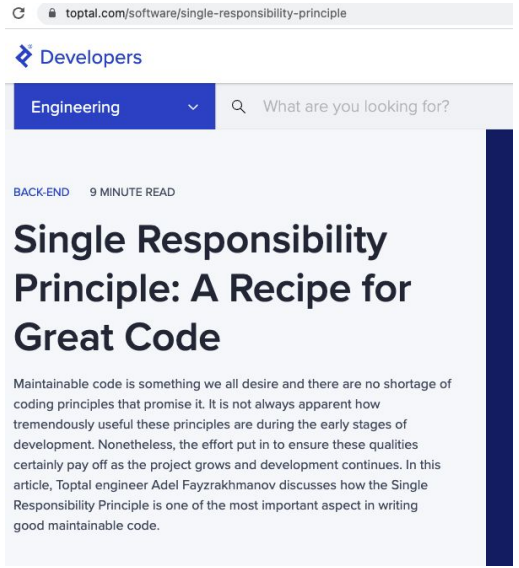
trybe

SOLID

Cada pessoa tem uma responsabilidade específica

Single Responsibility - Responsabilidade única

Esse princípio declara que uma classe deve ser **especializada em um único assunto** e possuir apenas uma responsabilidade dentro do software, ou seja, a classe deve ter uma única tarefa ou ação para executar.



*__TDD__ - Primeiro teste e depois a implementação auxilia a criar funções com responsabilidade única.

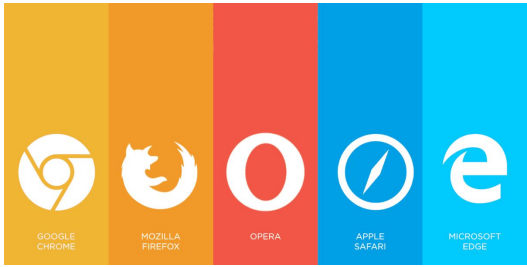
*__Função pura__ - Dado uma entrada, a função retorna a mesma saída e sem efeitos colaterais.

O alicerce da casa deve ser fechado para uma modificação

E aberto para a extensão caso queira adicionar novo cômodos

Open-Closed Principle

Princípio Aberto-Fechado — **Objetos ou entidades devem estar abertos para extensão, mas fechados para modificação**, ou seja, quando novos comportamentos e recursos precisam ser adicionados no software, devemos estender e não alterar o código fonte original.



Para a fácil manutenção

Se uma janela estragar, deve ser possível substituí-la por outra

Liskov Substitution

Uma classe derivada deve ser substituível por sua classe base.

Uma pessoa trabalhadora não precisa de ferramentas que não irá utilizar

Interface Segregation Principle

Uma classe não deve ser forçada a implementar interfaces e métodos que não irão utilizar.

A casa a ser construída não pode depender das pessoas engenheiras que lá estão, pois, podem sair para assumir outras obras.

Assim, é necessário criar alguma pessoa intermediária que tenha toda a obra em mente, e que quando chegue as novas pessoas, então possa continuar a ser construída.

Dependency Inversion Principle

Princípio da Inversão de Dependência — **É necessário depender de abstrações e não de implementações.**

SOLID

- Responsabilidade única (S)
- Aberto fechado (O)
- Inversão de dependência (D)

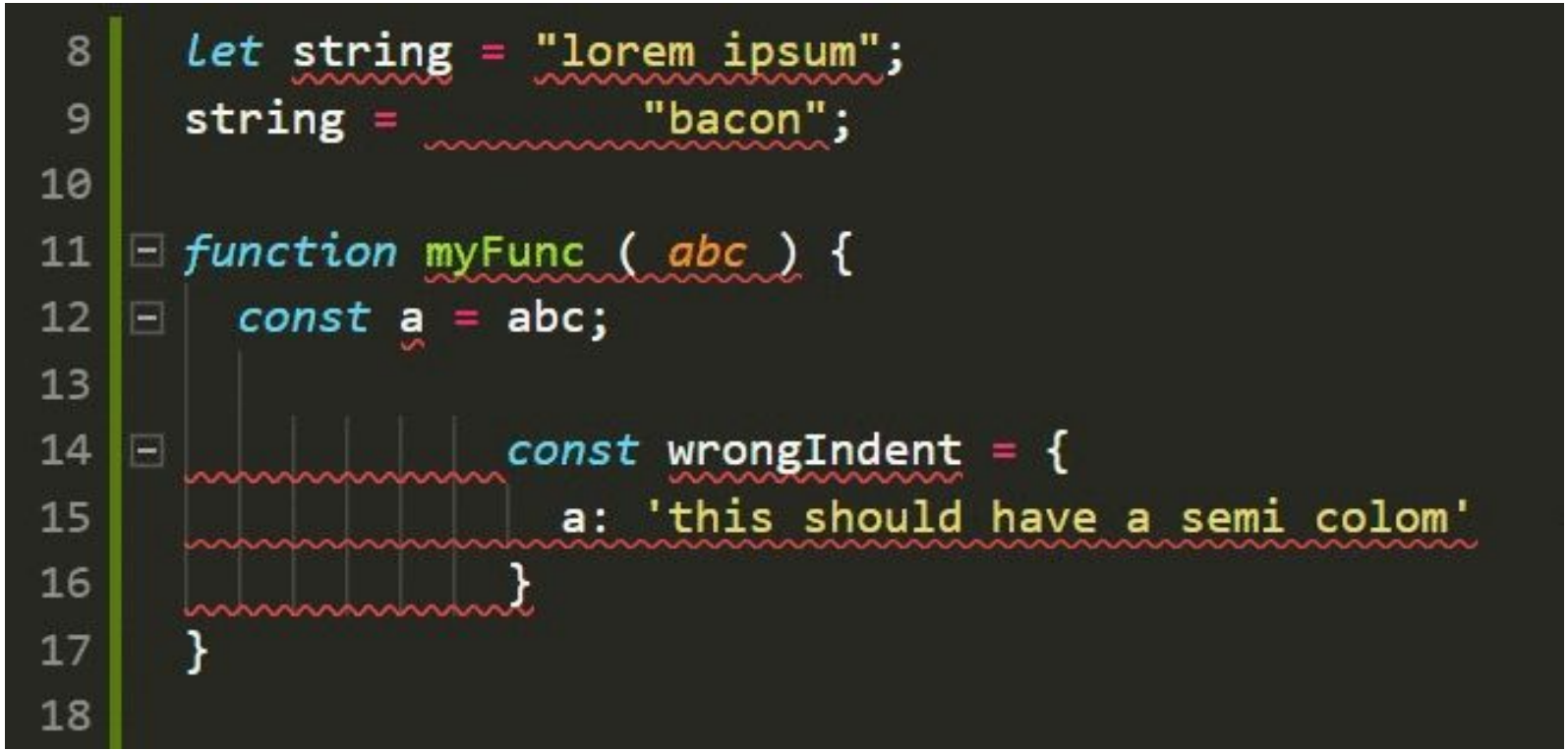
- ❏ Liskov (L)
- ❏ Separação por interfaces (I)


CLEAN YOUR CODE
indieYourFace.com

O Clean Code de Uncle Bob

- ✔ Como distinguir um código bom de um ruim
- ✔ Como escrever códigos bons e como transformar um ruim em um bom
- ✔ Como criar bons nomes, boas funções, bons objetos e boas classes
- ✔ Como formatar o código para ter uma legibilidade máxima
- ✔ Como implementar completamente o tratamento de erro sem obscurecer a lógica
- ✔ Como aplicar testes de unidade e praticar o desenvolvimento dirigido a testes

*Arquitetura de Software - MSC, e, há por exemplo o Clean Architecture e DDD.


8 let string = "lorem ipsum";
9 string = "bacon";
10
11 function myFunc (abc) {
12 const a = abc;
13
14 const wrongIndent = {
15 a: 'this should have a semi colom'
16 }
17 }
18

O SOLID, também, nos ajuda a evitar os *Code smells*

Alguns smells

- **Rigidez:** O software é difícil de mudar. Uma pequena mudança causa uma cascata de mudanças subsequentes
- **Fragilidade:** O software quebra em muitos lugares devido a uma única alteração.
- **Imobilidade:** Você não pode reutilizar partes do código em outros projetos devido aos riscos envolvidos e ao alto esforço
- **Complexidade** Desnecessária
- **Repetição** Desnecessária
- **Opacidade:** O código é difícil de entender

Uncle Bob


trybe