

M3105 - TD n°2 : Ecrire du code SOLID

Kata Car Racing : tirePressureMonitoringSystem

Système de surveillance de pression des pneus

Ce code pourrait être du code existant que vous venez d'hériter d'un collègue.

La classe d'alarme (**Alarm**) est conçue pour surveiller la pression des pneus et déclencher une alarme si la pression sort de la plage prévue.

```
public class Alarm {
    private final double lowPressureThreshold = 17;
    private final double highPressureThreshold = 21;

    private Sensor sensor = new Sensor();

    private boolean alarmOn = false;

    public void check() {
        double psiPressureValue = sensor.popNextPressurePsiValue();

        if (psiPressureValue < lowPressureThreshold || highPressureThreshold < psiPressureValue) {
            alarmOn = true;
        }
    }

    public boolean isAlarmOn() {
        return alarmOn;
    }
}
```

La classe de capteurs **Sensor** simule le comportement d'un capteur de pneu réel, fournissant des valeurs aléatoires mais réalistes.

```
public class Sensor
{
    public static final double OFFSET = 16;

    public double popNextPressurePsiValue()
    {
        double pressureTelemetryValue;
        pressureTelemetryValue = samplePressure();

        return OFFSET + pressureTelemetryValue;
    }

    private static double samplePressure()
    {
        // placeholder implementation that simulate a real sensor in a real tire
        Random basicRandomNumbersGenerator = new Random();
        double pressureTelemetryValue = 6 * basicRandomNumbersGenerator.nextDouble() *
                                         basicRandomNumbersGenerator.nextDouble();

        return pressureTelemetryValue;
    }
}
```

1. Revue de code...

Cette question doit être traitée en **mode déconnecté** :
Laissez vos ordinateurs éteints pour le moment et prenez une feuille de papier ☹️

1.a Réalisez le diagramme de classe de ce code *legacy*.

1.b Ce code ne suit pas certains principes SOLID. Identifiez ces principes, rappelez leur définition et indiquez en quoi ces principes ne sont pas respectés ici.

1.c Identifiez-vous d'autres mauvaises odeurs (*code smell*) dans ce code ?

Remarque : Pour vous aider dans votre réflexion, essayez de répondre aux questions suivantes :

- Que pensez-vous de l'instanciation du capteur ?
- Que pensez-vous des seuils de pressions haut et bas ?
- Que pensez-vous de l'expression qui permet de vérifier que la pression se situe exactement dans un intervalle souhaité ?

2. Ecrire du code SOLID...

Vous pouvez maintenant passer en **mode connecté** 😊

2.a Dans votre IDE préféré, créez un projet maven **tirepressuremonitoringsystem** dans lequel vous importerez dans **src/main/java** les classes **Alarm** et **Sensor**, à récupérer dans le répertoire **ressources/ racingcar/tirepressuremonitoringsystem** du dépôt <https://github.com/iblasquez/enseignement-iut-m3105-conception-avancee>

Avant de vous lancer dans un quelconque refactoring, **il vous faut mettre en place un harnais de tests** pour garantir le comportement actuel contre d'éventuelles régressions.

La logique métier se trouvant actuellement seulement dans la méthode **check** de la classe **Alarm**. Vous devez commencer par mettre en place des tests unitaires autour de cette méthode afin de garantir (et couvrir) le comportement suivant :

- o l'alarme se déclenche en cas de valeur de pression trop basse.
- o l'alarme se déclenche en cas de valeur de pression trop forte.
- o l'alarme ne se déclenche pas pour une valeur dans le seuil de sécurité.
- o Une fois que l'alarme s'est déclenchée, elle reste déclenchée quelle que soit la valeur sondée suivante, même si cette dernière revient dans le seuil de sécurité.

A vous d'écrire une classe de tests : **AlarmTest** (dans **src/test/java** : clic droit → New → JUnit Test Case) permettant de couvrir 100% du code écrit à refactorer 😊 !

N'oubliez pas de versionner votre projet et de commiter régulièrement !!

2.b Refactorisez pour rendre le code plus propre et SOLID ☺

Pour guider votre refactoring, nous vous proposons de suivre les étapes suivantes :

- 2.b.1 Pour commencer montrez l'intention c-a-d aidez-vous (de plusieurs) **Extract Method** pour rendre la méthode **check** la plus expressive possible.
- 2.b.2 Travaillez ensuite **autour des constructeurs** pour redistribuer les instanciations des attributs non constants au sein des constructeurs et ne conserver en début de classe que les déclarations de ces attributs.
- 2.b.3 Travaillez ensuite **autour du capteur** pour rendre votre code plus **SOLID** en étant par exemple moins dépendant des détails ...

Assurez-vous que sémantiquement vos classes soient « correctes »
*c-a-d un nommage en accord (et seulement en accord)
avec l'intention métier de la classe ...*

- 2.b.4 Intéressez-vous enfin au concept d'intervalle, notamment à la notion d'**intervalle de sécurité** (SafetyRange) qui pourrait notamment vous aider à rendre plus expression l'expression du genre :

```
return value < lowThreshold || highThreshold < value
```

en gérant l'appartenance d'une valeur à un intervalle fixé par des seuils...
La responsabilité des seuils se verrait ainsi plutôt attribuée à un intervalle qu'à une alarme (dans l'idée que les seuils n'apparaissent plus directement comme des attributs de la classe Alarm)...
Bien sûr, dans ces conditions, une alarme ne pourra exister que si elle est créée à partir d'un capteur et d'un intervalle de sécurité.
Essayez de mettre en place ce nouveau concept de vous-même. Si vous avez un peu de mal à vous organiser, vous pouvez consulter en annexe quelques étapes à suivre pour mener à bien ce refactoring ☺

**Veillez bien qu'à la fin de votre refactoring
les classes soient sémantiquement « correctes »
et ne contiennent que des termes métiers simples qui leur sont propre ☺**

2.c Analysez le code avec SonarLint pour voir rapidement si vous pouvez y apporter quelques petites améliorations supplémentaires.

2.d Générez le diagramme de classes à partir de votre code (via [ObjectAid UML Explorer](#) sous Eclipse). Comparez ce diagramme avec celui que vous aviez dessiné à la question 1. Le code est-il plus SOLID ?

2.e Poussez votre code vers votre dépôt distant ...

Remarque : Le refactoring est une activité subjective...
Quand faut-il s'arrêter ? C'est vous qui jugez ☺...

Annexe : Mise en place du concept d'intervalle de sécurité (SafetyRange)

Suggestions de quelques étapes pour mener à bien ce refactoring...

→ **Créer une nouvelle classe métier SafetyRange** pour modéliser le concept d'intervalle de sécurité qui sera responsable de gérer ce qui attrait aux seuils de sécurité.

→ **Attribuer un intervalle de sécurité à l'Alarm**

→ **Déléguer** la recherche sur la sécurité (dans la classe Alarm) à l'intervalle de sécurité

→ **Injecter l'intervalle de sécurité dans le constructeur de la classe Alarm** et faire en sorte que la dépendance de la classe Alarm avec les seuils d'intervalle disparaisse.