Clean Code

Qualité de Développement — R5.A.08

C. Raïevsky

2026-2027

IUT A Valence
Université Grenoble Alpes

Département Informatique

BUT Informatique 3ème année

Qualité de développement - Savoir Coder

Un développeur passe dix fois plus de temps à lire du code qu'à en écrire

Il faut donc écrire le code pour les dix fois où il sera lu

- Lisible
- Compréhensible
- Structuré

Critères de lisibilité

- Convention de nommage
- Convention de formatage

Conventions de nommage

Types utilisateur

- Commencent par une majuscule : `Point`
- "*CamelCase*" si plusieurs mot : `BotTeam`

Variables, attributs, méthodes, fonctions associées

- Commencent par une minuscule

Constantes

En majuscules : `THIS_WILL_NOT_CHANGE`

Conventions Rust

On écrit tout en anglais

Sauf si une règle de l'entreprise impose autre chose

Formatage

- Ne ré-inventez pas la roue
- Les outils font ça très bien : IDE, `cargo fmt`

```
1 struct Point{x:i32,
2 y: i32,}
3
4 fn main(){let p=Point{x:1,y:2};
5 if p.x.abs()+p.y.abs()<10{
6 println!("p is close");}
7 else{println!("p is far");}
8 println!("{}{}",p.x,p.y);
9 }
```

```
1  struct Point {
2      x: i32,
3      y: i32,
4  }
5
6  fn main() {
7      let p = Point { x: 1, y: 2 };
8      if p.x.abs() + p.y.abs() < 10 {
9          println!("p is close");
10     } else {
11         println!("p is far");
12     }
13     println!("{}{}", p.x, p.y);
14 }
```

Critères de compréhensibilité

- Intention Revealing Interfaces
- Expressivité
- Concision

Intention Revealing Interfaces

Les noms utilisés doivent faciliter la compréhension

Critères généraux - Expressivité

Expressivité - le nom a un sens

- Rôle
- Données contenues

```
let d = 10; // Distance to player
let flag = true; // True if player active
let l = 0; // Length of the vector
```

```
let dist_to_player = 10;
let is_player_active = true;
let vec_length = 0;
```

Éviter les termes génériques dans les noms :

data, value, info, variable, table, string, object, etc.

Critères généraux - Concision

Le plus court possible mais pas trop

- Ajouter l'information pertinente, le contexte
- Mais pas le reste (type, interface, etc.)

```rust
struct Data {
    point: Point,
    flag: bool,
    name_string: String,
    i: u32,
}
```

```rust
struct Player {
    position: Point,
    is_active: bool,
    name: String,
    id: u32,
}
```

Dans les langages typés, on ne répète pas le type dans le nom

Critères généraux - Incohérence

```
impl Data {
    fn difference_of_points(one: &Data, two: Point) -> u32 {
        Point::compute_distance(one.point, two)
    }
}
```

Incohérence

- Nom de la fonction → pourquoi une différence entre points retourne un entier ?
- Nommage des paramètres

Critères généraux - Incohérence

```rust
impl Point {
    fn move_point(&self, vector: Vec2D) -> Point {
        let new_x = self.x + vector.get_x();
        let new_y = self.y + vector.get_y();
        Point::by_parametre(new_x, new_y)
    }
}
```

Incohérence

- Nom de la fonction → suggère une modification du point

Critères généraux - Cohérence

```rust
impl Player {
    fn distance(&self, pos: Point) -> u32 {
        self.position.distance(pos)
    }
}
```

Cohérence

- Nom de la fonction cohérent → calcule la distance entre un Player et un point
- Nommage des paramètres → le point passé en paramètres est une position

Critères généraux - Cohérence

```
impl Point {
    fn moved(&self, v: &Vec2D) -> Self {
        Point::new(self.x + v.x, self.y + v.y)
    }

    fn translate(&mut self, v: &Vec2D) -> &mut Self {
        self.x += v.x;
        self.y += v.y;
        self
    }
}
```

Noms cohérents avec :

- La mutabilité de l'instance (`self`)
- Le type de retour

Classe/Struct méhtodes/fonctions associées

Classes / Struct

- Nom ou groupe nominal
- Pas de verbe
- Éviter les termes génériques (Data, Info, Table, etc.)

Méthodes, fonctions associées

- Verbes ou groupes verbaux
- *En Rust*, pas de "get_" ou "set_"

KISS - ne pas faire le malin

Keep It Simple Stupid

You're smart but others don't care, at all !

À moins d'avoir une très bonne raison

Un code clair sera de meilleure qualité qu'un code "optimisé"

Ne pas ré-inventer la roue

Les outils modernes font le gros du travail

Conclusion

Il faut écrire le code pour les dix fois où il sera lu

- Lisible
- Compréhensible
- Structuré
- *Pas forcément tout de suite*