Mohammad Faisal
Apr 20, 2021 · 4 min read ★ · Listen

Applying the Liskov Substitution Principle in React

A look at the SOLID principles in action

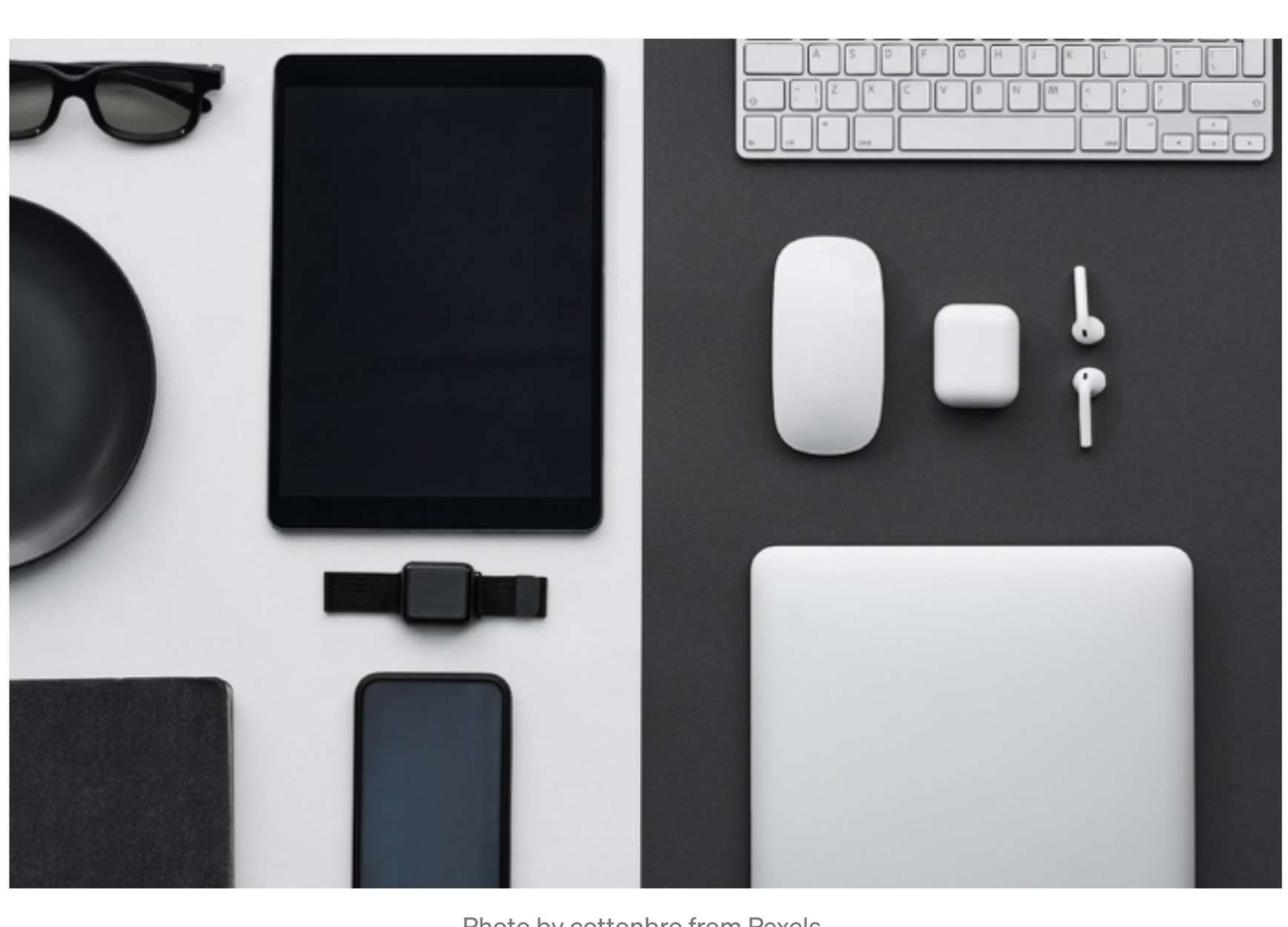


Photo by cottonbro from Pexels.

SOLID is a set of principles that are used as guidelines for creating a clean and maintainable application that is also less buggy and error-prone.

Today, we will take a deep dive into the third principle of SOLID: the Liskov Substitution Principle. We will try to understand how this principle can help us to create a better and cleaner React application.

· · ·

Other Articles in this Series

1. Single Responsibility Principle
2. Open Closed Principle
3. Interface Segregation Principle
4. Dependency Inversion Principle

· · ·

What Is the Liskov Substitution Principle?

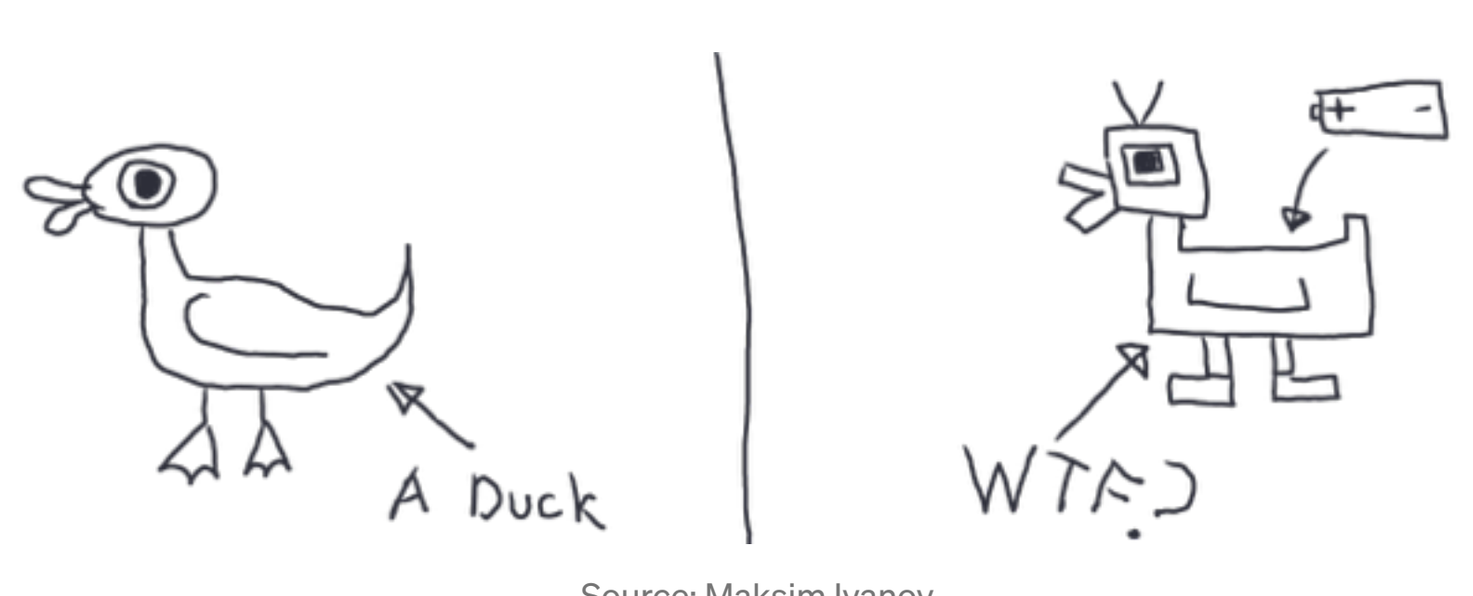
In simple terms, this principle says:

> "Subclasses should be substitutable for their superclasses."

That means subclasses of a particular class should be able to replace the superclass without breaking any functionality.

Example

If `PlasticDuck` is a subclass of `Duck`, then we should be able to replace instances of `Duck` with `PlasticDuck` without any surprises.



Source: Maksim Ivanov

That means `PlasticDuck` should fulfill all the expectations set by the `Duck` class.

· · ·

What Does This Mean in React?

React is not an object-oriented framework because it's basically JavaScript. In the context of React, the main idea behind this principle is:

> "Components should abide by some kind of contract."

At its core, this means there should be some kind of contract between components. So whenever a component uses another component, it shouldn't break its functionality (or create any surprises).

· · ·

Let's Take a Deeper Dive

Let's take a `ModalHolder` component. This component takes `contentToShow` as a prop and shows it inside a modal:

```jsx
import {useState} from "react";
import Modal from 'react-modal';

export const ModalHolder = ({contentToShow}) => {

    const [visibility , setVisibility] = useState(false);

    return <>
        <button onClick={() => setVisibility(true)}> Show Modal</button>

        <Modal isOpen={visibility}>
           <div>{contentToShow}</div>
        </Modal>
    </>
}
```

ModalHolder.jsx hosted with ❤ by GitHub — view raw

ModalHolder.jsx

What's the issue here?

Well, the problem is now there are no restrictions on what can be passed into the `ModalHolder` component. Absolutely anything can be passed into this through the variable `contentToShow`.

First, let's check if our code works and everything goes as expected:

```jsx
import React  , {useEffect}from'react';
import {ModalHolder} from "./views/liskov-substitution-principle/ModalHolder";

function App() {

  const modalContent = <div> This is shown inside modal </div>

  return (
    <div>
        <ModalHolder contentToShow = {modalContent} />
    </div>
  );
}

export default App;
```

App.jsx hosted with ❤ by GitHub — view raw

App.jsx

Now if you open the modal, it will work just fine and show you the modal:

Show Modal

Let's take advantage of the flaw we described earlier and see how it can destroy our application.

Let's try to pass an object into the `ModalHolder` and see what happens:

```jsx
import React  , {useEffect}from'react';
import {ModalHolder} from "./views/liskov-substitution-principle/ModalHolder";

function App() {

  const modalContent = { key:" value" }

  return (
    <div>
        <ModalHolder contentToShow = {modalContent} />
    </div>
  );
}

export default App;
```

App.jsx hosted with ❤ by GitHub — view raw

App.jsx

This code is perfectly fine and will give no compilation error. Now let's open our application and see what happens if we click on the button:

```
▶Uncaught Error: Objects are not   react-dom.development.js:13231
valid as a React child (found: object with keys {}). If you meant
to render a collection of children, use an array instead.
    at throwOnInvalidObjectType (react-dom.development.js:13231)
    at reconcileChildFibers (react-dom.development.js:14133)
    at reconcileChildren (react-dom.development.js:16990)
    at updateHostComponent (react-dom.development.js:17632)
    at beginWork (react-dom.development.js:19080)
    at HTMLUnknownElement.callCallback (react-dom.development.js:3
945)
    at Object.invokeGuardedCallbackDev (react-dom.development.js:3
```

Error

So our application is crashing even though our code has no error. What went wrong here?

Our `Modal` component is allowed to contain another React component. But other components are not bonded to follow that because there is no contract.

· · ·

What's the Solution?

Now we will see the importance of using TypeScript in our application and why it's important. Let's refactor our `ModalHolder` component to TypeScript and see what happens:

```tsx
import { ReactElement, useState } from 'react'
import Modal from 'react-modal'

interface ModalHolderProps {
    contentToShow: JSX.Element
}

export const ModalHolder = ({ contentToShow }: ModalHolderProps) => {
    const [visibility, setVisibility] = useState(false)

    return (
        <>
            <button onClick={() => setVisibility(true)}> Show Modal</button>

            <Modal isOpen={visibility}>
                <div>{contentToShow}</div>
            </Modal>
        </>
    )
}
```

ModalHolder.tsx hosted with ❤ by GitHub — view raw

ModalHolder.tsx

So now we have refactored our component to accept the prop `contentToShow` only when it gets a `JSX.Element`.

If someone wants to pass anything that's not a valid component to render, we will get an error:

```tsx
return (
    <div>
        <ModalHolder contentToShow={<div>This is valid</div>} />

        <ModalHolder contentToShow={{ key: 'invalidAsChildren' }} />
    </div>
)
```

```
TS2739: Type '{ key: string; }' is missing the following properties from type
'Element': type, props
```

Valid usage of ModalHolder

Voila! Now all other components that want to plug into the `ModalHolder` component need to follow a contract so that they don't create any unexpected behavior.

· · ·

Did We Do It?

We have designed our `ModalHolder` component in such a way that no child component that uses this component is able to create any unexpected behavior because they must abide by the rules set by the parent.

That's exactly what **Liskov Substitution Principle** is all about.

So yes, we did it!

· · ·

I hope you enjoyed this article as much I enjoyed writing it.

Leave your thoughts below. And have a Great Day :D

Have something to say? Get in touch with me via LinkedIn