R3.04 – Une SOLIDe revue de code

*Ce TP sera réalisé en **mode déconnecté** :*
Laissez vos ordinateurs éteints pour le moment et prenez une feuille de papier 😉
Les deux premiers exercices proposés dans cet énoncé
sont des exercices qui sont inspirés des contrôles passés 😉

Exercice 1 : Principes S.O.L.I.D : Définition

Pour chaque lettre du mot **S.O.L.I.D,** donnez l'acronyme du principe (sigle).
Donnez la signification de chaque initiale de cet acronyme.
Enoncez le principe correspondant et dites en une phrase en quoi/comment ce principe vous permet de prendre du recul sur votre code. Proposez des pistes de refactoring pour chaque principe.

S	

O

L

I

D	

Exercice 2 : S *or* O *or* L *or* I *or* D ?

❑ Bout de code n°1 :

```
public class Greeter {
	String formality;

	public String greet() {
		if (this.formality == "formal") {
			return "Good evening, sir.";
		} else if (this.formality == "casual") {
			return "Sup bro?";
		} else if (this.formality == "intimate") {
			return "Hello Darling!";
		} else {
			return "Hello.";
		}
	}

	public void setFormality(String formality) {
		this.formality = formality;
	}
}
```

Identifiez le principe SOLID non respecté :

Enoncé de ce principe :

Pourquoi ce principe n'est-il pas respecté ? :

Vers un code plus SOLID… (suggérez des pistes possibles de refactoring) :

❑ **Bout de code n°2 :**

```
public class Radio {
	private DuracellBattery battery;

	public Radio() {
		this.battery = new DuracellBattery();
	}

	public void play() {
		this.battery.start();
	}
}


public class DuracellBattery {

	public void start() {
		// avec du code qui permet de démarrer la batterie ;-)
	}

}
```

Identifiez le principe SOLID non respecté :

Enoncé de ce principe :

Pourquoi ce principe n'est-il pas respecté ? :

Vers un code plus SOLID… (suggérez des pistes possibles de refactoring) :

❑ Bout de code n°3 :

```
import java.util.ArrayList;

public class Board {
	ArrayList<String> spots;

	public Board() {
		this.spots = new ArrayList<String>();
		for (int i = 0; i < 9; i++) {
			this.spots.add(String.valueOf(i));
		}
	}

	public ArrayList<String> firstRow() {
		ArrayList<String> firstRow = new ArrayList<String>();
		firstRow.add(this.spots.get(0));
		firstRow.add(this.spots.get(1));
		firstRow.add(this.spots.get(2));
		return firstRow;
	}

	public ArrayList<String> secondRow() {
		ArrayList<String> secondRow = new ArrayList<String>();
		secondRow.add(this.spots.get(3));
		secondRow.add(this.spots.get(4));
		secondRow.add(this.spots.get(5));
		return secondRow;
	}

	public ArrayList<String> thirdRow() {
		ArrayList<String> thirdRow = new ArrayList<String>();
		thirdRow.add(this.spots.get(6));
		thirdRow.add(this.spots.get(7));
		thirdRow.add(this.spots.get(8));
		return thirdRow;
	}

	public void display() {
		String formattedFirstRow =
	  this.spots.get(0) + " | " + this.spots.get(1) + " | " + this.spots.get(2) + "\n"
	+ this.spots.get(3) + " | " + this.spots.get(4) + " | " + this.spots.get(5) + "\n"
	+ this.spots.get(6) + " | " + this.spots.get(7) + " | " + this.spots.get(8);
	System.out.print(formattedFirstRow);
	}
}
```

Identifiez le principe SOLID non respecté :
Enoncé de ce principe :

Pourquoi ce principe n'est-il pas respecté ? :

Vers un code plus SOLID… (suggérez des pistes possibles de refactoring) :

❑ Bout de code n°4 :

```
public interface Animal {
	void voler();
	void courir();
	void aboyer();
}

public class Oiseau implements Animal {

	public void aboyer() {
		throw new RuntimeException("opération non définie pour un oiseau");
	}

	public void courir() {
		// du code pour faire courir l'oiseau
	}

	public void voler() {
		// du code pour faire voler l'oiseau
	}
}


public class Chien implements Animal {

	public void voler() {
		throw new RuntimeException("opération non définie pour un chien");
	}

	public void aboyer() {
		// du code pour faire aboyer le chien
	}

	public void courir() {
		// du code pour faire courir le chien
	}
}


public class Chat implements Animal {
	public void voler() {
		throw new RuntimeException("opération non définie pour un chat");
	}

	public void aboyer() {
		throw new RuntimeException("opération non définie pour un chat");
	}

	public void courir() {
		// du code pour faire courir le chat
	}
}
```

Identifiez le principe SOLID non respecté :
Enoncé de ce principe :

Pourquoi ce principe n'est-il pas respecté ?

Vers un code plus SOLID… (suggérez des pistes possibles de refactoring) :

❑ Bout de code n°5 :

```
public interface Cheese {}
public class Cheddar implements Cheese {}
public class Maroilles implements Cheese {}
```

Un sandwich au fromage contient un morceau de fromage qui peut être changé:

```
public class CheeseSandwich {
	protected Cheese filling;

	public void setFilling(Cheese c) {
		this.filling = c;
	}

	public Cheese getFilling() {
		return this.filling;
	}
}
```

Un sandwich au cheddar ne peut contenir que du cheddar…

```
public class CheddarSandwich extends CheeseSandwich {
	public void setFilling(Cheddar c) {
		this.filling = c;
	}
}
```

Le code suivant compile. Mais que se passe-t-il à l'exécution ?

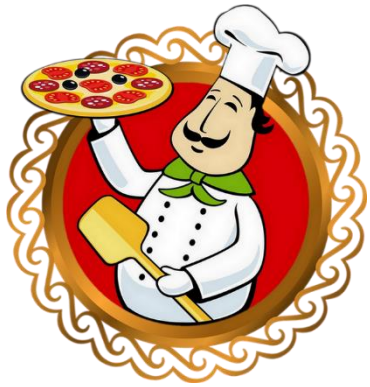
```
public class ClientClass {
	public static void main(String[] args) {
		CheddarSandwich sandwich = new CheddarSandwich();
		sandwich.setFilling(new Cheddar());
		Cheddar cheddar = (Cheddar) sandwich.getFilling();

		sandwich.setFilling(new Maroilles());
		cheddar = (Cheddar) sandwich.getFilling();
	}
}
```

Principe non respecté :
Enoncé de ce principe :

Pourquoi ce principe n'est-il pas respecté ? :

Pour ce bout de code, on ne vous demande pas de piste de refactoring 😉

Exercice 3 : La pizza : un SOLIDe entraînement !



```
public class Pizza {

	public static final int COOKED = 0;
	public static final int BAKED = 1;
	public static final int DELIVERED = 2;

	private String name;

	int state = COOKED;

	public boolean bake() {

		if (state == COOKED) {
			state = BAKED;
		} else if (state == BAKED) {
			return false; // Can't bake a pizza already baked !
		} else if (state == DELIVERED) {
			return false; // Can't bake a pizza already delivered !
		}

		return true;
	}

	public boolean deliver() {

		if (state == COOKED) {
			return false; // Can't deliver a pizza not baked yet;
		} else if (state == BAKED) {
			state = DELIVERED;
		} else if (state == DELIVERED) {
			return false; // Can't deliver a pizza already delivered;
		}

		return true;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public int getState() {
		return state;
	}

	public void setState(int state) {
		this.state = state;
	}
}
```

1. **Que fait ce code ?**

2. **Quelles sont les mauvaises odeurs de ce code ?**
   Pourquoi ce code n'est-il pas SOLID ? Si vous identifiez un principe SOLID non respecté : nommer ce principe, donner sa définition et expliquer pourquoi ce principe n'est pas respecté

3. **Vers un code plus SOLID…**
   Suggérez une piste possible de refactoring en proposant un diagramme de classes qui permettrait de rendre ce code SOLID

4. **Let's code !!!!**

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

La première chose à faire est de resynchroniser votre dépôt local avec le remote
https://github.com/iblasquez/enseignement-but2-developpement

Dans votre IDE préféré, **dans un répertoire local qui vous est propre ,** faites-en sorte d'importer le projet maven **`pizza`** du répertoire `ressources` du dépôt

Faites en sorte que ce **code soit correctement formaté et qu'il compile** !
Faites en sorte que les **tests passent AU VERT** !!! Vérifiez que le harnais de tests fourni couvre bien le code.

Versionnez ce projet : et **procédez à un premier commit** dont le message pourrait être **initial commit**

En vous inspirant de ce qui a été fait précédemment avec le kata Parrot, refactorez pas à pas ce code pour obtenir un code SOLID et propre. N'oubliez pas de commiter régulièrement !

Il est à noter que cet exercice ne sera pas corrigé en TP, mais qu'il vous est fortement conseiller de faire des revues de code entre vous,
de partager et comparer vos différentes implémentations 😉