Principes SOLID

Qualité de Développement — R5.A.08

C. Raïevsky

2026-2027

IUT A Valence
Université Grenoble Alpes

Département Informatique

BUT Informatique 3ème année

Comment organiser son code ?

On parle de :

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

Principes SOLID

Comment organiser, structurer, son code ?

Pas de méthode systématique mais des grands principes

5 Principes de structuration - organisation du code

- **S**ingle Responsability Principle (SRP)
- **O**pen-closed Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle (DIP)

En Rust, un ensemble d'outils

Hiérarchie des composants en Rust :

- Package
- Crate
- Module
- Struct - Enum

Single Responsability Principle

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

Exemple d'un panier d'achat, types définis :

```rust
struct ShoppingCart {
    items: Vec<Item>,
}

struct Item {
    name: String,
    price: f64,
    quantity: u32,
}
```

```rust
struct User {
    name: String,
    discount_rate: f64, // 0.1 for 10%
}
```

Single Responsability Principle - Contre-exemple

```rust
impl ShoppingCart {
    fn add_item(&mut self, item: Item) {
        self.items.push(item);
    }

    // ... other methods

    fn checkout_price(&self, user: &User) -> f64 {
        let total: f64 = self.items.iter()
            .map(|item| item.price * item.quantity as f64)
            .sum();
        total * (1.0 - user.discount_rate)
    }
}
```

Single Responsability Principle - Contre-exemple

Problèmes :

- Si `User` change, `ShoppingCart` doit changer
- Si le calcul du prix change, `ShoppingCart` doit changer

`ShoppingCart` a plusieurs responsabilités

Solution :

- Séparer `User` et `ShoppingCart`, ce sont des concepts différents
- Le calcul du prix est une responsabilité à part entière

Single Responsability Principle - Une solution

```rust
impl ShoppingCart {
    fn add_item(&mut self, item: Item) {
        self.items.push(item);
    }
}

struct Pricing;

impl Pricing {
    fn calculate_price(cart: &ShoppingCart, user: &User) -> f64 {
        let total: f64 = cart.items.iter()
            .map(|item| item.price * item.quantity as f64)
            .sum();
        total * (1.0 - user.discount_rate)
    }
}
```

Améliorations :

- `ShoppingCart` et `User` sont indépendants
- `ShoppingCart` ne fait que gérer les produits
- `Pricing` gère le calcul du prix

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

Open-Closed Principle

Open

Un composant doit être ouvert à l'extension

Closed

Un composant doit être correctement protégé des modifications

Open-Closed Principle - Contre Exemple

```rust
pub enum Shape {
    Circle { radius: f64 },
    Square { side: f64 },
}

impl Shape {
    pub fn draw(&self) {
        match self {
            Shape::Circle { radius } => {
                println!("Circle with radius {}", radius);
            }
            Shape::Square { side } => {
                println!("Square with side {}", side);
            }
        }
    }
}
```

Open-Closed Principle - Contre Exemple

Ajouter une forme nécessite :

- Modifier la structure de données
- Modifier la fonction de calcul de la surface

La strucutre `Shape` est fermée à l'extension

Open-Closed Principle - Exemple

```rust
pub trait Shape {
    fn area(&self) -> f64;
}

pub struct Circle {
    radius: f64,
}

pub struct Square {
    side_length: f64,
}
```

```rust
impl Shape for Circle {
    fn area(&self) -> f64 {
        PI * self.radius.powi(2)
    }
}

impl Shape for Square {
    fn area(&self) -> f64 {
        self.side_length.powi(2)
    }
}
```

Open-Closed Principle - Exemple

Possibilité d'ajouter une forme :

- Définir une structure de données
- Définir une fonction de calcul de la surface

```rust
pub struct Rectangle {
    width: f64,
    height: f64,
}
```

```rust
impl Shape for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
}
```

Le trait Shape est ouvert à l'extension

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, une "`Shape`" doit avoir une fonction "`area`" **cohérente**

Interface Segregation Principle

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 - Contre Exemple

Trait "Animal"

- Espèce
- Régime alimentaire
- Cri

```
1 trait Animal {
2     fn species(&self) -> String;
3     fn diet(&self) -> Diet;
4     fn call(&self) -> String;
5 }
```

Interface Segregation Principle - Contre Exemple

Un chien...

```rust
struct Dog {
    name: String,
}

impl Animal for Dog {
    fn species(&self) -> String {
        String::from("Canis lupus familiaris")
    }

    fn diet(&self) -> Diet { Diet::Omnivore }

    fn call(&self) -> String { String::from("Woof!") }
}
```

Interface Segregation Principle - Contre Exemple

Un chat...

```rust
struct Cat {
    name: String,
}

impl Animal for Cat {
    fn species(&self) -> String {
        String::from("Felis catus")
    }

    fn diet(&self) -> Diet { Diet::Carnivore }

    fn call(&self) -> String { String::from("Meow!") }
}
```

Et une giraffe ?

```rust
struct Giraffe {
    name: String,
}

impl Animal for Giraffe {
    fn species(&self) -> String {
        String::from("Giraffa camelopardalis")
    }

    fn diet(&self) -> Diet { Diet::Herbivore }

    fn call(&self) -> String { String::from(" ?????????? ") }
}
```

Ajouter la méthode "`call`" à "`Animal`"

- l'a rendu trop large (*fat*)
- pose problème pour les animaux sans cri

Découpage de l'interface "Animal"

- `Animal` : espèce, régime alimentaire
- `Vocal` : cri

```
1  trait Animal {
2      fn species(&self) -> String;
3      fn diet(&self) -> Diet;
4  }
5
6  trait Vocal {
7      fn call(&self) -> String;
8  }
```

Interface Segregation Principle - Solution

```rust
impl Animal for Cat {
    fn species(&self) -> String {
        String::from("Felis catus")
    }

    fn diet(&self) -> Diet {
        Diet::Carnivore
    }
}

impl Vocal for Cat {
    fn call(&self) -> String {
        String::from("Meow!")
    }
}
```

```rust
impl Animal for Giraffe {
    fn species(&self) -> String {
        String::from(
            "Giraffa camelopardalis")
    }

    fn diet(&self) -> Diet {
        Diet::Herbivore
    }
}
```

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