Les 10 commandements du clean Code

Pourquoi faire du clean code ?

Quels sont les caractéristiques d’un mauvais code :

- Un code “pourri”
- Rigidité
- Fragilité
- Inséparabilité
- Opacité
- Rush son code

Pourquoi faire du clean code ?

Les avantages d’écrire son code proprement :

- Qualité du code
- Lisibilité
- Facilité de maintenance
- Pour se la péter 😎

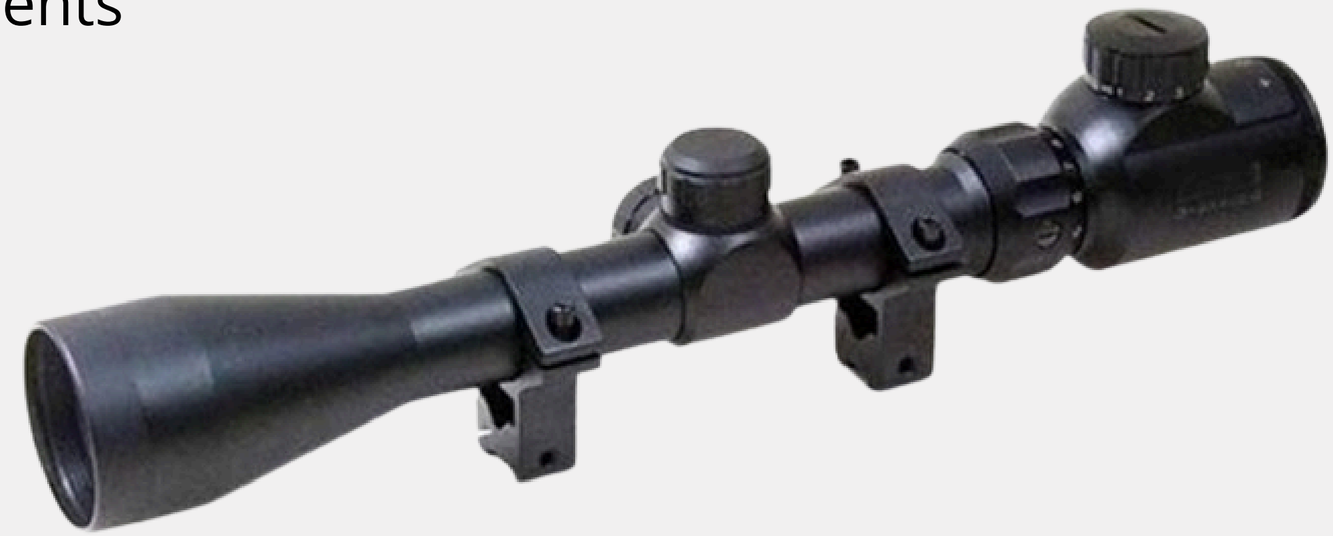
1. **Le nommage des variables**

Quel est le meilleur choix ici ?

```
let searchUser = (phone) => {
  //do something
}

//ou

let searchUserByPhoneNo = (phone) => {
  //do something
}
```

1. **Le nommage des varibles**

- Les “règles d’écriture” : camelCase + SNAKE_UPPER_CASE
- Un nom significatif
- Privilégier un détail plutôt qu’un résumé
- Utiliser des verbes cohérents
- “The Scope Length Rule”



2. Les nombres magiques

Quel est le meilleur choix ici ?

```
const NUMBER_OF_STUDENTS = 50{
for(let i= 0; i<NUMBER_OF_STUDENTS; i++){
  //do something
}

// ou

for(let i= 0; i<50; i++){
  //do something
}
```

2. Les nombres magiques

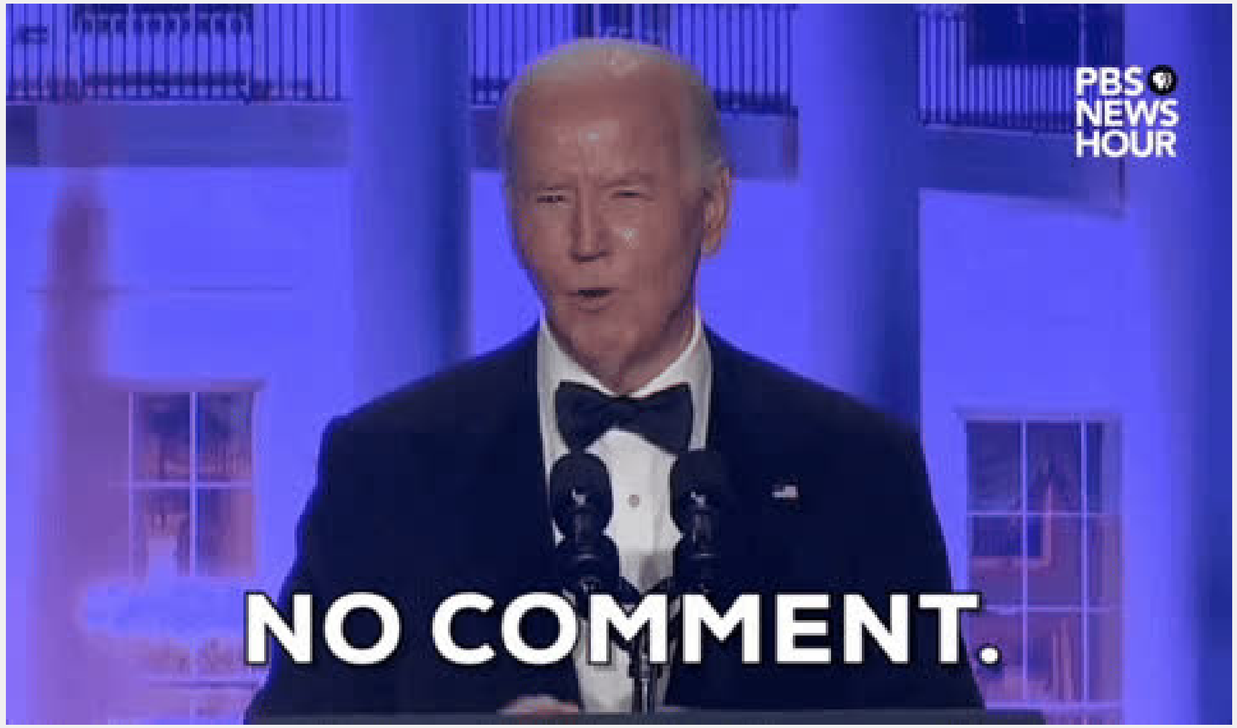
Cela signifie que nous attribuons un nombre sans signification claire.
Parfois, nous utilisons une valeur dans un but précis,
cependant nous ne l’affectons pas à une variable significative.
Le problème est que lorsque quelqu’un travaille avec votre code,
il ne connaît pas la signification de cette valeur directe.

3. Les imbrications biscornues

Quel est le meilleur choix ici ?

```
const ARRAY = [[['une plume']]]
array.forEach((firstArray) =>{
  firstArray.forEach((secondArray) =>{
    secondArray.forEach(element) =>{
      console.log(element);
    })
  })
})

//ou

const getValuesOfNestedArray = (element) => {
  if(Array.isArray(element)){
    return getValuesOfNestedArray(element[0])
  }
  return element
}
getValuesOfNestedArray(ARRAY)
```

4. Les commentaires



// Cela permet d’aider les gens
// à mieux comprendre le code
// ,plus tard dans le temps
// à mieux collaborer sur le
// même projet.

5. Eviter les trop grandes fonctions

Quel est le meilleur choix ici ?

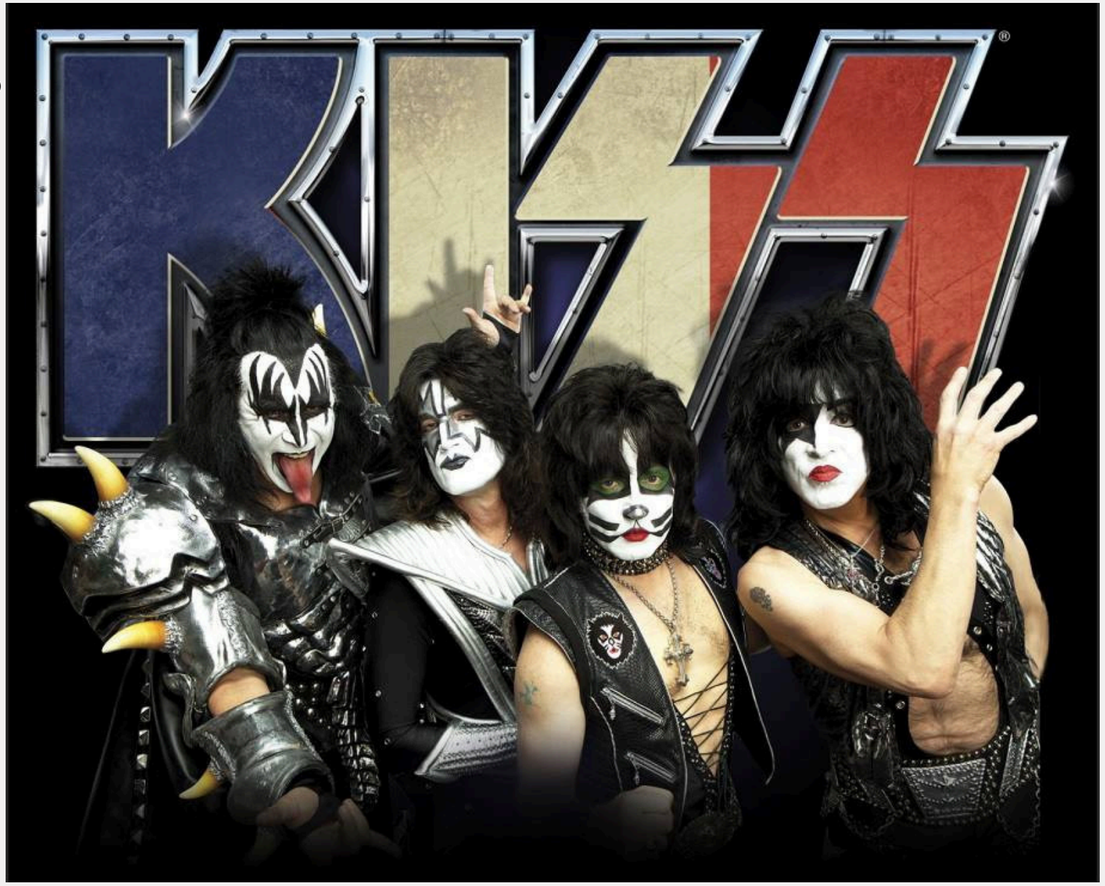
```
let addSub = (a, b) => {
  let addition = a + b
  let sub = a - b
  return `${addition}${sub}`

//ou

let add = (a, b) => {
  return a + b
}
let sub = (a, b) => {
  return a - b
}
```

5. Eviter les trop grandes fonctions

- Un adage en dev est “Diviser pour mieux régner”
- On m’a également dit : “Si ta fonction fait plus de 30 lignes, c’est que tu peux la séparer en plusieurs petites fonctions”
- En divisant nos fonctions -> réutilisable + lisible

6.DRY

“Don’t Repeat Yourself”
En fait, dès qu’on se répète dans le code, cela signifie que l’on peut créer une fonction pour ce bout de code.

7. 

“Keep it simple, stupid”

8.YAGNI

“You Aren’t Gonna Need It”
Ce principe exprime l’idée suivante : il faut uniquement ajouter des fonctionnalités supplémentaires au code lorsqu’elles sont nécessaires.

9.SOLID

- Responsabilité unique (Single responsibility principle)

Une classe, une fonction ou une méthode doit avoir une et une seule unique raison d'être modifiée. Cela favorise la modularité et facilite la maintenance en évitant les classes surchargées de responsabilités.

- Ouvert/fermé (Open/closed principle)

Une entité applicative (classe, fonction, module ...) doit être fermée à la modification directe mais ouverte à l'extension. L'objectif est de permettre l'ajout de nouvelles fonctionnalités sans altérer le code existant.

9.SOLID

- Substitution de Liskov (Liskov substitution principle)

Une instance de type T doit pouvoir être remplacée par une instance de type G, tel que G sous-type de T, sans que cela ne modifie la cohérence du programme. Cela garantit que les sous-classes peuvent être utilisées de manière interchangeable avec leurs classes de base.

- Ségrégation des interfaces (Interface segregation principle)

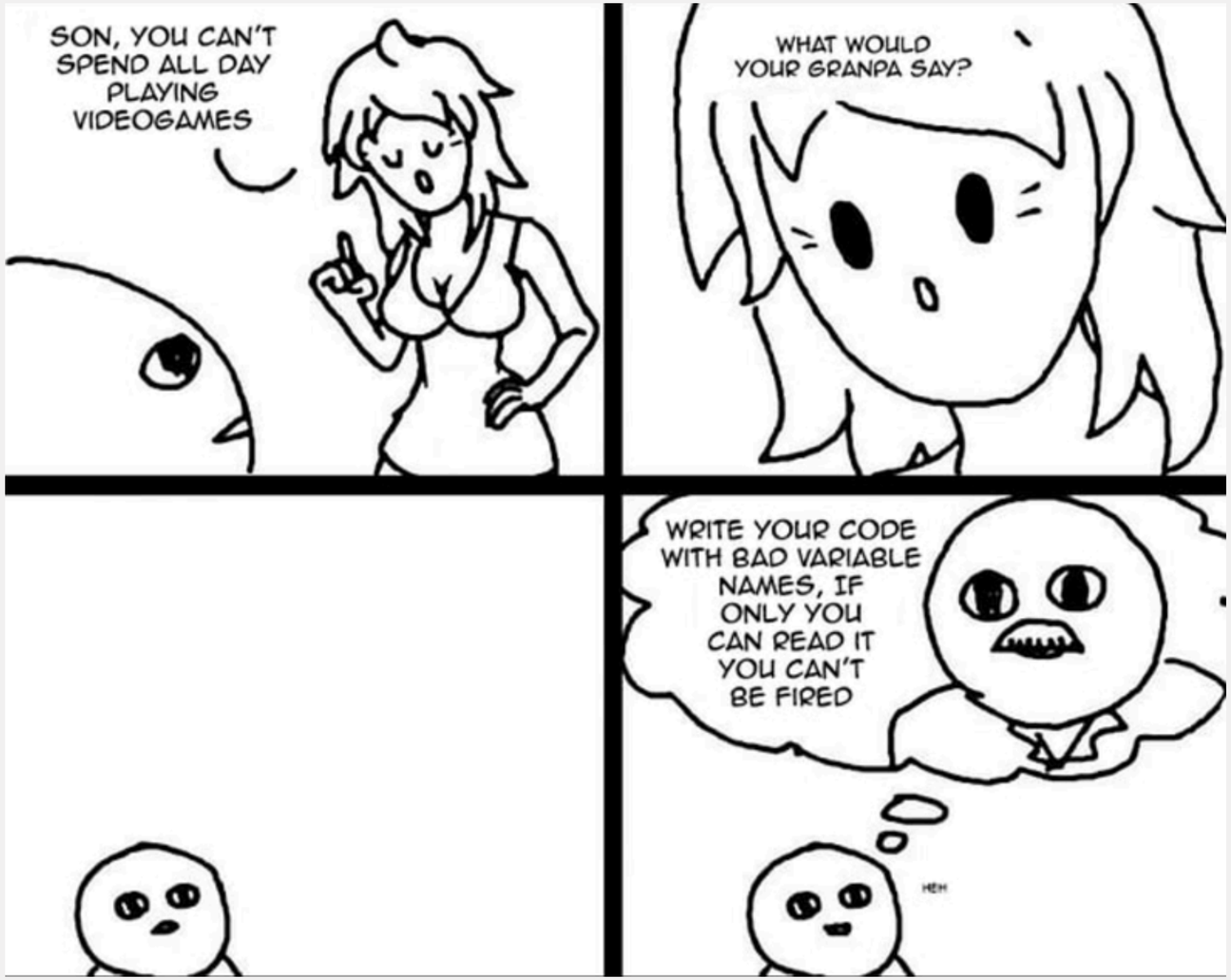
Préférer plusieurs interfaces spécifiques pour chaque client plutôt qu'une seule interface générale. Cela évite aux classes de dépendre de méthodes dont elles n'ont pas besoin, réduisant ainsi les couplages inutiles.

9.SOLID

- Inversion des dépendances (Dependency inversion principle)

Il faut dépendre des abstractions, pas des implémentations. Cela favorise la modularité, la flexibilité et la réutilisabilité en réduisant les dépendances directes entre les modules.

10. Petit tips supplémentaire



Conclusion

En respectant ces bonnes pratiques, cela permet d’améliorer la lisibilité, la maintenance et favoriser la collaboration
L’application de ces pratiques vous évitera d’avoir un code spaghetti et de passer pour un noob.
Mais rassurez vous, on est tous des noobs a l’heure actuelle et c’est en forgeant qu’on devient un bon codeur. Donc forgez