M3105 – TD : Améliorer la lisibilité des tests à l'aide de patterns de création
Kata Car Racing : `tirePressureMonitoringSystem`

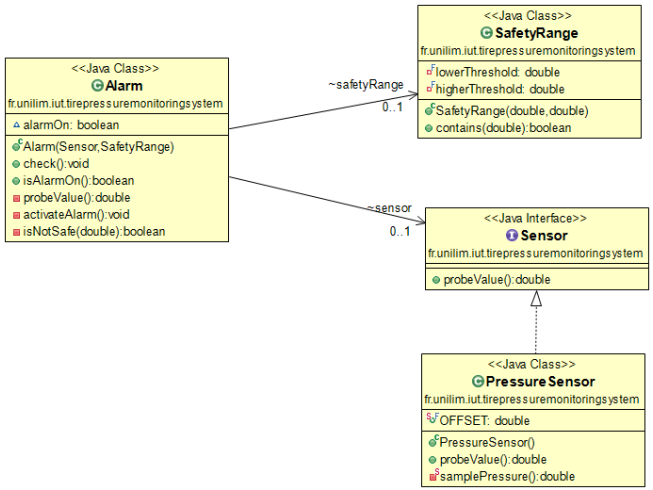
Système de surveillance de pression des pneus

Il vous est conseillé de faire ce TD en pair-programming 😉

Dans ce TD, nous allons reprendre le code produit lors d'un TD précédent :
Ecrire du code SOLID Kata Car Racing : `tirePressureMonitoringSystem`.

Ré-ouvrez donc le projet **`tirePressureMonitoringSystem`** dans votre IDE.

Suite au refactoring mis en place lors du précédent TD, vous devriez disposer :
- au niveau du code production **(`src/main/java`)** : d'une classe **`Alarm`**, d'une classe **`PressureSensor`**, d'une interface **`Sensor`**, d'une classe **`SafetyRange`**
- au niveau du code de test **(`src/test/java`)** : d'une classe **`AlarmTest`**



Remarque : Si nécessaire, vous pouvez récupérer le code de production (**`Alarm`**, **`PressureSensor`**, **`Sensor`**, **`SafetyRange`**) et le code de test (**`AlarmTest`**) via le lien suivant :
http://unil.im/tireSOLID

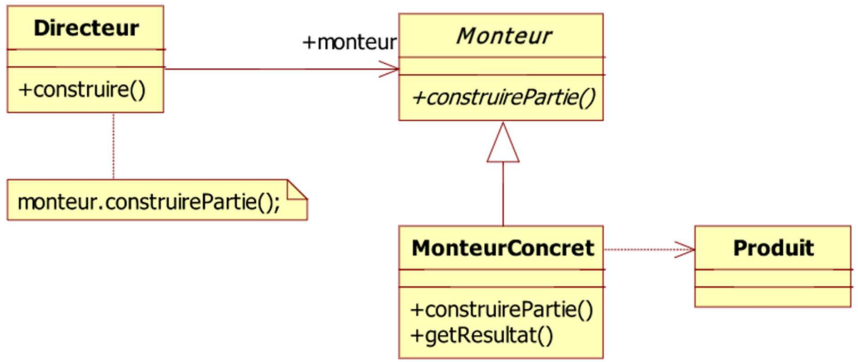
Assurez-vous que vos tests passent AU VERT avant de continuer !!!

Ce TD va permettre d'améliorer la lisibilité des tests du projet **`tirePressureMonitoringSystem`**.
Pour créer des objets de manière plus explicite,
nous allons mettre en place des patterns de création de type **`Builder`** et **`Factory Method`**,
en s'appuyant sur l'intention de ces patterns,
mais pas forcément en reprenant l'implémentation originale proposée par le Gof 😉

Exercice n°1 : Créer une alarme à l'aide d'un `Builder` fluent (`Monteur`)

Le Gang of Four (GoF) définit le pattern **Builder** (**Monteur**) de la manière suivante :
« Le pattern Builder est utilisé pour la création d'objets complexes dont les différentes parties doivent être créées suivant un certain ordre ou algorithme spécifique. Une classe externe contrôle l'algorithme de construction »

1. Dans le Gof, le diagramme de classes de ce pattern est le suivant :
   (extrait de http://www.goprod.bouhours.net)



Participants au patron :

- **Monteur**
  Spécifie une interface abstraite pour la création de parties d'un objet Produit.
- **MonteurConcret**
  Construit et assemble des parties du produit par implémentation de l'interface Monteur.
  Définit la représentation qu'il crée et en conserve la trace.
  Fournit une interface pour la récupération du produit final.
- **Directeur**
  Construit un objet en utilisant l'interface de Monteur.
- **Produit**
  Représente l'objet complexe en cours de construction. MonteurConcret construit la représentation interne du produit et définit le processus par lequel il est assemblé.
  Comporte les classes qui définissent les parties constitutives, y compris les interfaces nécessaires à l'assemblage des parties pour donner le résultat final.

Rendez-vous dans le dépôt https://github.com/iblasquez/enseignement-iut-m3105-conception-avancee où vous retrouverez les travaux de vos collègues autour de ce pattern, jetez-y un petit coup d'œil.

L'intention de ce pattern est de
dissocier la construction d'un objet de sa représentation,
de sorte que le même processus de construction permette des représentations différentes.

En pratique, le **pattern builder** est très utilisé,
Son intention est respectée, mais son implémentation ne s'appuie pas toujours sur l'implémentation suggérée par le **Gof** au travers du diagramme de classes précédent 😉

En effet, suivant le contexte, le **pattern builder** peut être implémenté de manière plus ou moins complexe. L'article ***Design pattern : Builder et Builder sont dans un bateau*** sur le blog de Xebia présente différentes implémentations possibles pour ce pattern :

- de la plus simple, sous forme de **builder fluent**, avec enchaînement de méthodes : `new UneClasseAMonter().avecUnArgument().avecUnAutreArgument().build()`
- … à une implémentation classique du **Gof**

Jetez un petit coup d'oeil à cet article : http://unil.im/builderxebia

Dans notre contexte (système de surveillance de pression des pneus), nous disposons de peu de paramètres et de peu de complexité (seulement une alarme composée d'un capteur et un intervalle de sécurité). Nous allons donc nous contenter d'implémenter un **pattern builder** sous sa forme la plus simple c-a-d **sous la forme d'un builder fluent**.

Vérifiez que vos tests passent AU VERT avant de continuer !!!

2. Nous souhaitons mettre en place un **builder** pour améliorer la lisibilité des tests.
   Pour commencer, focalisons-nous sur le premier test qui est actuellement écrit de la manière suivante :

```
@Test
public void alarmeSeDeclenche_EnCasDeValeurTropBasse() {
	Alarm alarm = new Alarm(sensorThatProbes(0.0), new SafetyRange(17, 21));
	alarm.check();
	assertTrue(alarm.isAlarmOn());
}
```

   En s'inspirant de l'article de Xebia, et comme une alarme *utilise* un capteur (`Sensor`) *avec* un certain intervalle de sécurité (`SafetyRange`), on se dit que l'étape **A**rrange de ce test (construction de l'`Alarm`) pourrait être implémentée via un **builder fluent** de la manière suivante :

```
@Test
public void alarmeSeDeclenche_EnCasDeValeurTropBasse() {
	Alarm alarm = new AlarmBuilder()
			.usingSensor(sensorThatProbes(0.0))
			.withSafetyRange(17, 21)
			.build();
	alarm.check();
	assertTrue(alarm.isAlarmOn());
}
```

2.1 Refactoring autour de la construction de l'`Alarm` pour y faire apparaître un builder fluent

Autrement dit, modifiez l'implémentation de l'étape **Arrange** du premier test comme indiqué ci-dessus (changement à effectuer pour l'instant uniquement dans ce test : petit pas par petit pas 😉)
Pour que ce code puisse compiler, il ne reste plus qu'à créer la classe **`AlarmBuilder`** et à implémenter ses méthodes **`withSafetyRange, usingSensor`** et **`build`**, c'est l'objet des questions suivantes …

2.2 Création de la classe `AlarmBuilder` :

Où créer la classe **`AlarmBuilder`** ?
Dans `src/test/java` (oui bien dans `test` puisqu'on souhaite que ***cette nouvelle classe nous aide*** à améliorer la lisibilité de nos **tests**).
Créez donc dans `src/test/java` un nouveau package `helpers` dans lequel vous ajouterez la classe **`AlarmBuilder`** (générée automatiquement via l'IDE depuis le code de test 😉)

2.3 Implémentation des méthodes `withSafetyRange, usingSensor` et `build` :

Créez les méthodes **`withSafetyRange, usingSensor`** et **`build`** de la classe **`AlarmBuilder`** via l'IDE et implémentez ces méthodes afin de faire passer le test AU VERT !!!

2.4 Améliorer la lisibilité en proposant une méthode statique `anAlarm` au lieu d'un appel direct au constructeur… `withSafetyRange, usingSensor` et `build` :

- Commencez par modifier l'étape Arrange du test de la manière suivante

```
@Test
public void alarmeSeDeclenche_EnCasDeValeurTropBasse() {
	Alarm alarm = AlarmBuilder.anAlarm()
			.usingSensor(sensorThatProbes(0.0))
			.withSafetyRange(17, 21)
			.build();
	alarm.check();
	assertTrue(alarm.isAlarmOn());
	}
```

- Puis implémentez la méthode **`static`** **`anAlarm`** dans la classe `AlarmBuilder` pour faire passer ce test !

- Positionnez votre curseur sur ***`anAlarm()`***, sur clic droit **`Source -> Add Import`** pour que l'IDE ajoute automatiquement l'**`import static`** sur `AlarmBuilder.anAlarm`
  et améliorant ainsi encore un peu la lisibilité du test qui devrait ressembler à :

```
@Test
public void alarmeSeDeclenche_EnCasDeValeurTropBasse() {
	Alarm alarm = anAlarm()
			.usingSensor(thatProbes(0.0))
			.withSafetyRange(17, 21)
			.build();
	alarm.check();
	assertTrue(alarm.isAlarmOn());
	}
```

- Vérifiez que vos tests passent toujours AU VERT !!!

- Renommez `sensorThatProbes` par `thatProbes` (pour supprimer la redondance autour du terme `sensor` et ne lire que : **`usingSensor(thatProbes(0.0))`**

- Vérifiez que vos tests passent toujours AU VERT !!!

3. Il ne vous reste plus qu'à modifier test par test l'étape **Arrange** de chaque test de sorte que cette étape soit désormais implémentée par un **builder fluent**.
   Petit pas par petit pas c-a-d un test à la fois et relance des tests à chaque fois 😉

Exercice n°2 : Qui a la responsabilité de faire sonder les capteurs ?

Pensez-vous que ce soit de la responsabilité de la classe **`AlarmTest`** de faire sonder les capteurs c-a-d de proposer des services (méthodes **`thatProbes`**) qui permettent de *fabriquer* un capteur (mocké) qui sonde une (ou plusieurs) valeur(s)?
Si non, proposez et implémentez une solution simple qui permettrait de mieux répartir cette responsabilité 😉