M3105 – Une SOLIDe revue de code

*Ce TD sera réalisé en **mode déconnecté** :*
Laissez vos ordinateurs éteints pour le moment et prenez une feuille de papier 😉

❑ 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 😉