R3.04 : Kata Car Racing : Le retour !
Encore plus SOLID et clean !

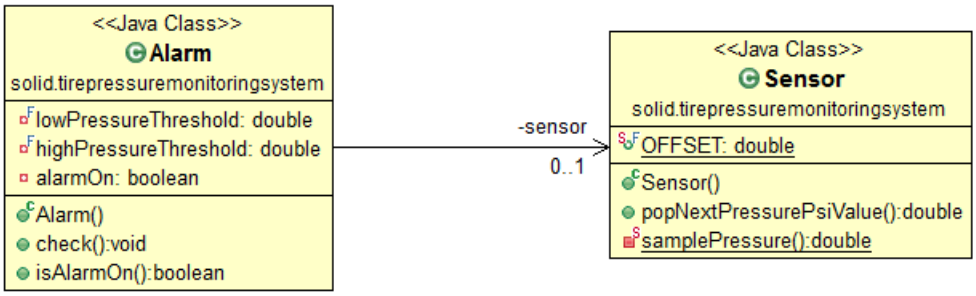
Un code de production encore plus SOLID
Un code de tests encore plus propre (patterns de création à la rescousse 😉)

Exercice 1 : Quid de la SOLIDité du code ?

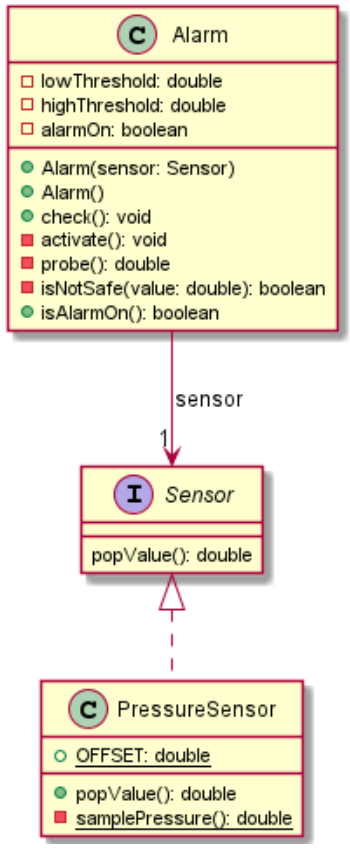
a. Quid de l'existant ? (OCP DIP)

Dans un précédent TP, vous avez eu l'occasion de travailler sur le code existant d'un projet issu d'un *Système de surveillance de pression des pneus* qui mettait en œuvre une **alarme** et un **capteur** dans un projet nommé **`tirePressureMonitoringSystem`**.

Vous avez commencé par récupérer le code existant de ce projet (**code legacy**), qui après rétroconception, vous a permis de visualiser l'état actuel de la conception du projet au travers du diagramme de classes suivant :



Après avoir détecté quelques ***mauvaises odeurs dans ce code*** (*bad smells*) et après **avoir analysé le manque de SOLIDité** de ce code legacy, vous avez procédé pas à pas à un premier ***refactoring*** **(remaniement de code) autour des principes OCP et DIP** qui vous a amené à faire évoluer la conception vers une conception similaire au diagramme de classes ci-contre.



Pour mener à bien ce nouveau TP, vous devez au minimum **repartir d'une telle conception.**

Dans votre IDE préféré, ouvrez le projet **`tirePressureMonitoringSystem`** et retrouvez le code que vous avez remanié au TP précédent 😉
Vérifiez que ce code compile bien et que les tests passent AU VERT.

Procédez maintenant à une **rétroconception de votre code,** qui va vous permettre de continuer en mode « *Le TP dont vous êtes le héros !* » 😉

⇨ **Si le diagramme de classes obtenu ressemble au diagramme de classes ci-contre principes de OCP et DIP respectés)**, vous pouvez passer à la question suivante (b. Quid des responsabilités de l'alarme)

⇨ **Si le diagramme de classes obtenu ne fait pas apparaître l'interface *Sensor* et la classe `PressureSensor`,** vous pouvez récupérer le code correspondant à la conception précédente en allant sur ce lien : **http://unil.im/tireSOLID**
(raccourci de https://gist.github.com/iblasquez/5d410e20e5b6620c88f6496f2a639152)
Vérifiez que ce code compile bien et que les tests passent AU VERT.
Commitez avec un message du genre : ***refacto with a Sensor abstraction (DIP OCP).***

⇨ **Si le diagramme de classes obtenu fait apparaître un intervalle de sécurité,** (c-a-d qu'en plus de OCP et DIP, vous avez été attentif à SRP), vous pouvez passer directement à l'exercice 2 (mais rien ne vous empêche pas de lire quand même la question suivante) 😉

b. Quid des responsabilités de l'alarme ? (SRP)

Une alarme est en général conçue pour se déclencher si la valeur qu'on lui fournit (ici plus spécifiquement la pression des pneus) n'est pas une valeur sûre.
Il est donc normal que la classe `Alarm` :

- **offre un service** qui permet de vérifier si la valeur est déclenché ou non :
  ⇨ **méthode publique** : `isAlarmOn`

- **offre un service** qui permet de « commander » l'alarme en lui demandant de **vérifier** si la valeur qu'elle doit surveiller est toujours sûre et si ce n'est pas la cas, l'alarme doit se déclencher.
  ⇨ **méthode publique** : `check`

- quand l'alarme se déclenche, elle **modifie son état**.
  Pour respecter le principe d'encapsulation, il faut que la modification de cet état ne puisse pas être accessible de l'extérieur, mais que **seule l'alarme puisse le contrôler**.
  ⇨ **méthode privée** : `activate`

- retour sur la méthode `check` qui est composée de deux étapes :

  - **la première étape consiste à récupérer une valeur issue d'un capteur.**
    Pour améliorer la lisibilité du code cette première étape a été extraite dans une méthode `probe` qui est donc une **méthode privée**.
    Dans cette méthode, l'instruction suivante `sensor.popValue() ;`
    indique très explicitement que la responsabilité de récupérer la valeur (`popValue`) est déléguée au capteur(`sensor`)

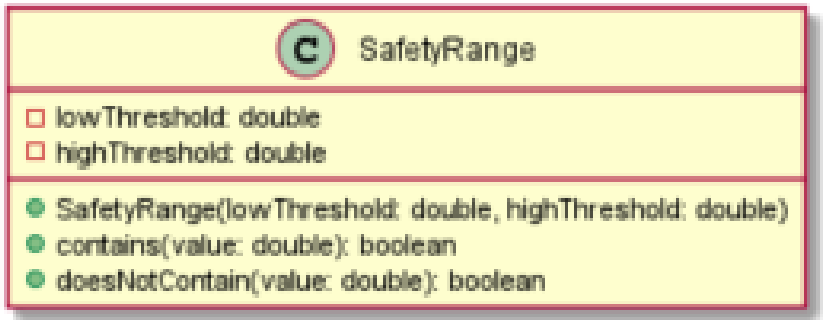
  - **la deuxième étape consiste à traiter cette valeur et réagir en conséquence**
    c-a-d que l'alarme doit s'activer si la valeur n'est pas sûre.
    Mais que signifie une valeur sûre ?
    Une valeur sûre est une valeur qui appartient à un intervalle de sécurité.
    En théorie, est-ce que cet intervalle de sécurité doit être le même pour tous les capteurs ? ❑ OUI ❑NON

    Si votre réponse est NON, vous avez bien compris que **l'alarme n'est pas responsable de vérifier si une valeur donnée appartient (ou pas) à un intervalle de sécurité donné**, mais que cette **vérification doit être déléguée à l'intervalle de sécurité** et donc, en plus d'un capteur (pour respecter OCP et DIP) une alarme doit avoir un capteur de sécurité (pour respecter SRP) 😉

b.1 Ajouter une nouvelle classe métier `SafetyRange` (intervalle de sécurité)

❑ Pour refactorer en *toute sécurité*, sans *casser* le code opérationnel existant, placez-vous sur votre dernier commit est, à l'aide de votre IDE ou directement en ligne de commande à côté de votre IDE**, créer une branche que vous appelerez *refacto_SRP* et assurez-vous maintenant que vous êtes bien en train de travailler sur cette branche.**

❑ Pour bien débuter ce refactoring, vous ne toucherez pas au code existant, mais vous commencerez par **créer une nouvelle classe métier `SafetyRange`** dont le diagramme de classes est le suivant :



En effet, un intervalle de sécurité :

- est nécessairement défini par deux valeurs, une pour chaque seuil de l'intervalle : **`lowThreshold`** et **`HighThreshold`**
- doit être capable d'offrir des services qui permettent par exemple de savoir :
  - si une valeur appartient à l'intervalle : **`contains(value :double)`**
  - si une valeur n'appartient à l'intervalle : **`doesNotContains(value :double)`**

❑ Une fois que vous aurez implémenté cette classe, vous ajouterez une nouvelle classe de tests **`SafetyRangeTest`** avec 2 tests et 4 assertions permettant de tester de manière exhaustive les deux méthodes **`contains`** et **`doesNotContains`** c-a-d qu'une valeur est bien contenue ou pas dans dans intervalle de sécurité.

❑ Si votre code compile et que tous vos tests passent AU VERT, procédez à un petit commit avec un message du genre : **`add SafetyRange`** avant de passer à la suite 😉

b.2 Faire en sorte qu'une alarme *délègue* la vérification des seuils à un intervalle de sécurité

Pour que l'alarme puisse déléguer la sécurité à un intervalle de sécurité, il faut qu'une alarme soit associée à un intervalle de sécurité c-a-d

une **`Alarm`** A-UN **SafetyRange**

⇒ Modélisez la phrase suivent sous forme de diagramme de classe UML 😉

Avant de vous lancer dans quoi que ce soit, bien réfléchir à l'impact que pourrait avoir l'introduction d'un intervalle de sécurité sur le code existant de l'alarme, et notamment :

- quel est l'impact de ce refactoring sur les attributs actuels de la classe `Alarm` ?
- quel est l'impact de ce refactoring sur le constructeur de la classe `Alarm` ?

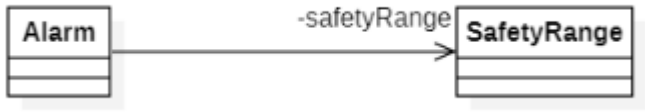
Réfléchir à ces questions en terme d'**ajout** de code, de **modification** de code, de **suppression** de code en gardant bien à l'esprit qu'aucun comportement existant ne doit être perdu lors de ce refactoring 😉

Pour procéder en toute sécurité à ce refactoring, vous allez bien sûr effectuer des **petits pas et utiliser l'IDE** pour répercuter automatiquement les modifications que vous allez effectuer autour du constructeur dans le fichier tests 😉

2.1 Quid de *l'ajout de code* lié à l'intervalle de sécurité ?

Une bonne pratique pour refactorer un code en toute confiance est de s'assurer qu'à chaque petit pas, tous les tests passent AU VERT !
Commencer par **ajouter du code**, est une bonne pratique car le simple ajout de nouveau code ne devrait, dans un premier temps, n'avoir aucun impact sur le code existant et donc les tests devraient continuer à passer AU VERT en toute quiétude.

La conception suivante :



a comme impact dans votre implémentation, **l'apparition d'un nouvel attribut** (de type **`SafetyRange`**) dans la classe `Alarm`.

⇒ Procédez à ce petit *ajout de code* dans votre classe **`Alarm`**.

Après cet ajout, vérifiez que le code compile
et que les tests continuent à passer AU VERT !

2.2 Quid de la *modification de code* lié à l'ajout de l'intervalle de sécurité ?

La présence d'un attribut de type `SafetyRange` dans la classe `Alarm` nous interroge maintenant sur la pertinence de la présence des attributs suivants dans la classe `Alarm` :

```
private final double lowThreshold = 17;
private final double highThreshold = 21;
```

Il est également à noter que pour l'instant, le seul intervalle de sécurité utilisé dans le projet est celui qui possède les valeurs 17 et 21.

❑ **2.2.a** Pour pouvoir être en mesure de supprimer les deux précédentes instructions (déclaration de `lowThreshold` et de `highThreshold`) de la classe `Alarm`, il faut **spécifier cet intervalle spécifique lors de la création de l'alarme**, c-a-d ajouter directement dans le constructeur primaire l'instruction suivante :

```
public Alarm(Sensor sensor) {
      this.sensor=sensor;
      this.alarmOn = false;
      this.safetyRange = new SafetyRange(17, 21);
      }
```

⇒ Procédez à ce petit *ajout de code* dans le constructeur de votre classe **`Alarm`**.

Après voir procédez à ce petit pas de refactoring
(ajout de code pour introduire SafetyRange)
vérifiez que le code compile et que les tests continuent à passer AU VERT !

❑ **2.2.b** Associer un intervalle de sécurité à l'alarme doit permettre, au final, de pouvoir passer n'importe quel intervalle de sécurité en paramètre à l'alarme et faire ainsi en sorte que **le constructeur primaire prenne en paramètres tous les attributs de la classe :** l'alarme (ce qui est déjà le cas) **et l'intervalle de sécurité !**

c-a-d que la signature actuelle du constructeur primaire

```
public Alarm(Sensor sensor)
```

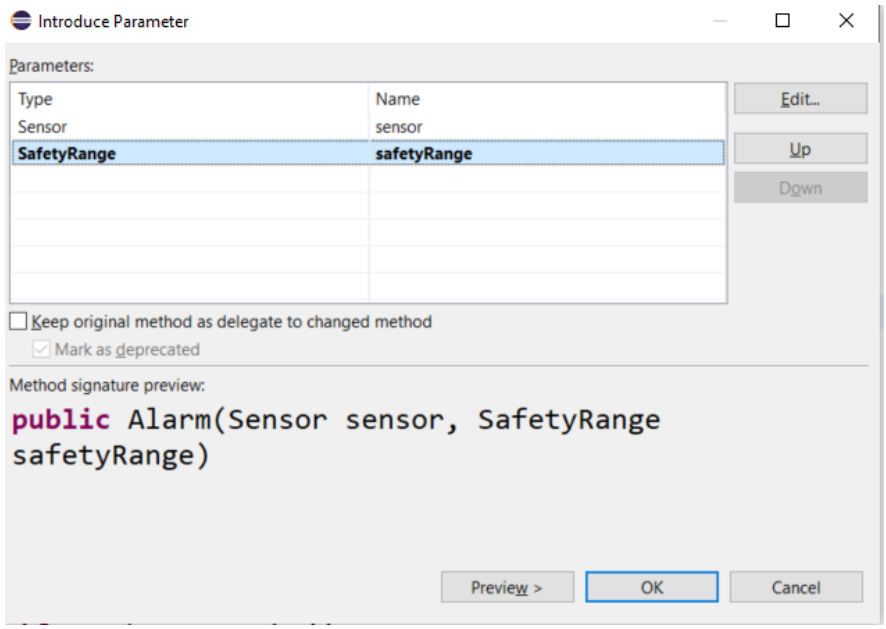
devienne la signature suivante :

```
public Alarm(Sensor sensor, SafetyRange safetyRange)
```

Encore une fois, l'IDE va vous aider à effectuer ce refactoring en toute confiance.

Sélectionnez dans le constructeur, le bout d'instruction :
`new SafetyRange(17, 21)`

Cherchez dans les options de refactoring de votre IDE préféré, un refactoring du type **Introduce Parameter** et faites en sorte qu'à l'issue de ce refactoring la signature de la méthode corresponde à la signature attendue 😉



Que se passe-t-il dans votre IDE lorsque vous lancer ce refactoring ?

➔ dans la classe **`Alarm`** :
- **dans le constructeur primaire** : **`safetyRange`** est désormais en paramètre
- **dans le constructeur par défaut**, l'intervalle par défaut **`new SafetyRange(17, 21)`** est explicitement appelé.

➔ dans la classe **`AlarmTest`**, toutes les alarmes sont explicitements instanciées avec l'intervalle par défaut **`new SafetyRange(17, 21)`**

Après voir procédez à ce petit pas de refactoring
(modification de code au travers du refactoring de l'IDE IntroduceParameter)
vérifiez que le code compile et que tous les tests continuent à passer AU VERT !

❑ **2.2.c** Maintenant que nous nous sommes assuré que l'intervalle de sécurité était bien pris en compte dans la création de l'alarme, **il est temps de déléguer la responsabilité de la vérification de la non-appartenance d'une valeur à l'intervalle de sécurité .**

Faites en sorte que la méthode que cette délégation (dans la classe **`Alarm`**) se fasse dans la méthode concernée à savoir `private boolean isNotSafe(double value).`

Cette dernière doit donc se réduire à une seule instruction qui appellera la *bonne* méthode de la classe `SafetyRange` 😉

Après voir procédez à ce petit pas de refactoring
(modification de code au travers de la mise en place de la délégation de la non-appartenance d'une valeur à l'intervalle de sécurité)
vérifiez que le code compile et que tous les tests continuent à passer AU VERT !

❑ **2.2.d Terminer par un petit nettoyage du *code mort***

Balayez la classe **`Alarm`** et constatez que les deux lignes suivantes sont du code *mort*

```
private final double lowThreshold = 17;
private final double highThreshold = 21;
```

car `lowThreshold` et `highThreshold` ne sont pas utilisées dans le code de la classe !

Supprimez donc ces deux instructions,
vérifiez que le code continue de compiler et
que tous les tests continuent à passer AU VERT !

❑ **2.2.e … sans oublier le commit pour marque la fin de ce refactoring !**

Comme votre code compile et que tous vos tests passent AU VERT, il est temps de procéder un petit **commit** avec un message du genre : **`alarm with SafetyRange delegation (SRP)`** avant de passer à la suite 😉

c. Quid de `master` et du refactoring autour de SRP ?

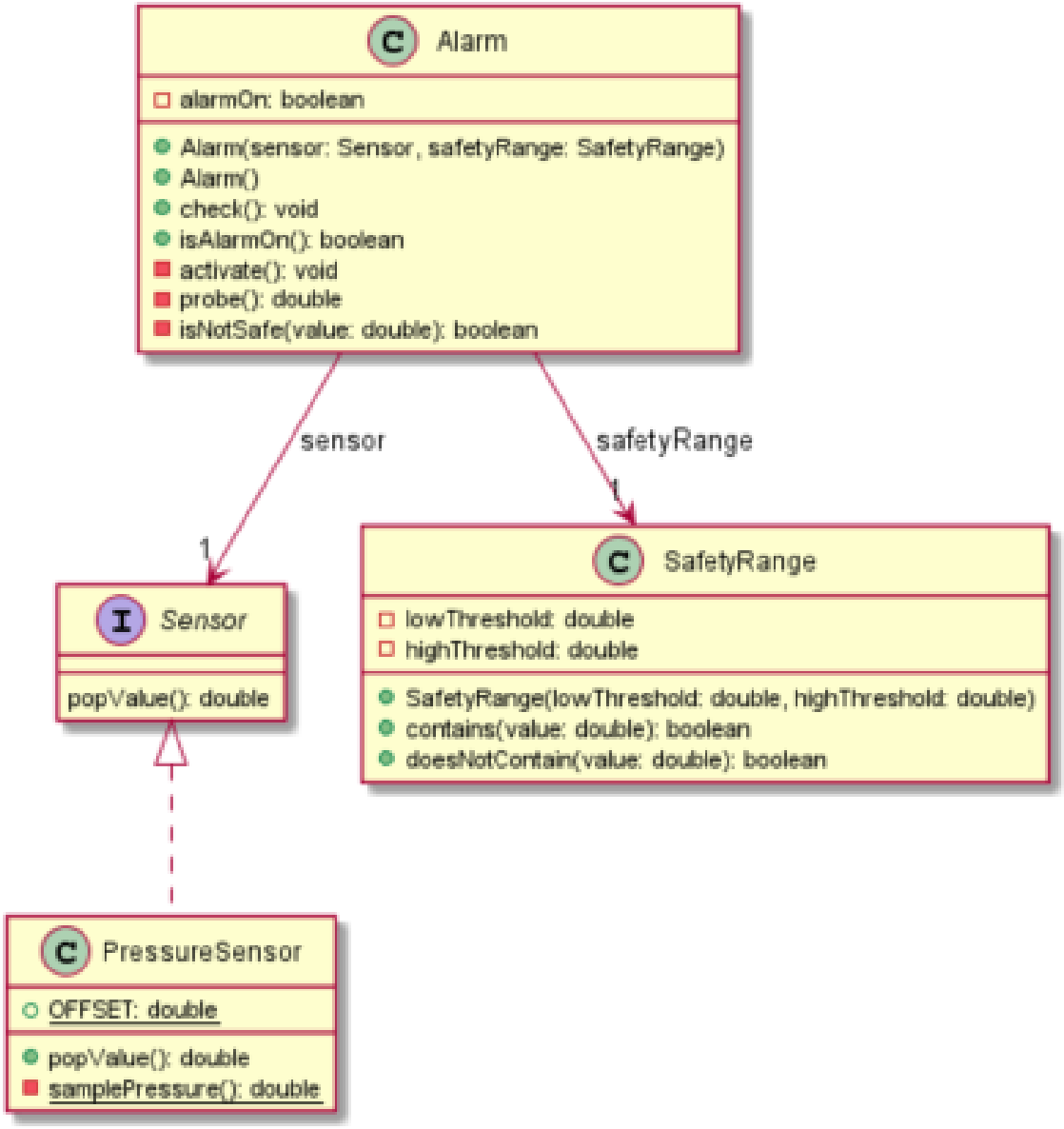
Nous venons de procéder tranquillement à un refactoring autour de SRP dans la **branche `refacto_SRP`** tout en conservant un master fonctionnel 😉

Il est temps de merger les modifications apportées par le refacto dans master, avant que master dispose d'un code fonctionnel et SOLID 😉

Pour merger la branche refacto_SRP dans master, vous pouvez, selon vos préférences :

- Soit le faire directement avec le plug-in git de votre IDE préféré.
- Soit le faire en ligne de commande (hors IDE)
- Soit le faire avec un outil de versionning visuel de type Gitkraken (hors IDE)

Quoi qu'il en soit, avant de commencer l'exercice suivant, vous devez **vous placer sur le dernier commit de la branche `master`** c-a-d sur le commit issu de la fusion (merge) et vérifiez que le projet offre un code SOLID avec une alarme (**`Alarm`**) qui dispose d'un capteur (**`Sensor`**) et d'un intervalle de sécurité (**`SafetyRange`**) et donc que la rétroconception est similaire à la conception suivante :



Exercice n°2 : Créer une alarme à l'aide d'un `Builder` fluent (`Monteur`)

Pour réaliser cet exercice, vous devez partir d'un projet dont la rétroconception est similaire au diagramme de classes de la feuille précédente !
… avec bien sûr un code qui compile et tous les tests qui passent AU VERT …

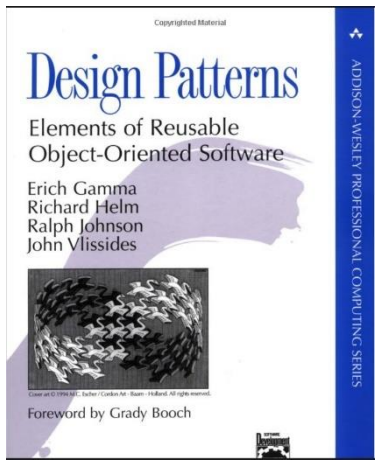
Cet exercice vise à **améliorer la qualité du code des tests**,
au travers de la **lisibilité de la création des objets utilisés dans les tests** du projet **`tirePressureMonitoringSystem`**.

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 😉

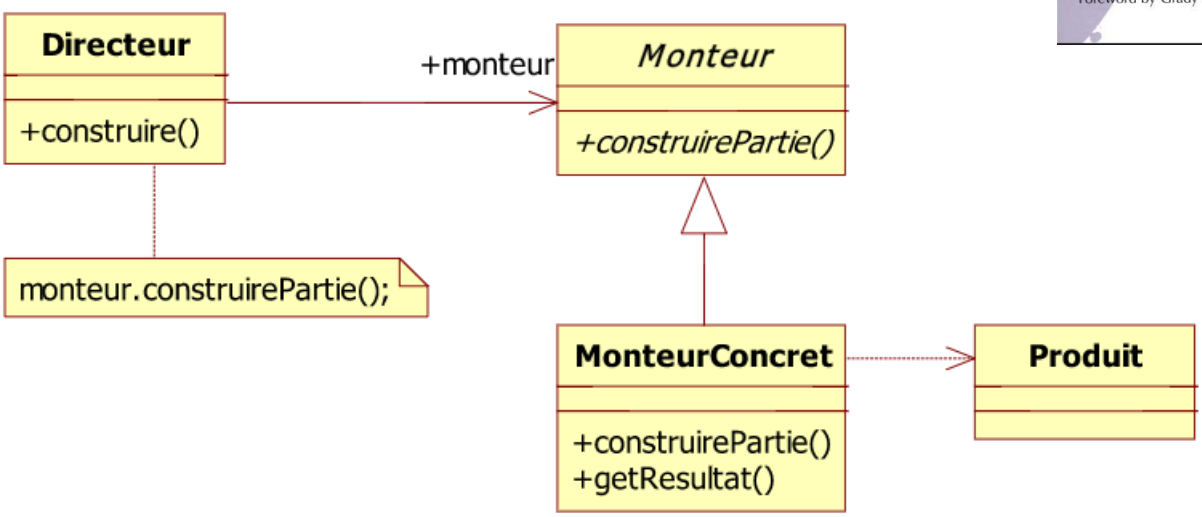
❑ Nous allons donc réaliser cet exercice dans une **branche** que nous mergerons ensuite à **`master`** 😉
⇨ A partir du dernier commit de **`master`**, créez une nouvelle branche **refacto_clean_test_with_builder** et placez-vous sur le premier commit de cette branche 😉

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.

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 rapide petit coup d'oeil à cet article : http://unil.im/builderxebia
(https://blog.engineering.publicissapient.fr/2016/12/28/design-pattern-builder-et-builder-sont-dans-un-bateau/)

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 *fluent*** pour améliorer la lisibilité des tests. Pour commencer, focalisons-nous sur le premier test qui est actuellement écrit de la manière suivante :

```
@Test
void alarm_On_when_value_too_low() {
      Sensor sensor = probeValue(1.0);

      Alarm alarm = new Alarm(sensor, new SafetyRange(17, 21));
      alarm.check();

      assertTrue(alarm.isAlarmOn());
}
```

En s'inspirant de l'article du blog 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
void alarm_On_when_value_too_low() {
      Sensor sensor = probeValue(1.0);

      Alarm alarm = new AlarmBuilder()
                  .withSensor(sensor)
                  .withSafetyRange(new SafetyRange(17, 21))
                  .build();
      alarm.check();

      assertTrue(alarm.isAlarmOn());
                  }
```

Remarque : A la place de **`withSafetyRange(new SafetyRange(17, 21))`** nous aurions très bien pu écrire **`withSafetyRange(17, 21)`**.
Rien ne vous empêche, s'il vous reste du temps à la fin du TP, d'implémenter dans la classe **`AlarmBuilder`** cette surcharge à la méthode `withSafetyRange` et de la tester 😉

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`**, **`withSensor`** et **`build`**,
étapes qui vont être réalisées pas à pas en répondant aux questions suivantes 😉

2.2 Création de la classe `AlarmBuilder` :

Où créer la classe **`AlarmBuilder`** ?

Dans `src/test/java` (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, withSensor` et `build` :

A l'aide de l'IDE, créez les méthodes **`withSafetyRange, withSensor`** et **`build`** dans la classe **`AlarmBuilder.`**
Puis implémentez ces méthodes afin de faire passer le test AU VERT !!!

Remarque : pour pouvoir chaîner les appels des méthodes **`withSafetyRange`** et **`withSensor`**, Il faut bien évidemment que ces deux méthodes renvoient un objet de type **`AlarmBuilder`** 😉

Avant de continuer, vérifiez que le code continue de compiler et que les tests continuent à passer AU VERT !

2.4 Améliorer (encore plus) la lisibilité en proposant une méthode statique `anAlarm` au lieu d'un appel direct au constructeur

Pour supprimer l'appel direct au constructeur, nous allons le *cacher*
dans une méthode statique plus explicite que nous appellerons **`anAlarm`** 😉

⇨ Commencez par modifier **l'étape Arrange** du test de la manière suivante c-a-d en appelant la méthode statique **`anAlarm`** de la classe **`AlarmBuilder.`**

```
@Test
void alarm_On_when_value_too_low() {
      Sensor sensor = probeValue(1.0);


      Alarm alarm = AlarmBuilder.anAlarm()
                  .withSensor(sensor)
                  .withSafetyRange(new SafetyRange(17, 21))
                  .build();

      alarm.check();

      assertTrue(alarm.isAlarmOn());
}
```

⇨ Implémentez la méthode **`static anAlarm`** dans la classe `AlarmBuilder` pour faire passer ce test AU VERT !

⇨ Il est encore possible d'améliorer un peu plus la lisibilité en faisant en sorte que l'IDE ajoute automatiquement l'**`import static`** sur `AlarmBuilder.`*`anAlarm`*

Sous Eclipse, sans la classe **`AlarmTest`**, il suffit de positionner votre curseur sur ***`anAlarm`*`()`** puis à l'aide d'un clic droit : **`Source -> Add Import.`** Le test devient alors :

```
@Test
void alarm_On_when_value_too_low() {
      Sensor sensor = probeValue(1.0);

      Alarm alarm = anAlarm()
                  .withSensor(sensor)
                  .withSafetyRange(new SafetyRange(17, 21))
                  .build();
      alarm.check();

      assertTrue(alarm.isAlarmOn());
}
```

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

Avant de continuer, vérifiez que tout le code continue de compiler et que les tous tests continuent à passer AU VERT !

2.5 Mise en place, *petit pas par petit pas*, du *builder fluent* dans toutes les méthodes de tests

Maintenant que la création d'un objet de type **`Alarm`** est *refactoré* dans la première méthode de test à l'aide **d'un *builder fluent***, il ne vous reste plus qu'à vous inspirer de cette implémentation pour modifier **test par test** l'étape **Arrange** et améliorer ainsi la qualité de code de cette étape dans chaque test.

Pour chaque méthode de test, procédez ***petit pas par petit pas***, c.-à-d. :

⇨ Modifiez l'étape Arrange manière à construire l'alarme avec la méthode `build` de l'`AlarmBuilder`.
⇨ Vérifiez que le code compile et que les tests continuent à passer AU VERT !!!
⇨ Passez à la méthode de test suivante.

***Avant de continuer,* faîtes en sorte que toutes les méthodes de tests fassent appel à `AlarmBuilder`, tout en ayant toujours tous les tests passent AU VERT !**

2.6 … sans oublier le commit pour marquer la fin de ce refactoring !

Avant de continuer, comme votre code compile et que tous vos tests passent AU VERT, il est temps de procéder un petit **commit** avec un message du genre :

`add AlarmBuilder in AlarmTest`

Le pattern builder fluent permet
d'améliorer la qualité du code de test
lors de la création des objets de test
en insistant sur deux points :

⇨ **faciliter la *lisibilité* du test** lors de la création des objets de test (étape **A**rrange du pattern AAA).

⇨ **assurer la *sécurité* du code** en garantissant qu'un objet de test est bien construit avec les *bonnes* valeurs.
Le côté ***secure design*** est assurée par les méthodes **with** qui permettent pour chaque attribut, de savoir **sans ambiguïté, de manière très explicite** où, quand et comment passer une valeur à cet attribut pour créer l'objet en toute confiance selon les besoins.

A visionner autour de la notion de builder
pour (re)découvrir et approfondir cette bonne pratique très utile et utilisée pour la création d'objets complexes

→ Commencez par la vidéo de Simon Dieny : **Le Design Pattern Contre-Intuitif le plus utilisé par les Développeurs Professionnels…**
https://www.youtube.com/watch?v=zto9F3ANVXA

→ Puis, pour aller plus loin, continuez par la série des 4 vidéos suivantes extraites de la rubrique **Principes SOLID et Design Pattern** des vidéos de José Paumard : **https://unil.im/but2r204**
(URL complète : https://www.youtube.com/playlist?list=PLzzeuFUy_CngSfFq9-TJ0r8NC7Y3hSNpe)

❑ **37 - Pattern Builder pour la création d'objets complexes (https://www.youtube.com/watch?v=7Oerphqa6jE)**

❑ **38 - Création d'objets complexes avec une factory (https://www.youtube.com/watch?v=IjaRKCfE1EI)**

❑ **39 - Création d'un objet complexe avec un Builder (https://www.youtube.com/watch?v=WVfyd1BRhwc)**

❑ **40 - Implémentation du Builder (https://www.youtube.com/watch?v=vzHMeMsjOcw)**
Attention, l'implémentation du Builder proposée dans cette vidéo se fait directement dans la classe métier, elle se fait donc sous forme d'une classe statique d'où la notation : `new Contrat.Builder()`
Dans le module R2.04, quand nous construirons un ***builder fluent***, nous l'implémenterons de manière *séparé*, dans sa propre classe nommée avec le nom de la classe dont il doit construire un objet suivi de Builder. Si on appliquait ce standard de codage à l'exemple de la vidéo, il faudrait implémenter les méthodes `withXXX` et `build()`dans une classe `ContratBuilder` et la première instruction pour construire un objet deviendrait : `new ContratBuilder()`

Pour les plus rapides :

Exercice n°3 : Du code de test propre ! (Clean test)

Les tests permettent de se substituer en partie à une spécification. Ils permettent également de modifier et de faire évoluer le code de production en toute confiance. ***Le code de test devient alors aussi important que le code de production****. En devenant un citoyen de première classe,* ***un test agile se doit donc de respecter un niveau de qualité équivalent à celui du code de production : l'expression test propre (clean test) est d'ailleurs employée en écho à l'expression code propre (clean code).***

Extrait : « Les tests dans le développement logiciel, du cycle en V aux méthodes agiles »
Isabelle BLASQUEZ · Hervé LEBLANC · Christian PERCEBOIS
Tech. Sci. Informatiques 36(1-2): 7-50 (2017)

Cet exercice va se focaliser sur la classe **`AlarmTest`** et vous guider pas à pas dans le reactoring d'un code de test encore plus propre 😉

⇨ Commencez par relire le code de la classe **`AlarmTest`**.
Ce code respecte à première vue certains critères de lisibilité : code concis, correctement indenté, moins de dix lignes par méthode. Cependant, l'utilisation de mauvaises pratiques de conception (*code smells*) subsitent.
Quelles mauvaises odeurs identifiez-vous *encore* dans le code ?
Quid des nombres magiques ? Quid du nommage des variables ?
Quid de la complexité du code ? Quid de la duplication ?

1. La classique mauvaise odeur du nombre magique (*magic number smell code*) 😉

Vous avez surement remarqué que jusqu'à présent, à *presque* toutes les revues, nous avons détecter un **nombre magique**, qui est sans doute un des *code smell* les plus répandu.

Le code d'**`AlarmTest`** ne fait pas exception à cette règle : le *code smell* du nombre magique est bien présent dans ce code 😉.
⇨ Serez-vous le(s) trouver ?

Peut-être avez-vous détecté 17 et 21 comme nombres magiques …
Pour ma part, j'aurais plutôt envie de dire que l'objet `new SafetyRange(17, 21)` est un nombre magique qui représente l'intervalle de sécurité par défaut d'un capteur de pression.

⇨ Rappelez à quelle **opération de refactoring** vous devez procéder pour pallier à la mauvaise odeur du nombre magique
(**code smell magic number**) :

⇨ Utilisez cette option de refactoring automatique de votre IDE préféré pour faire apparaître ***SAFETY_RANGE_PRESSURE*** à la place du(es) nombre(s) magique(s) 😉

Avant de continuer, vérifiez que le code continue de compiler et que tous les tests continuent à passer AU VERT !

2. Toujours se poser, à la fin, la fameuse question du DRY (**D**on't **R**epeat **Y**ourself) 😉

Avez-vous remarqué quelque chose de particulier quand vous avez refactoré pas à pas toutes les étapes Arrange des tests en vous inspirant de la première étape ?
⇨ Avez-vous senti comme une petite odeur de duplication de code à ce moment ?

En effet, comme c'est toujours le même intervalle qui est utilisé dans l'étape Arrange (désormais appelé ***SAFETY_RANGE_PRESSURE***), une partie du code de l'étape Arrange est donc identique dans chaque méthode de test.
Il y a donc de la duplication de code car **exactement le même code apparaît 4 fois dans la classe `AlarmTest`** (en principe, la duplication au sens DRY s'entend dès 3 répétitions).

Remarque : Veillez à ne pas vous poser la question du **DRY** trop tôt dans votre refactoring pour vous éviter de factoriser le code trop tôt. En effet, parfois, il vaut mieux laisser apparaître la duplication pour mieux refactorer ensuite. Si vous voulez en savoir plus sur ce sujet, vous pourrez visionner, tranquillement chez vous, cette vidéo de refactoring sur le kata gilded rose qui illustre bien ce fait : https://www.youtube.com/watch?v=g11gydDAMSo

⇨ Rappelez à quelle **opération de refactoring** vous devez le plus souvent procéder pour pallier à la mauvaise odeur de la duplication (**code smell DRY**) :

⇨ Utilisez cette option de refactoring automatique de votre IDE préféré pour faire apparaître un appel à la méthode `alarmWithSafetyRangePressure` dans chaque méthode de test à place de la duplication de code actuelle 😉

Avant de continuer, vérifiez que le code continue de compiler et que tous les tests continuent à passer AU VERT !

Si tout s'est bien passé, à l'image de la première méthode de tests, toutes vos méthodes de tests d'**`AlarmTest`** doivent avoir actuellement une structure similaire à la structure suivante :

```
@Test
void alarm_On_when_value_too_low() {
      Sensor sensor = probeValue(1.0);
      Alarm alarm = alarmWithSafetyRangePressure(sensor);

      assertTrue(alarm.isAlarmOn());
      }
```

Rappelons que le refactoring est une action subjective, et que l'activité de refactoring prend fin lorsque vous le jugez opportun, c'est-à-dire que vous trouvez votre code de meilleure qualité, plus facile à lire, à maintenir et à étendre en fonction de vos besoins !
… Mais comme nous sommes actuellement dans un kata, pour un meilleur entraînement, nous décidons de pousser encore un peu plus loin nos bonnes pratiques de refactoring 😉

3. Pinaillons avec deux dernières petites améliorations (règle des boy scout) …

a. En effet pourquoi ne pas améliorer encore un peu plus la lisibilité du test en ***inlinant*** **`probeValue`** à la place de `sensor` c-a-d en faisant apparaître **`probeValue`** directement au moment de l'instanciation de l'alarme.

⇨ Dans la première méthode de test, placez-vous sur le `sensor` de l'instruction :

```
Alarm alarm = alarmWithSafetyRangePressure(sensor);
```

⇨ Sélectionnez alors l'opération de refactoring **Inline…** de votre IDE préféré pour transformer les deux instructions suivantes :

```
Sensor sensor = probeValue(150.0);
Alarm alarm = alarmWithSafetyRangePressure(sensor);
```

en une seule instruction (sur la même ligne)

```
Alarm alarm = alarmWithSafetyRangePressure(probeValue(1.0));
```

⇨ Pas à pas, **pour chaque méthode de test**, faites en sorte que **`probeValue`** **soit *inliné***. Et que tous **les tests continuent à passer AU VERT !**

Remarque : L'intérêt de procéder à un **refactoring de type *`inline`*** permet :

- de **supprimer une variable intermédiaire bien souvent inutile** et inutilisée dans la suite du code (comme c'était le cas de `sensor`)
- de **permettre au code de se lire de manière très fluide** presque naturelle : **`alarmWithSafetyRangePressure(probeValue(1.0));`** cette instruction *algorithmique* se lit comme presque comme une phrase d'un texte 😉

b. **Dernier renommage** : Pour encore plus de fluidité dans la lecture de test, on pourrait maintenant également avoir envie de renommer à l'aide de l'IDE la méthode **`isAlarmOn`** de la classe en méthode **`isOn`** afin que la première méthode de test devienne :

```
@Test
void alarm_On_when_value_too_high() {
      Alarm alarm = alarmWithSafetyRangePressure(probeValue(1.0));
      assertTrue(alarm.isOn());
}
```

***Avant de continuer,* faîtes en sorte tous les tests AU VERT !**

4. Il ne reste plus qu'à versionner le tout !

⇨ Procédez à un petit **commit** avec un message du genre : **`refacto AlarmTest : clean code`**
⇨ **Mergez dans** **`master`** la branche actuelle `refacto_clean_test_with_builder`

Exercice n°4 : Qui a la responsabilité de faire sonder les capteurs ?

(pour les plus rapides)

Si vous examinez la visibilité des méthodes de la classe **`AlarmTest`**, vous constaterez que votre classe dispose actuellement de trois méthodes privées :

```
private Alarm alarmWithSafetyRangePressure(Sensor sensor);
private Sensor probeValue(double value);
private Sensor probeValue(double value1, double value2);
```

La question légitime à se poser dans ce cas-là est :
Est-ce que toutes les méthodes privées sont de la **responsabilités** de **`AlarmTest`** ?
(**SOLID autour de SRP)**

En effet, 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 **`probeValue`**) qui permettent de *fabriquer* un capteur (mocké) qui sonde une (ou plusieurs) valeur(s)?

Evidemment non, il serait bien plus préférable de disposer d'une **fabrique de capteurs** …
⇨ A partir du dernier commit de **`master`**, créez un nouvelle branche **`refacto_factory_sensor`** dans laquelle vous pourrez vous essayez à une implémentation qui permettrait de mieux répartir cette responsabilité.
⇨ Si votre solution vous satisfait et est opérationnelle, vous pourrez alors la merger dans **`master`** 😉