sourcemind

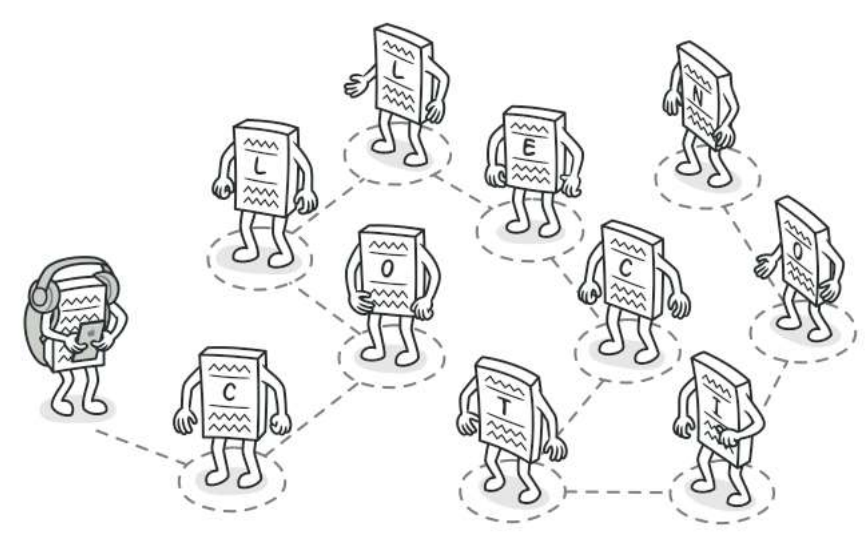
DESIGN PATTERNS

Design Patterns in React

1. Higher-Order Components (HOC): HOC is a pattern where a component wraps another component to add some additional behavior or data. This is useful for reusing logic across multiple components.
2. Render Props: Render props are a way to share code between components using a prop that renders a component. This is useful for implementing reusable logic that can be used across multiple components.
3. Context API: The Context API is a way to pass data down the component tree without having to pass props manually through each level. This is useful for data that needs to be shared across multiple components.
4. Container and Presentational Components: The container and presentational component pattern involves separating components into two categories: containers, which handle data and state, and presentational components, which handle rendering and presentation.
5. State Management Libraries: For larger applications, managing state can become a complex task. State management libraries, such as Redux and MobX, provide a way to manage state in a centralized and organized manner.

Behavioral patterns

1. Iterator



Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

Design Pattern Link

JS Implementation

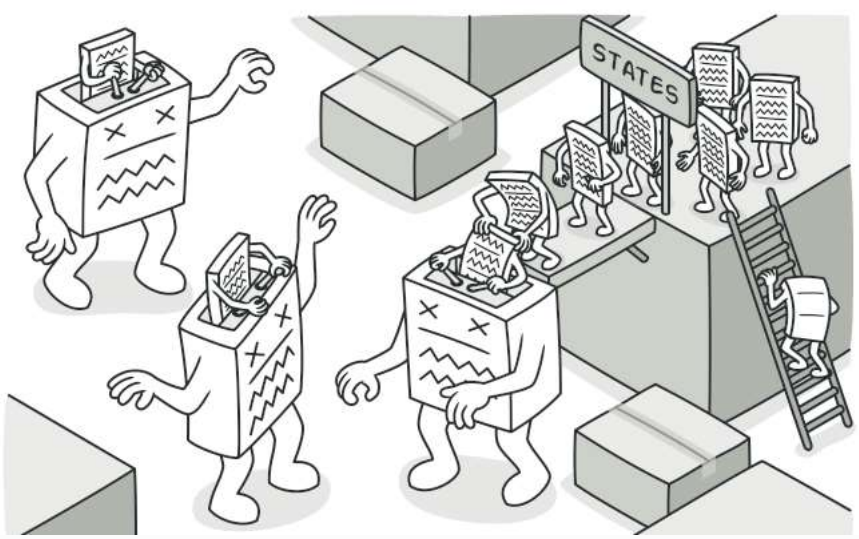
sourcemind

Iterator Design Pattern in React

```
function ItemList({ items }) {
  return (
    <ul>
      {items.map((item, index) => (
        <li key={index}>{item}</li>
      ))}
    </ul>
  );
}
```

Behavioral patterns

2. State



State is a behavioral design pattern that lets an object alter its behavior when its internal state changes.
It appears as if the object changed its class.

Design Pattern Link

JS Implementation

sourcemind

State Design Pattern in React

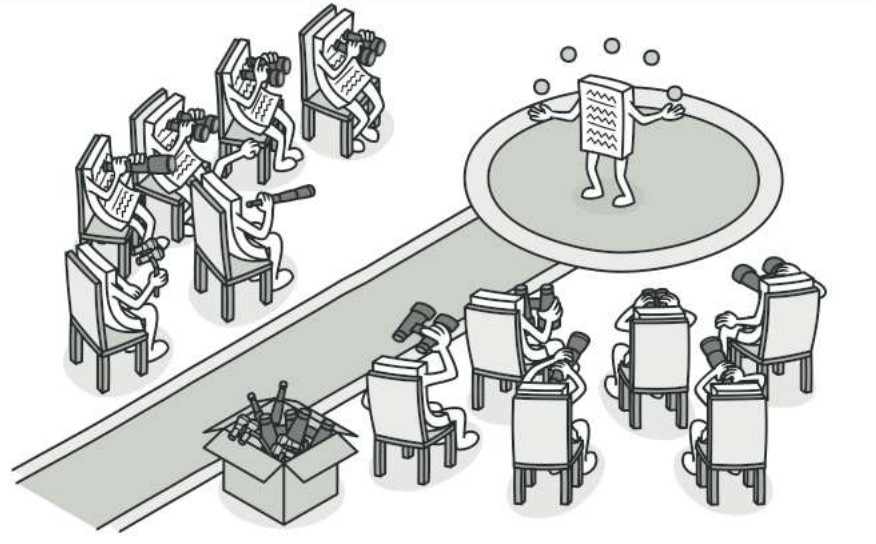
```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  incrementCount = () => {
    this.setState({
      count: this.state.count + 1
    });
  }

  render() {
    return (
      <div>
        <h1>{this.state.count}</h1>
        <button onClick={this.incrementCount}>Increment</button>
      </div>
    );
  }
}
```

The State design pattern is a fundamental concept in React, where components can manage their own state and render their views based on that state. In React, state is an object that hol data that can change over time and trigger re-rendering of the component.

sourcemind

Behavioral patterns

3. Observer



Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Design Pattern Link

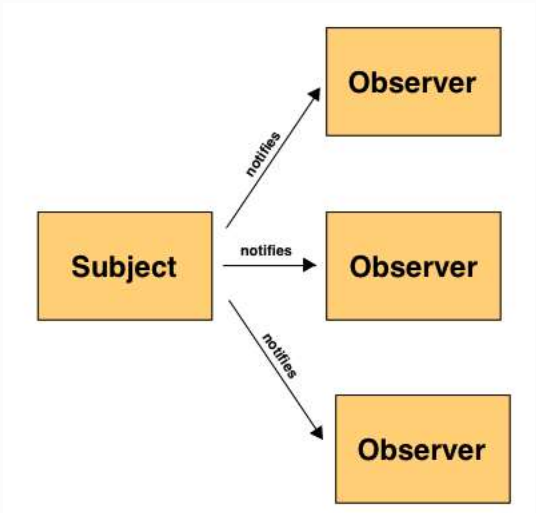
JS Implementation

sourcemind

Observer Design Pattern in React

MobX is a popular library for state management in React applications. It can be used to implement the Observer desi pattern, which involves a subject that is being observed by one or more observers, and the observers are notified whenever the subject changes.

With MobX, you can create an observable store that acts as the subject, and components that observe the store and re-render whenever the store changes.



sourcemind

Example

```
import React from "react"
import ReactDOM from "react-dom"
import { makeAutoObservable } from "mobx"
import { observer } from "mobx-react-lite"

class Timer {
    secondsPassed = 0

    constructor() {
        makeAutoObservable(this)
    }

    increaseTimer() {
        this.secondsPassed += 1
    }
}

const myTimer = new Timer()

// A function component wrapped with `observer` will react
// to any future change in an observable it used before.
const TimerView = observer(({ timer }) => <span>Seconds passed: {timer.secondsPassed}</span>)

ReactDOM.render(<TimerView timer={myTimer} />, document.body)

setInterval(() => {
    myTimer.increaseTimer()
}, 1000)
```

Lazy Loading Design pattern

- Lazy loading is one of the most common design patterns used in web and mobile development. It is widely used with frameworks like Angular and React **to increase an application's performance by reducing initial loading time**.

- In the earlier versions of React, lazy loading was implemented using third-party libraries. However, React introduced two new native features to implement lazy loading with the React v16.6 update.

React.lazy()

```
// Without React.lazy()
import AboutComponent from './AboutComponent ';

// With React.lazy()
const AboutComponent = React.lazy(() => import('./AboutComponent '));

const HomeComponent = () => (
    <div><AboutComponent /></div>
)
```

```
import React, { Suspense } from "react";
const AboutComponent = React.lazy(() => import('./AboutComponent'));

const HomeComponent = () => (
    <div><Suspense fallback = { <div> Please Wait... </div> } >
            <AboutComponent /></Suspense></div>
);
```

- As discussed, lazy loading has many benefits. But overusing it can have a significant negative impact on your applications. So, it is essential to understand when we should use lazy loading and when we should not.
- You should not opt for lazy loading if your application has a small bundle size. There is no point in splitting a small bundle into pieces, and it will only increase coding and configuring effort.
- Also, there are special application types like e-commerce sites that can be negative impacted by lazy loading. For example, users like to scroll through quickly when searching for items. If you lazy load shopping items, it will break the scrolling speed and result in a bad user experience. So, you should analyze the company's website usage before using lazy loading.