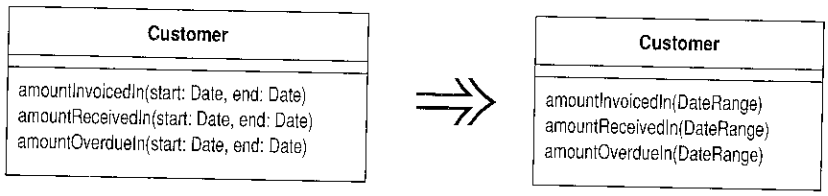
Introduce Parameter Object

You have a group of parameters that naturally go together.

Replace them with an object.



Motivation

Introduce Parameter Object

Often you see a particular group of parameters that tend to be passed together. Several methods may use this group, either on one class or in several classes. Such a group of classes is a data clump and can be replaced with an object that carries all of this data. It is worthwhile to turn these parameters into objects just to group the data together. This refactoring is useful because it reduces the size of the parameter lists, and long parameter lists are hard to understand. The defined accessors on the new object also make the code more consistent, which again makes it easier to understand and modify.

You get a deeper benefit, however, because once you have clumped together the parameters, you soon see behavior that you can also move into the new class. Often the bodies of the methods have common manipulations of the parameter values. By moving this behavior into the new object, you can remove a lot of duplicated code.

Mechanics

- Create a new class to represent the group of parameters you are replacing. Make the class immutable.
- Compile.

- Use *Add Parameter (275)* for the new data clump. Use a null for this parameter in all the callers.

  ⇒ *If you have many callers, you can retain the old signature and let it call the new method. Apply the refactoring on the old method first. You can then move the callers over one by one and remove the old method when you're done.*

- For each parameter in the data clump, remove the parameter from the signature. Modify the callers and method body to use the parameter object for that value.

- Compile and test after you remove each parameter.

- When you have removed the parameters, look for behavior that you can move into the parameter object with *Move Method (142)*.

  ⇒ *This may be a whole method or part of a method. If it is part of a method, use* Extract Method (110) *first and then move the new method over.*

Introduce Parameter Object

Example

I begin with an account and entries. The entries are simple data holders.

```
class Entry...
  Entry (double value, Date chargeDate) {
      _value = value;
      _chargeDate = chargeDate;
  }
  Date getDate(){
      return _chargeDate;
  }
  double getValue(){
      return _value;
  }
  private Date _chargeDate;
  private double _value;
```

My focus is on the account, which holds a collection of entries and has a method for determining the flow of the account between two dates:

```
class Account...
  double getFlowBetween (Date start, Date end) {
      double result = 0;
      Enumeration e = _entries.elements();
      while (e.hasMoreElements()) {
          Entry each = (Entry) e.nextElement();
          if (each.getDate().equals(start) ||
```

```
            each.getDate().equals(end) ||
            (each.getDate().after(start) && each.getDate().before(end)))
        {
            result += each.getValue();
        }
    }
    return result;
}

private Vector _entries = new Vector();

client code...
double flow = anAccount.getFlowBetween(startDate, endDate);
```

I don't know how many times I come across pairs of values that show a range, such as start and end dates and upper and lower numbers. I can understand why this happens, after all I did it all the time myself. But since I saw the range pattern [Fowler, AP] I always try to use ranges instead. My first step is to declare a simple data holder for the range:

Introduce Parameter Object

```
class DateRange {
  DateRange (Date start, Date end) {
      _start = start;
      _end = end;
  }
  Date getStart() {
      return _start;
  }
  Date getEnd() {
      return _end;
  }
  private final Date _start;
  private final Date _end;
}
```

I've made the date range class immutable; that is, all the values for the date range are final and set in the constructor, hence there are no methods for modifying the values. This is a wise move to avoid aliasing bugs. Because Java has pass-by-value parameters, making the class immutable mimics the way Java's parameters work, so this is the right assumption for this refactoring.

Next I add the date range into the parameter list for the `getFlowBetween` method:

```
class Account...
  double getFlowBetween (Date start, Date end, DateRange range) {
      double result = 0;
      Enumeration e = _entries.elements();
      while (e.hasMoreElements()) {
          Entry each = (Entry) e.nextElement();
          if (each.getDate().equals(start) ||
```

```
            each.getDate().equals(end) ||
            (each.getDate().after(start) && each.getDate().before(end)))
        {
            result += each.getValue();
        }
    }
    return result;
}

client code...
  double flow = anAccount.getFlowBetween(startDate, endDate, null);
```

At this point I only need to compile, because I haven't altered any behavior yet.

The next step is to remove one of the parameters and use the new object instead. To do this I delete the start parameter and modify the method and its callers to use the new object instead:

Introduce Parameter Object

```
class Account...
  double getFlowBetween (Date end, DateRange range) {
      double result = 0;
      Enumeration e = _entries.elements();
      while (e.hasMoreElements()) {
          Entry each = (Entry) e.nextElement();
          if (each.getDate().equals(range.getStart()) ||
              each.getDate().equals(end) ||
              (each.getDate().after(range.getStart()) && each.getDate().before(end)))
          {
              result += each.getValue();
          }
      }
      return result;
  }

client code...
  double flow = anAccount.getFlowBetween(endDate, new DateRange (startDate, null));
```

I then remove the end date:

```
class Account...
  double getFlowBetween (DateRange range) {
      double result = 0;
      Enumeration e = _entries.elements();
      while (e.hasMoreElements()) {
          Entry each = (Entry) e.nextElement();
          if (each.getDate().equals(range.getStart()) ||
              each.getDate().equals(range.getEnd()) ||
              (each.getDate().after(range.getStart()) && each.getDate().before(range.getEnd())))
```

```
            {
                result += each.getValue();
            }
        }
        return result;
    }

client code...
    double flow = anAccount.getFlowBetween(new DateRange (startDate, endDate));
```

I have introduced the parameter object; however, I can get more value from this refactoring by moving behavior from other methods to the new object. In this case I can take the code in the condition and use *Extract Method (110)* and *Move Method (142)* to get

```
class Account...
  double getFlowBetween (DateRange range) {
      double result = 0;
      Enumeration e = _entries.elements();
      while (e.hasMoreElements()) {
          Entry each = (Entry) e.nextElement();
          if (range.includes(each.getDate())) {
              result += each.getValue();
          }
      }
      return result;
  }

class DateRange...
  boolean includes (Date arg) {
      return (arg.equals(_start) ||
              arg.equals(_end) ||
              (arg.after(_start) && arg.before(_end)));
  }
```

I usually do simple extracts and moves such as this in one step. If I run into a bug, I can back out and take the two smaller steps.