# Coding Rules

## Qualité de Développement — R4.02

---

C. Raïevsky

2023-2024

IUT A Valence
Université Grenoble Alpes

Département Informatique

*BUT Informatique 2ème 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 capitales : `THIS_WILL_NOT_CHANGE`

Conventions Rust

**On écrit tout en anglais**

# Formatage

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

## Intention Revealing Interfaces

**Les noms utilisés doivent faciliter la compréhension**

# Critères généraux - Expressivité

## Expressivité - le nom a un sens

- Rôle
- Données contenues

```rust
struct Entity {
    position: Point,
    is_active: bool,
    id: u32,
}
```

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

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

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

# Critères généraux - Concision

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

```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)
    }
}
```

# 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

# Comment organiser son code ?

**On parle de :**

- **Structuration**
- **Découpage en composants logiciels**
- **Décomposition**

# Composants Rust

**Hiérarchie des composants en Rust :**

- Package
- Crate
- Module
- Struct - Enum

# Principes SOLID

## 5 Principes de structuration - organisation du code

- Single Responsability Principle (SRP)
- Open-closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

## Single Responsability Principle

SOLID

Objectif principal : limiter l'impact d'un changement

- Un composant n'a qu'une seule responsabilité
- Un composant regroupe les éléments qui changent pour la même raison

### Classes - Struct

- Un seul concept
- Le nom permet de saisir les responsabilités

### Fonctions - Méthodes

- Une seule action
- Nom explicite, un verbe
- Nombre de paramètres "raisonnable"

## Single Responsability Principle - Contre-exemple

S OLID

```
1  struct Simulation {
2      cars: Vec<Cars>,
3      window: sfml::graphics::RenderWindow,
4  }
5
6  impl Simulation {
7      fn step(&mut self) { /* Update cars*/ }
8
9      fn update_display(&mut self) {
10         // Draw cars on "window"
11     }
12 }
```

Il faudra changer "Simulation" si on choisit une autre bibliothèque graphique ou si on décide de travailler avec une autre structure de données pour les entités

## Single Responsability Principle

S OLID

Un mieux (avec d'autres problèmes)

```rust
struct Simulation {
    cars: Vec<Car>,
}

impl Simulation {
    fn run(&mut self) {
        /* Update cars */
    }
}
```

```rust
struct ScreenDisplay {
    window: sfml::graphics::RenderWindow,
}

impl ScreenDisplay {
    fn display(&mut self, vehicles: &Vec<Car>)
        /* Draw cars on the screen */
    }
}
```

Attention au "surdécoupage". L'objectif reste de minimiser l'impact du changement

# Open-Closed Principle

S O LID

**Open**

Un composant doit être ouvert à l'extension

**Closed**

Un composant doit être correctement protégé des modifications

# Open-Closed Principle

S O LID

**Open**

Un composant doit être ouvert à l'extension

```
1  struct Simulation {
2      cars: Vec<Car>,
3  }
4
5  impl Simulation {
6      fn run(&mut self) {
7          for c in self.cars.iter_mut() {
8              c.advance();
9          }
10     }
11 }
```

**Problème :**

- Simulation ne traite que des voitures
- Ajouter des avions ???

→ La simulation est "**fermée**" à l'extension

# Open-Closed Principle

S O LID

**Open**

Un composant doit être ouvert à l'extension

```
1 trait Vehicle {
2     fn advance(&mut self);
3 }
4
5 struct Simulation {
6     vehicles:
7         Vec<Box<dyn Vehicle>>,
8 }
```

```
1 impl Simulation {
2     fn run(&mut self) {
3         for v in
4             self.vehicles.iter_mut() {
5                 v.advance();
6         }
7     }
8 }
```

## Open-Closed Principle

S O LID

```rust
trait Vehicle {
    fn advance(&mut self);
}
```

```rust
impl Vehicle for Car {
    fn advance(&mut self) {
        self.position.x += self.speed.x;
        self.position.y += self.speed.y;
    }
}
```

```rust
impl Vehicle for Plane {
    fn advance(&mut self) {
        self.x += self.vx;
        self.y += self.vy;
        self.z += self.vz;
    }
}
```

Simulation peut traiter tous type implémentant le trait "**Vehicle**" → **ouverte**

## Liskov Substitution Principle

SO L ID

**Avec héritage :**

Un objet de type T doit pouvoir être remplacé par un objet d'un sous type de T

**Pas d'héritage en Rust donc**

- Un objet implémentant un trait T doit pouvoir être remplacé par un autre objet implémentant ce trait T
- Un trait est un **contrat**
- Par exemple, un "`Vehicle`" doit avoir une fonction "`advance`" **cohérente**

## Interface Segregation Principle

SOL **I** D

L'utilisateur d'une interface ne doit pas dépendre de méthodes qu'il n'utilise pas

**En Rust**

- Un trait ne comporte que des méthodes utiles pour celui qui l'implémente
- Sinon il faut découper le trait

$\rightarrow$ Diminution du couplage

## Interface Segregation Principle

SOL | D

Ajout d'une méthode "refuel" pour un véhicule

```rust
trait Vehicle {
    fn advance(&mut self);
    fn traveled_distance_km(&self) -> u32;
    fn refuel(&mut self, amount: u32);
}
```

```rust
impl Vehicle for Car {
    fn advance(&mut self) {...}
    fn traveled_distance(&self) {...}
    fn refuel(&mut self) {...}
}
```

```rust
impl Vehicle for Plane {
    fn advance(&mut self) {...}
    fn traveled_distance(&self) {...}
    fn refuel(&mut self) {...}
}
```

Paraît une bonne idée

## Interface Segregation Principle

SOL I D

Ajout des vélos...

```
trait Vehicle {
    fn advance(&mut self);
    fn traveled_distance_km(&self) -> u32;
    fn refuel(&mut self, amount: u32);
}
```

```
impl Vehicle for Bike {
    fn advance(&mut self) {...}
    fn traveled_distance(&self) {...}
    fn refuel(&mut self) {
        ?????
    }
}
```

## Interface Segregation Principle

SOL **I** D

### Le type "Simulation" :

- À besoin de "`advance`" et "`traveled_distance`"
- Pourrait traiter des vélos
- Mais l'abstraction "`Vehicle`" est devenue trop large

### Abstractions cohérentes

- Aussi petites que nécessaires
- mais pas plus petites

# Dependency Inversion Principle

SOLI D

## Entre composants

- Les composants de **haut niveau** ne doivent pas dépendre des composants de **bas niveau**
- Les deux dépendent d'**abstractions** (*traits*)

## Abstractions (*traits*)

- Les abstractions ne doivent pas dépendre des implémentations, des détails
- Les implémentations dépendent des abstractions