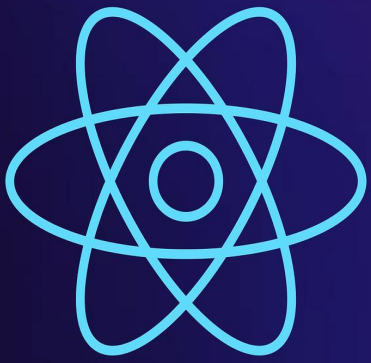
REACT SOLID

Elevating the Quality of Your Front-End Code



By Davi Silva

DAVI SILVA

React Solid

Elevating the Quality of Your Front-End Code

First edition

This book was professionally typeset on Reedsy.
Find out more at reedsy.com

Contents

1	Intro	1
2	Single Responsibility Principle (SRP)	2
3	Open/Closed Principle (OCP)	8
4	Liskov Substitution Principle (LSP)	13
5	Interface Segregation Principle (ISP)	17
6	Dependency Inversion Principle (DIP)	23
About the Author		28

1

Intro

The acronym SOLID refers to five principles in software development. These principles allow us to create flexible, readable, and easier-to-maintain software. In this e-book, we will discuss these principles to demonstrate their practical application in front-end software development using one of the leading libraries in the market—React.

Accessing the Book Code

All examples are available in a public GitHub repository at this link: https://github.com/davi1985/react-solid.

The code is organized into branches corresponding to different principles. For instance, to view the Single Responsibility Principle, switch to the **"01-srp"** branch. Here, you'll find two commits: **"bad implementation"** and **"good implementation."** This setup allows you to compare the two versions and see the differences in the code. Other branches follow the same format.

2

Single Responsibility Principle (SRP)

> *A class should serve a single, clearly defined purpose, reducing the need for frequent changes.*

Let's take a look at the ***ListRepos*** component and analyze its responsibilities:

```
import axios from 'axios'
import { useEffect, useState } from 'react'

type Repo = { id: string; language: string; name:
string }

export const ListRepos = () => {
  const [repos, setRepos] = useState<Repo[]>([])
  const [isLoading, setIsLoading] = useState(true)
  const [error, setError] = useState(false)

  useEffect(() => {
    (async () => {
      try {
```

```
        const { data } = await axios.get(
          'https://api.github.com/users/davi1985/repos'
        )
        setRepos(data)
      } catch {
        setError(true)
      } finally {
        setIsLoading(false)
      }
    })()
  }, [])

  if (error) {
    return <p>Ops, something wrong </p>
  }

  if (isLoading && !error) {
    return <p>Loading data..</p>
  }

  return (
    <main>
      {repos.map(({ id, language, name }) => (
        <div key={id}>
          <h1>{name}</h1>
          <span>{language}</span>
        </div>
      ))}
    </main>
  )
}
```

In the component above, we have data fetching, list rendering, and state management. Now, considering the Single Responsibility Principle (SRP), let's start separating these responsibilities. First, we'll isolate the data fetching into a custom hook:

```
import axios from 'axios'
import { useState, useEffect } from 'react'

type Repo = { id: string; language: string; name:
string }

export const useGetRepos = () => {
  const [repos, setRepos] = useState<Repo[]>([])
  const [isLoading, setIsLoading] = useState(true)
  const [error, setError] = useState(false)

  useEffect(() => {
    (async () => {
      try {
        const { data } = await axios.get(
          'https://api.github.com/users/davi1985/repos'
        )
        setRepos(data)
      } catch {
        setError(true)
      } finally {
        setIsLoading(false)
      }
    })()
  }, [])

  return {
    isLoading,
    error,
    repos,
  }
}
```

Now, our ***ListRepos*** component is cleaner and has fewer responsibilities.

```
import { useGetRepos } from './useGetRepos'

export const ListRepos = () => {
  const { error, isLoading, repos } = useGetRepos()

  if (error) {
    return <p>Ops, something wrong </p>
  }

  if (isLoading && !error) {
    return <p>Loading data..</p>
  }

  return (
    <main>
      {repos.map(({ id, language, name }) => (
        <div key={id}>
          <h1>{name}</h1>
          <span>{language}</span>
        </div>
      ))}
    </main>
  )
}
```

We can further improve this by noting that each item in our list is being rendered directly in the component. Once again, we can separate the responsibilities by creating a new component—***Repo***.

```
type Props = { language: string; name: string }

export const Repo = ({ language, name }: Props) => (
  <div>
    <h1>{name}</h1>
```

```
    <span>{language}</span>
  </div>
)
```

Now, let's incorporate our newly created component:

```
import { Repo } from './Repo'
import { useGetRepos } from './useGetRepos'

export const ListRepos = () => {
  const { error, isLoading, repos } = useGetRepos()

  if (error) {
    return <p>Ops, something wrong </p>
  }

  if (isLoading && !error) {
    return <p>Loading data...</p>
  }

  return (
    <main>
      {repos.map(({ id, language, name }) => (
        <Repo key={id} name={name}
        language={language} />
      ))}
    </main>
  )
}
```

By applying the Single Responsibility Principle, we now have isolated components that are easy to test and maintain, each with a single responsibility. We can continue enhancing our application, and I challenge you to implement the following:

1. The custom hook ***useGetRepos*** should now depend on the ***Axios*** library for fetching data.
2. For better architecture, we can create a types folder to centralize any repeated types throughout the application.

With the first principle complete, let's move on to explore and apply the next one.

3

Open/Closed Principle (OCP)

> *Software entities (classes, modules, functions, etc) should be OPEN for extension and CLOSED for modification.*

When we start studying this principle, we need to observe two important points that the theory shows us:

- Open for extension - it means that after developing a React component, one should avoid directly manipulating its implementation unless it is unavoidable.
- Closed for modification - it means that after developing one React component, we need to avoid manipulating your root implementation directly. Instead, we must extend or add new elements without modifying the code, following this principle, the changes need to avoid the already implemented code.

Let's look at the code below where we have the Button com-

ponent. This component has two variants: 'primary' and 'secondary' and based on one of these properties the styles are modified.

```
type ButtonVariant = 'primary' | 'secondary'

type ButtonProps = { label: string; variant:
ButtonVariant }

export const Button = ({ label, variant }:
ButtonProps) => {
  const mapVariant = {
    primary: 'bg-blue-400 text-gray-100 rounded-full',
    secondary: 'text-blue-500 border rounded-full
    border-blue-500',
  }

  return (
    <button className={`py-2 px-4 shadow-md
    ${mapVariant[variant]}`}>
      {label}
    </button>
  )
}
```

Now, we need to render one optional icon in this component, so we need to do something like this:

```
type ButtonVariant = 'primary' | 'secondary'

type ButtonProps = {
  label: string;
  variant: ButtonVariant
  icon?: string;
}
```

```
export const Button = ({ label, variant, icon }:
ButtonProps) => {
  const mapVariant = {
    primary: 'bg-blue-400 text-gray-100 rounded-full',
    secondary: 'text-blue-500 border rounded-full
    border-blue-500',
  }

  return (
    <button className={`py-2 px-4 shadow-md
    ${mapVariant[variant]}`}>
      {icon && <span className="p-2">{icon}</span>}

      {label}
    </button>
  )
}
```

And now using this component we have this:

```
import { Button } from './Button'

export const App = () => (
  <div className="flex gap-2">
    <Button label="First Button" variant="primary" />
    <Button
      label="Second Button"
      variant="secondary"
      icon=">>>"
     />
  </div>
)
```

In other words, we modified the root implementation to add

a new feature, which doesn't align well with the ***Open/Closed Principle*** (OCP). If the icon is another React component, how would the Button component be rendered? Again, we would need to alter the root code of the component to accommodate this new scenario. However, we want to structure our Button component with both OCP and the Single Responsibility Principle (SRP) in mind. Here's how we can achieve that:

```
import { ButtonHTMLAttributes, ReactNode } from
'react'

type ButtonVariant = 'primary' | 'secondary'

type ButtonProps =
ButtonHTMLAttributes<HTMLButtonElement> & {
  label: string
  variant: ButtonVariant
  children?: ReactNode | string
}

export const Button = ({ label, variant, children,
...rest }: ButtonProps) => {
  const mapVariant = {
    primary: 'bg-blue-400 text-gray-100 rounded-full',
    secondary: 'text-blue-500 border rounded-full
    border-blue-500',
  }

  return (
    <button
      className={`py-2 px-4 shadow-md
      ${mapVariant[variant]}`}
      {...rest}
    >
      {children && <span
      className="p-2">{children}</span>}
```

```
      {label}
    </button>
  )
}
```

Now, we have a button component that is extensible, reusable, and flexible. We utilize ***ButtonHTMLAttributes<HTMLButton Element>*** to ensure the button inherits all the default HTML properties of a button. Additionally, the icon can now be either a ***ReactNode*** or a ***string***, allowing us to pass in an icon as an emoji or any other React element from an icon library.

In conclusion, by adhering to the Open/Closed Principle and the Single Responsibility Principle, we've transformed our Button component into a more robust and adaptable solution. This design allows for seamless integration of new features, such as icons, without altering the core implementation.

4

Liskov Substitution Principle (LSP)

> *Objects of a subclass can replace objects of a superclass without affecting the program's correctness*

While the name may seem complex, the concept is straightforward: subclasses of a component should be able to substitute the parent component seamlessly, ensuring that no existing functionality breaks.

For instance, if ***PasswordInput*** is a subclass of ***TextInput***, we should be able to replace instances of ***TextInput*** with ***PasswordInput*** without any unexpected behavior. This implies that ***PasswordInput*** must fulfill all the requirements and expected behaviors defined by the ***TextInput*** component.

Let's look at the simple Input component below and your use:

```
type InputProps = { type: string; placeholder: string
}

export const Input = ({ type, placeholder }:
InputProps) => (
  <input
    className="bg-gray-200 py-2 px-4 rounded-md"
    type={type}
    placeholder={placeholder}
  />
)
```

In the accompanying App component, the Input component works as intended:

```
import { Input } from './Input'

export const App = () => (
  <div className="flex gap-2">
    <Input type="text" placeholder="Write some text"
    />
    <Input type="number" placeholder="Write some
    number" />
  </div>
)
```

While this implementation of the Input component is functional, it lacks flexibility. There's no straightforward way to extend or modify its behavior without altering the root implementation. To address this, let's rewrite the component while adhering to the LSP:

```
import { InputHTMLAttributes } from 'react'
```

```
type InputProps =
InputHTMLAttributes<HTMLInputElement>

export const Input = ({ placeholder, ...props }:
InputProps) => (
  <input
    className="bg-gray-200 py-2 px-4 rounded-md"
    placeholder={placeholder}
    {...props}
  />
)

export const TextInput = ({ ...props }: InputProps)
=> (
  <input
    {...props}
    className="bg-gray-200 py-2 px-4 rounded-md"
  />
)

export const Password = ({ ...props }: InputProps) =>
(
  <input
    {...props}
    className="bg-gray-200 py-2 px-4 rounded-md"
  />
)
```

Now, the Input component is effectively isolated. By leveraging the ***InputHTMLAttributes<HTMLInputElement>*** type, any other input component that extends this type inherits all the properties and behaviors of a standard input element. This design is made possible through the implementation of the Liskov Substitution Principle, which promotes code reusability and maintainability.

In conclusion, the Liskov Substitution Principle enhances the extensibility of our components while maintaining their core functionality. By ensuring that subclasses can replace their parent classes without issues, we create a more robust and flexible codebase. This approach not only simplifies future modifications but also fosters a more organized structure in our React applications. Embracing LSP leads to better design practices, ultimately resulting in cleaner, more maintainable code.

5

Interface Segregation Principle (ISP)

> *No code should be forced to depend on methods it does not use.*

This principle emphasizes the importance of avoiding large interfaces that encompass numerous methods or values. Instead, we should aim to create smaller, more focused interfaces tailored to the specific needs of the functions or classes that utilize them.

Let's look at the dashboard component below:

```
export type Widget = {
  type: 'chart' | 'table'
  data: any
  title?: string
  columns?: string[]
  rows?: string[][]
}
```

```
type ChartProps = { data: number[]; title: string }
type TableProps = { columns: string[]; rows:
string[][] }
type DashboardProps = { widgets: Widget[] }

const Chart = ({ title, data }: ChartProps) => (
  <div>
    <h2>{title}</h2>
    <div>{JSON.stringify(data)}</div>
  </div>
)

const Table = ({ columns, rows }: TableProps) => (
  <table>
    <thead>
      <tr>
        {columns.map((col) => (
          <th>{col}</th>
        ))}
      </tr>
    </thead>

    <tbody>
      {rows.map((row, index) => (
        <tr key={index}>
          {row.map((cell, cellIndex) => (
            <td key={cellIndex}>{cell}</td>
          ))}
        </tr>
      ))}
    </tbody>
  </table>
)

export const Dashboard = ({ widgets }:
DashboardProps) => (
  <div>
```

```
    {widgets.map(({ title, data, type, columns, rows
    }, index) => {
      if (type === 'chart') {
        return <Chart
                 key={index}
                 data={data as number[]}
                 title={title!}
                />
      }

      return <Table
               key={index}
               columns={columns!} rows={rows!}
              />
    })}
  </div>
)
```

Wow, this code is a little confusing, we have an unnecessary coupling and hard maintenance, but look at the uses before starting with refactoring guidance by ISP.

```
import { Dashboard, Widget } from './Chart'

export const App = () => {
  const widgets: Widget[] = [
    {
      type: 'chart',
      data: [10, 20, 30, 40],
      title: 'Sample Chart',
    },
    {
      data: [],
      type: 'table',
      columns: ['Name', 'Age', 'City'],
      rows: [
```

```
        ['Alice', '30', 'New York'],
        ['Bob', '25', 'Los Angeles'],
        ['Charlie', '35', 'Chicago'],
      ],
    },
  ]

  return <Dashboard widgets={widgets} />
}
```

So, first of all, let's separate into small parts, the ***ChartWidget*** and ***TableWidget*** types only include properties relevant to their respective components. The Dashboard component can handle an array of widgets without unnecessary properties for each widget type.

```
export type ChartWidget = {
  type: 'chart'
  data: number[]
  title: string
}

export type TableWidget = {
  type: 'table'
  columns: string[]
  rows: string[][]
}

export type Widget = ChartWidget | TableWidget
type DashboardProps = { widgets: Widget[] }

const Chart = ({ title, data }: ChartWidget) => (
  <div>
    <h2>{title}</h2>
    <div>{JSON.stringify(data)}</div>
```

```
  </div>
)

const Table = ({ columns, rows }: TableWidget) => (
  <table>
    <thead>
      <tr>
        {columns.map((col) => (
          <th>{col}</th>
        ))}
      </tr>
    </thead>

    <tbody>
      {rows.map((row, index) => (
        <tr key={index}>
          {row.map((cell, cellIndex) => (
            <td key={cellIndex}>{cell}</td>
          ))}
        </tr>
      ))}
    </tbody>
  </table>
)

export const Dashboard = ({ widgets }:
DashboardProps) => (
  <div>
    {widgets.map((widget, index) => {
      if (widget.type === 'chart') {
        return <Chart key={index} {...widget} />
      }

      return <Table key={index} {...widget} />
    })}
  </div>
)
```

With this setup, you have a clear, modular design adhering to the ISP. Each widget type is responsible for only its properties, leading to a more maintainable and understandable codebase.

6

Dependency Inversion Principle (DIP)

> *1. High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces).*
>
> *2. Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.*

The Dependency Inversion Principle (DIP) in React emphasizes that high-level components should not directly depend on low-level components. Instead, both should rely on common abstractions, such as props or context. This approach promotes flexibility, modularity, and maintainability while simplifying unit testing by allowing easy mockups of low-level components.

By adhering to this principle, developers can create a scalable architecture with a clear separation of concerns. In this context, "component" can refer to any part of the application, including

React components, functions, modules, class-based components, or third-party libraries. Let's explore an example.

Consider the following ***LoginForm*** component:

```
import { FormEvent, useState } from 'react'
import axios from 'axios'

export const LoginForm = () => {
  const [name, setName] = useState('')
  const [email, setEmail] = useState('')

  const handleSubmit = async (ev: FormEvent) => {
    ev.preventDefault()

    try {
      const formData = new FormData(ev.currentTarget)
      await axios.post('https://myapi.com/login',
      formData)
    } catch (error) {
      console.log(error)
    }
  }

  return (
    <form onSubmit={handleSubmit}>
      <div>
        <label>Name:</label>
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
          required
        />
      </div>
```

```
      <div>
        <label>Email:</label>
        <input
          type="email"
          value={email}
          onChange={(e) => setEmail(e.target.value)}
          required
        />
      </div>

      <button type="submit">Submit</button>
    </form>
  )
}
```

In this implementation, the form directly handles user login data. If we want to change the submission behavior, we need to modify the form itself. Additionally, testing the form in isolation becomes challenging because the submission logic is embedded in the component.

To refactor this, we can create a separate Form component that inverts the dependency:

```
import { FormEvent, useState } from 'react'

type FormProps = {
  onSubmit(ev: FormEvent): Promise<void>
}

const Form = ({ onSubmit }: FormProps) => {
  const [name, setName] = useState('')
  const [email, setEmail] = useState('')
```

```
  return (
    <form onSubmit={onSubmit}>
      <div>
        <label>Name:</label>
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
          required
        />
      </div>

      <div>
        <label>Email:</label>
        <input
          type="email"
          value={email}
          onChange={(e) => setEmail(e.target.value)}
          required
        />
      </div>

      <button type="submit">Submit</button>
    </form>
  )
}

export const LoginForm = () => {
  const handleSubmit = async (ev: FormEvent) => {
    ev.preventDefault()

    try {
      const formData = new FormData(e.currentTarget)
      await axios.post('https://myapi.com/login',
      formData)
    } catch (err) {
      console.error(err.message)
```

```
    }
  }

  return <Form onSubmit={handleSubmit} />
}
```

In this refactored version, we created an isolated Form component that accepts a submission handler as a prop, effectively inverting the dependencies. This change allows us to test each component independently without worrying about unintended side effects, as they are no longer tightly coupled.

By applying the Dependency Inversion Principle, we enhance our code's maintainability and testability.

About the Author

With four years of experience in technology, I currently work as a software engineer at Midway, specializing in React, TypeScript, and JavaScript. I made a career transition at 35, and since then, I have truly found my passion as a software development professional.

Philippians 4:13, "I can do all things through Christ who strengthens me".

You can connect with me on:

- https://www.linkedin.com/in/davisilva85
- https://github.com/davi1985