TP1 - Une remise en route sur Rust

Qualité de Développement - R5.A.08

Pour dérouiller votre Rust (pun intended…) vous allez :

- Créer une bibliothèque pour gérer des formes (*shapes*).
- Fiabiliser cette bibliothèque à l'aide tests unitaires.
- Vous assurer de son utilisabilité par un programme externe à la bibliothèque à l'aide de tests d'intégration.
- Ajouter un exécutable illustrant l'utilisation de votre bibliothèque.
- Enrichir votre bibliothèque pour fournir des fonctions liées à la gestion d'une liste de formes.

> L'objectif de ce TP est de vous faire réaliser un projet complet qui pourra vous servir de référence pour les TP et projets à venir. Notamment en ce qui concerne les tests, la structuration en modules et la documentation.

Si vous travaillez sur un poste de l'IUT, Rust est déjà installé, sous linux, évidemment.

Si vous souhaitez travailler sur un ordi personnel, utilisez la méthode d'installation utilisant `rustup` pour installer tous les outils utiles au développement Rust.

Avant tout : Git

Avant de vous lancer dans le code, vous devrez, comme d'habitude :

- Créer un dépôt Git pour la matière avec le nom : **QualDev-R5.A.08-Nom1-Nom2**
- Ajouter votre binôme et votre encadrant comme Maintainer de ce dépôt.
- Cloner le dépôt en local sur votre compte.

Comme vous n'êtes plus des débutants, vous créerez, **pour chaque TP**

- une branche pour le développement du TP, et idéalement une "merge request" pour cette branche.
- un répertoire dédié à ce TP, qui sera créé au moment de l'initialisation de votre *crate*.

Création du *crate* pour le TP

Pour créer le projet dédié à votre bibliothèque de gestion de formes, utilisez la commande :

```
cargo new shapes --lib
```

À ce stade, vous pouvez déjà commiter et pousser votre code.

Création des types de données correspondants aux formes

Créez les nouveaux types permettant de gérer des formes :

- Circle, possédant un "*radius*"
- Square, possédant un "*side_length*"
- Rectangle, possédant deux champs : "*width*" et "*height*"

Premiers tests

Afin de valider cet ébauche de projet, ajoutez un test de création pour chaque type de donnée en vous inspirant du test présent par défaut dans le fichier lib.rs.

Vous devriez obtenir quelque chose comme ça :

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn can_create_circle(){
        let c = Circle {radius: 3.14};
        assert_eq!(c.radius, 3.14);
    }

    // Other creation tests...
}
```

Ajout de fonctions

Pour chaque type de forme, ajoutez deux fonctions permettant de calculer l'aire ("***area***") et le périmètre ("***perimeter***") de la forme.

Pour rappel, associer une fonction à une instance en Rust ressemble à ça :

```
impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * self.radius.powi(2)
```

```
    }

    // Perimeter

    // Other functions...
}
```

N'oubliez pas d'ajouter les tests unitaires correspondants.

Affichage des formes dans la console

Pour faciliter le débogage, ajoutez l'implémentation du trait `Display` pour chaque type de forme.

Implémenter le trait "Display" en Rust ressemble à ça :

```
use std::fmt;

impl Display for Circle {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "Circle with radius {}", self.radius)
    }
}
```

Fonction *scale*

On souhaite pouvoir modifie la taille d'une forme en utilisant une fonction `scale` qui prend un facteur d'échelle en argument.

Commencez par écrire les **tests unitaires** permettant de tester cette fonction puis ajoutez-la aux blocs "impl" de chaque type.

Cette fonction doit modifier la forme sur laquelle elle est appelée.

À ce stade, vous devriez avoir des tests unitaires pour les fonctions associées aux différents type de formes.

Tests d'intégration

Jusqu'à présent, votre bibliothèque n'a été validée et donc utilisée que par des tests unitaires, **internes** à la bibliothèque.

Afin de vous assurer que votre bibliothèque peut être utilisée sans erreur depuis un code extérieur, vous allez ajouter des tests d'intégration.

En Rust, les tests d'intégration sont placés dans un répertoire séparé, `tests`, à la racine de votre projet.

Création du répertoire et du fichier de tests d'intégration

Créez le répertoire `tests` à la racine de votre projet puis ajoutez-y un fichier `tests_integration.rs`.

> Chaque fichier Rust dans le répertoire "tests" est traité comme un *crate* à part entière.
>
> Pour utiliser les types et fonctions de votre bibliothèque, vous devrez tout d'abord les importer.

Ajout d'un premier test d'intégration

Ainsi, dans votre fichier `tests_integration.rs`, vous devrez importer les types que vous souhaitez tester :

```
use shapes::Circle;
```

Avant de pouvoir écrire une fonction de test :

```
#[test]
fn can_create_circle(){
    let c = Circle {radius: 3.14};
    assert_eq!(c.radius, 3.14);
}
```

Lancez les tests d'intégration

Pour lancer les tests d'intégration, utilisez la commande :

```
cargo test
```

À ce stade le compilateur ne devrait pas être très content.

Pour ne lancer que les tests d'intégration, sans les autres tests (documentation, unitaires), utilisez la commande :

```
cargo test --test tests_integration
```

Le dernier argument étant le nom de votre fichier de tests d'intégration.

Problèmes de visibilité

Pour que ce premier test d'intégration fonctionne, vous devrez rendre les types de votre bibliothèque visibles depuis le code extérieur.

> Le choix de la **visibilité** des éléments d'un module ou d'un type est un **point crucial** de la structure de votre code.
>
> Tout mettre publique n'est pas une bonne idée.

Dans le cas du type `Circle`, vous devrez choisir les éléments à rendre publics et ceux à garder privés, quitte à ajouter des fonctions d'accès et de modification.

C'est dans ces choix que les principes de l'**encapsulation** que vous avez vu en Programmation Orientée Objet doivent être appliqués.

Choisissez judicieusement les éléments que vous souhaitez rendre publiques et ajoutez les fonctions d'accès et de modification nécessaires pour que votre test d'intégration fonctionne, quitte à modifier celui-ci pour qu'il utilise ces fonctions.

Finaliser les tests d'intégration

Ajoutez les tests d'intégration pour les autres types de formes et les fonctions associées.

> Les tests d'intégration sont un élément important de la qualité logicielle car :
>
> - Ils permettent de valider le bon fonctionnement de votre code depuis l'extérieur.
> - Ils fournissent des exemples d'utilisation de votre code, soit une forme de documentation.

Tests dans la documentation

Afin de conserver la documentation à jour par rapport au code, il est possible d'inclure des tests dans la documentation. Cela permet également de fournir un exemple d'utilisation de votre code.

Ajoutez un test de documentation à la fonction `area` de la structure `Circle` :

```
/// Returns circle's area
/// ```
/// # use shapes::Circle;
/// let c = Circle {radius: 1};
/// assert_eq!(c.area(), std::f64::consts::PI);
/// ```
```

Le bout de code entre triple backticks est un test de documentation. Il est lancé lors des tests de votre projet :

```
cargo test
```

Le test sera également visible dans la documentation de votre projet, générée avec la commande :

```
cargo doc
```

Ce qui donnera quelque chose comme ça :

Les plus attentifs auront noté que la ligne `# use shapes::Circle;` n'est pas visible dans la documentation mais est nécessaire pour que le test compile.

Ajouter un exécutable qui utilise la bibliothèque

Pour tester plus complètement ou pour fournir un exemple plus complet d'utilisation de votre bibliothèque, il peut être utile d'ajouter un exécutable qui utilise la bibliothèque que vous venez de créer.

Pour cela, placez les exécutables dans un répertoire `bin` dans le répertoire `src` votre projet.

Ajoutez un fichier `example.rs` dans ce répertoire et écrivez un programme qui utilise votre bibliothèque, un exemple minimal :

```
use shapes::Circle;

fn main() {
    let c = Circle::new(3.14);
    println!("Circle with radius {} has area {}", c.radius(), c.area());
}
```

Pour lancer cet exécutable, utilisez la commande :

```
cargo run --bin example
```

Si vous n'avez qu'un seul fichier dans le répertoire bin, `cargo run` suffit.

Généralisation par introduction d'un enum

Vous l'avez sans doute vu venir : il est temps d'appeler les méthodes `area` et `perimeter` sur une liste de formes par exemple pour calculer la surface totale d'un ensemble de formes.

Dans l'état actuel de votre bibliothèque, vous ne pouvez pas le faire : vous ne pouvez pas mettre des

`Circle` et des `Square` dans la même liste.

Pour faire cela, Rust offre deux options :

- Les "***enums***",
- Les "***traits objects***".

Les *enums* fonctionnent bien quand les types sont connus à la compilation. Dans le cas présent, si l'ensemble des formes possibles est déterminé à la compilation.

Les *traits objects* permettent de gérer des types "dynamiques". Dans le cas présent, cela permet à l'utilisateur de votre bibliothèque d'ajouter des types de formes sans avoir à changer votre code.

Regrouper des types dans un *enum*

Afin de pouvoir mettre des formes différentes dans une même liste, vous allez définir un nouveau type *enum* `Shape` qui est formellement "l'union disjointe" des types `Circle`, `Square` et `Rectangle`.

```
pub enum Shape  {
    Circle(Circle),
    Square(Square),
    Rectangle(Rectangle),
}
```

Une fois ce type défini, vous pourrez créer une liste de formes de type `Vec<Shape>` :

```
let mut v: Vec<Shape> = Vec::new();
v.push(Shape::Circle(Circle::new(3.14)));
v.push(Shape::Square(Square::new(2.0)));
```

C'est un bon début, mais pour pouvoir appeler une méthode sur chaque élément de la liste, il faudra l'ajouter au type `Shape`. Pour cela vous procéderez comme pour un `struct` :

```
impl Shape {
    pub fn area(&self) -> f64 {
        match self {
            Shape::Circle(c) => c.area(),
            Shape::Square(s) => s.area(),
            Rectangle(r) => r.area(),
        }
    }
}
```

Ajoutez les autres fonctions associées aux formes à ce nouveau type `Shape`.

Ajoutez les tests d'intégration de ce nouveau type. Il permettront de valider son utilisation en dehors de la bibliothèque ainsi que de donner un exemple de son utilisation.

Un exemple de test d'intégration pour la création d'une `Shape` de type `Circle` :

```
let shape = Shape::Circle(shapes::Circle::new(3.14));
match shape {
    Shape::Circle(c) => {  // Test if the shape is a `Circle` variant
        assert_eq!(c.radius(), 3.14);
    }
    _ => panic!("Expected Circle variant"),
}
```

Cela pourrait aussi s'écrire :

```
let shape = Shape::Circle(shapes::Circle::new(3.14));
if let Shape::Circle(c) = shape {
        assert_eq!(c.radius(), 3.14);
} else {
    panic!("Expected Circle variant");
}
```

À ce stade vous avez une bibliothèque fonctionnelle et normalement fiable et documentée.

Commitez et poussez votre code.

Généralisation par introduction d'un trait

Une autre façon de permettre la manipulation de formes différentes dans une même liste est d'utiliser des **traits**.

Nettoyage

Avant d'attaquer cette partie, mettez de côté votre code actuel en le déplaçant dans un sous module "shapes_enum". Pour cela, ajoutez la ligne `rust mod shapes_enum;` dans votre fichier `lib.rs` et créez un fichier `shapes_enum.rs`. Déplacez-y le code actuel.

Renommez votre fichier de tests d'intégration en `tests_enum.rs` et adaptez-le pour utiliser le sous-module `shapes_enum`.

Corrigez les erreurs pour faire passer tous les tests.

Commitez et poussez votre code.

Définition du trait *Shape*

Dans le fichier `lib.rs` qui devrait maintenant ne contenir que la déclaration du sous-module `shapes_enum`, ajoutez un ***trait*** `Shape` :

```
pub trait Shape {
```

```
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}
```

Un trait en Rust définit un comportement.

Ce comportement pourra être implémenté par différents types, permettant ainsi de les traiter de manière uniforme.

Ici, vous avez déclaré le comportement attendu pour une forme : pouvoir calculer son aire et son périmètre.

Il n'est pas possible d'associer des données à un trait.

Implémentation du trait *Shape*

Pour pouvoir utiliser ce trait, il faudra l'implémenter pour chaque type de forme. Pour `Circle`, cela ressemblera à ça :

```
use std::f64::consts::PI;

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

    fn perimeter(&self) -> f64 {
        2.0 * PI * self.radius
    }
}
```

Ajoutez les implémentations pour les autres types de formes.

Utilisation du trait `Shape`

Une fois le trait défini et implémenté pour les différentes formes, il est possible de créer une liste de formes. Cependant, la première intuition **ne fonctionnera pas** :

```
let mut v: Vec<Shape> = Vec::new(); // Error: `Shape` is not a concrete type
v.push(Circle::new(3.14));
```

Rust ne peut pas faire une liste d'éléments dont il ne connaît pas la taille à la compilation. Or le trait `Shape` spécifie un comportement mais ne définit pas la taille des éléments qui l'implémentent.

Il va donc falloir stocker dans le vecteur, non plus des instances implémentant le trait `Shape`, mais des références ou des pointeurs vers ces instances.

Pour faire cela en Rust, il faut utiliser les pointeurs vers le tas : `Box`.

La déclaration d'un vecteur d'éléments implémentant le trait `Shape` devient donc :

```
let mut v: Vec<Box<dyn Shape>> = Vec::new();
```

Et pour ajouter des éléments à ce vecteur :

```
v.push(Box::new(Circle::new(1.0)));
v.push(Box::new(Square::new(2.0)));
```

Ajoutez les tests d'intégration pour cette nouvelle version de votre bibliothèque. Vérifiez notamment que la somme des aires des formes d'une liste est correcte.

Ajoutez un binaire qui utilise cette nouvelle version de votre bibliothèque dans un fichier `example_trait.rs` dans le répertoire `bin`.

Commitez et poussez votre code une fois que tous les tests fonctionnent.

> Cette façon de définir des comportements communs à plusieurs types est plus proches de ce qui se fait dans les langages orientés objets plus classiques. Un trait correspond à une interface dans ce contexte.

Nouveau type de forme dans un programme externe

Cette deuxième façon de généraliser le type `Shape`, en utilisant un trait, est plus flexible pour les utilisateurs de la bibliothèque puisqu'elle leur permet de définir leur propres formes.

Pour illustrer cela, définissez **dans votre programme d'exemple** un type `RightTriangle` et implémentez le trait `Shape` pour ce type.

Constatez que vous pouvez mélanger des formes de la bibliothèque et des triangles rectangles dans une même liste.

Factorisation de code

Définissez, dans votre fichier `lib.rs`, une fonction `total_area` qui prend en argument une liste de formes et renvoie la somme des aires de ces formes.

Ajoutez également les tests unitaires correspondants.

Pouvez-vous utiliser cette fonction dans votre programme d'exemple en lui passant une liste de formes contenant également des triangles rectangles ?

Open Closed Principle

Ce que vous venez de faire est un exemple du Open Closed Principle : votre bibliothèque est ouverte à l'ajout de nouveaux types de formes sans avoir à connaître ou modifier le code existant.

Ainsi la fonction `total_area` peut être utilisée pour n'importe quel type défini par l'utilisateur qui implémente le trait `Shape`, sans avoir à dévoiler le code de la fonction ou les détails internes de la bibliothèque.

Évaluation

Vous serez évalués sur :

- La présence de tests :
  - unitaires,
  - d'intégration,
- La présence de documentation et de tests dans la documentation
- La généralisation de votre bibliothèque par l'introduction d'un *enum* **et** d'un *trait*
- Le programme d'exemple d'utilisation de votre bibliothèque et notamment :
  - la définition du nouveau type `RightTriangle`,
  - l'utilisation de la fonction `total_area` avec une liste de formes contenant des éléments de ce nouveau type (mais pas que).
- La présence de votre branche de développement,
- La pertinence des messages et la fréquence de commit,

Si vous avez terminé

Lisez cet article sur le polymorphisme en Rust.

Quels sont les avantages et inconvénients de chaque méthode de généralisation ?