Fixing Bloating Code

Andrejs Doronins
TEST AUTOMATION ENGINEER

Bloaters

Very large methods and classes that are hard to work with. Bloaters usually accumulate over time as software evolves.

Method Bloaters

```
double calculateTotal(...){
    // rule of thumb - up to 10 lines
    // 10-20 lines is still often OK
    // 20+ - consider refactoring
}
```

Class Bloaters

```
class Order {
    // 2+ Responsibilities
}
```

SRP
Single Responsibility Rule

Bloating Code

Refactoring – Split It Up!

Refactoring Bloaters

Yes	No
Split large classes into smaller classes	Shorter class, variable, and method names
Split large methods into smaller methods	

Bloaters

- Long parameter list
- Long method
- Contrived complexity
- Primitive obsession
- Data clumps
- Large class

Demo

Introduction to the project

Long Parameter List

Your method has 4 or more parameters

Long Parameter List Issues

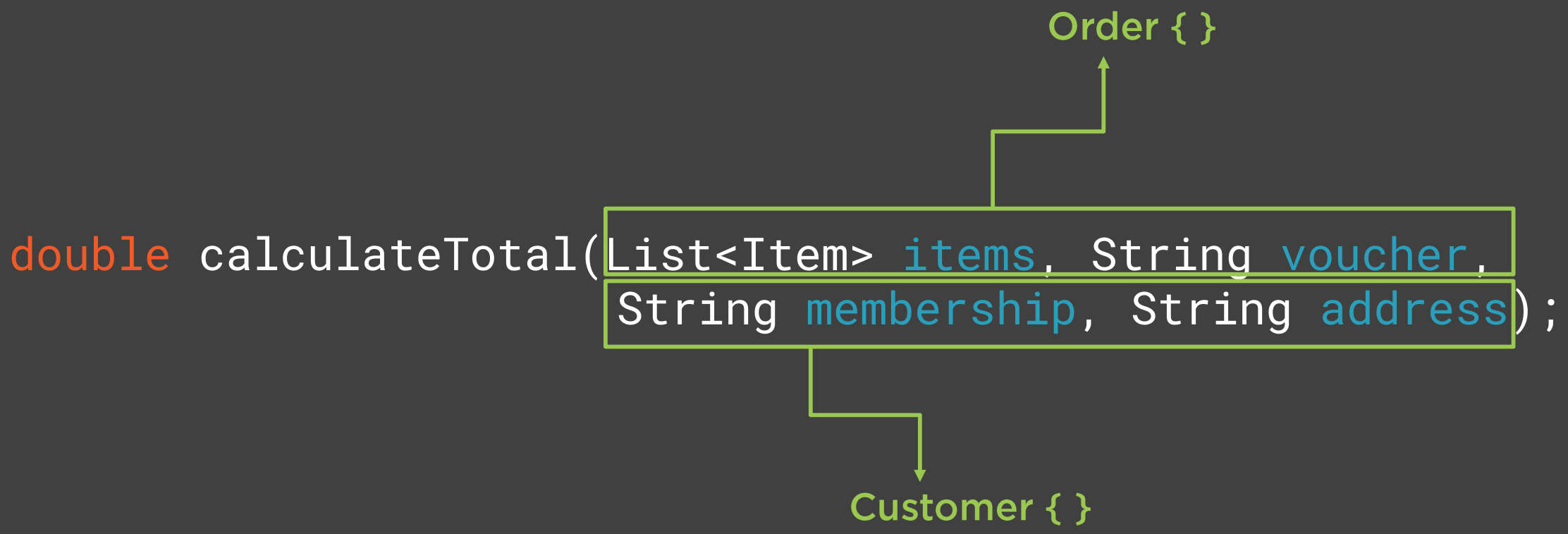
Hard to understand

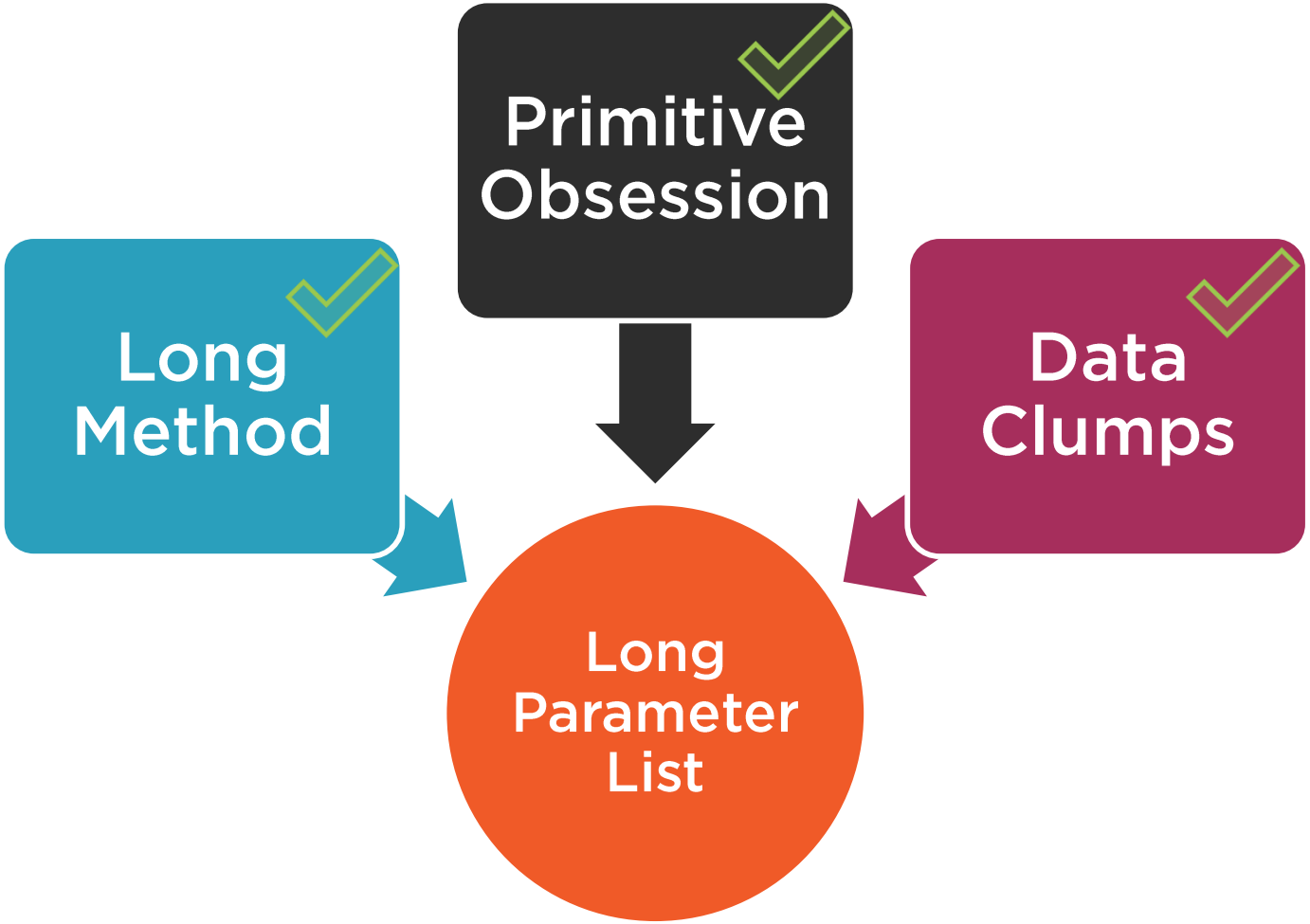
Difficult to remember the position of each argument

Acts like a magnet for even more arguments and code

Clear What the Method Does?

```
calculate(10, false, “US”, new Order());
```


Order { }
double calculateTotal(List<Item> items, String voucher,
String membership, String address);
Customer { }


Primitive Obsession
Long Method
Data Clumps
Long Parameter List

Long Method

A method contains too many lines of code

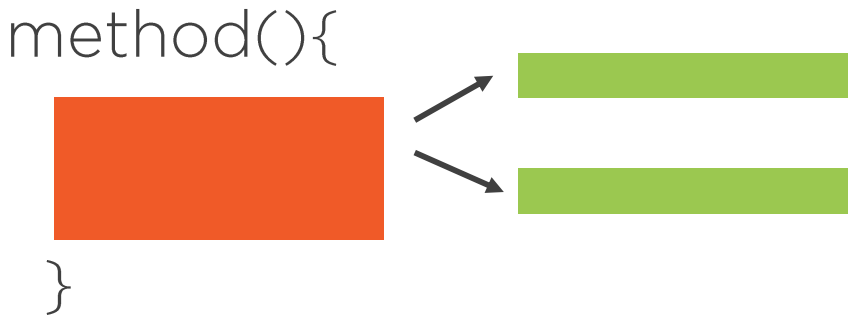
Long Method Issues

Difficult to maintain

Easy to break

Attracts even more code

How to Fix Long Method



Extract Method

(split a method into several smaller methods)

Decompose Conditional

(a form of Extract method)

Before

```
calculateTotal(...){
// sum up prices
// apply voucher
// add delivery fee
}
```

After

```
calculateTotal(...){
 // delegate to other methods
}

sumPrices(...){ }

applyVoucher(...){ }

addDeliveryFee(...){ }
```

Benefits Achieved

Shorter and easier to understand

More reusable code

Contrived Complexity

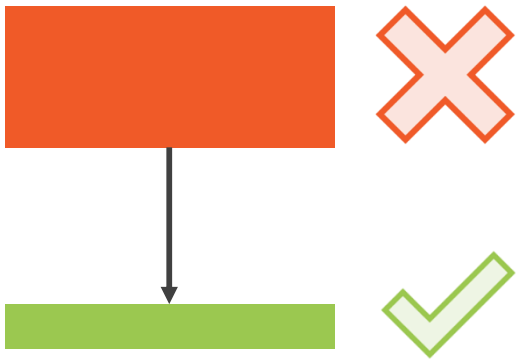
Code that achieves an objective but is too complex. An easier, shorter or more elegant way exists to achieve the same goal.

Contrived Complexity

Hard to understand

Higher chance of breaking when changing

How to Fix Contrived Complexity



Substitute Algorithm

(Replace with simpler and more elegant code)

Before

```
List<Double> prices = new ArrayList<>();


for(Item item : items){

   prices.add(item.price());

  }


for(double price : prices){

   baseTotal = baseTotal + price;

 }
```

After

```
for(Item item : items){
 baseTotal = baseTotal+item.price();
  }
```

Benefits Achieved

Shorter and easier to understand

Primitive Obsession

Use of primitives (int, String, etc.) instead of objects

Primitive Obsession Issues

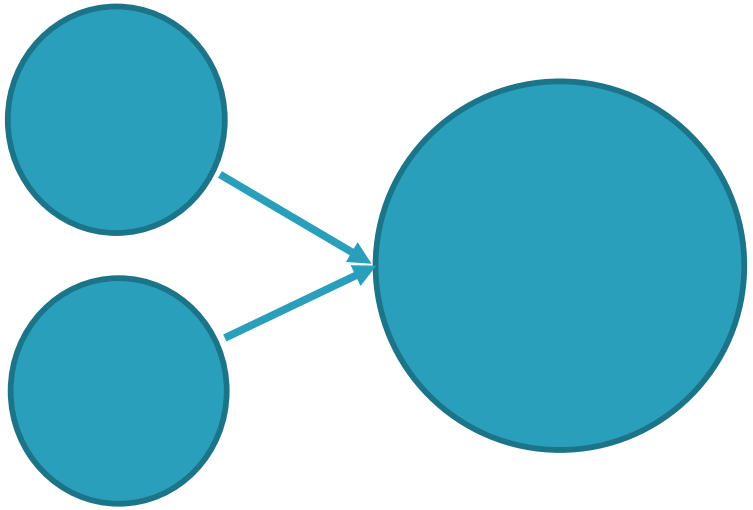
Major cause of long parameter lists

Causes code duplication

Not type safe and prone to errors

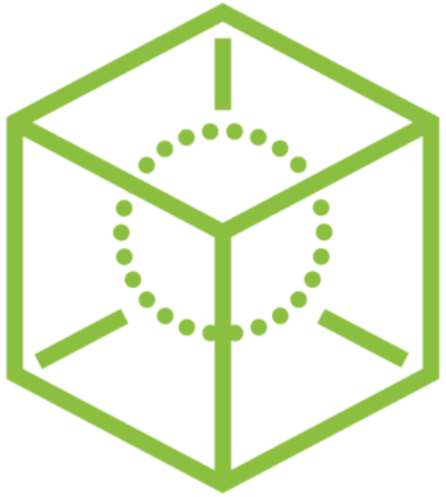
- getCountryVat (“Fance”);

How to Fix Primitive Obsession



Preserve Whole Object

(Just pass in the
whole object as a parameter)



Introduce Parameter Object

(Create an object, move the primitives there, pass in the object)

Preserve Whole Object

```
String a = customer.getA();
String b = customer.getB();
double calculateTotal(a, b){ }
```

```
double calculateTotal(customer){
    String a = customer.getA();
    String b = customer.getB();
 }
```

Introduce Parameter Object

```
double calculateTotal(a, b){ }
```

```
class Order {
 String a;
 String b;
}

Order order = new Order();
double calculateTotal(order){ }
```

Benefits Achieved

- Safer code (typos won't compile)
- Reduced number of parameters
- Better encapsulation

Data Clumps

A group of variables which are passed around together (in a clump) throughout various parts of the program

Data Clumps Examples

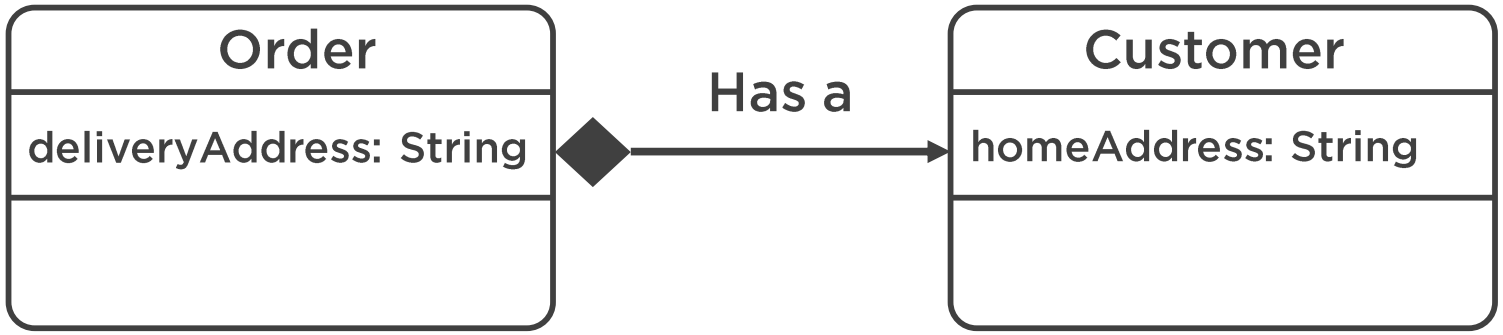
```
sender.sendEmailTo(“Mr.”, “John”, “Doe”);

system.bookRoom(startTime, endTime);

system.bookReturnFlight(there, back);

manager.setDeliveryTimeWindow(from, to);

checkout.calculateTotal(order, customer);
```

Data Clumps Issues

Major cause of long parameter lists

Causes code duplication

Both Data Clumps and Primitive Obsession often suggest a specific problem: missing domain objects


Order
deliveryAddress: String
Has a
Customer
homeAddress: String

Before

```
Order {
   String address;
}

deliverTo(address);
```

After

```
Order {
   Customer customer;
}

deliverTo(customer.getAddress());
```

Large Class

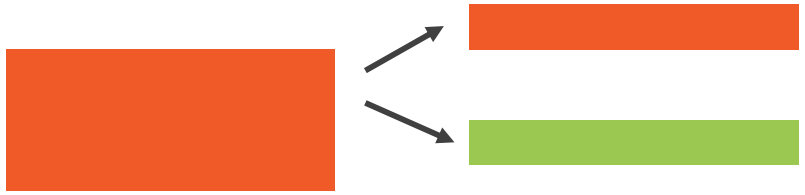
A class that does many things (many responsibilities). A "God Object" is a class that does almost everything.

Large Class Issues

Very difficult and slow to maintain

Violates the Single Responsibility Principle

How to Fix Large Class



Extract Class

(Split into several smaller classes)

Summary

Methods and Classes can and should be split

Strive for the simplest and most elegant solution

Too many primitives indicates you should introduce new classes