1. The following code is a **Long Method (within the Bloaters group)**. It is extensive and contains multiple if and switch statements. This is both hard to read and difficult to alter if needed. This can be fixed by factoring the larger method, calculateTotalPrice(), into smaller methods. One such example is to make the if statements at the bottom of the method into another method to and make a single call. This reduces the complexity of this method while achieving the same result.

```
  public double calculateTotalPrice() {
    double total = 0.0;
    for (Item item : items) {
          double price = item.getPrice();
          switch (item.getDiscountType()) {
                case PERCENTAGE:
                price -= item.getDiscountAmount() * price;
                break;
                case AMOUNT:
                price -= item.getDiscountAmount();
                break;
                default:
                // no discount
                break;
          }
          total += price * item.getQuantity();
              if (item instanceof TaxableItem) {
              TaxableItem taxableItem = (TaxableItem) item;
              double tax = taxableItem.getTaxRate() / 100.0 * item.getPrice();
              total += tax;
          }
      }
    if (hasGiftCard()) {
          total -= 10.0; // subtract $10 for gift card
    }
    if (total > 100.0) {
          total *= 0.9; // apply 10% discount for orders over $100
    }
    return total;
  }
```

2. The following code, within the sendConfirmationEmail() method, is highly dependent on the Item class. This behavior indicates that this method is more interested in the framework of the Item class than the Order class, which is the class this method is part of. This can be fixed by addressing and extracting the necessary properties of the Item class in the Order class and passing them as parameters to the sendConfirmationEmail() method. This allows for minimal dependency of the Item class and can dissolve the concept of **Feature Envy (within the Couplers group)**.

```
public void sendConfirmationEmail() {
    String message = "Thank you for your order, " + customerName + "!\n\n" +
            "Your order details:\n";
    for (Item item : items) {
        message += item.getName() + " - " + item.getPrice() + "\n";
    }
    message += "Total: " + calculateTotalPrice();
    EmailSender.sendEmail(customerEmail, "Order Confirmation", message);
}
```

3. The following code illustrates the concept of **Data Clumps (within the Bloaters group)**. The Order class implements multiple parameters within the class and constructor, such as customerName and customerEmail. These are related variables/parameters that are being treated as separate entities. Nevertheless, to treat data clumps, we can create a Customer object to encapsulate these two variables and implementing this class within the Order class as a parameter.

```
public class Order {
    private List<Item> items;
    private String customerName;
    private String customerEmail;

    public Order(List<Item> items, String customerName, String customerEmail) {
        this.items = items;
        this.customerName = customerName;
        this.customerEmail = customerEmail;
    }
```

4. The following code shows the presence of **Long Parameter Lists (within the Bloaters group)**. The Item class contains five distinct parameters, making the list unnecessarily lengthy. When a method has too many parameters, it can be difficult to read and maintain the code. It can also make it harder to test the method. To fix this, it is possible to break up the algorithm and replace the parameters with method calls/setter methods.

```
    public Item(String name, double price, int quantity, DiscountType discountType,
double discountAmount) {
        this.name = name;
        this.price = price;
        this.quantity = quantity;
        this.discountType = discountType;
        this.discountAmount = discountAmount;
    }
```