



Clean Code

or think twice your next PR comment

Frontend Junior Program - 2022

CONFIDENTIAL | © 2022 EPAM Systems, Inc.

Does it run? Just leave it alone.



Writing Code that
Nobody Else Can Read

The Definitive Guide

ONLY?

@ThePracticalDev

A story of a Pull Request

In medias res

Let us start with an **imaginary situation**. Never happened, never will, still, maybe it will be useful to do so.

The story is about a small PR (**Pull Request**) and a comment, which was targeted this change:

“Instead of adding a new parameter, please introduce an object, and encapsulate all parameters into that.”

In future, there will be more parameters here, so it is cleaner to use an object here, instead.”

Let's sink this comment for a moment...

```
function getClubRules(  
    ruleSet: RuleSet,  
    clubType: string) {  
  
    // business logic here  
}
```



```
function getClubRules(  
    ruleSet: RuleSet,  
    clubType: string,  
    location: string) {  
  
    // business logic here  
}
```

just a **new parameter** has been added

The Proposal

Well, the colleague is right...

Sort of. When you have many-many parameters, and some of them (in the middle of the parameter list) could be optional, then things could go out of hand very easily.

```
const rules = getClubRules(  
  fightClubSet,  
  "FIGHT",  
  null,  
  null,  
  "UK",  
  null,  
  "Whoa, wait, this is crazy."  
);
```

the object of fears - more precisely, being *not an object...*

```
const rules = getClubRules({  
  ruleSet: fightClubSet,  
  clubType: "FIGHT",  
  location: "UK",  
  concern: "Whoa, wait, this is crazy."  
});
```

the proposal

this seems a reasonable and subtle change...

The “Interface”

But. And there is always a “but”...

Using **TypeScript**, you must create an **interface for that parameter object** (to avoid using *any*), something like this.

Rightly so. However. What is this *ClubRules*?

Is there any “*thing*” such as *ClubRules*?

Or is it just an arbitrary collection of random identifiers tangled together merely for the reason to **provide something as an interface**?



```
interface ClubRules {  
  ruleSet: string[];  
  clubType: string;  
  _foo?: FooBar;  
  _bar?: FooBar;  
  location: string;  
  _baz?: FooBar;  
  concern: string;  
};
```

this is not a real interface, just humble homies hanging here under the traffic sign: “interface”

```
function getClubRules(clubRules: ClubRules) {  
  
  const rules = getClubRules({  
    ruleSet: fightClubSet,  
    clubType: "FIGHT",  
    location: "UK",  
    concern: "Whoa, wait, this is crazy."  
  });  
}
```



*If so, we’ve just **introduced an added complexity** to the system.*



Systems are complex, by nature

And that's a [good thing](#). Building complex systems requires knowledge, imagination and competence. It is a wonderful being part of that*. Also, this is what we are paid for. Not anyone is capable of that: you need to work hard to acquire the necessary skillset.

**“An organism's astonishing gift of concentrating a stream of [order on itself](#) and thus escaping the decay into atomic chaos”*



But we should avoid of the additional complexity

Additional === unnecessary. We've already learnt that in Front-end (and in software development in general), even small changes can complicate things so much, that it could be very hard (if possible, at all) to handle. Writing clean code means developing a system from any unnecessary complexity.

So, we have *one* new interface now

The main problem with unclean code is that it is contagious: shortly, you will have *dozens* of similar interfaces. Obviously, you'd need to move these interfaces to a separate file.

a-collection-of-interfaces-which-we-really-use-only-once-and-now-we-have-this-thingy-to-poke-around.ts

a humble proposal for the filename

You may think, that this file name is just a joke

But really, it isn't. If this is the only option for a name that actually makes sense here, then *that is a problem*. A simpler file name, such as *interfaces.ts* would not work, because we do need that for the real interfaces. Also, you cannot mix them, otherwise how would you know the difference?

Meaningful names: the first rule.

Think about proper names


If you are unable to name something meaningfully, then it could be the clear sign of that you are doing something wrong*. The names must be descriptive and explicit.

We use interfaces to **encapsulate*** entities – which are tightly coupled – into one box

The reason is that those entities together form an independent unit, therefore we can handle them as one thing. This way we **decrease the outer complexity**, as we hide the complexity.

do we know what is **inside**? who cares!

what we think now, that it is **one thing**.
but in this case – it is not. It is many things, but we cannot realize that: this interface is lying, and we **cannot reason about** that.



```
function getClubRules(clubRules: ClubRules) {  
}
```

This interface is an **internal implementation detail**, a **technical addition**, a **workaround**. If we must have one, we should name it as **getClubRulesParams**.

It is still wrong to have, but at least it explicitly says that.

**encapsulating, that is the second*

Always encapsulate things which are tightly coupled

If you don't, soon you'll have a lot of things to consider at the same time, because you will see everything. And everything is too much.

Never encapsulate things which are not common

If you do, important things will be hidden. Hidden in a wrong place.

Everything seems good

until we call it. When passing an object to a function, there is temptation to create a variable for that – rightly so, because if that is a thing, we'd need to initialize that.

```
const rules = getClubRules({  
  ruleSet: fightClubSet,  
  clubType: "FIGHT",  
  location: "UK",  
  concern: "Whoa, wait, this is crazy."  
});
```

*"If this is an object, why not extract that?
In **future***, if we need to call it again, we
could reuse that object!"*

we had to **rename, import and add the interface here**

without it we could add easily a wrong property without an error. See? The unclean code is spreading, like a virus.



```
const clubRulesParams: getClubRulesParams = {  
  ruleSet: fightClubSet,  
  clubType: "FIGHT",  
  location: "UK",  
  concern: "Whoa, wait, this is crazy."  
};  
  
const rules = getClubRules(clubRulesParams);
```

so much **"better"**



As you can see, now we have another thing to do (create a variable for a parameter object), just because we developed something (a parameter object), which **was not even needed at the first place. This is the third.*

You Aren't Gonna Need It

Developing one *YAGNI* creates another one. Yes, that will be recursive.

top 10 reasons why unnecessary code is evil

1. Defect numbers correlates with LoC (Lines of Code).
same business value -> double the code size -> double the bugs
2. Slows down the review process
 - a) more code -> more time to review
 - b) it is a code smell -> even more time to review
 - c) will be spotted in review -> leads to needless discussions
 - d) needs to be removed -> committed -> pushed again
 - e) YAGNI code parts are chained (one justifies another) -> another YAGNI will be revealed -> again and again, until all will be removed
3. It is an added complexity, so it slows down the further development
4. It adds premature structures, which makes the code rigid, very hard to modify
5. It contains implicit, but false assumptions about the code
e.g., the properties of that object are connected -> the code is lying -> very hard to refactor or remove later
6. Still, it is a tech debt, so needs to be removed at some point
7. It is arrogant: it presumes that it is a necessary investment in future
8. It hides critical issues
because it “solves” a problem, it may seem like good code, but the problem is not solved, just it was hidden
9. Because it is hard to understand, it could hide security issues
10. Risks the delivery
the story was estimated according the business feature complexity, not the solution complexity -> hard to keep the estimation -> the late merge places extra pressure on the TL (review) and on the QA

You Aren't Gonna Need It – new variables

Never add a new variable if that is not absolutely necessary.
Especially do not add for wanting the code easier to read.

top 10 reasons why you should not add a new variable

1. You don't really need it
double check please, you will see that it is not necessary
2. It will prevent using an arrow function without a return
and it is a slap into the face of a compact and easily chainable functional code
3. It will prevent using a ternary instead of an if
using if (a statement) instead of a ternary (an expression) is definitely not a clean code
4. You will screw up the naming
and it will be misleading
5. It hides the fact that the code could be easily refactored
it is possible that 10 variables of 10 are unnecessary, but it is impossible to see that without removing them one by one
6. It does not make the code more readable
maybe it is perfectly readable without it, you just need to gain experience
7. If it makes the code more readable
then there is a problem in the surrounding code -> adding new code to the heap won't fix that
8. It makes the code even less readable
9. You will declare that as *let*
and the linter will be angry about not using a *const*
10. You could break the maximum LoC limit for that function
so you have to refactor the code, unnecessarily extracting code parts that you should not really



a software system containing some unnecessary code parts here and there

Let's back to coding – but now we have a bit of *issue*

By having an object as a parameter, there is a chance that we'll accidentally *overwrite a property there*.

we *change* something here



```
function getClubRules(clubRules: ClubRules) {  
    clubRules.location = 'LOC-' + clubRules.location;  
  
    // more business logic  
}  
  
const clubRules: ClubRules = {  
    ruleSet: fightClubSet,  
    clubType: "FIGHT",  
    location: "UK",  
    concern: "Whoa, wait, this is crazy."  
};  
  
const rules = getClubRules(clubRules);  
  
console.log(clubRules);
```

and it will *break* here



```
[LOG]: {  
  "ruleSet": [],  
  "clubType": "FIGHT",  
  "location": "LOC-UK",  
  "concern": "Whoa, wait, this is crazy."  
}
```

good luck with
debugging this




To prevent [leaking the property change...](#)

we'd need to clone the parameter object. Working with an input object *directly* is usually a code smell.

So, now we'd [need to clone the whole object](#).

```
function getClubRules(clubRules: ClubRules) {  
    const newClubRules = JSON.parse(JSON.stringify(clubRules));  
    newClubRules.location = 'LOC-' + clubRules.location;  
  
    console.log(newClubRules)  
  
    // more business logic  
}
```

yay! even [more unnecessary code](#)



Let's compare the solutions!

```
function getClubRules(  
  ruleSet: RuleSet,  
  clubType: string,  
  _foo: FooBar,  
  _bar: FooBar,  
  location: string,  
  _baz: FooBar,  
  concern: string) {  
  const newLocation = 'LOC-' + location;  
  
  // more business logic  
}  
  
const rules = getClubRules(  
  fightClubSet,  
  "FIGHT",  
  null,  
  null,  
  "UK",  
  null,  
  "Whoa, wait, this is crazy."  
);
```

do you think it looks wrong*?



```
interface getClubRulesParams {  
  ruleSet: RuleSet;  
  clubType: string;  
  _foo?: FooBar;  
  _bar?: FooBar;  
  location: string;  
  _baz?: FooBar;  
  concern: string;  
};  
  
function getClubRules(clubRules: getClubRulesParams) {  
  const newClubRules = JSON.parse(JSON.stringify(clubRules));  
  newClubRules.location = 'LOC-' + clubRules.location;  
  
  // more business logic  
}  
  
const clubRulesParams: getClubRulesParams = {  
  ruleSet: fightClubSet,  
  clubType: "FIGHT",  
  location: "UK",  
  concern: "Whoa, wait, this is crazy."  
};  
  
const rules = getClubRules(clubRulesParams);
```

**then this is the fourth*

Wrong code should look wrong

And it is never a solution to make that looking good.

1. You are an inexperienced developer
 - a) and that is fine, you will need years to gain experience
 - b) if you hides wrong code with a “good looking” one, you will be more experienced in one thing: *writing dangerous code*
 - c) if you don’t hide, others can help you to grow
2. While growing the codebase, the code structure did not evolve
 - > as a consequence, you can write only wrong looking code there, but refactoring is out of the scope in story development
3. It signs a problem of another abstraction level
 - a) the problem could be in a common / parent module -> the fix requires higher level domain knowledge or governance on the code
 - b) the problem could be in the business requirements -> you cannot fix wrong business requirements in the code
4. The codebase is not mature enough to implement that in a nice way
 - in this case the code could remain in that wrong looking state until the codebase evolves enough to see the solution
5. The codebase is too rigid
 - in this case the code could remain in that wrong looking state until it will be refactored
6. The project is in an early phase
 - where the main goal is to have a quick feedback loop with the client, and there is no point to build a throwaway code with a lot of effort making it rigid to modify (defying the quick feedback loop)
7. There is a bug in your code
 - > you should fix it. You should fix the *bug*, not the wrong looking code
8. There is a bug in other parts of the code
 - it just manifests the bug at your code -> you should escalate this -> it is not your responsibility to fix that another bug
9. You wrote your code to be able to test the wrong thing
 - > thing again your testing strategy
10. It is a good-looking code
 - your eyes are just not trained enough to see that -> you need to gain experience

Let's make this clear: this example is **not clean code**

But it is not for the reason some may think.

The problem with the elongated parameter list is not that it looks ugly, or there could be a lot of nulls there. **This is just the surface.**

The problem here is that **it signs architectural problems** with the code, and it is impossible to assess from this part what could be that:

1. why do we have **string literals** in a code at all?
2. maybe these code is a **configuration** (it looks to me)
3. maybe this code **does many things** at the same time
4. maybe we have to break down this code and **spread across different abstraction layers**
5. maybe some of the parameters belongs to each other. If so, this is a serious issue, and a possible root cause of that could be that many of **these arguments (values) should not exist here at all**. At this point here an ID would be needed only which could connect all the values together. (And you obviously cannot fix that by encapsulating them into an object literal)

is this a **clean code or not**, then?



```
const rules = getClubRules(  
  fightClubSet,  
  "FIGHT",  
  null,  
  null,  
  "UK",  
  null,  
  "Whoa, wait, this is crazy."  
);
```

the *only* proper way to handle this is to
comment it and add a *technical debt* to the code



```
// TODO: TL to check the architecture
const rules = getClubRules(
  fightClubSet,
  "FIGHT",
  null,
  null,
  "UK",
  null,
  "Whoa, wait, this is crazy."
);
```

The Bible

This book is unquestionably a mandatory read

However, please read it as a critical thinker. There could be some points which could actually lead serious issues [if following dogmatically](#).

Let's consider this one, for example:

THE BOY SCOUTS HAVE A RULE:*

"Always leave the campground cleaner than you found it."

If you find a mess on the ground, you clean it up regardless of who might have made it.



**this is the fifth*

Never ever practice the Boy Scout rule...

...during feature development. Refactoring (even the smallest) and feature development always must happen in different phases.

1. **Refactoring and feature development require different workflow**
a regression test *must* follow every refactoring -> it is expensive and may require significant manual QA effort as well -> if you do refactor in feature development you force a regression test
2. **Refactoring requires a careful planning**
it is not an ad-hoc job. With that attitude the chance that something will go badly is almost 100%
3. **Maybe your change just makes it worse**
if something looks wrong in a well-established codebase, then maybe there is a reason behind that – just probably you are not aware of that
4. **Your change in the refactored code may conflict**
with your teammate's change
5. **Refactoring is like pulling a thread from a fabric**
you just cannot see how it ends
6. **You may need to update the tests as well**
it could happen (in white box testing)
7. **You may introduce a critical bug**
under the radar, just from purely good intentions
8. **Maybe the refactoring is not needed at all**
or not at that time
9. **Refactoring does not have a direct business value**
or let's keep the team lead have a governance to decide that
10. **Every line of code must be covered by a story or a technical task**
if your story is about feature development then it is not a refactoring

Does that mean the Boy Scout rule is wrong?

Nope, that is actually a useful advice. However, it requires [strict processes](#) (separating the tech tasks from the feature development), [disciplined development team](#) of [experienced members](#) and [strong governance](#) on the code.

Unfortunately, there is a gravitation in the Clean Code book of taking the topics very dogmatically, basically presuming that developers are brainless idiots.

That could be fine (*remember, in Agile the most important thing is the transparency of problems, not the problems themselves*), but that is just totally opposite what we'd need to follow the Boy Scout rule.

Robert, you could have [dogmatic developers or an experienced team](#), that is able to follow this rule *in a way*.

Please choose, you can't have both at the same time, sorry. ͇(ツ)͇



rules that **do not** matter

Formatting

You must use Prettier anyway

DRY + LoC limits

Sonar will be nervous about these

Avoid comments which explain code

You won't write comments after all

Who do you write your code for?

*„Any fool can write code that a computer can understand.
Good programmers write code that humans can understand”*

Martin Fowler

*„Programs are meant to be read by humans, and only incidentally
for computers to execute.”*

Donald Knuth



Importance of readability

For yourself

- you understand now, but can you after 6 months?

For teamwork

- even more **crucial to make it clear** for the next people who touch or use your code
- don't even think that you'll be there forever to maintain
- on enterprise level your code can remain part of the system for years

```
ws.on("message", m => {
  let a = m.split(" ")
  switch(a[0]){
    case "connect":
      if(a[1]){
        if(clients.has(a[1])){
          ws.send("connected");
          ws.id = a[1];
        }else{
          ws.id = a[1]
          clients.set(a[1], {client: ws})
          ws.send("connected")
        }
      }else{
        let id = Math.random().toString()
        ws.id = id;
        clients.set(id, {client: ws})
      }
    }
  }
})
```

Clean Code != Beautiful code

Beauty is not measurable, there are no standards for beauty

Real requirements:

- a program **free of errors**
- **easy to understand** for humans
- it is not obvious that you can make it better
- **namings are clear and revealing intention**, pronounceable, searchable, nouns for variables, verbs for functions
- functions/modules **do or returns*** exactly the one single thing that you expect from it
- **easy to extend and modify** on demand
- **no needless complexity**: only add features that we need, extend later as we learn more about product requirements (YAGNI)
- **no needless repetition (DRY)** – but do not over-extract, it will add complexity!

*10th

Command–query separation

A function should *return something* or *do something* (with side effects). Never should do both at the same time.

Keeping code clean

Define and document [detailed code conventions](#) and team agreement

- not just for formatting, but also semantic rules
- great example: [Google JS style guide](#)

[Automate](#) what you can

- ESLint, Code coverage, Unit tests
- commit / push hooks in Git
- tools for checking LoC, Cyclomatic Complexity

Do [code review](#)

- reduces bugs tremendously
- knowledge sharing channel, collaborative
- prompt, polite; asking, not directing, includes praise

Remember [Broken Window Theory](#)



“Sometimes science is more art than science, Morty.”



Weöres Sándor: “[Tojáséi](#)” [en: Egg(s)hell]

The simple code is

expressive: readable, the intention is clear,
it documents itself

responsible over a single part only,
but it tells you the whole story about that

short as much as possible,
there is nothing left to take away

symbolic: reusable in its realm

APPENDIX

So how to handle the complexity?



Remember: we cannot hide...



... at least successfully

[We Can't Seem to Escape the Problem of Complexity in Software Development](#)

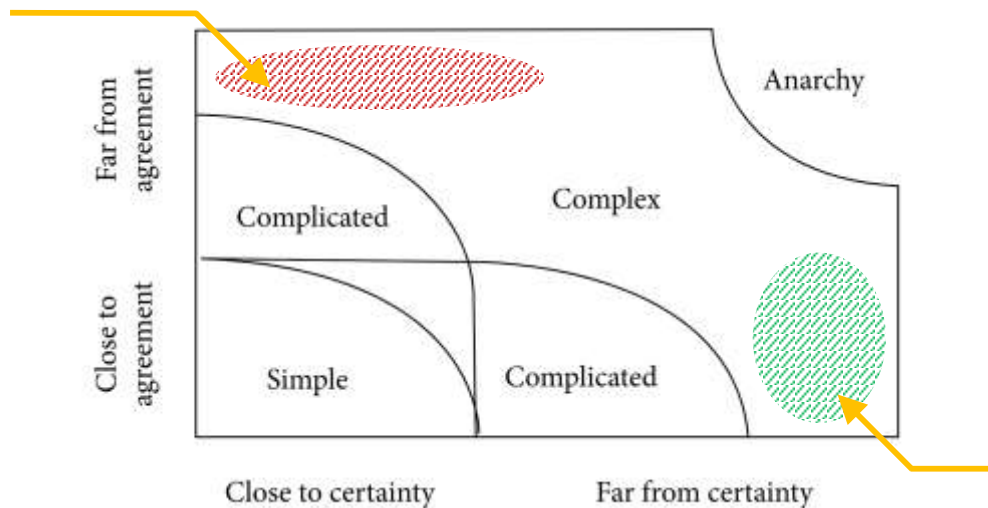
Let's master the complexity



Mastering the complexity

The **systems are complex** (*far from certainty*), so the only viable way of development is having a strong governance, and to **restrict the toolset significantly**.

- autonomy in tools and approaches
- loose coalition of experts or a *one-man army*
- compromises
- mock, prototype, Proof Of Concept (PoC)



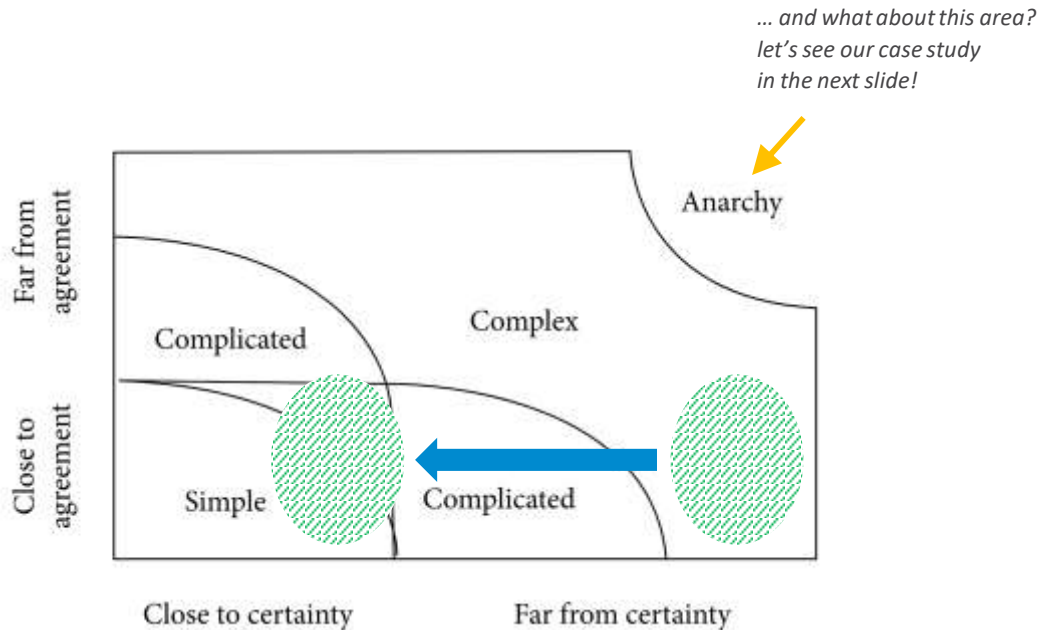
- strong governance
- strict requirements on NFRs
- restrictive conventions, principles, patterns, methodologies
- limited, industry standard toolset
- Clean Code
- Quality Gates
- testing (unit, e2e)
- teamwork centric
- suited for SDLC
- supported in long term
- final product

[Complexity and Project Management: A General Overview](#)

Mastering the complexity

One of the main goals of **handling the complexity** is to move the understanding closer to certainty with different techniques:

- **breaking down** the system into smaller parts that we can reason about
- creating layers that embodies **separation of concerns**
- components with **single responsibility**
- **encapsulating** complexity
- **loosen the coupling** between components
- reducing the **redundancy**



A case study of *The Neverending Story*

- I. The project of Fantastica has been falling down: the project's **Health Status is Red**. The **Architect** has left the project, and the **Complexity** has taken shape into a formless entity, called *The Nothing*, and it started to disintegrate the **delivery plan**.
- II. The **Delivery Manager** informs the **Resource Managers** that the **Client** feels bad, and only a **Team Lead** warrior, named Atreyu can save the project.
- III. The DM gives Atreyu a medallion, the **AGILE**, that guides and protects him. Atreyu and his **team** set off on his quest.
- IV. They soon find themselves in the *Swamps of Sadness*, formed by **bugs and tech debts**, in which the team drowns.
- V. In the Swamps Atreyu finds the **QA lead**, who tells him that the product can be saved only by receiving a new name (**MVP – Minimum Viable Product**), but only the **Product Owner** can do it.
- VI. After countless **overtimes**, wandering alone in a deserted city, Atreyu comes across a chained werewolf (**Scrum Master**), who tells him that when **User Stories** of Fantastica disappear into the *Nothing* they reappear in the next **Sprint** as **Spillovers**.
- VII. The Client invokes **Account Manager** in *Mountain of Destiny*, and they **postpone the Go Live date**. Again, and again.



While *Fantastica* can be saved the new Architect will commit the same mistake as the predecessor: the architecture will be way too complex and **the only thing which can save the project is** to remember the original objective, the...

simplicity

A [simple](#) code is far from superficial or primitive, just the opposite: it is sophisticated.

The problem is, it is very hard to achieve.
The gravity always pulls down* to overcomplicated solutions.

That's why we need to utilize the wisdom of the past, crystallised in [principles and methods](#).

** The level of disorder in the universe is steadily [increasing](#).
Systems tend to move from ordered behavior to more random behavior.*



Bauhaus was about simplicity and [usefulness](#).

Q&A