PEMROGRAMAN LANJUT

Code Smells: Bloater

Oleh

Tri Hadiah Muliawati

Politeknik Elektronika Negeri Surabaya

2021

Politeknik Elektronika Negeri Surabaya
Departemen Teknik Informatika dan Komputer

Bloater

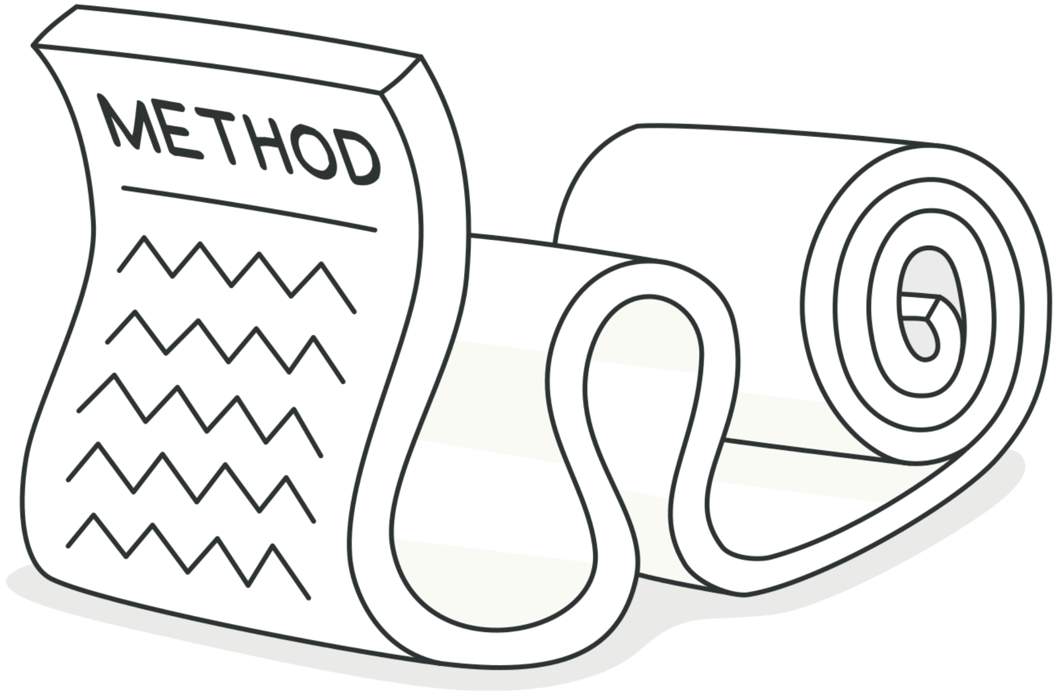
Code, methods and classes that have increased to
such gargantuan proportions that they’re hard to work with.

Bloater

- Long Method
- Long Parameter List
- Data Clump
- Primitive Obsession
- Large Class

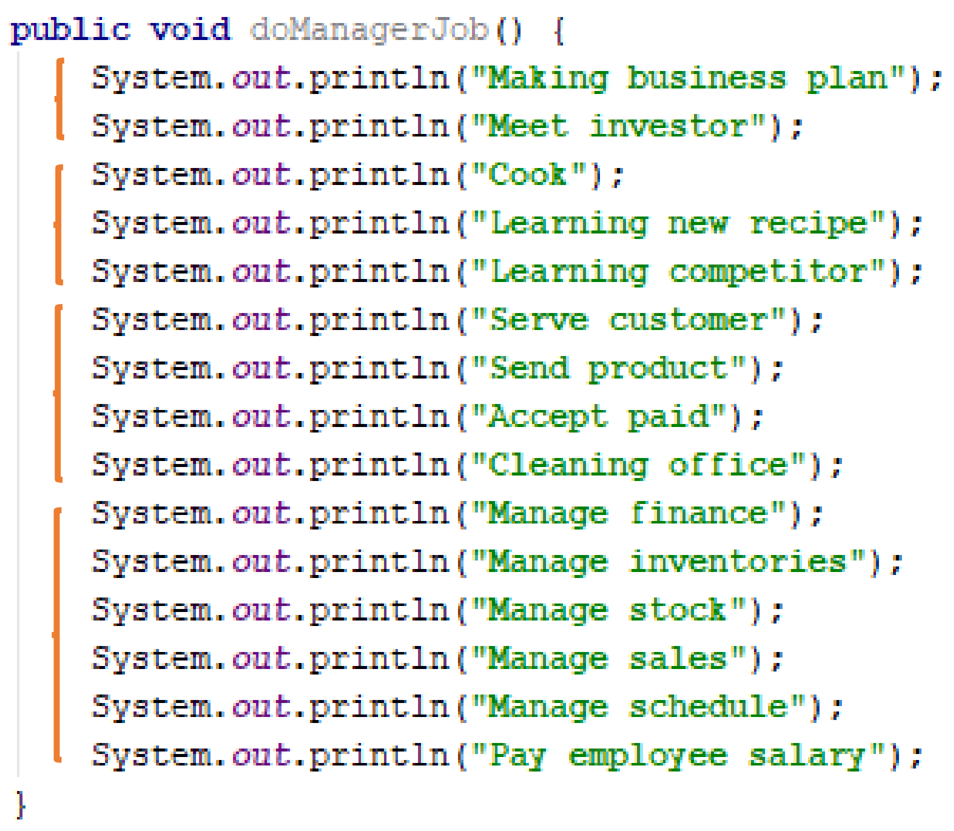
Long Method

- A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions.
- If you feel the need to comment on something inside a method, you should take this code and put it in a new method.

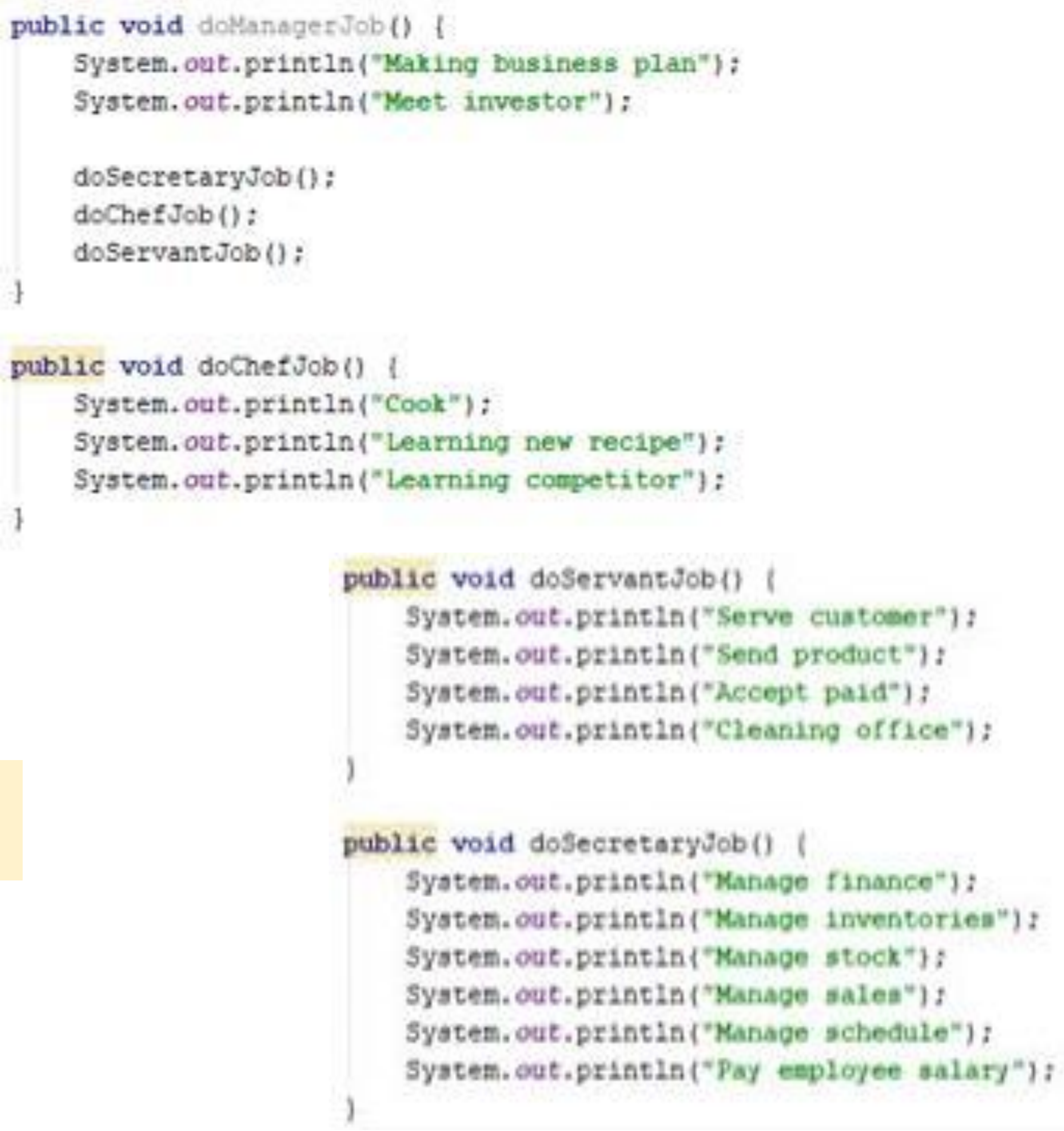


Long Method: Refactoring

- **Extract Method:** To reduce the length of method body.
- **Replace Temp with Query, Introduce Parameter Object, Preserve Whole Object:** If local variables and parameters interfere with extracting a method.
- **Replace Method with Method Object:** the local variables in long method are so intertwined that you can’t apply extract method.
- Conditional operators and loops are a good clue that code can be moved to a separate method.



```
public void doManagerJob() {
    System.out.println("Making business plan");
    System.out.println("Meet investor");
    System.out.println("Cook");
    System.out.println("Learning new recipe");
    System.out.println("Learning competitor");
    System.out.println("Serve customer");
    System.out.println("Send product");
    System.out.println("Accept paid");
    System.out.println("Cleaning office");
    System.out.println("Manage finance");
    System.out.println("Manage inventories");
    System.out.println("Manage stock");
    System.out.println("Manage sales");
    System.out.println("Manage schedule");
    System.out.println("Pay employee salary");
}
```



```
public void doManagerJob() {
    System.out.println("Making business plan");
    System.out.println("Meet investor");

    doSecretaryJob();
    doChefJob();
    doServantJob();
}

public void doChefJob() {
    System.out.println("Cook");
    System.out.println("Learning new recipe");
    System.out.println("Learning competitor");
}

public void doServantJob() {
    System.out.println("Serve customer");
    System.out.println("Send product");
    System.out.println("Accept paid");
    System.out.println("Cleaning office");
}

public void doSecretaryJob() {
    System.out.println("Manage finance");
    System.out.println("Manage inventories");
    System.out.println("Manage stock");
    System.out.println("Manage sales");
    System.out.println("Manage schedule");
    System.out.println("Pay employee salary");
}
```

The original method (doManagerJob()) consists of more than 10 Line of Code (LoC)

```java
public void printEmployeeInformation() {
    System.out.println("===========================");
    System.out.println("Print Employee Information");
    System.out.println("===========================");
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a employee ID: ");
    String ID = input.nextLine();
    String name = getName(ID);
    String age = getAge(ID);
    System.out.println("");
    System.out.println("ID\t: " + ID);
    System.out.println("Name\t: " + name);
    System.out.println("Age\t: " + age);
    System.out.println("===========================");
    System.out.println("End of Employee Information");
    System.out.println("===========================");
}
```

```java
private void printInformation(String ID, String name, String age) {
    System.out.println("");
    System.out.println("ID\t: " + ID);
    System.out.println("Name\t: " + name);
    System.out.println("Age\t: " + age);
}

private void printFooter() {
    System.out.println("===========================");
    System.out.println("End of Employee Information");
    System.out.println("===========================");
}

private void printHeader() {
    System.out.println("===========================");
    System.out.println("Print Employee Information");
    System.out.println("===========================");
}
```

```java
public void printEmployeeInformation() {
    printHeader();
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a employee ID: ");
    String ID = input.nextLine();
    String name = getName(ID);
    String age = getAge(ID);
    printInformation(ID, name, age);
    printFooter();
}
```

```java
public void printReceipt(ProductInfo info){
    System.out.println("=============================");
    System.out.println("===========Receipt===========");
    System.out.println("=============================");

    boolean eligibleForSpecialPrice = info.count > 10;
    if (eligibleForSpecialPrice) {
        info.unitPrice = 50;
    } else {
        info.unitPrice = 70;
    }
    System.out.println(info.unitPrice * info.count);

    System.out.println("=============================");
    System.out.println("=======End of Receipt========");
    System.out.println("=============================");
}
```

```java
public void printReceipt(ProductInfo info){
    printReceiptHeader();

    info = calculateUnitPrice(info);
    System.out.println(info.unitPrice * info.count);

    printReceiptFooter();
}
```

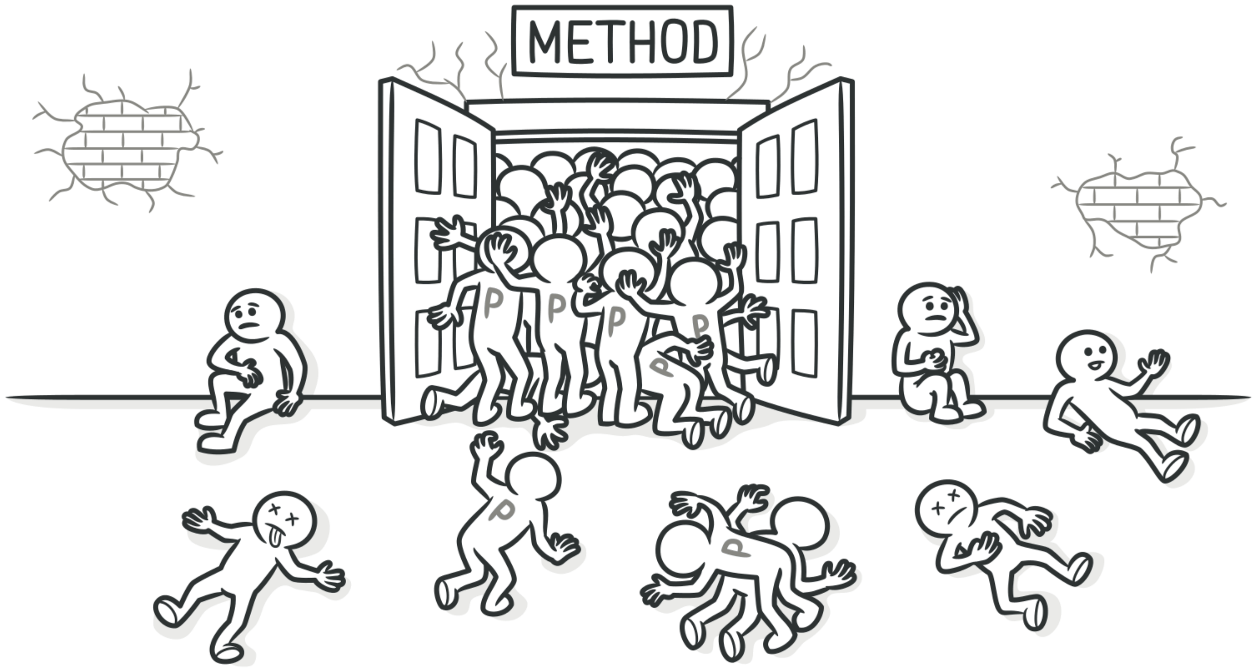
```java
private void printReceiptFooter() {
    System.out.println("=============================");
    System.out.println("=======End of Receipt========");
    System.out.println("=============================");
}

private void printReceiptHeader() {
    System.out.println("=============================");
    System.out.println("===========Receipt===========");
    System.out.println("=============================");
}

private ProductInfo calculateUnitPrice(ProductInfo info) {
    boolean eligibleForSpecialPrice = info.count > 10;

    if (eligibleForSpecialPrice) {
        info.unitPrice = 50;
    } else {
        info.unitPrice = 70;
    }
    return info;
}
```

```java
private ProductInfo calculateUnitPrice(ProductInfo info) {
    boolean eligibleForSpecialPrice = info.count > 10;

    if (eligibleForSpecialPrice) {
        info.unitPrice = 50;
    } else {
        info.unitPrice = 70;
    }
    return info;
}
```

```java
private ProductInfo calculateUnitPrice(ProductInfo info) {
    if (isEligibleForSpecialPrice(info)) {
        info.unitPrice = 50;
    } else {
        info.unitPrice = 70;
    }
    return info;
}

private boolean isEligibleForSpecialPrice(ProductInfo info) {
    return info.count > 10;
}
```

Long Parameter List

- More than three or four parameters for a method.



Long Parameter List: Refactoring

- **Preserve Whole Object:** Instead of passing a group of data received from another object as parameters, pass the object itself to the method.
- **Introduce Parameter Object:** If there are several unrelated data elements, sometimes you can merge them into a single parameter object.
- **Replace Parameter with Method Call:** If some of the arguments are just results of method calls of another object. Place object in the field of its own class or passed it as a method parameter.

```java
public void printReceipt(ProductInfo info) {
    printReceiptHeader();

    info.unitPrice = calculateUnitPrice(info.count, info.unitPrice,
            info.isLimitedEdition);
    System.out.println(info.unitPrice * info.count);

    printReceiptFooter();
}
```

```java
public void printReceipt(ProductInfo info) {
    printReceiptHeader();

    info.unitPrice = calculateUnitPrice(info);
    System.out.println(info.unitPrice * info.count);

    printReceiptFooter();
}
```

```java
private int calculateUnitPrice(int count, int unitPrice,
        boolean isLimitedEdition) {

    if (!isLimitedEdition) {
        if (count > 20) {
            return unitPrice - 10;
        } else if (count > 10) {
            return unitPrice - 5;
        } else {
            return unitPrice;
        }
    }
    return unitPrice;
}
```

```java
private int calculateUnitPrice(ProductInfo info) {

    if (!info.isLimitedEdition) {
        if (info.count > 20) {
            return info.unitPrice - 10;
        } else if (info.count > 10) {
            return info.unitPrice - 5;
        } else {
            return info.unitPrice;
        }
    }
    return info.unitPrice;
}
```

```java
public void printEmployeeInformation() {
    printHeader();

    Scanner input = new Scanner(System.in);
    System.out.print("Enter a employee ID: ");
    String ID = input.nextLine();
    String name = getName(ID);
    String province = getProvince(ID);
    String city = getCity(ID);
    String country = getCountry(ID);

    printInformation(ID, name, city, province, country);
    printFooter();
}
```

```java
public void printEmployeeInformation() {
    printHeader();

    Scanner input = new Scanner(System.in);
    System.out.print("Enter a employee ID: ");
    String ID = input.nextLine();
    String name = getName(ID);
    Address address = getAddress(ID);

    printInformation(ID, name, address);
    printFooter();
}
```

```java
private void printInformation(String ID, String name, String city,
        String province, String country) {
    System.out.println("");
    System.out.println("ID\t: " + ID);
    System.out.println("Name\t: " + name);
    System.out.println("City\t: " + city);
    System.out.println("Province\t: " + province);
    System.out.println("Country\t: " + country);
}
```

```java
private void printInformation(String ID, String name, Address address) {
    System.out.println("");
    System.out.println("ID\t: " + ID);
    System.out.println("Name\t: " + name);
    System.out.println("City\t: " + address.city);
    System.out.println("Province\t: " + address.province);
    System.out.println("Country\t: " + address.country);
}
```

```
int basePrice = quantity * itemPrice;
double seasonDiscount = this.getSeasonalDiscount();
double fees = this.getFees();
double finalPrice = discountedPrice(basePrice, seasonDiscount, fees);
```

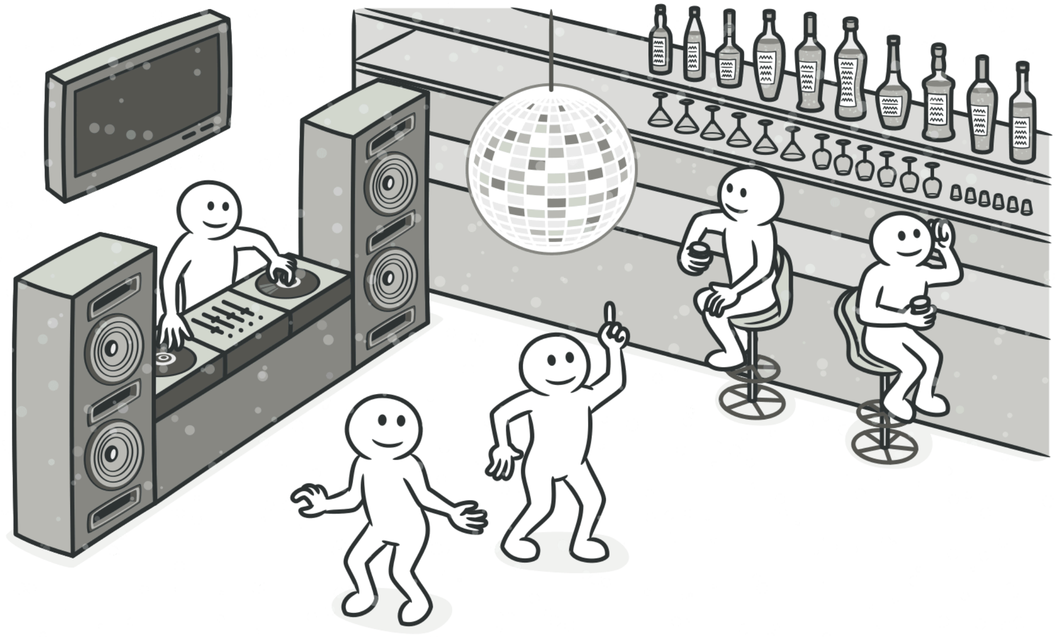
```
int basePrice = quantity * itemPrice;
double finalPrice = discountedPrice(basePrice);
```

Parameter `seasonDiscount` and `fees` can be placed inside `discountedPrice()` method.

We can also move method calls (`getSeasonalDiscount()` and `getFees()`) into `discountedPrice()` method.

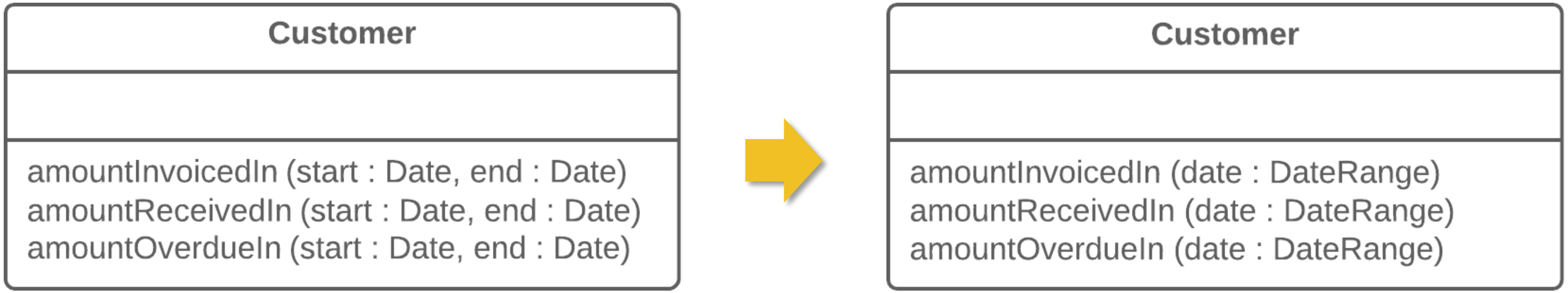
Data Clumps

- Sometimes different parts of the code contain identical groups of variables (such as parameters for connecting to a database).
- These clumps should be turned into their own classes.



Data Clumps: Refactoring

- **Extract Class:** If repeating data comprises the fields of a class, move them to their own class.
- **Introduce Parameter Object:** If the same data clumps are passed in the parameters of methods.
- **Preserve Whole Object:** If some of the data is passed to other methods, think about passing the entire data object to the method instead of just individual fields.


Customer
amountInvoicedIn (start : Date, end : Date)
amountReceivedIn (start : Date, end : Date)
amountOverdueIn (start : Date, end : Date)
Customer
amountInvoicedIn (date : DateRange)
amountReceivedIn (date : DateRange)
amountOverdueIn (date : DateRange)

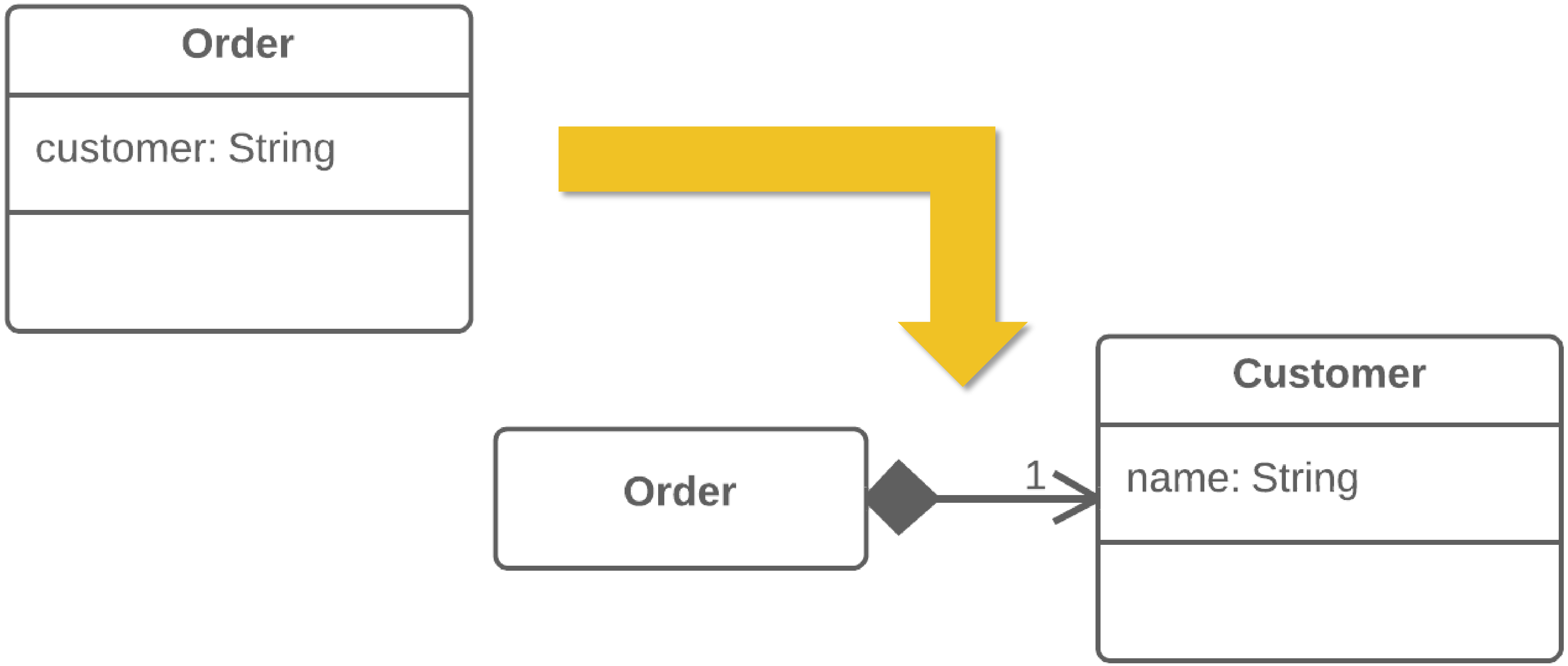
Primitive Obsession

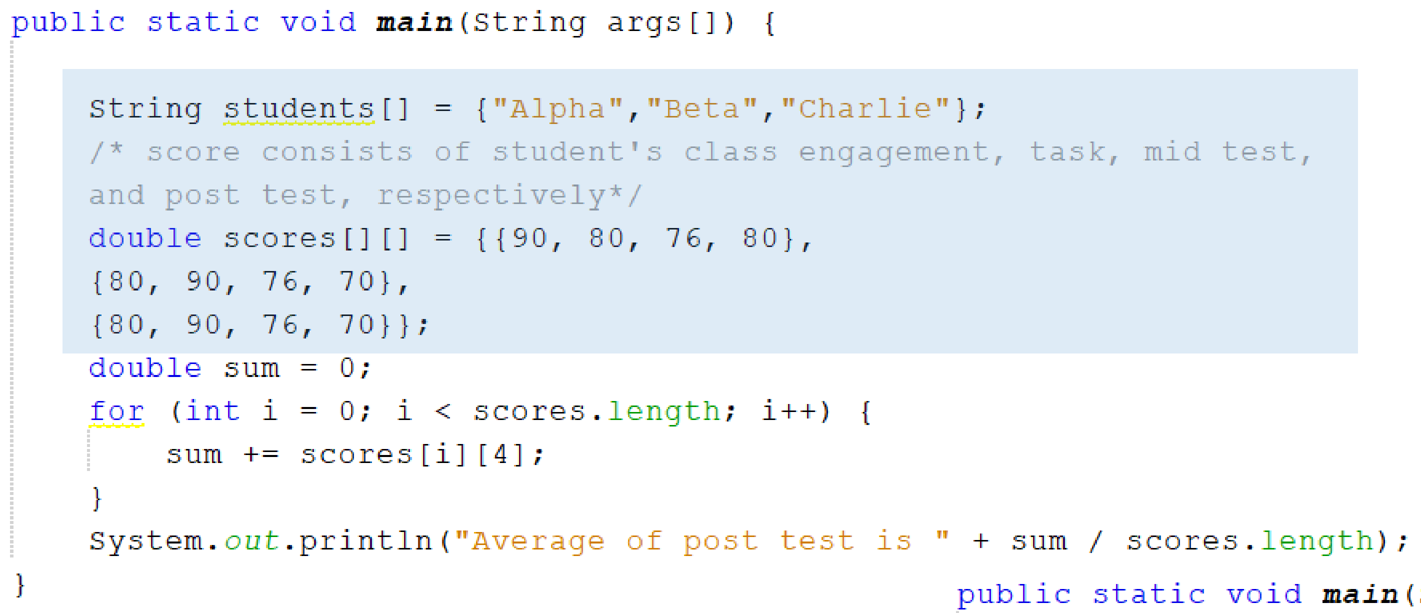
- Use of primitives instead of small objects for simple tasks (such as currency, ranges, special strings for phone numbers, etc.)
- Use of constants for coding information (such as a constant USER_ADMIN_ROLE = 1 for referring to users with administrator rights.)
- Use of string constants as field names for use in data arrays.

Primitive Obsession: Refactoring

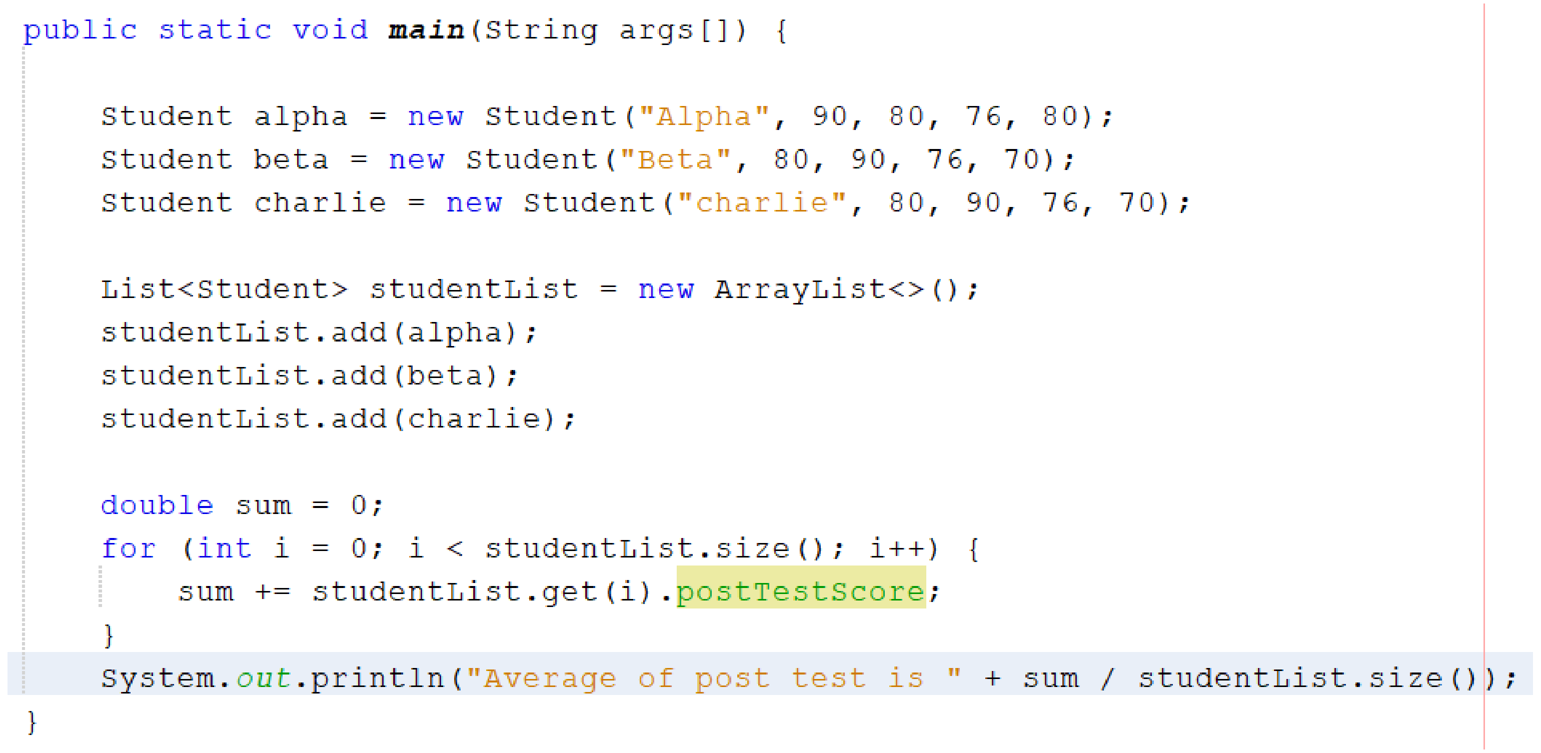
- **Replace Value with Object:** If you have a large variety of primitive fields, it may be possible to logically group some of them into their own class.
- **Introduce Parameter Object** or **Preserve Whole Object:** If the values of primitive fields are used in method parameters.
- **Replace Array with Object:** If there are arrays among the variables.
- **Replace Type Code with Class, Replace Type Code with Subclasses or Replace Type Code with State/Strategy:** If complicated data is coded in variables.

customer field has its own behavior and associated data. Thus, it is better to turn it into class.

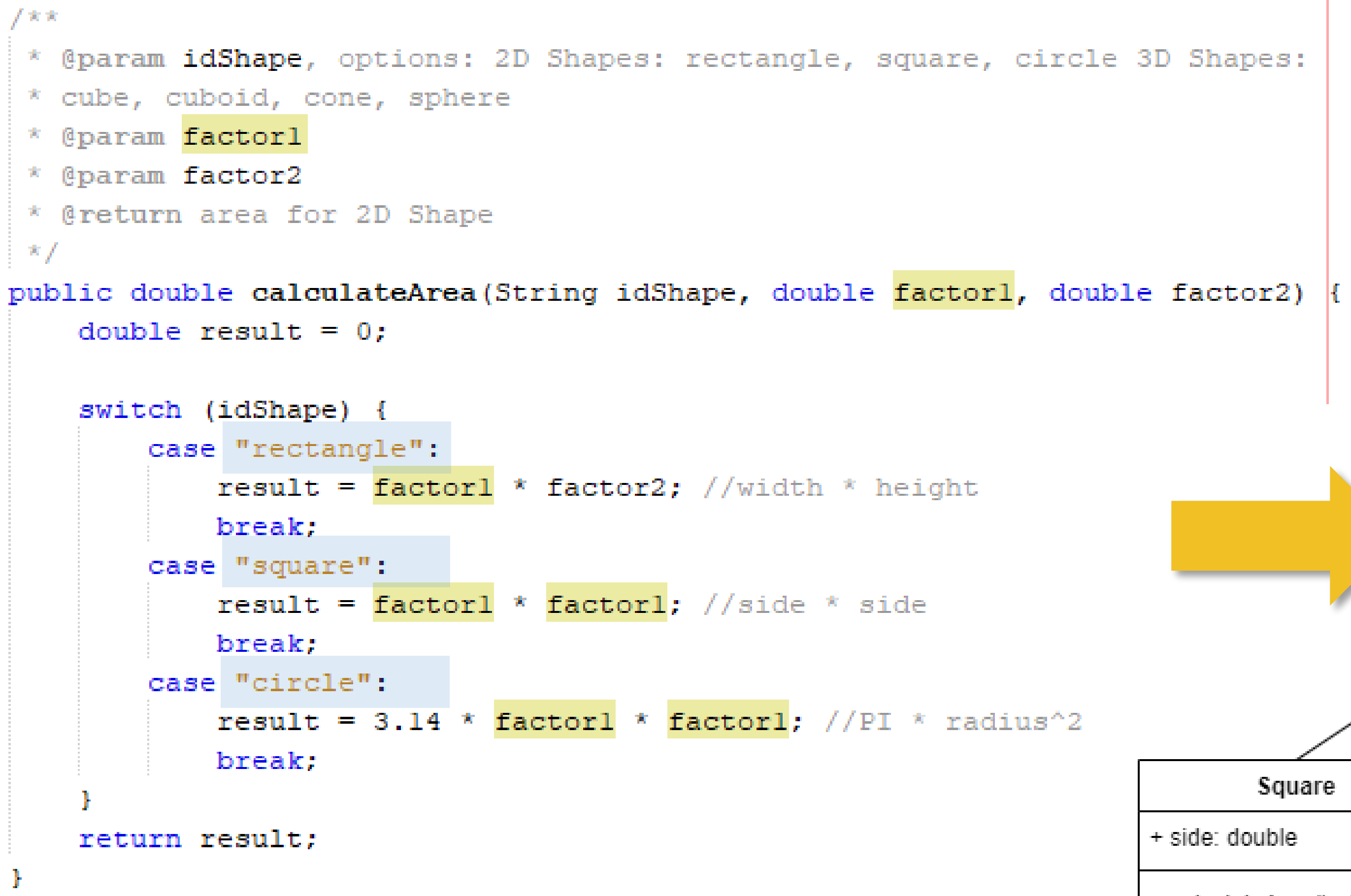




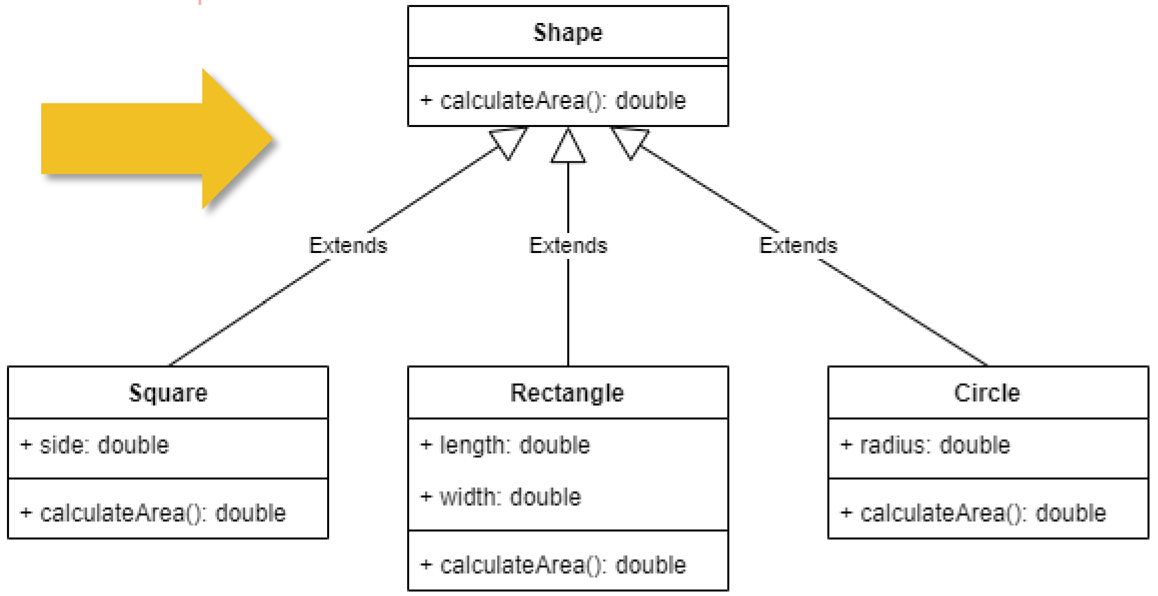
```
public static void main(String args[]) {

    String students[] = {"Alpha","Beta","Charlie"};
    /* score consists of student's class engagement, task, mid test,
    and post test, respectively*/
    double scores[][] = {{90, 80, 76, 80},
    {80, 90, 76, 70},
    {80, 90, 76, 70}};
    double sum = 0;
    for (int i = 0; i < scores.length; i++) {
        sum += scores[i][4];
    }
    System.out.println("Average of post test is " + sum / scores.length);
}
```



```
public static void main(String args[]) {

    Student alpha = new Student("Alpha", 90, 80, 76, 80);
    Student beta = new Student("Beta", 80, 90, 76, 70);
    Student charlie = new Student("charlie", 80, 90, 76, 70);

    List<Student> studentList = new ArrayList<>();
    studentList.add(alpha);
    studentList.add(beta);
    studentList.add(charlie);

    double sum = 0;
    for (int i = 0; i < studentList.size(); i++) {
        sum += studentList.get(i).postTestScore;
    }
    System.out.println("Average of post test is " + sum / studentList.size());
}
```

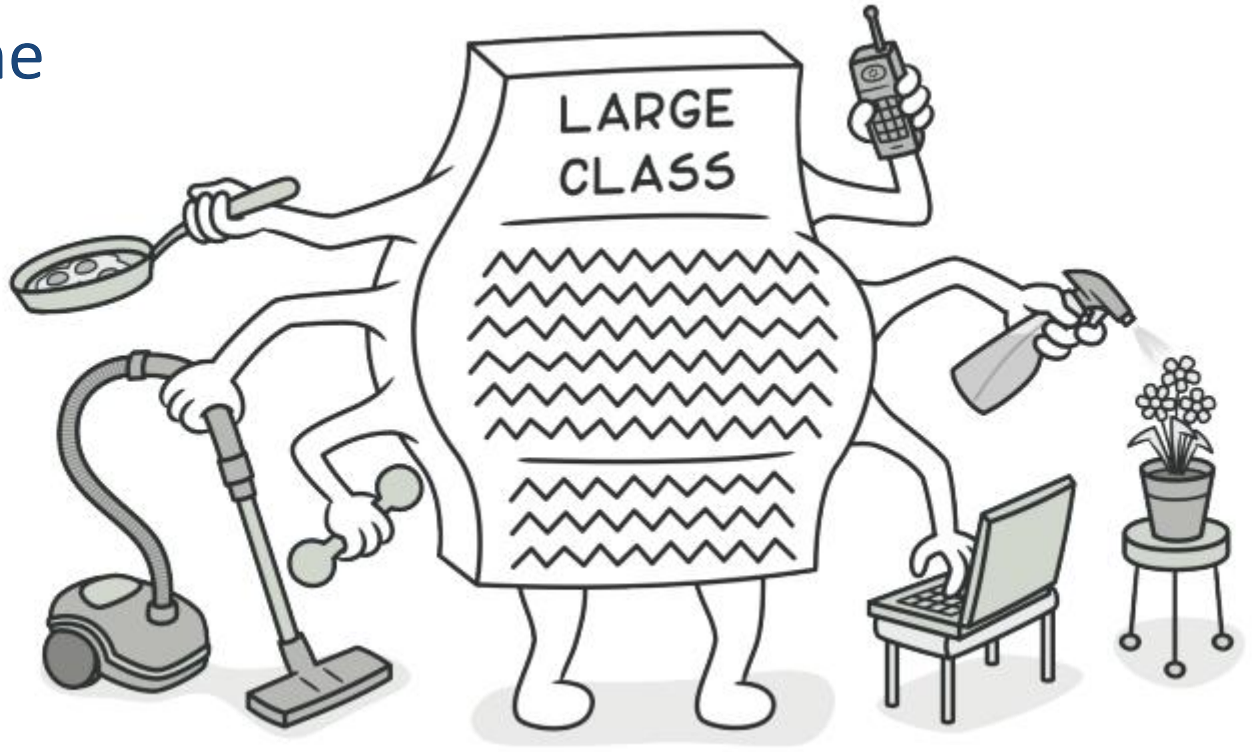


```java
/**
 * @param idShape, options: 2D Shapes: rectangle, square, circle 3D Shapes:
 * cube, cuboid, cone, sphere
 * @param factor1
 * @param factor2
 * @return area for 2D Shape
 */
public double calculateArea(String idShape, double factor1, double factor2) {
    double result = 0;

    switch (idShape) {
        case "rectangle":
            result = factor1 * factor2; //width * height
            break;
        case "square":
            result = factor1 * factor1; //side * side
            break;
        case "circle":
            result = 3.14 * factor1 * factor1; //PI * radius^2
            break;
    }
    return result;
}
```



Large Class

- Classes usually start small. But over time, they get bloated as the program grows.



Large Class: Refactoring

- **Extract Class:** if part of the behavior of the large class can be spun off into a separate component.
- **Extract Subclass:** if part of the behavior of the large class can be implemented in different ways or is used in rare cases.
- **Extract Interface:** if it's necessary to have a list of the operations and behaviors that the client can use.
- If a large class is responsible for the graphical interface, you may try to move some of its data and behavior to a separate domain object.

References

- Martin, Robert C. Clean Code: A Handbook of Agile Software Craftsmanship. Pearson. 2008.
- Fowler, Martin. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1999.
- https://refactoring.guru/
- Putra, F. Z. P., 2019. Rancang Bangun Pustaka untuk Refactoring Otomatis terhadap Long Method Code Smell.

bridge to the future

http://www.eepis-its.edu