TP2 - TDD

Qualité de Développement - R4.02

Première mise en pratique du TDD.

Afin de mettre en pratique le TDD et de poursuivre la prise en main du langage Rust, vous allez développer deux petits projets :

- un programme capable de déterminer si une chaîne de caractères est un palindrome.
- une petite bibliothèque de géométrie permettant de manipuler des points et des vecteurs 2D.

Palindrome

Spécifications

Votre vérificateur devra respecter les contraintes suivantes :

- Détecter qu'une chaîne de caractères est un palindrome : c'est-à-dire qu'elle correspond au même mot à l'envers.
  - Des mots comme "non" ou "radar" sont des palindromes.
  - Des mots comme "aspirateur" ou "petit" ne sont pas des palindromes.
- La casse n'a pas d'influence sur le fait d'être un palindrome : "Non" ou "Radar" sont également des palindromes.
- Il devra également être capable de détecter des phrases palindromes comme "Was It A Rat I Saw" ou "Engage le jeu que je le gagne".

Mise en pratique du TDD sur le problème du palindrome

Règles du TDD

Vous allez mettre en pratique le TDD classique "Red-Green-Refactor". Les règles sont :

1. Vous devez écrire un test qui échoue avant de pouvoir écrire le code de production correspondant.
2. Vous devez écrire une seule assertion à la fois, qui fait échouer le test ou qui échoue à la compilation.
3. Vous devez écrire le minimum de code de production pour que l'assertion du test actuellement en échec soit satisfaite.

Premier test

Créez votre premier test dans le fichier `main.rs`, en dessous de la fonction `main`, de votre nouveau projet. Adaptez le nom du test à l'exigence que vous testez :

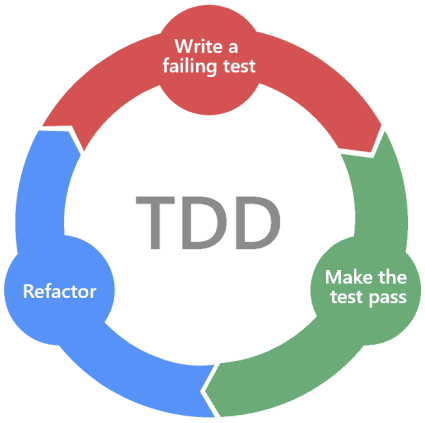
```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn radar_is_palindrome() {
        assert!(is_palindrome("radar"));
    }
}
```

Votre IDE devrait vous offrir la possibilité de passer les tests, sinon avec `cargo` :

```
cargo test
```

Vous devriez constater que l'étape de compilation ne passe logiquement pas. Vous êtes arrivés à la fin de la première étape du cycle TDD :



Bravo ! vous avez pratiqué le TDD. Mais ce n'est pas fini : il va falloir continuer le cycle.

Code pour le premier test

Votre tâche est maintenant d'écrire **le minimum** de code pour que le test passe. Dans un cas simple comme celui du palindrome, vous serez peut-être tentés de commencer à écrire des choses compliqués. Ce n'est pas la bonne façon de faire dans le cas général.

Le premier bout de code doit ressembler à ça :

```rust
fn is_palindrome(chaine: &str) -> bool {
    true
}
```

Constatez que le test passe maintenant.

Le principal intérêt de ce test est de valider le prototype de la fonction et de valider que les outils de test fonctionnent.

Vous êtes arrivés à la deuxième étape du cycle. Ce serait le moment de faire du refactoring. En l'occurrence le code et les tests sont tellement simples qu'il n'y a rien à modifier.

Itérez

Maintenant que le test passe, revenez au début du cycle : **Écrire un nouveau test qui échoue**.

Vous n'avez pas besoin de ré-écrire tout le module de test, uniquement d'ajouter une fonction avec l'annotation '#[test]'

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

    #[test]
    fn radar_is_palindrome() {
        assert!(is_palindrome("radar"));
    }

    #[test]
    fn aspirateur_is_not() {
        assert!(!is_palindrome("aspirateur"));
    }
}
```

Écrivez un code qui fait passer les tests

Ne vous souciez pas tout de suite des autres contraintes sur le palindrome, uniquement ce qui fera passer les tests.

Une version valide pourrait être :

```
fn is_palindrome(chaine: &str) -> bool {
    if chaine == "radar" {
        return true;
    } else {
        return false;
    }
}
```

Ou sa version plus *idiomatique* :

```
fn is_palindrome(chaine: &str) -> bool {
    chaine == "radar"
}
```

Mais on va arrêter la torture à ce stade, **écrivez un premier algorithme qui détecte un mot**

palindrome.

Pensez à bien utiliser les nombreuses fonctions disponibles sur le type `str`

Refactoring

À ce stade vous avez une première version de l'algorithme de détection de palindrome qui fait passer deux tests.

Vous êtes donc dans la phase de refactoring. Il peut être utile à ce stade d'examiner votre code **et vos tests** pour vérifier si une amélioration ne pourrait pas être faites. Si vous décidez de modifier votre code ou vos tests, vous devez valider que les tests continuent de passer **avant** d'écrire un nouveau test.

Terminez le détecteur de palindrome

Continuez d'itérer dans le cycle TDD jusqu'à respecter les spécifications.

Considérations de qualité

Prenez le temps d'évaluer :

- La confiance que vous avez dans votre code.
- L'impact qu'aurait l'ajout d'une contrainte en terme de remise en question de l'existant.
- Le risque associé à l'optimisation de votre algorithme de détection.
- La facilité pour un développeur à comprendre les garanties offertes par votre code.

Bibliothèque de géométrie

Création du projet et mise en Git

Avant de plonger : **Git**.

Cette partie sera évaluée, il vous faudra donc la mettre sous Git pour le rendu. Normalement à ce stade de votre formation, vous devriez maîtriser mais au cas où, un rappel sur les consignes relative au rendu sous Git est là : Consignes pour le rendu Git

Ne faites rien avant d'avoir un dépôt pour la matière sur GitLab, nommé `R4_QualiteDev_Nom1_Nom2` ou `R5_QualiteDev_Nom1_Nom2` suivant l'année, et d'avoir **cloné** ce dépôt en local.

Une fois votre dépôt cloné, ajoutez y un fichier ".gitignore" contenant "*/target".

Création du projet

Dans le clone de votre dépôt Git, créez un nouveau projet Rust nommé "geometry" à l'aide de `cargo` :

```
cargo new --lib geometry
```

Projet de type bibliothèque

Le projet que vous venez de créer est de type bibliothèque. Il ne contient pas de `main.rs`. Vous ne pouvez donc pas lancer d'exécution. Votre code ira dans le fichier `lib.rs`.

Vous pouvez constater qu'en revanche le squelette de tests est déjà présent dans ce fichier. C'est donc dans le fichier lib.rs que vous allez ajouter les fonctionnalités demandées ainsi que les tests correspondant.

Ajoutez et commitez ce projet dans votre dépôt et poussez le sur Gitlab pour être sûr que tout fonctionne.

Mise en pratique du TDD sur la bibliothèque `geometry`

Pour rappel, les règles du TDD :

1. Vous devez écrire un test qui échoue avant de pouvoir écrire le code de production correspondant.
2. Vous devez écrire une seule assertion à la fois, qui fait échouer le test ou qui échoue à la compilation.
3. Vous devez écrire le minimum de code de production pour que l'assertion du test actuellement en échec soit satisfaite.

Structures de données

Avant de commencer à ajouter des fonctionnalités à votre bibliothèque, définissez la structure de données `Vec2D` avec deux composantes, x et y, de type flottant sur 64 bits : `f64`.

Définissez également une structure `Point` avec les mêmes caractéristiques.

Vous pouvez consulter le chapitre du livre à ce sujet.

Fonctionnalités attendues

Créations

1. Des fonctions associées au type `Vec2D` doivent permettre de créer :
    a. un vecteur nul,

   b. un vecteur de coordonnées (1.0, 1.0),
   c. un vecteur à partir de deux flottants donnés en paramètre,
   d. un vecteur à partir de deux points.
2. Des fonctions associées au type `Point` doivent permettre de créer :
   a. Un point d'origine avec les coordonnées (0.0, 0.0),
   b. Un point à partir des coordonnées données en paramètre.

Opérations

1. Une fonction permettant d'ajouter deux vecteurs,
2. Une fonction permettant de déplacer un point en fonction d'un vecteur donné,
3. Une fonction permettant de calculer la norme d'un vecteur,
4. Une fonction permettant d'appliquer une homothétie à un vecteur,
5. Une fonction permettant de normaliser un vecteur,
6. Une fonction permettant d'appliquer une rotation à un vecteur,
7. Une fonction permettant de calculer l'angle entre deux vecteurs,
8. Une fonction permettant de calculer le produit scalaire entre deux vecteurs,
9. Une fonction permettant de déterminer si deux vecteurs sont co-linéaires,

Appliquez les règles du TDD pour réaliser les fonctionnalités demandées.

Commencez par écrire le test qui valide la création d'un vecteur unitaire:

```
#[test]
fn unit_is_unit() {
    let u = Vec2D::unit();
    assert_eq!(u.x, 1.0);
    assert_eq!(u.y, 1.0);
}
```

Le compilateur doit vous insulter puisque vous n'avez pas encore créé cette fonction. Pour satisfaire le compilateur, créez la fonction associée `unit` dans un bloc `impl Vec2D`.

Une fois que cela fonctionne, créez le test pour la fonction pour le vecteur nul, puis la fonction de création `nul`.

Et ainsi de suite.

Structuration du code

Une fois ce processus terminé, pour améliorer la qualité de votre bibliothèque en terme de lisibilité et donc de maintenabilité, vous allez structurer votre code.

Commencez en plaçant les éléments relatifs au type Vec2D dans un module dédié. Consultez la page

pertinente dans le livre de Rust

Il vous faudra créer deux **modules** : `vec2d` et `point`.

Pour créer un module, vous avez deux options :

- Créer un **fichier** `nom_module.rs` au même niveau que le fichier `lib.rs` ou
- Créer un **répertoire** `nom_module` au même niveau que le fichier `lib.rs` et dans ce répertoire, créer un fichier `mod.rs`.

Pour que le compilateur prenne en compte ces nouveaux modules, il faudra ajouter en haut du fichier `lib.rs` : `mod nom_module`.

Pour pouvoir utiliser les fonctions définies dans votre module, vous devez ajouter :

```
use nom_module::NomType;
```

Vous pourrez ensuite utiliser les fonctions de votre type avec :

```
NomType::uneFonction();
```

Chaînage des opérations

L'objectif ici est de rendre possible le chaînage des opérations :

```
let mut p = Point::origin();
let mut v = Vec2D::unit();
p.translate(v.scale(3.0).rotate(PI).add(Vec2D::unit()));
```

Pour cela, il faudra que les opérations de modification retournent une référence vers le vecteur modifié.

Évaluation

Seule la partie sur la bibliothèque "geometry" sera évaluée.

Les critères d'évaluation sont les suivants :

- Présence de tests pour valider les fonctionnalités et notamment les **cas limites** (translation par un vecteur nul, rotation d'un angle de 2π).
- Organisation du code :
  - Modules placés dans des fichiers ou répertoires dédiés et formant des unités logicielles cohérentes.
  - Cohérence et uniformité dans l'organisation des déclarations de structures et de

fonction associées.

- Documentation des modules, structures et fonctions.
  - Idéalement en ajoutant des exemples d'utilisation qui passeront en tests.
- Respect de la norme de codage Rust (formatage).
- Réalisation des fonctionnalités demandées.
- Réalisation d'un Readme décrivant rapidement la bibliothèque et pointant vers la documentation générée.
- Présence de la documentation générée dans le dépôt Git, dans le répertoire "doc".

N'oubliez pas de pousser votre travail pour qu'il puisse être évalué.

Si vous ne travaillez pas dans la branche 'main', précisez-le **clairement** dans le Readme.