DRY (Do Not Repeat Yourself)

Bad Example (Violating DRY)

```
class Bag {
        private int[] elements;
        public void process() {
                for (int i = 0; i < elements.length; i++) {
                        System.out.println(elements[i]);
                }
        }
        public void print() {
                for (int i = 0; i < elements.length; i++) {
                        System.out.println(elements[i]);
                }
        }
}
```

The above code does not following the DRY principle and the updated correct code which follows DRY principle is as follows:

Good Example (Following DRY)

```
class Bag {
        private int[] elements;
        public void process() {
                print(); // Call the sub functions to avoid repetition. Avoid repeating the logic, call
print() instead.
        }
        public void print() {
                for (int i = 0; i < elements.length; i++) {
                        System.out.println(elements[i]);
        }
    }
}
```

KISS (Keep It Simple)

Bad Example (Violating KISS)

```
class Fibonnaci {
    public int getFib(int n) { // Not a simple logic.
        if (n <= 1) return n;
        return getFib(n - 1) + getFib(n - 2);
    }
}
```

How can the above code is not having a simple logic?

The recursion is harder to understand and inefficient because it doesn't scale well for larger inputs.

It is complex in terms of performance and can be hard to debug or optimize.

Good Example (Following KISS)

```
class Fibonnaci {
    public int getFib(int n) { // Simple logic.
        int a = 0;
        int b = 1;
        int c = 0;
        for (int i = 1; i <= n; i++) {
            c = a + b;
            a = b;
            b = c;
        }
        return c;
    }
}
```

The above iterative approach is simple and efficient. The above simple logic solution is clearer, more efficient, and easier to understand than the recursive version.

YAGNI (You Aren't Gonna Need It)

It's a software development principle that encourages developers to focus on the immediate requirements of the project and avoid implementing the features or functionality that may never be needed.

The idea is to reduce complexity by not adding code or features until they are actually required.

Bad Example (Violating YAGNI)

```
class Order {
    private double totalPrice;
    private List<Item> items;
    private Boolean isDiscountAvailable;

    public Order(List<Item> items) {
        this.items = items;
        this.totalPrice = calculateTotalPrice(items);
    }

    private double calculateTotalPrice(List<Item> items) {
        double total = 0;
        for (Item item : items) {
            total += item.getPrice();
        }
```

```
		return total;
	}

	public void applyDiscount(Discount discount) {
		if(isDiscountAvailable)   {
			totalPrice -= discount.calculateDiscount(totalPrice);
		}
	}

	public void applyCoupon() {
		// Code to handle the coupon functionality here
		// even though coupons are not part of the current requirements.
	}

	public void calculateShippingCostForRegion(String region) {
		// Complex code for calculating international shipping cost.
		// even though it is not needed right now.
		return 0;
	}
}
```

Good Example (Following YAGNI)

```
class Order {
	private double totalPrice;
	private List<Item> items;
	private Boolean isDiscountAvailable;

	public Order(List<Item> items) {
		this.items = items;
		this.totalPrice = calculateTotalPrice(items);
```

```
        }

        private double calculateTotalPrice(List<Item> items) {
                double total = 0;
                for (Item item : items) {
                        total += item.getPrice();
                }
                return total;
        }

        public void applyDiscount(Discount discount) {
                if(isDiscountAvailable)   {
                        totalPrice -= discount.calculateDiscount(totalPrice);
                }
        }
}
```