M3105 – TD : Kata Parrot

Prenez connaissance du code suivant …

```
public enum ParrotTypeEnum {
	EUROPEAN, AFRICAN, NORWEGIAN_BLUE;
}

public class Parrot {

	private ParrotTypeEnum type;
	private int numberOfCoconuts = 0;
	private double voltage;
	private boolean isNailed;

	public Parrot(ParrotTypeEnum _type, int numberOfCoconuts, double voltage,
	                                                          boolean isNailed) {
		this.type = _type;
		this.numberOfCoconuts = numberOfCoconuts;
		this.voltage = voltage;
		this.isNailed = isNailed;
	}

	public double getSpeed() {
		switch (type) {
		case EUROPEAN:
			return getBaseSpeed();
		case AFRICAN:
			return Math.max(0, getBaseSpeed() - getLoadFactor() *numberOfCoconuts);
		case NORWEGIAN_BLUE:
			return (isNailed) ? 0 : getBaseSpeed(voltage);
		}
		throw new RuntimeException("Should be unreachable");
	}

	private double getBaseSpeed(double voltage) {
		return Math.min(24.0, voltage * getBaseSpeed());
	}

	private double getLoadFactor() {
		return 9.0;
	}

	private double getBaseSpeed() {
		return 12.0;
	}

}
```


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 classes de ce code *legacy*.

1.b Que fait ce code ?

1.c Ce code pourrait être du code existant que vous venez d'hériter d'un collègue développeur. En réalité, c'est celui d'un kata créé par Emilie Bache (https://github.com/emilybache/Parrot-Refactoring-Kata) inspiré d'un exemple du livre "Refactoring, Improving the Design of Existing Code" de Martin Fowler. Ce code fait donc apparaître des signes de mauvais design (mauvaises pratiques de conception, code smells). Quels sont-ils ?

1.d Quelle solution proposeriez-vous pour rendre ce code plus SOLID ?
Exposez votre solution au travers d'un nouveau diagramme de classes.

2. Vers du code SOLID…

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

2.a Dans votre IDE préféré, créer un projet maven **`kataparrot`** dans lequel vous ajouterez dans **src/main/java** les classes **`Parrot`** et **`ParrotTypeEnum`** et dans **src/test/java** la classe **`ParrotTest`** à récupérer dans le répertoire **`ressources/kataparrot`** du dépôt https://github.com/iblasquez/enseignement-iut-m3105-conception-avancee

Faites en sorte que ce code soit correctement formaté et qu'il compile !

Faites en sorte que les tests passent AU VERT !!!

Comme vous le savez avant de vous lancer dans un quelconque refactoring, **il est Indispensable de disposer d'un harnais de tests** pour garantir le comportement actuel du système contre d'éventuelles régressions.
Cette fois-ci le harnais de tests vous est fourni 😉.
Vérifiez que ce harnais de tests couvre bien le code.

Mettez votre projet sous git et procédez à un premier commit du genre :
« commit initial»

Le refactoring que vous allez mettre en place pour résoudre les problèmes de design identifiés précédemment est connu sous le nom de ***Replace Conditional with Polymorphism.***
Martin Fowler préconise son utilisation lorsque : *You have a conditional that chooses different behavior depending on the type of an object***.**
La solution qu'il propose consiste à : *Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract.*
(Extrait : https://refactoring.com/catalog/replaceConditionalWithPolymorphism.html*)*

Nous allons vous guider petits pas par petits dans la mise en place de ce refactoring.

Avant de commencer, jetez un petit coup d'œil au fichier de tests. Que constatez-vous ?
Ce fichier propose un (ou plusieurs) tests pour chaque type de perroquet.
Nous allons donc procéder au refactoring type de perroquet par type (petit pas par petit pas) en commençant par focaliser notre refactoring autour du comportement propre au perroquet européen…

2.b Refactoring autour du perroquet européen

2.b.1 Un polymorphisme ne peut exister que sur une hiérarchie de classes.
Créez donc une nouvelle classe **`EuropeanParrot`** qui hérite de **`Parrot`**.
⇨ Relancez les tests, ils doivent continuer à passer AU VERT !!!!
Le comportement initial n'a pas été modifié pour le moment 😉

2.b.2 Pour remplacer la structure conditionnelle par du polymorphisme, Martin Fowler indique qu'il faut *Move each leg of the conditional to an overriding method in a subclass.* Appliqué au cas du perroquet européen, cela revient à faire en sorte de *Move leg of the conditional* ***`EUROPEAN`*** *to an overriding method* **`getSpeed`** *in a subclass* **`EuropeanParrot`** c-a-d redéfinir la méthode **`getSpeed`** avec les instructions présentes actuellement dans la clause ***`EUROPEAN`***
⇨ A vous de jouer !
⇨ Relancez les tests, ils doivent continuer à passer AU VERT !!!!
Le comportement initial n'a pas été modifié pour le moment 😉

2.b.3 Pour vous assurer que c'est bien l'implémentation de la méthode **`getSpeed`** de La classe fille **`EuropeanParrot`** qui est exécutée (c-a-d que le polymorphisme fonctionne correctement), il faut :

➔ D'une part, s'assurer que l'implémentation du comportement dans la classe mère n'est plus exécuté. Pour cela une solution consiste à remplacer cette implémentation par un déclenchement d'exception. Ainsi, si ce bout de code venait à être malencontreusement appelé, il déclencherait une exception et les tests correspondants ne passeraient plus AU VERT 😉

```
public double getSpeed() {
   switch (type) {
         case EUROPEAN:
               throw new RuntimeException("Should be unreachable");
   //…
}
```


➔ D'autre part, pour que le polymorphisme soit effectif, l'instruction **`parrot.getSpeed()`** doit s'appliquer sur un objet **`parrot`** de la classe **`EuropeanParrot.`**
Il est donc nécessaire de modifier l'étape ***A***rrange du test sur le perroquet européen afin que

```
@Test
public void getSpeedOfEuropeanParrot() {
      Parrot parrot = new Parrot(ParrotTypeEnum.EUROPEAN, 0, 0, false);
      assertEquals(parrot.getSpeed(), 12.0, 0.0);
}
```

devienne :

```
@Test
public void getSpeedOfEuropeanParrot() {
      Parrot parrot = new EuropeanParrot();
      assertEquals(parrot.getSpeed(), 12.0, 0.0);
}
```

Procédez à cette modification.
Que constatez-vous ? Le code ne compile pas !
En effet, pour compiler correctement, il manque dans la classe **`EuropeanParrot`** un constructeur par défaut qui doit pour l'instant juste faire appel au constructeur de la classe mère avec les bons paramètres !

⇨ A vous de jouer !
⇨ Relancez les tests, ils doivent passer AU VERT !!!!

2.b.4 Commitez en mentionnant de manière explicite dans le message de commit le petit pas de refactor que vous venez d'effectuer, par exemple :
« Refactor polymorphisme pour obtenir la vitesse d'un perroquet européen (EuropeanParrot) »

2.c Refactoring autour du perroquet africain

2.c.1 Nous souhaitons maintenant focaliser notre refactoring autour du comportement du perroquet africain. En suivant les mêmes petits pas que précédemment, faites en sorte de *Move leg of the conditional* ***`AFRICAN`*** *to an overriding method* **`getSpeed`** *in a subclass* **`AfricanParrot`**

Ne pas oubliez :
- de remplacer l'ancienne implémentation par une exception pour vous assurer que c'est bien la nouvelle implémentation qui est exécutée !
- de faire en sorte que l'objet **`parrot`** des méthodes de tests sur le perroquet africain soit instancier à partir d'un constructeur de la classe `AfricanParrot`.
  Combien de paramètres doit avoir ce constructeur ?
  Pour répondre à cette question, posez-vous la question suivante :
  Quel(s) paramètre(s) affecte(nt) la vitesse du perroquet africain ?
  Implémentez ce constructeur (de la classe `AfricanParrot`) en faisant appel au constructeur de la classe mère avec les bons paramètres !

⇨ Relancez les tests, ils doivent passer AU VERT !!!!
⇨ **Commitez en mentionnant de manière explicite dans le message de commit le petit pas de refactor que vous venez d'effectuer.**

2.c.2 Les tests étant actuellement au VERT, on peut s'interroger sur une éventuelle **amélioration de la qualité du code autour du perroquet africain** …

⇨ **En terme de responsabilité**, on peut se poser la question de savoir quelle classe est responsable du facteur de charge (actuellement implémenté via `getLoadFactor`) **?**
Cette méthode est-elle actuellement dans la bonne classe ? Si non, faites le nécessaire (`Refactor-> Move…`)
⇨ Relancez les tests, ils doivent passer AU VERT !!!!

⇨ Que contient la méthode `getLoadFactor` ?
Qu'en pensez-vous ? N'y aurait-il pas là comme un **code smell ?**
Comment pourriez-vous nettoyer ce code ?
⇨ Relancez les tests, ils doivent passer AU VERT !!!!

⇨ **Améliorer la lisibilité** de votre code en inlinant le contenu de la méthode `getLoadFactor` dans `getSpeed` (`Refactor ->Inline…`).
⇨ Relancez les tests, ils doivent passer AU VERT !!!!

⇨ **A propos de l'encapsulation : attribut de la classe `AfricanParrot`**
Quel paramètre affecte la vitesse d'un perroquet africain ?
Ce paramètre affecte-t-il la vitesse des autres types de perroquet ?
Si non…ne serait-il pas judicieux de déplacer cet attribut de la classe mère vers la classe fille où il est réellement utilisé et donc d'en faire un attribut de la classe `AfricanParrot`. Pour cela :
Dans la classe `Parrot` :
- déplacer la déclaration de l'attribut `numberOfCoconuts` vers `AfricanParrot` (`Refactor -> Move…`)
- modifier à l'aide du refactor de l'IDE, la signature du constructeur de manière à supprimer « de manière automatique » le paramètre `numberOfCoconuts` de la liste des paramètres (`Refactor -> Change Method Signature`)

➔ Dans la classe `AfricanParrot` :
- vérifier que l'attribut `numberOfCoconuts` est bien déclaré
- faites en sorte que le constructeur initialise correctement cet attribut

⇨ Relancez les tests, ils doivent passer AU VERT !!!!

⇨ **Commitez en mentionnant de manière explicite dans le message de commit le petit pas de refactor que vous venez d'effectuer par exemple « AfricanParrot : encapsulation du nombre de noix de coco et responsabilité du facteur de charge »**

2.d Refactoring autour du comportement du perroquet norvégien bleu

Nous souhaitons maintenant focaliser notre refactoring autour du comportement du perroquet africain. En suivant les mêmes petits pas que précédemment, faites en sorte de *Move leg of the conditional* `NORWEGIAN_BLUE` *to an overriding method* `getSpeed` *in a subclass* `NorwegianBlueParrot`

Ne pas oubliez :
- de remplacer l'ancienne implémentation par une exception pour vous assurer que c'est bien la nouvelle implémentation qui est exécutée !
- de faire en sorte que l'objet `parrot` des méthodes de tests sur le perroquet africain soit instancié à partir d'un constructeur de la classe `NorwegianBlueParrot`.
- de vous questionner sur la « bonne » encapsulation, les « bonnes » responsabilités, la lisibilité pour soigner la qualité de votre code 😉

⇨ Relancez les tests, ils doivent passer AU VERT !!!!
⇨ **Commitez en mentionnant de manière explicite dans le message de commit le petit pas de refactor que vous venez d'effectuer.**

2.e Améliorer la qualité du code dans la classe `Parrot`

2.e.1 Supprimer le code mort (code obsolète)
Lancez une couverture de tests.
Jetez un coup d'œil sur vos différentes classes, voyez-vous du code non couvert ? (instructions surlignées en rouge avec Coverage).

Auparavant 100% du code était couvert, pourquoi ce code n'est-il plus couvert ? …
Peut-être parce qu'il s'agit désormais de code mort qui ne sera plus utilisé et on peut aller jusqu'à dire qu'il n'est même plus atteignable en raison de la mise en place du polymorphisme autour de la méthode `getSpeed`.

⇨ Supprimez ce code mort et relancez les tests pour vous assurer que vous ne modifiez pas le comportement : ils doivent passer AU VERT !!!!

2.e.2 Rendre abstrait ce qui doit l'être…
Une méthode peut-elle devenir abstraite ?
Quid de la classe `Parrot` ?
⇨ Relancez les tests, ils doivent passer AU VERT !!!!

2.e.3 Code smell…
Voyez-vous un code smell dans le code restant ?
Si oui, faites en sorte de le nettoyer 😉
⇨ Relancez les tests, ils doivent passer AU VERT !!!!

2.e.4 Améliorer la lisibilité…
Pouvez-vous encore améliorer la lisibilité de code ?
⇨ Relancez les tests, ils doivent passer AU VERT !!!!

⇨ **Commitez en mentionnant de manière explicite dans le message de commit le petit pas de refactor que vous venez d'effectuer.**

2.f Quid de l'énumération `ParrotTypeEnum` ?

A quoi sert désormais l'énumération **`ParrotTypeEnum`** dans le code ?
Si vous considérez qu'elle est devenue obsolète, vous pouvez essayer de la supprimer.
Relancez les tests pour voir si cette suppression était bien justifiée et n'impacte pas le comportement de votre projet …

2.g Générez le diagramme de classes à partir de votre code.
Comparez ce diagramme avec celui de la question 1.
Le code est-il plus SOLID ?
Jetez un petit coup d'œil sur :
https://refactoring.com/catalog/replaceConditionalWithPolymorphism.html

2.h Encore un petit refactoring ?
Consultez le code dont vous disposez actuellement.
Le refactoring est subjectif. Avez-vous envie de refactorer autre chose dans ce projet ?

Pousser votre code vers votre dépôt distant
après vous être assuré que vos tests sont AU VERT !!!

Et un peu de lecture pour terminer : Seven Ways to Refactor Java switch Statements
https://www.developer.com/java/data/seven-ways-to-refactor-java-switch-statements.html