Clean Code vs. Dirty Code: React Best Practices .

Part 1

Day 81/100 .

What is clean code, and why do I care?

Clean code is a consistent style of programming that makes your code easier to write, read, and maintain. Often a developer spends time on a problem, and once the problem is solved, they make a pull request. I contend that you aren't done just because your code "works." Now is your chance to clean it up by removing dead code (zombie code), refactoring, and removing any commented-out code! as you Strive for maintainability. Ask yourself, "Will someone else be able to understand this code six months from now?"

In simpler terms, write code that you would be proud to take home and show your mother. LOL.

Why do you care? Because if you're a good developer, you're lazy. Hear me out — I mean that as a compliment. A good developer, when faced with a situation where they have to do something more than once, will generally find an automated (or better) solution to complete the task at hand. So because you're lazy, subscribing to clean-code techniques will decrease the frequency of changes from pull-request code reviews and the need to come back to the same piece of code over and over.

Clean code passes the "smell test"

Clean code should pass the smell test. What do I mean by that? We've all looked at code (our own or others') and said, "Something's not quite right here." Remember, if it doesn't feel right, it probably isn't. Code that's well thought out just comes together. If it feels like you're trying to fit a square peg into a round hole, then pause, step back, and take a break. Nine times out of 10, you'll come up with a better solution.

Clean code is DRY.

DRY is an acronym that stands for "Don't Repeat Yourself." If you are doing the same thing in multiple places, consolidate the duplicate code. If you see patterns in your code, that is an indication it is prime for DRYing. Sometimes this means standing back from the screen until you can't read the text and literally looking for patterns.

Example 1: Dirty Code

```
// Dirty Code
const MyComponent = () => (
  <div>
    <OtherComponent type="a" className="colorful" foo={123}
bar={456} />
    <OtherComponent type="b" className="colorful" foo={123}
bar={456} />
  </div>
);
```

Example 2: Clean & DRY code

```
//DRY/Clean code
const MyOtherComponent = ({ type }) => (
  <OtherComponent type={type} className="colorful" foo={123}
bar={456} />
);
const MyComponent = () => (
  <div>
    <MyOtherComponent type="a" />
    <MyOtherComponent type="b" />
  </div>
);
```

Note That: Sometimes (as in our examples above) DRYing your code may actually increase code size. However, DRYing your code also generally improves maintainability.

Be warned that it's possible to go too far with DRYing up your code, so know when to say when.

Clean code is predictable and testable

Writing unit tests is not just a good idea, it's become almost mandatory. After all, how can you be sure that your latest shiny new feature didn't introduce a bug somewhere else?

Many React developers rely on Jest for a zero-configuration test runner and to produce code coverage reports.

Clean code is self-commenting.

Has this happened to you before? You wrote some code and made sure that it was fully commented. As will happen, you found a bug, so you went back and changed the code. Did you remember to change your comments as well to reflect the new logic? Maybe. Maybe not. The next person who looked at your code then may have gone down a rabbit hole because they focused on the comments.

Add comments only to explain complex thoughts; that is, don't comment on the obvious. Fewer comments also reduces visual clutter.

```
// Dirty
const fetchUser = (id) => (
  fetch(buildUri`/users/${id}`) // Get User DTO record from REST API
    .then(convertFormat) // Convert to snakeCase
    .then(validateUser) // Make sure the the user is valid
);
```

In the clean version, we rename some of the functions to better describe what they do, thus eliminating the need for comments and reducing visual clutter. This limits the potential confusion of the code not matching the comments later.

```
// Clean
const fetchUser = (id) => (
  fetch(buildUri`/users/${id}`)
    .then(snakeToCamelCase)
    .then(validateUser)
);
```

Naming things

We should all give serious thought to variable names, function names, and even filenames.

Here are a few guidelines:

- Boolean variables, or functions that return a boolean value, should start with "is," "has" or "should."

```
// Dirty
const done = current >= goal;
```

```
// Clean
const isComplete = current >= goal;
```

– Functions should be named for what they do, not how they do it. In other words, don't expose details of the implementation in the name. Why? Because how you do it may change some day, and you shouldn't need to refactor your consuming code because of it. For example, you may load your config from a REST API today, but you may decide to bake it into the JavaScript tomorrow.

```
// Dirty
const loadConfigFromServer = () => {
  ...
};
```

```
// Clean
const loadConfig = () => {
  ...
};
```

Clean code follows proven design patterns and best practices.

Computers have been around a long time. Throughout the years, programmers discovered patterns in the way they solved certain problems. These are called design patterns. In other words, there are algorithms that have been proved over time to work, so you should stand on the shoulders of those who preceded you so that you don't have to make the same mistakes.

Then there are best practices. They are similar to design patterns but broader, not specific to a coding algorithm. They might cover things like, "You should lint your code" or "When writing a library package, include React as a peerDependency."

Here are some best practices to follow when architecting your React applications.

- Use small functions, each with a single responsibility. This is called the single responsibility principle. Ensure that each function does one job and does it well. This could mean breaking up complex components into many smaller ones. This also will lead to better testability.
- Be on the lookout for leaky abstractions. In other words, don't impose your internal requirements on consumers of your code.

- Follow strict linting rules. This will help you write clean, consistent code.

Clean code doesn't (necessarily) take longer to write.

I hear the argument all the time that writing clean code will slow productivity. That's a bunch of hooey. Yes, initially you may need to slow down before you can speed up, but eventually your pace will increase as you are writing fewer lines of code.

And don't discount the "rewrite factor" and time spent fixing comments from code reviews. If you break your code into small modules, each with a single responsibility, it's likely that you'll never have to touch most modules again. There is time saved in "write it and forget it."

Part 2: Practical examples of dirty code vs. clean code