]="step" />
  <p>Buy at {{ minValue$ | async | currency }}</p>
  <input
    type="range"
    formControlName="max"
    [min]="min"
    [max]="max"
    [step]="step" />
  <p>Sell at {{ maxValue$ | async | currency }}</p>
</form>
```

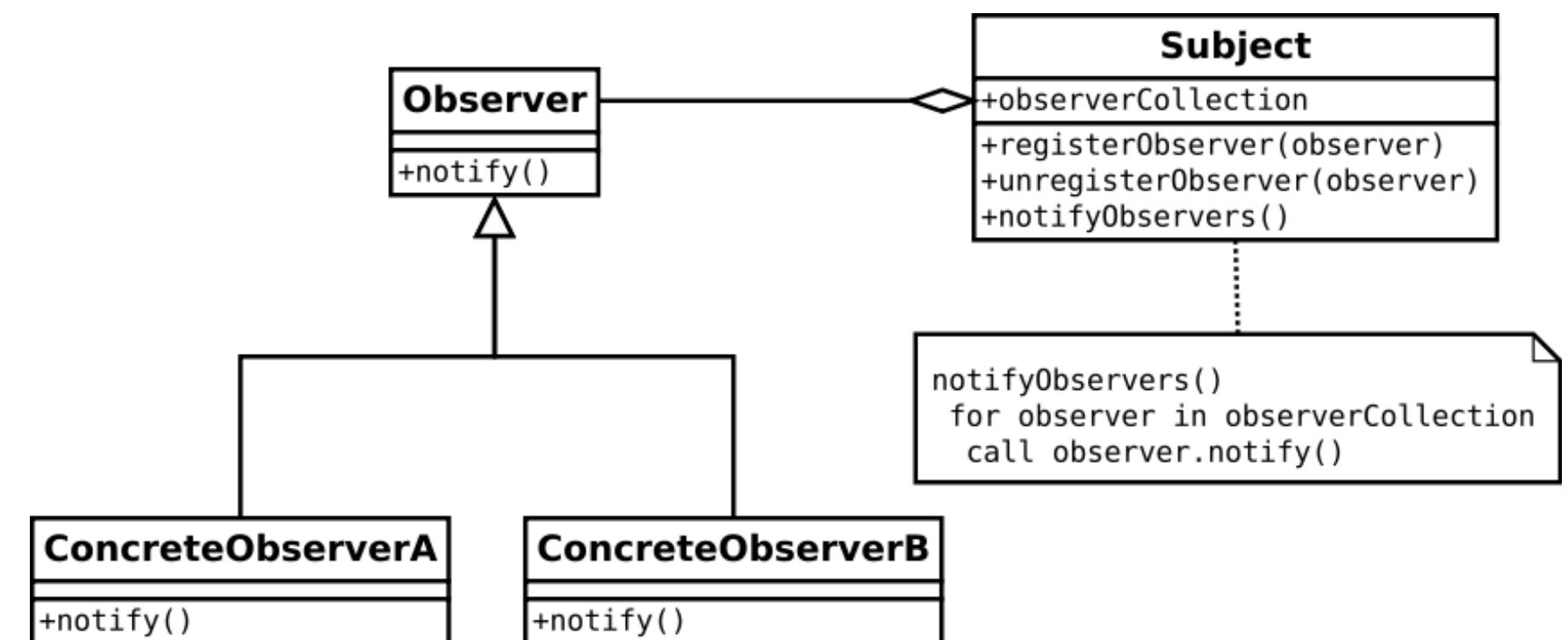
Can I use a stream to
communicate events
to the rest of my app?

Communicate
state over time

Iterator Pattern



Observer Pattern



State

Communication

```
export class NotificationService {
  private subject = new Subject();
  notifications$ = this.subject.asObservable();

  constructor() {
  }

  emit(notification) {
    this.subject.next(notification);
  }
}
```

```
export class AppComponent implements OnInit {
  constructor(private snackbar: MatSnackBar, private ns: NotificationService)
  {
  }

  ngOnInit() {
    this.ns.notifications$
      .subscribe(notification => this.showNotification(notification));
  }

  showNotification(notification) {
    this.snackbar.open(notification, 'OK', {
      duration: 3000
    });
  }
}
```

Practical Examples

Basic
Observable
Sequence

Challenges

- Open **input.component.ts**
- Capture the **searchControl** output and input it into **queryString**
- Map the output to all uppercase letters
- Reverse the output i.e. **Lukas** becomes **sakuL**
- **BONUS** How would you URL encode the output?

Preserving State

Challenges

- Open **counter.component.ts**
- Capture the **btn** click and increment count by 1

Merging Streams

Challenges

- Open **slideshow.component.ts**
- Create a **previous\$** stream to capture the previous button click
- Create a **next\$** stream to capture the next button click
- Return a **Target** object that looks like this `{shift: -1, direction: 'right'}`
- Combine both streams to update the same slideshow
- Use **startWith** to set the initial value to **this.currentPosition**
- Use the **scan** method to calculate the adjusted index and return a **Position** object
HINT: `this.getAdjustedIndex(acc.index, value.shift)`
- Update the slideshow to use the new **Position** object

Mapping to Functions

Challenges

- Open **game.component.ts**
- Complete the **leftArrow\$** stream to capture the correct keystroke
- Complete the **rightArrow\$** stream to capture the correct keystroke
- Complete the **leftArrow\$** stream to pass the appropriate value
- Complete the **rightArrow\$** stream to pass the appropriate value
- Add a **scan** function to act on the stream data in the **merge** operator

Grouping Stream Data

Challenges

- Open **map.component.ts**
- Create a stream to capture clicks on the document
- Map the **MouseEvent** to an appropriate position object
- Identify and implement the operator needed to draw a straight line
- Hints have been given to help keep you focused

Sequencing Streams

Challenges

- Open **annotate.component.ts**
- Think about the event sequence involved with drawing a smooth line
- Create and sequence the streams to model that interaction
- Hints have been given to help keep you focused

Distributed Complexity

A photograph of a group of eight people sitting around a campfire at night. The scene is dimly lit by the fire, with bright orange and yellow flames visible. The people are dressed in dark clothing, and some are wearing headlamps or have flashlights. They are sitting on the ground, facing the fire. The background is dark with silhouettes of trees.

Story Time

DOM



```
$("form").submit(function( event ) {  
  if ( $("input:first").val() === "correct" ) {  
    $("span").text( "Validated..." ).show();  
    return;  
  }  
  
  $("span").text( "Not valid!" ).show().fadeOut( 1000 );  
  event.preventDefault();  
});
```



jQuery "Application"

DOM

CONTROLLER



First Generation AngularJS Application

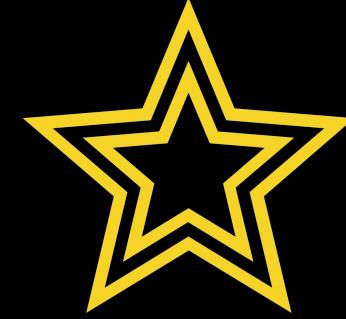
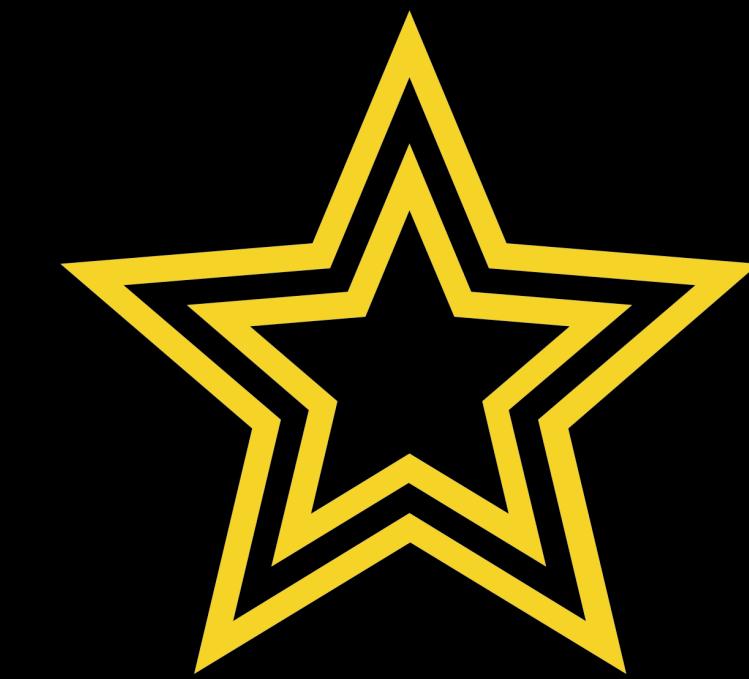
DOM

CONTROLLER

SERVICE

DOM

CONTROLLER

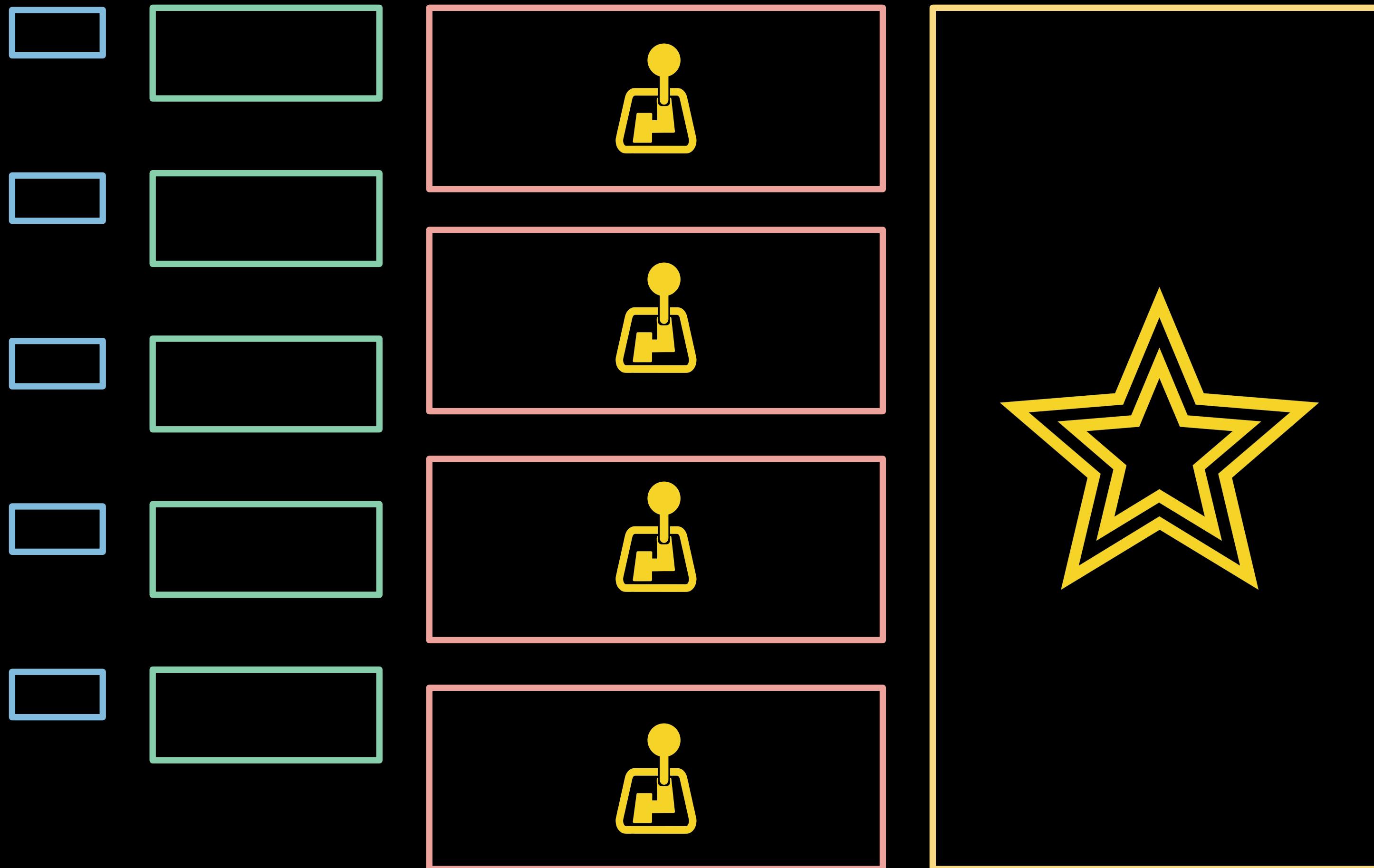


Second Generation AngularJS Application



Typical Stateful Service

Enter Redux



REDUCERS

STORE

In classical physics, if you know everything about a system at some instant of time, and you also know the equations that govern how the system changes then you can predict the future. That's what we mean when we say that the classical laws of physics are deterministic. If we can say the same thing, but with the past and future reversed, the same equations tell you everything about the past. Such a system is called reversible.

The Theoretical Minimum by Leonard Susskind

WAT.



In classical physics, if you know everything about a system at some instant of time, and you also know the equations that govern how the system changes then you can predict the future. That's what we mean when we say that the classical laws of physics are deterministic. If we can say the same thing, but with the past and future reversed, the same equations tell you everything about the past. Such a system is called reversible.

The Theoretical Minimum by Leonard Susskind

STORE!

In classical physics, if you know everything about a system at some instant of time, and you also know the equations that govern how the system changes then you can predict the future. That's what we mean when we say that the classical laws of physics are deterministic. If we can say the same thing, but with the past and future reversed, the same equations tell you everything about the past. Such a system is called reversible.

The Theoretical Minimum by Leonard Susskind

REDUCERS!

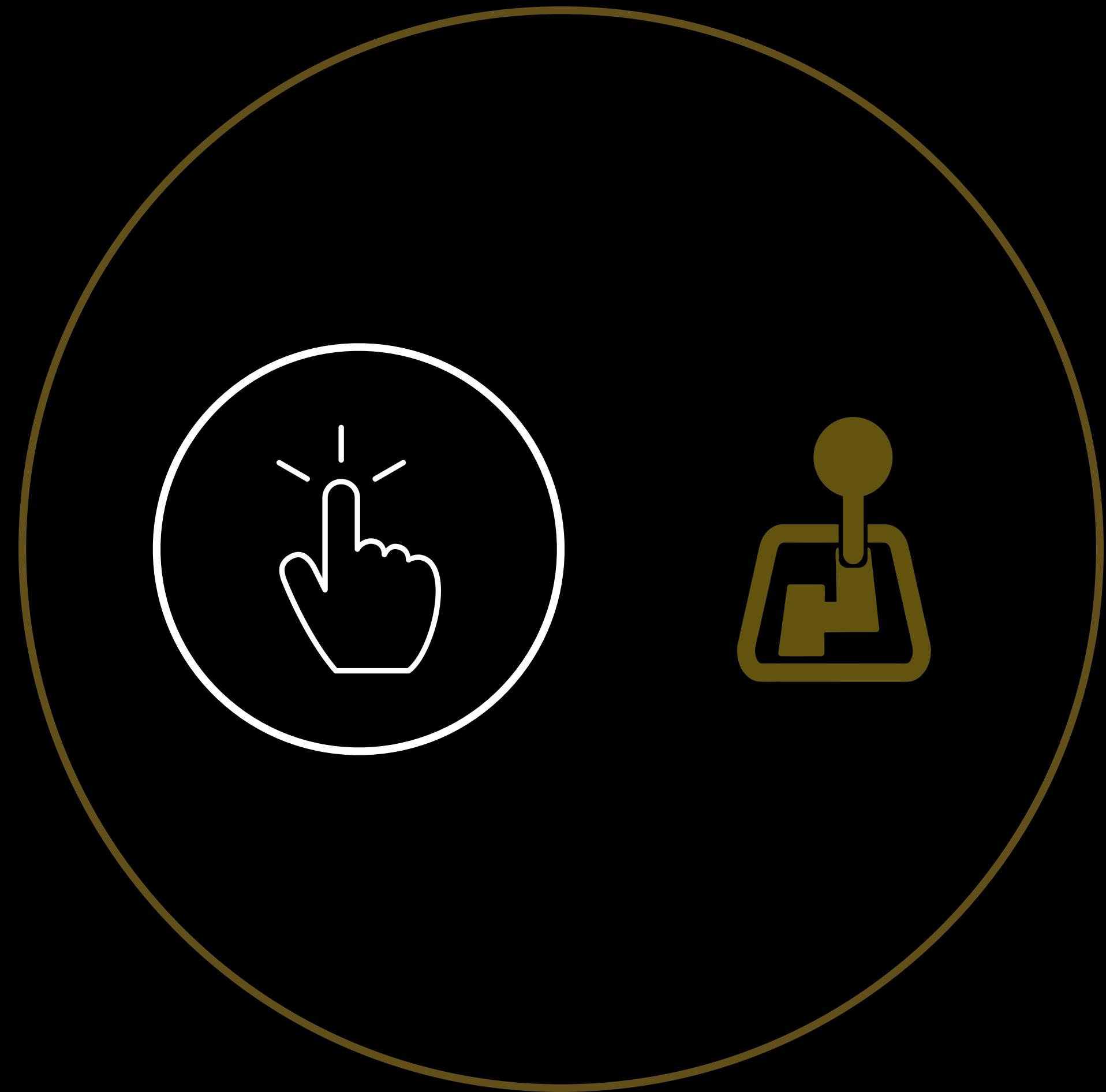
In classical physics, if you know everything about a system at some instant of time, and you also know the equations that govern how the system changes then you can predict the future. That's what we mean when we say that the classical laws of physics are deterministic. If we can say the same thing, but with the past and future reversed, the same equations tell you everything about the past. Such a system is called reversible.

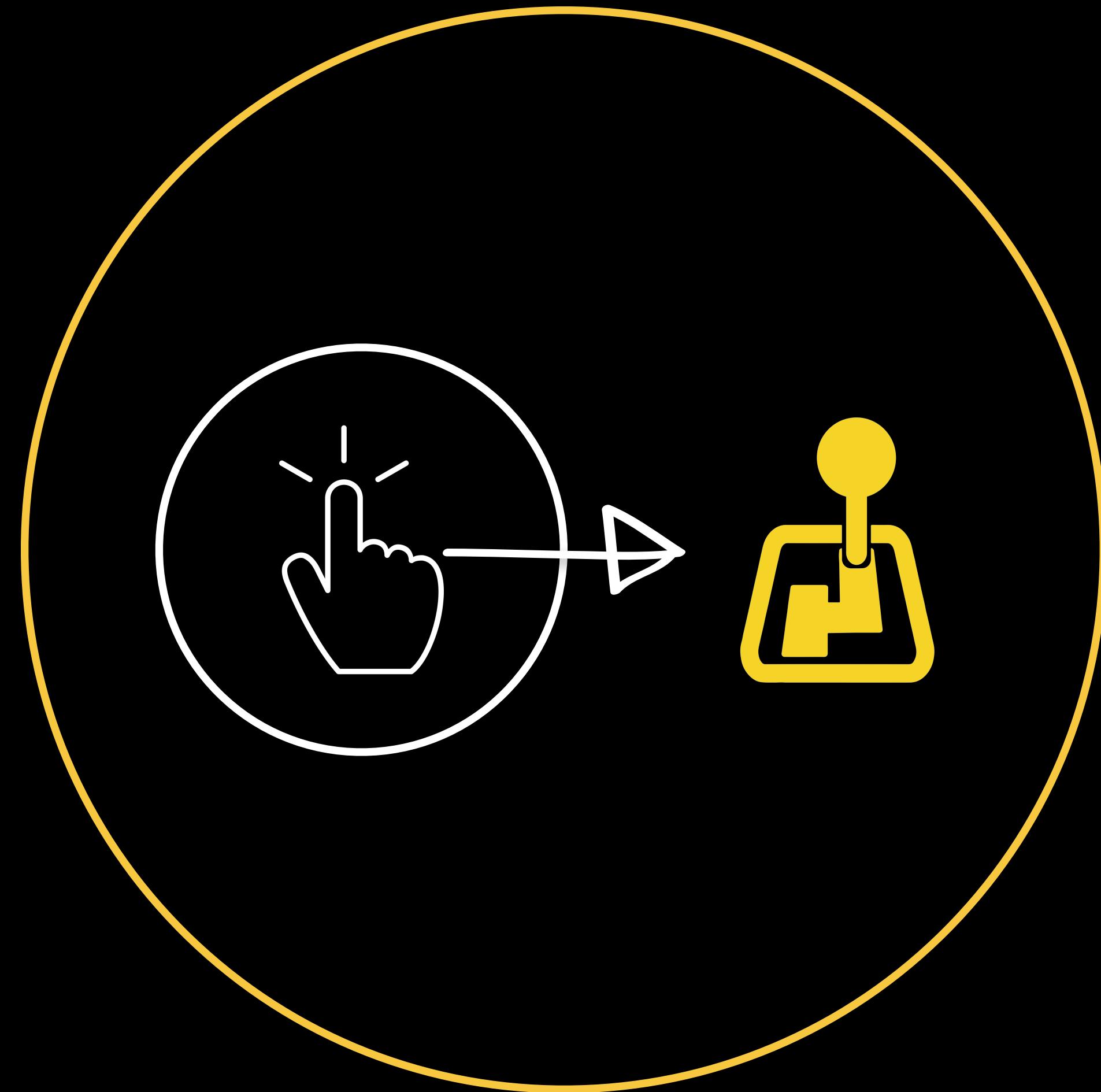
The Theoretical Minimum by Leonard Susskind

In classical physics, if you know everything about a system at some instant of time, and you also know the equations that govern how the system changes then you can predict the future. That's what we mean when we say that the classical laws of physics are deterministic. If we can say the same thing, but with the past and future reversed, the same equations tell you everything about the past. Such a system is called reversible.

The Theoretical Minimum by Leonard Susskind

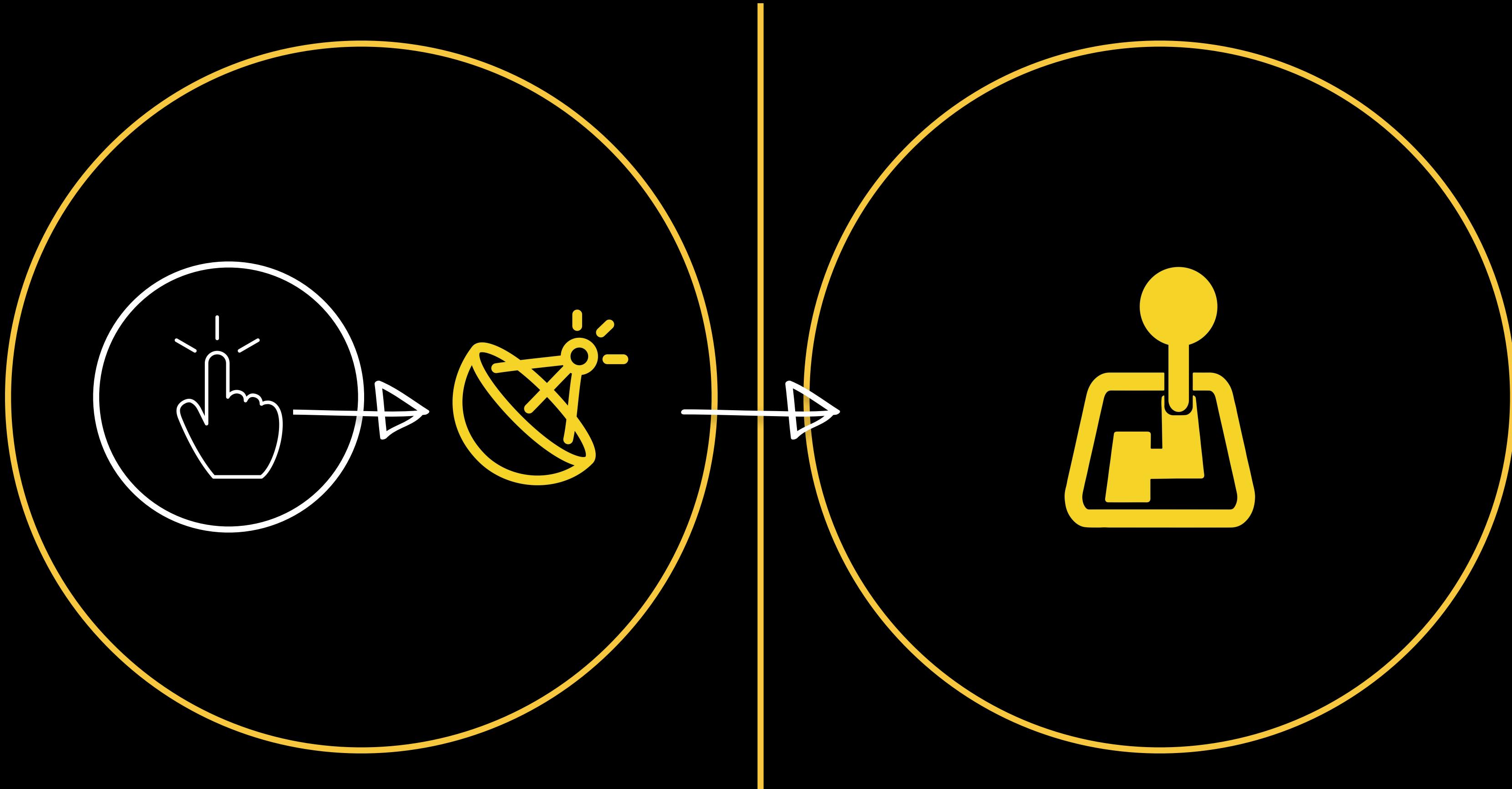
User Input and Response

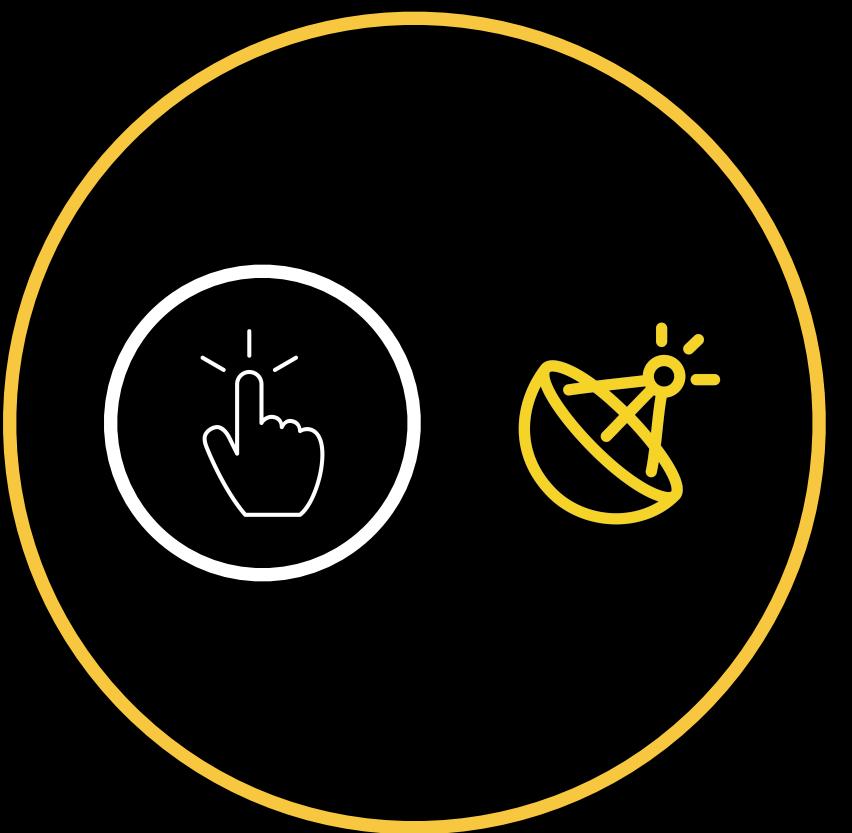


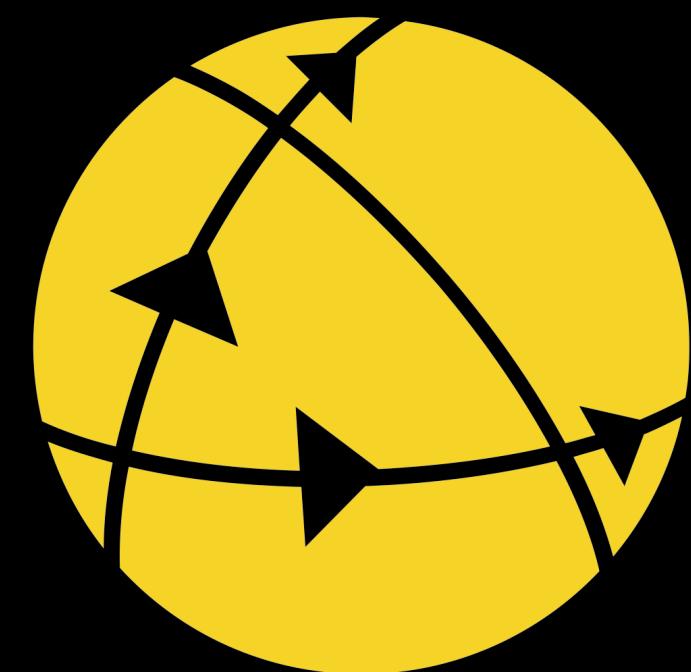
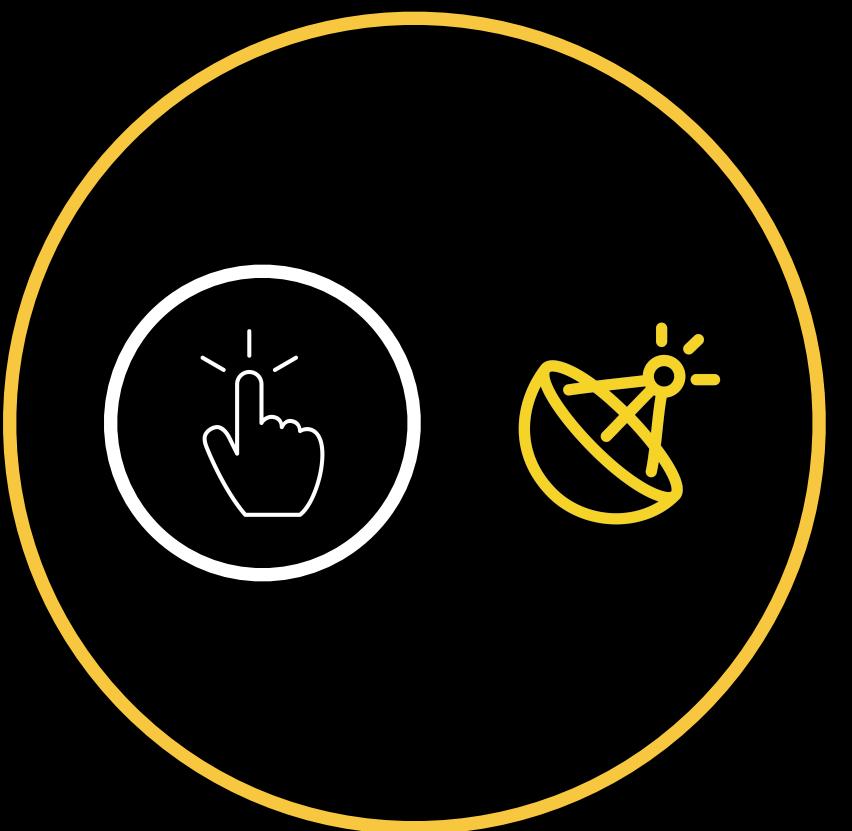


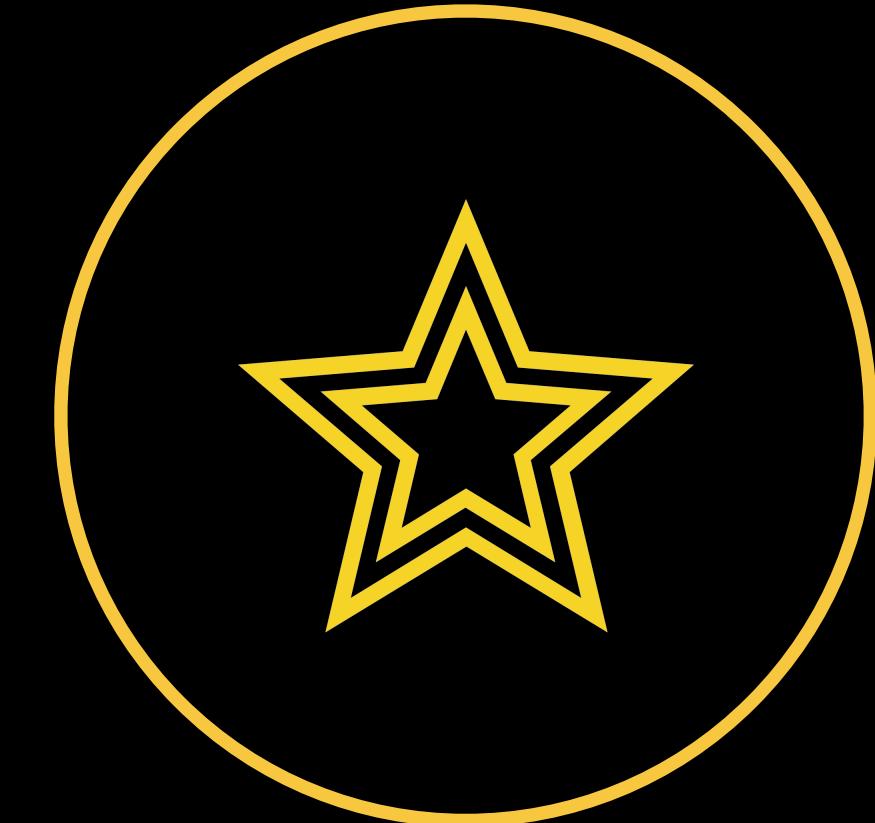
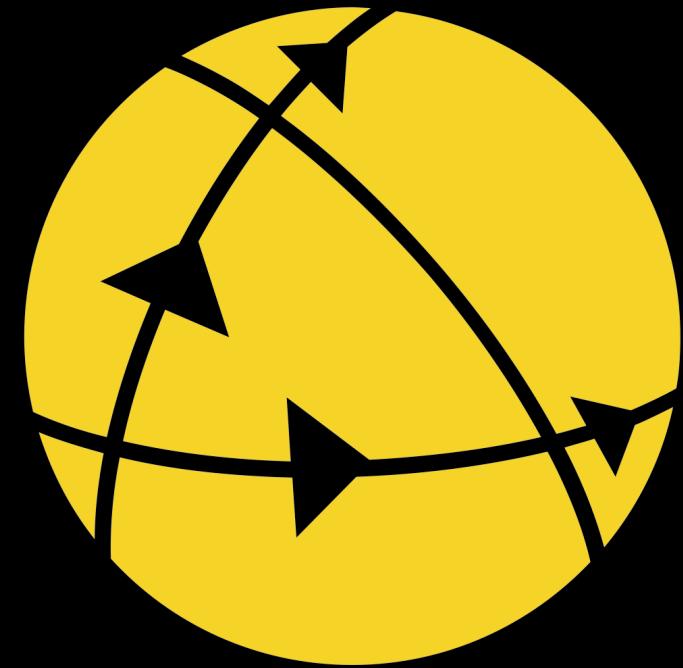
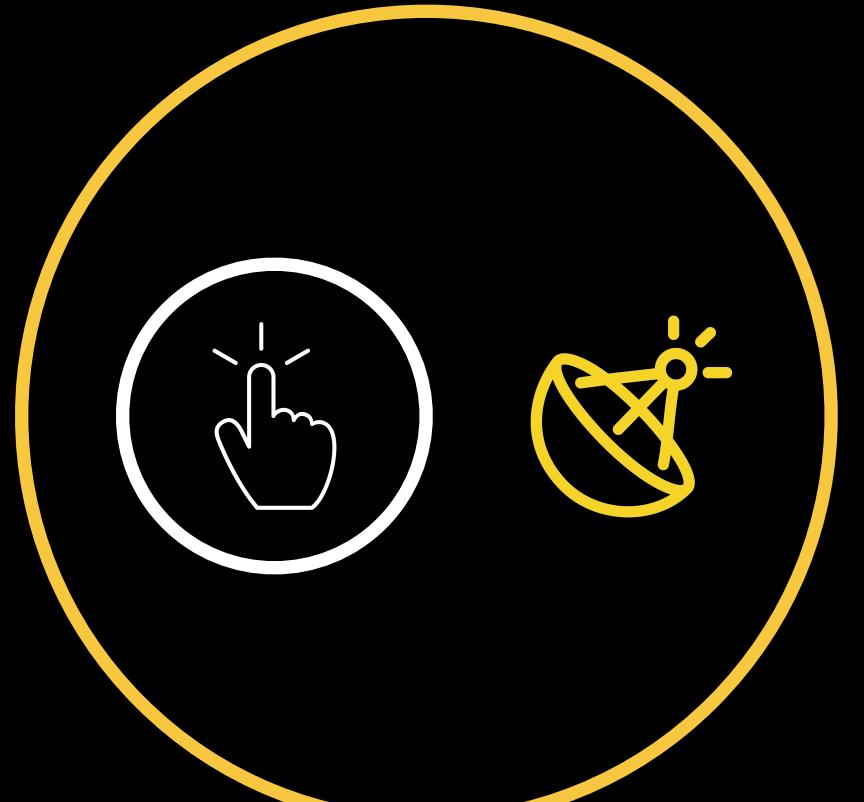
Time and Space

ACTIONS!

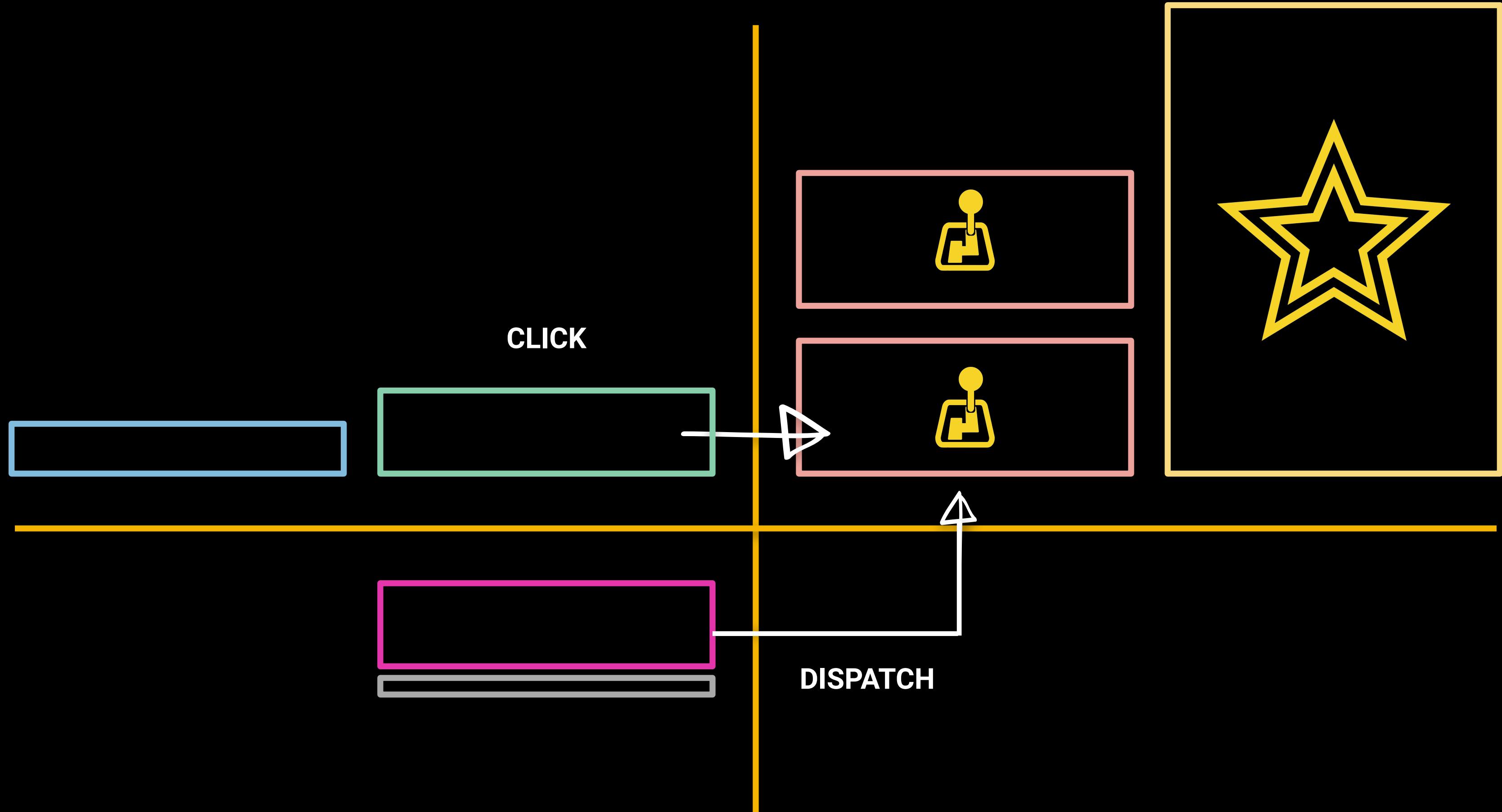








Space: Manual Dispatch

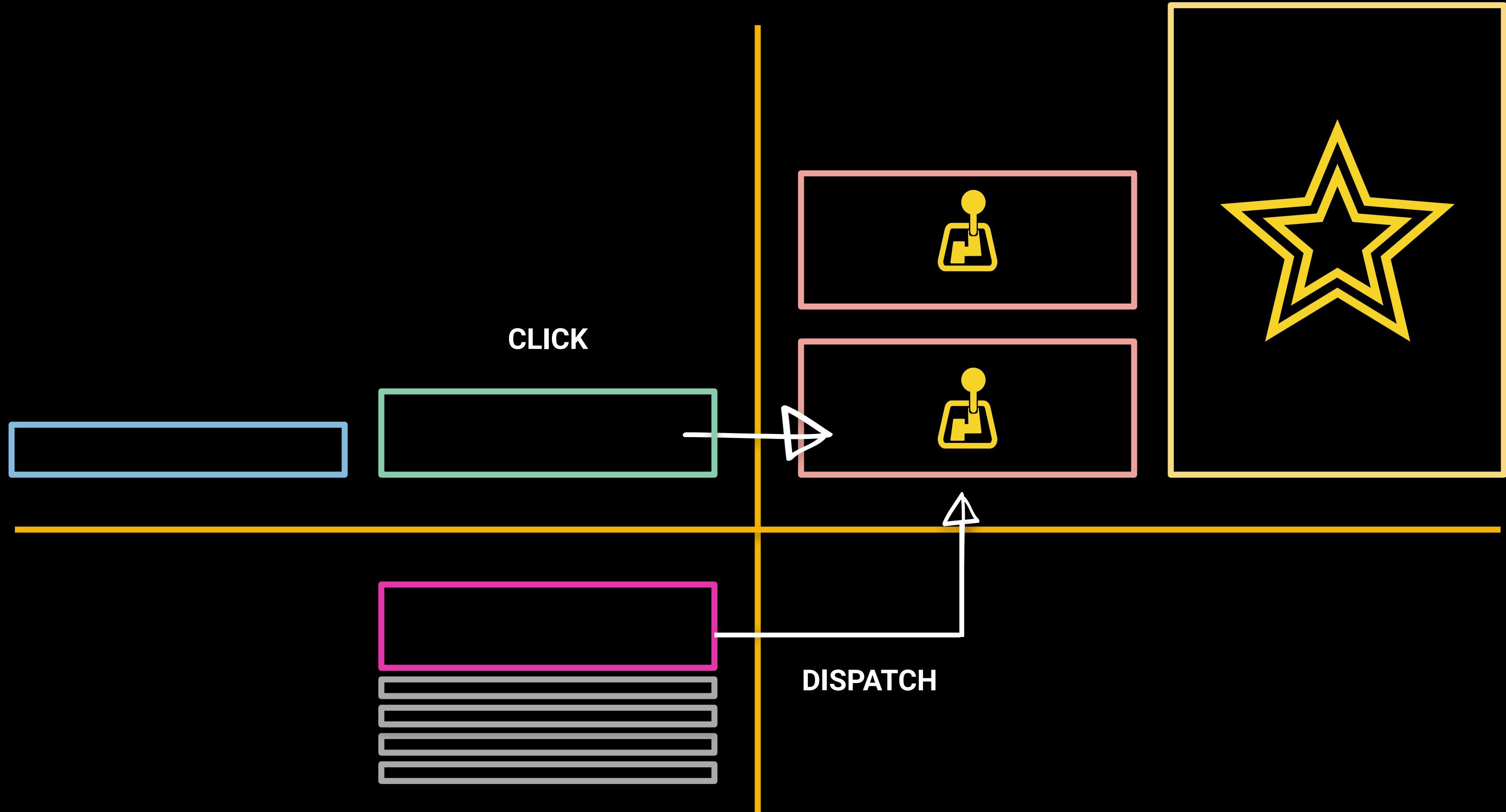


```
actions = [
  {type: '[Client] Select', payload: {id: '1', firstName: 'John', lastName: 'Doe', company: 'Acme, Inc'}},
  {type: '[Client] Select', payload: {id: '2', firstName: 'Jane', lastName: 'Smith', company: 'Super, Inc'}},
  {type: '[Client] Select', payload: {id: '1', firstName: 'John', lastName: 'Doe', company: 'Acme, Inc'}},
  {type: '[Client] Select', payload: {id: '2', firstName: 'Jane', lastName: 'Smith', company: 'Super, Inc'}},
  {type: '[Client] Select', payload: {id: '1', firstName: 'John', lastName: 'Doe', company: 'Acme, Inc'}},
  {type: '[Client] Select', payload: {id: '2', firstName: 'Jane', lastName: 'Smith', company: 'Super, Inc'}},
];
index = 0;

step() {
  this.index = this.index < this.actions.length - 1 ? this.index + 1 : 0;
  this.store.dispatch(this.actions[this.index]);
}
```

Manual Dispatch

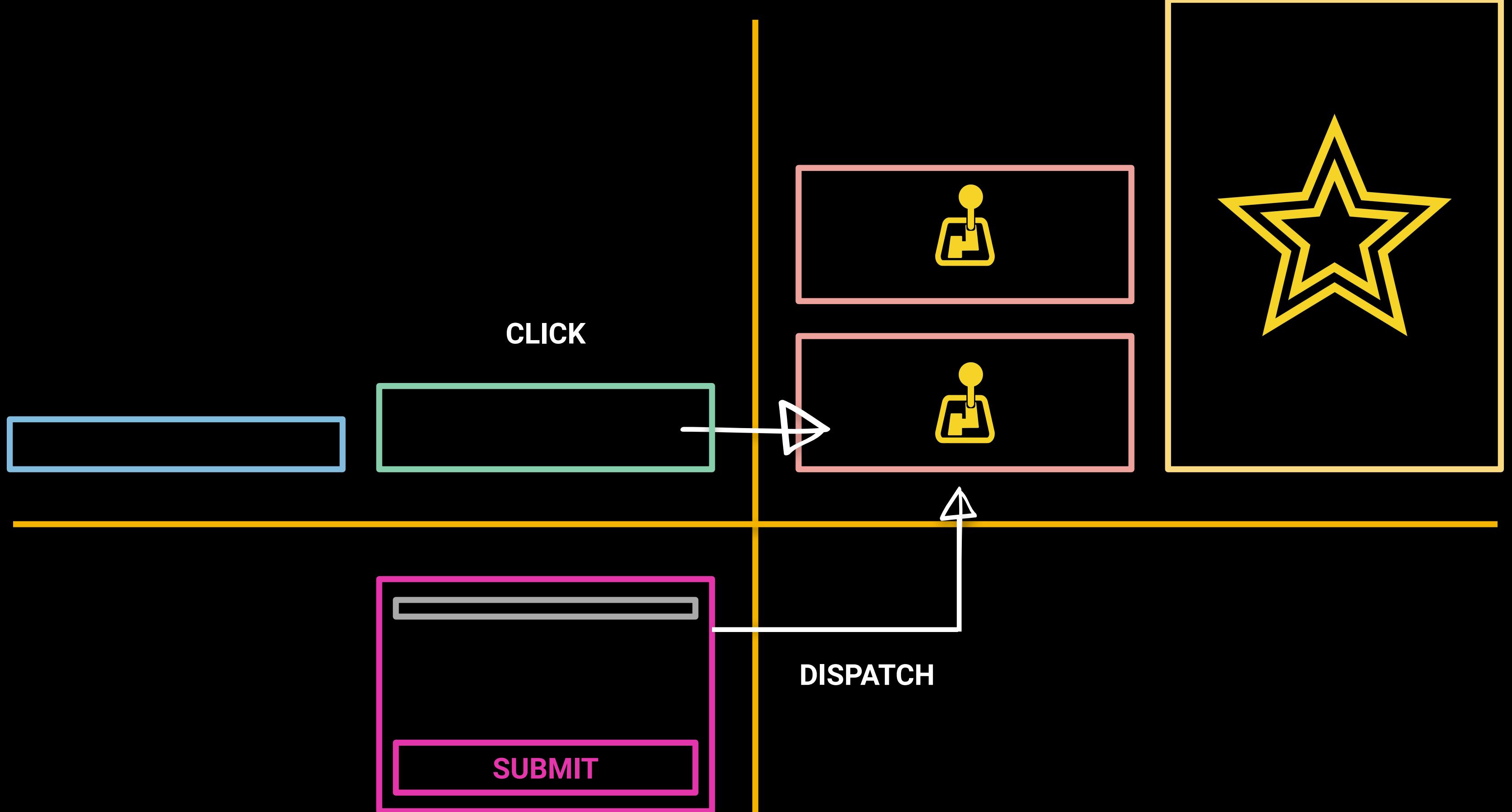
Time: Manual Cycle



```
actions = [
  {type: '[Client] Select', payload: {id: '1', firstName: 'John', lastName: 'Doe', company: 'Acme, Inc'}},
  {type: '[Client] Select', payload: {id: '2', firstName: 'Jane', lastName: 'Smith', company: 'Super, Inc'}},
  {type: '[Client] Select', payload: {id: '1', firstName: 'John', lastName: 'Doe', company: 'Acme, Inc'}},
  {type: '[Client] Select', payload: {id: '2', firstName: 'Jane', lastName: 'Smith', company: 'Super, Inc'}},
  {type: '[Client] Select', payload: {id: '1', firstName: 'John', lastName: 'Doe', company: 'Acme, Inc'}},
  {type: '[Client] Select', payload: {id: '2', firstName: 'Jane', lastName: 'Smith', company: 'Super, Inc'}},
];
cycle() {
  const result = Observable
    .from(this.actions)
    .zip(Observable.interval(this.timerInterval), (a, b) => a)
  ;
  result.subscribe(action => this.store.dispatch(action));
}
```

Manual Cycle

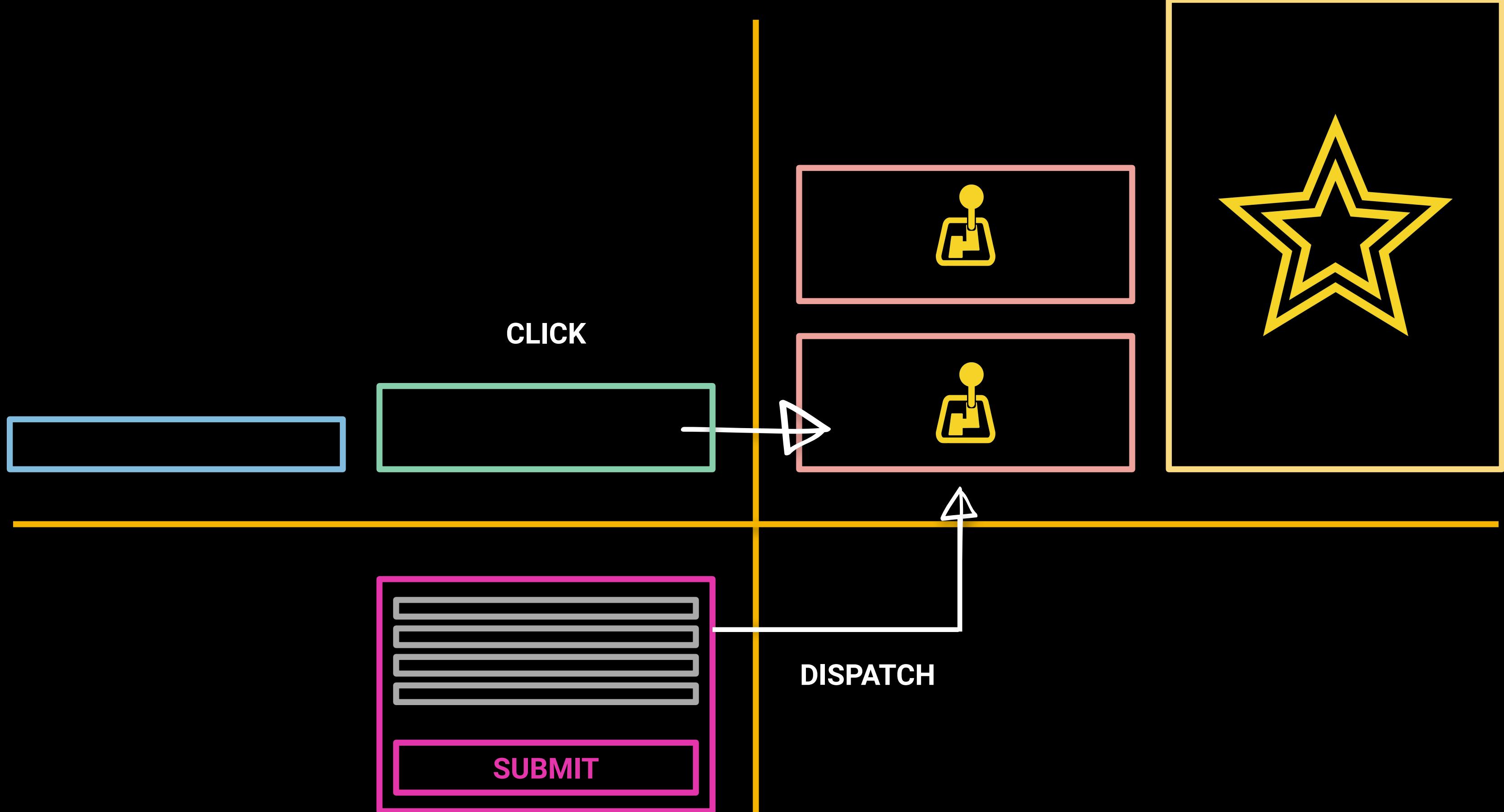
Space: Dynamic Dispatch



```
fetchSingle() {  
    this.actionsService  
        .single()  
        .subscribe(action => this.action = JSON.stringify(action, null , '\t'));  
}  
  
dispatch(action) {  
    this.store.dispatch(JSON.parse(action));  
}
```

Dynamic Dispatch

Time: Dynamic Cycle

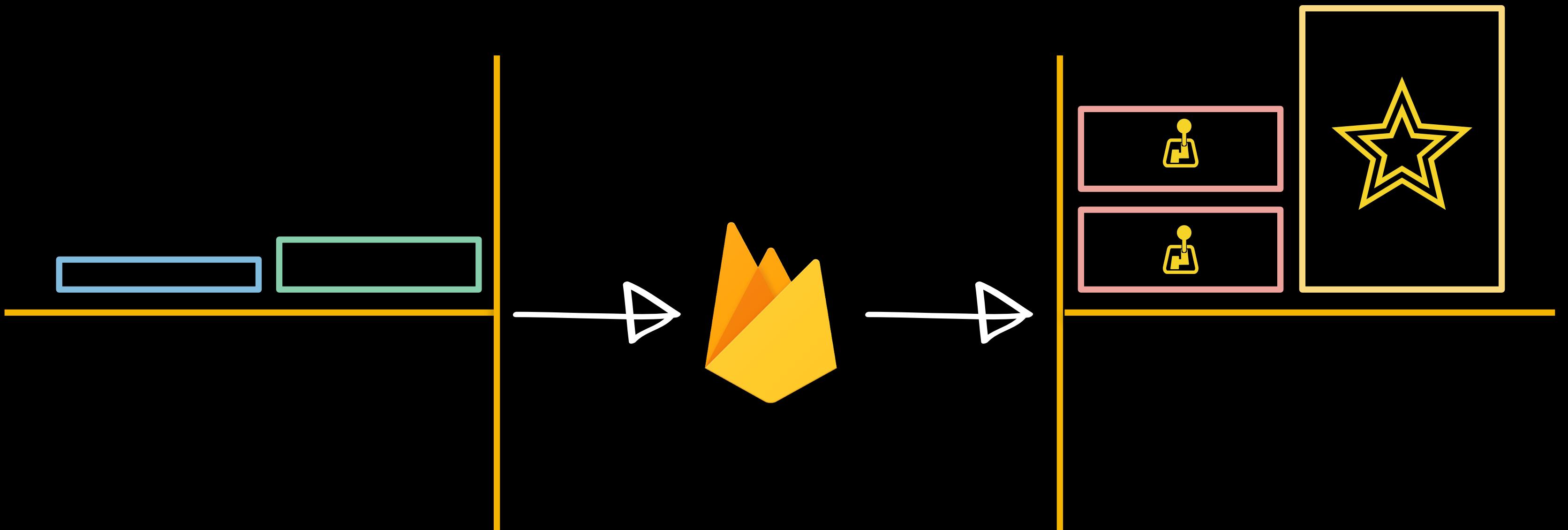


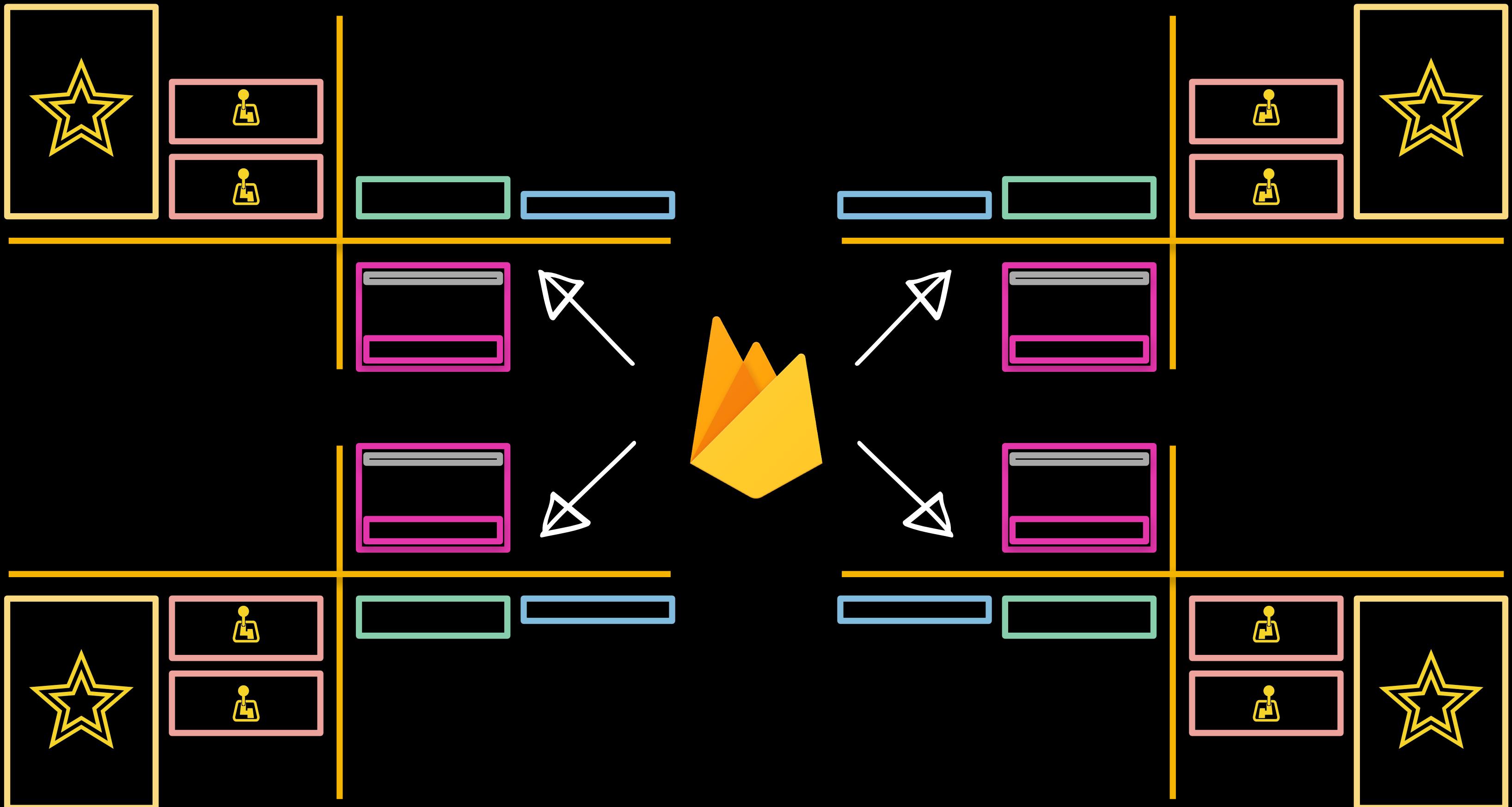
```
fetchAll() {
  this.actionsService
    .all()
    .subscribe(actions => this.rawActions = JSON.stringify(actions, null , '\t'));
}

dispatchCycle(rawActions) {
  const actions = JSON.parse(rawActions);
  const result = Observable
    .from(actions)
    .zip(Observable.interval(this.timerInterval), (a, b) => a)
  ;
  result.subscribe((action: any) => this.store.dispatch(action));
}
```

Dynamic Dispatch

Space: Remote Dispatch





```
constructor(  
  private actionsService: ActionsService,  
  private store: Store<reducers.AppState>,  
  private afs: AngularFirestore  
) {  
  this.remoteActions = afs.collection('actions');  
}  
  
ngOnInit() {  
  this.remoteActions.valueChanges()  
    .skip(1)  
    .subscribe((actions: any) => {  
      this.store.dispatch(actions[0]);  
    });  
}  
  
dispatchRemote(action) {  
  this.remoteActions.add(JSON.parse(action));  
}
```

Remote Dispatch

Time: Undo and Redo

```
export function undoable(reducer: ActionReducer<any>): ActionReducer<any> {
  // Call the reducer with empty action to populate the initial state
  const initialState = {
    past: [],
    present: reducer(undefined, { type: ''}),
    future: []
  };

  // Return a reducer that handles undo and redo
  return function (state = initialState, action) {
    const { past, present, future } = state;

    switch (action.type) {
      case 'UNDO':
        // UNDO LOGIC
      case 'REDO':
        // REDO LOGIC
      default:
        // REGULAR LOGIC
    }
  };
}
```

Meta Reducer

**WAT
DOES IT
MEAN?!**



REAL WORLD APPLICATIONS

Space and Time: Thin Component

```
const CLIENT_LOAD      = '[Client] Load';
const CLIENT_CREATE    = '[Client] Create';
const CLIENT_UPDATE    = '[Client] Update';
const CLIENT_DELETE    = '[Client] Delete';
const CLIENT_SELECT    = '[Client] Select';
const CLIENT_CLEAR     = '[Client] Clear';
```

Actions

```
const reducer = (state = initialState, {type, payload}) => {
  switch (type) {
    case CLIENT_LOAD:
      return state;
    case CLIENT_SELECT:
      return selectClient(state, payload);
    case CLIENT_CREATE:
      return createClient(state, payload);
    case CLIENT_UPDATE:
      return updateClient(state, payload);
    case CLIENT_DELETE:
      return deleteClient(state, payload);
    case CLIENT_CLEAR:
      return clearClient(state, payload);
    default:
      return state;
  }
};

const store = new Store(reducer, initialState);
```

Reducer

```
export class ClientsComponent implements OnInit {
  clients$: Observable<Client[]>;
  currentClient$: Observable<Client>;

  constructor(private store: Store) {
    this.clients$ = this.store.select('clients');
    this.currentClient$ = this.store.select('currentClient');
  }

  ngOnInit() {
    this.store.dispatch(new ClientActions.LoadAction());
    this.resetCurrentClient();
  }

  resetCurrentClient() {
    const newClient: Client = { id: null, firstName: '', lastName: '', company: '' };
    this.store.dispatch(new ClientActions.SelectAction(newClient));
  }
}
```

ClientsComponent

```
class Store {  
  constructor(reducer) {  
    this.reducer = reducer;  
    this.state = reducer(undefined, { type: ''});  
  }  
  
  getState() {  
    return this.state.present;  
  }  
  
  dispatch(action) {  
    this.state = this.reducer(this.state, action);  
  }  
}
```

Store

```
io.on('connection', (socket) => {
  console.log('user connected');
  io.emit('update', store.getState());

  socket.on('dispatch', action => {
    store.dispatch(action);
    io.emit('update', store.getState());
  });

  socket.on('disconnect', function () {
    console.log('user disconnected');
  });
});

http.listen(5000, () => {
  console.log('started on port 5000');
});
```

Socket.io

```
const BASE_URL = 'http://localhost:5000';

export class SocketService {
  private socket;
  private subjects = {};

  constructor() {
    this.socket = io(BASE_URL);
    this.socket.on('update', data => this.next(data));
  }

  select(key) {
    this.subjects[key] = new BehaviorSubject(null);
    return this.subjects[key].asObservable();
  }

  next(data) {
    for (const key in this.subjects) {
      if (this.subjects.hasOwnProperty(key)) {
        this.subjects[key].next(data[key]);
      }
    }
  }

  dispatch(action) {
    this.socket.emit('dispatch', action);
  }
}
```

SocketService

GRAND FINALE!





I  YOU!



@simpulton



Thanks!

BONUS!
Thoughts on
managing complexity



Make it work

Make it right

Make it fast

Make it work

Make it known

Make it right

Make it fast

Code should be **fine**
grained

Code should do one
thing

Code should be self
documenting

Favor pure, immutable
functions

Abstractions should
reduce complexity

Abstractions should
reduce coupling

Abstractions should
increase cohesion

Abstractions should
increase portability

Refactor through
promotion

Composition over inheritance

Do not confuse
convention for
repetition

Well-structured code
will naturally have a
larger surface area

Team rules for
managing complexity

Be mindful over the
limitations of your
entire team and
optimize around that

Favor best practices
over introducing idioms
however clever they
may be

Consistency is better
than righteousness

Follow the style guide
until it doesn't make
sense for your
situation

Tactical rules for
managing complexity

Eliminate hidden state
in functions

Eliminate nested logic
in functions

Do not break the Single
Responsibility
Principle

Extracting to a method
is one of the most
effective refactoring
strategies available

If you need to clarify
your code with
comments then it is
probably too complex

It is *impossible* to write
good tests for bad
code

