

Workshop Guide

Restructuring Code: From “Push” To “Pull”

Overview

The Initial Situation

The restructuring pattern “From Push to Pull” is applicable to code that follows the general pattern described here.

Goal of the Code

The code produces a sequence of data objects, each of which may hold several data (e.g. several numbers).

Shape of the Code

The general structure of the code is similar to this:

- one class for a single input data (here: the class `Transaction`)
- a sequence of these input data elements (here: `List<Transaction>`)
- very few classes with logic (here: the class `PushingBalancesCalculator`)
- one class for a single output data entry (here: `BalancesOfMonth`)
- a sequence of these output data elements (here: `List<BalancesOfMonth>`) which is constructed early in the program run

The basic mechanics of the logic class(es) is as follows:

- initially, the sequence of output data is not filled
- the main logic part consists of two interlaced loops:
 - the outer loop iterates over the sequence of output data objects (here: the months for which the output data is to be determined)
 - the inner loop iterates over the sequence of input data objects and calculates the data for one output data object (here: the sum of the transactions for each month)
 - at the end of the inner loop, the calculated data is written to that output data object (“push”)
- finally, the sequence of output data is filled
- the logic class(es) may contain multiple of these interlaced loops in sequence, where a subsequent loop may access and modify the output data that a previous loop has calculated and pushed to the sequence of output data

First Impressions of the Code

The code doesn't look too bad. It has some issues. The algorithm is not easy to understand. Testing is only possible on an integration base. There is no isolation or abstraction.

Desired Final Situation

The code should be crisper. That means: easy to read, understand, test and extend. The overall goal is to extract the calculations of the individual values into separate methods. This can be broken up into three major parts:

1. Disentangle the for loops, isolate the loop body and extract it
2. Build one data object for each loop iteration (instead of reusing the same object over and over again)
3. Isolate the calculations of the distinct values into separate methods and calculate them on demand

In the following, we will split up these two parts into fine-grained steps that can be applied to the code.

We do this by:

- Defining one interaction point with outside objects (Step 1)
- Separating value calculation from iteration (Steps 2-3)
- Encapsulating the calculation object (Step 4)
- Separating calculation for each value (Step 5)
- Eliminating unwanted state (Step 6-8)
- Tidying up (Step 9)
- Achieving the “Pull” Structure (Final Step)

Restrictions / Rules:

- Change one thing at a time.
- Right now, we neither want to change the outside world nor our calculator class API, only the internals of our calculator class. The result objects (`BalancesOfMonth`) represent the outside world in our example. Therefore, we will leave the result objects and the `fillData` method signature untouched.

Note: The workspace with the step number contains the **solution** to the step!

Part I: Disentangle the For Loops and Extract the Loop Body

Overall Goal:

Extract most of the outer for loop body into a separate method. This new method must not know about the result object!

Pre-Arrangements (Workspace: Push)

Tidy up the code, rename local variables to make their names more descriptive. Push the loops as far to the outside of the method as possible (move as much code as possible inside the loop bodies).

Step 1: Decoupling from the outside world

Observations:

- The outer loop iterates over the list of result objects that need to be filled with the calculated data.
- The method calculates two values for each result object: `balance` and `averageBalance`.
- These two values cannot both be returned from a single method.

Solution:

Create an object that will transport the calculated values. This intermediate object will only hold the values, it will not contain any logic. It will have one setter that sets all calculated values simultaneously. It will also have getters, one for each of the calculated values.

In the main logic part, we will create one such intermediate object at the beginning of the calculation (can also go inside the outer loop). Instead of directly writing the calculated values to the output data structure, we write them to the intermediate object and then read them again from the intermediate object in order to write them to the output data structure. We reuse the same intermediate object for each run through the loop.

Modified:

`ValuesOfMonth` is created

`PushingBalancesCalculator.fillData()`

The screenshot shows a Java Source Compare tool comparing two files: `01FromPushToPull/src/push/PushingBalancesCalculator.java` and `01FromPushToPull/src/pull/PushingBalancesCalculator.java`. The left pane (`Push`) contains the original code with several lines highlighted in blue, indicating they have been moved or modified. The right pane (`Pull`) shows the refactored code where the highlighted lines have been moved into a new class `ValuesOfMonth`.

```

01FromPushToPull/src/push/PushingBalancesCalculator.java
18 }
19
20 @Override
21 public void fillData(List<BalancesOfMonth_API> balancesOfMonthList) {
22     int balance = 0;
23
24     for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) {
25         int ultimo = balancesOfMonth.getDate().getDayOfMonth();
26
27         double averageBalance = 0;
28         int dayOfLatestBalance = 1;
29         List<Transaction_API> transactionsOfMonth = transactionsOfMonth(balancesOfMonth.get
30         for (Transaction_API transaction : transactionsOfMonth) {
31             int day = transaction.getDate().getDayOfMonth();
32             averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, day);
33             balance += transaction.getAmount();
34             dayOfLatestBalance = day;
35         }
36         averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, ultimo);
37
38         balancesOfMonth.setBalance(balance);
39         balancesOfMonth.setAverageBalance((int) averageBalance);
40     }
41 }
42
43 private double calculateProportionalBalance(int dayOfLatestBalance, int balance, int
44     int countingDays = day - dayOfLatestBalance;
45     if (countingDays == 0) {
46
01FromPushToPull/src/pull/PushingBalancesCalculator.java
18 }
19
20 @Override
21 public void fillData(List<BalancesOfMonth_API> balancesOfMonthList) {
22     ValuesOfMonth valuesOfMonth = new ValuesOfMonth();
23     int balance = 0;
24
25     for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) {
26         int ultimo = balancesOfMonth.getDate().getDayOfMonth();
27
28         double averageBalance = 0;
29         int dayOfLatestBalance = 1;
30         List<Transaction_API> transactionsOfMonth = transactionsOfMonth(balancesOfMonth.get
31         for (Transaction_API transaction : transactionsOfMonth) {
32             int day = transaction.getDate().getDayOfMonth();
33             averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, day);
34             balance += transaction.getAmount();
35             dayOfLatestBalance = day;
36
37         averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, ultimo);
38
39         valuesOfMonth.setBalance(balance);
40         valuesOfMonth.setAverageBalance((int) averageBalance);
41
42         balancesOfMonth.setBalance(valuesOfMonth.getBalance());
43         balancesOfMonth.setAverageBalance(valuesOfMonth.getAverageBalance());
44     }
45

```

Step 2: Preparing for method extraction: Purifying the dependencies

Observations:

- We only need the date from the `BalancesOfMonth` object, namely for calculating the current `ultimo`.
- If we extracted the body of the outer loop into a method, this method would depend on our outer world, i.e. `BalancesOfMonth`.
- If we extracted the body of the outer loop into a method, this method would depend on state of our object, namely `transactions` (inside `transactionsOfMonth`).

Both dependencies are undesirable.

Solution:

- Move the transaction filtering to the top of the loop.
- Extract the date from the `BalancesOfMonth` object, move it to the top of the loop and use it in the rest of the method body.

Modified:

`PushingBalancesCalculator.fillData()`

```

Java Source Compare ▾
01FromPushToPull
@Override
public void fillData(List<BalancesOfMonth> balancesOfMonthList)
{
    ValuesOfMonth valuesOfMonth = new ValuesOfMonth();
    int balance = 0;

    for (BalancesOfMonth balancesOfMonth : balancesOfMonthList)
    {
        int ultimo = balancesOfMonth.getDate().getDayOfMonth();

        double averageBalance = 0;
        int dayOfLatestBalance = 1;
        List<Transaction> transactionsOfMonth = transactionsOfMonth(balancesOfMonth.getDate());
        for (Transaction transaction : transactionsOfMonth)
        {
            int day = transaction.getDate().getDayOfMonth();
            averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, day,
                balance += transaction.getAmount();
                dayOfLatestBalance = day;
            }

            averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, ultimo +
                valuesOfMonth.setBalanceAndAverage(balance, averageBalance);

            balancesOfMonth.setBalance(valuesOfMonth.getBalance());
            balancesOfMonth.setAverageBalance(valuesOfMonth.getAverageBalance());
        }
    }

    private double calculateProportionalBalance(int dayOfLatestBalance, int balance, int day, int
}

```

```

02FromPushToPull
public void fillData(List<BalancesOfMonth> balancesOfMonthList)
{
    ValuesOfMonth valuesOfMonth = new ValuesOfMonth();
    int balance = 0;

    for (BalancesOfMonth balancesOfMonth : balancesOfMonthList)
    {
        LocalDate dateOfMonth = balancesOfMonth.getDate();
        List<Transaction> transactionsOfMonth = transactionsOfMonth(dateOfMonth);

        int ultimo = dateOfMonth.getDayOfMonth();

        double averageBalance = 0;
        int dayOfLatestBalance = 1;
        for (Transaction transaction : transactionsOfMonth)
        {
            int day = transaction.getDate().getDayOfMonth();
            averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, day,
                balance += transaction.getAmount();
                dayOfLatestBalance = day;
            }

            averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, ultimo +
                valuesOfMonth.setBalanceAndAverage(balance, averageBalance);

            balancesOfMonth.setBalance(valuesOfMonth.getBalance());
            balancesOfMonth.setAverageBalance(valuesOfMonth.getAverageBalance());
        }
    }

    private double calculateProportionalBalance(int dayOfLatestBalance, int balance, int day,
}

```

Step 3: Separation of Calculation and Iteration

Observations:

- We can now extract the major part of the loop body into a separate method.

Solution:

Let's do it!

Modified:

`PushingBalancesCalculator.fillData()`

`PushingBalancesCalculator.calculateValuesForMonth()` is created

The screenshot shows a Java Source Code Compare tool interface. On the left, the code for '02FromPushToPull/src/push/PushingBalancesCalculator.java' is displayed. On the right, the code for '02FromPushToPull/src/push/PushingBalancesCalculator.java' is shown, which is identical to the left version. The code implements a method to calculate average balance based on transaction dates and amounts. A red box highlights the 'ultimo' variable, indicating a potential bug or inconsistency between the two versions.

```
02FromPushToPull/src/push/PushingBalancesCalculator.java

23     int balance = 0;
24
25     for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) {
26         LocalDate dateOfMonth = balancesOfMonth.getDate();
27         List<Transaction_API> transactionsOfMonth = transactionsOfMonth(dateOfMonth);
28
29         int ultimo = dateOfMonth.getDayOfMonth();
30
31         double averageBalance = 0;
32         int dayOfLatestBalance = 1;
33         for (Transaction_API transaction : transactionsOfMonth) {
34             int day = transaction.getDate().getDayOfMonth();
35             averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, day, ultimo);
36             balance += transaction.getAmount();
37             dayOfLatestBalance = day;
38         }
39         averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, ultimo + 1, ultimo);
40
41         valuesOfMonth.setBalanceAndAverage(balance, averageBalance);
42
43         balancesOfMonth.setBalance(valuesOfMonth.getBalance());
44         balancesOfMonth.setAverageBalance(valuesOfMonth.getAverageBalance());
45     }
46 }
47
48 private double calculateProportionalBalance(int dayOfLatestBalance, int balance, int day, int ultimo) {
49     int countingDays = day - dayOfLatestBalance;
50     if (CountingDays == 0) {
51         return 0;
52     }
53     double rate = (double) countingDays / daysInMonth;
54     return (balance * rate);
55 }

02FromPushToPull/src/push/PushingBalancesCalculator.java

23     int balance = 0;
24
25     for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) {
26         LocalDate dateOfMonth = balancesOfMonth.getDate();
27         List<Transaction_API> transactionsOfMonth = transactionsOfMonth(dateOfMonth);
28
29         balance = calculateValuesForMonth(valuesOfMonth, balance, dateOfMonth, transactions);
30
31         balancesOfMonth.setBalance(valuesOfMonth.getBalance());
32         balancesOfMonth.setAverageBalance(valuesOfMonth.getAverageBalance());
33     }
34
35     private int calculateValuesForMonth(ValuesOfMonth valuesOfMonth, int balance, LocalDate dateOfMonth, List<Transaction_API> transactionsOfMonth) {
36         int ultimo = dateOfMonth.getDayOfMonth();
37
38         double averageBalance = 0;
39         int dayOfLatestBalance = 1;
40         for (Transaction_API transaction : transactionsOfMonth) {
41             int day = transaction.getDate().getDayOfMonth();
42             averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, day, ultimo);
43             balance += transaction.getAmount();
44             dayOfLatestBalance = day;
45         }
46         averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, ultimo + 1, ultimo);
47
48         valuesOfMonth.setBalanceAndAverage(balance, averageBalance);
49
50         valuesOfMonth.setBalance(valuesOfMonth.getBalance());
51         return balance;
52     }
53
54     private double calculateProportionalBalance(int dayOfLatestBalance, int balance, int day, int ultimo) {
55         int countingDays = day - dayOfLatestBalance;
56         if (CountingDays == 0) {
57             return 0;
58         }
59         double rate = (double) countingDays / daysInMonth;
60         return (balance * rate);
61     }
62 }
```

Step 4: Improve Naming and Clean Up Code

Observations:

- The extracted method writes to a parameter
 - The parameter name is rather unspecific

Solution:

Rename the parameter and assign it to a local variable that will then get modified.

Modified:

`PushingBalancesCalculator.calculateValuesForMonth()`

The screenshot shows a Java Source Compare tool interface. It displays two side-by-side code snippets for the class `PushingBalancesCalculator.java`.

Left Side (Original Code):

```
02ZFromPushToPull/src/push/PushingBalancesCalculator.java
31     }
32 }
33 }
34 }
35 }
36 private int calculateValuesForMonth(ValuesOfMonth valuesOfMonth, int balance, LocalDate date) {
37     List<Transaction_API> transactionsOfMonth) {
38     int ultimo = dateOfMonth.getDayOfMonth();
39
40     double averageBalance = 0;
```

Right Side (Modified Code):

```
03FromPushToPull/src/push/PushingBalancesCalculator.java
31     }
32 }
33 }
34 }
35 }
36 private int calculateValuesForMonth(ValuesOfMonth valuesOfMonth, int precedingBalance,
37     List<Transaction_API> transactionsOfMonth) {
38     int balance = precedingBalance;
39     int ultimo = dateOfMonth.getDayOfMonth();
40
41     double averageBalance = 0;
```

The right side shows changes where the variable `balance` is initialized to `precedingBalance` and the `ultimo` variable is removed.

Part II: One object for each loop iteration

Overall Goal:

Build one data object for each loop iteration (instead of reusing the same object over and over again).

Step 5: Separation of concerns (Putting the Code Where it Belongs)

Observations:

- The extracted method uses the intermediate data object as in-out-parameter, so it can read values from the previous iteration (if required) and write the calculation results into the data object.
- The extracted method only operates on this parameter but not on the state of the class it is in. (*Code smell: Feature Envy*)

Solution:

The extracted calculation method can be moved to the class of the intermediate object. (If this is not done via an automated refactoring: This changes the invocation site of the method as well.)

And don't forget to manually move the method calculateProportionalBalance as well! - After that you can (automatically) change the signature of the extracted method to remove the PushingBalancesCalculator parameter.

Attention: Isolate the balance in the method body from the parameter that gets passed in.

Modified:

PushingBalancesCalculator.calculateValuesForMonth() is moved to
ValuesOfMonth.calculateValues()

PushingBalancesCalculator.calculateProportionalBalance() is moved to
ValuesOfMonth.calculateProportionalBalance()

```
03FromPushToPull/src/push/ValuesOfMonth.java
1 package push;
2
3 public class ValuesOfMonth {
4     private int balance;
5     private double averageBalance;
6
7     public int getBalance() {
8         return balance;
9     }
10
11    public int getAverageBalance() {
12        return (int) averageBalance;
13    }
14
15    public void setBalanceAndAverage(int balance, double averageBalance) {
16        this.balance = balance;
17        this.averageBalance = averageBalance;
18    }
19}
20
21
22}
23
24 int calculateValues(int precedingBalance, LocalDate dateOfMonth, List<Transac-
25 int balance = precedingBalance;
26 int ultimo = dateOfMonth.getDayOfMonth();
27
28 double averageBalance = 0;
29 int dayOfLatestBalance = 1;
30 for (Transaction_API transaction : transactionsOfMonth) {
31     int day = transaction.getDate().getDayOfMonth();
32     averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance,
33     balance += transaction.getAmount();
34     dayOfLatestBalance = day;
35 }
36 averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance,
37 setBalanceAndAverage(balance, averageBalance);
38
39 return balance;
40 }
41
42 private double calculateProportionalBalance(int dayOfLatestBalance, int balan-
43 int countingDays = day - dayOfLatestBalance;
44 if (CountingDays == 0) {
45     return 0;
46 }
47 double rate = (double) countingDays / daysInMonth;
48 return (balance * rate);
49 }
50 }
```

PushingBalancesCalculator.fillData() (method invocation is adapted to moved method)

<pre> 03FromPushToPull/src/push/PushingBalancesCalculator.java 27 List<Transaction_API> transactionsOfMonth = transactionsOfMonth(dateOfMonth); 28 29 balance = calculateValuesForMonth(valuesOfMonth, balance, dateOfMonth, transaction); 30 31 balancesOfMonth.setBalance(valuesOfMonth.getBalance()); 32 balancesOfMonth.setAverageBalance(valuesOfMonth.getAverageBalance()); 33 } 34 35 36 private int calculateValuesForMonth(ValuesOfMonth valuesOfMonth, int precedingBalance) { 37 List<Transaction_API> transactionsOfMonth = transactionsOfMonth(dateOfMonth); 38 39 int balance = precedingBalance; 40 int ultimo = dateOfMonth.getDayOfMonth(); 41 42 double averageBalance = 0; 43 int dayOfLatestBalance = 1; 44 for (Transaction_API transaction : transactionsOfMonth) { 45 int day = transaction.getDate().getDayOfMonth(); 46 averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance); 47 balance += transaction.getAmount(); 48 dayOfLatestBalance = day; 49 } 50 averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance); 51 52 valuesOfMonth.setBalanceAndAverage(balance, averageBalance); 53 return balance; 54 } 55 56 private double calculateProportionalBalance(int dayOfLatestBalance, int balance) { 57 int countingDays = day - dayOfLatestBalance; 58 if (countingDays == 0) { 59 return 0; 60 } 61 double rate = (double) countingDays / daysInMonth; 62 return (balance * rate); 63 } 64 65 private List<Transaction_API> transactionsOfMonth(LocalDate date) { 66 List<Transaction_API> results = new ArrayList<Transaction_API>(); 67 for (Transaction_API transaction : transactions) { 68 LocalDate dateOfTransaction = transaction.getDate(); 69 } 70 }</pre>	<pre> 04FromPushToPull/src/push/PushingBalancesCalculator.java 6 import org.joda.time.LocalDate; 7 8 import common.BalancesOfMonthCalculator_API; 9 import common.BalancesOfMonth_API; 10 import common.Transaction_API; 11 12 public class PushingBalancesCalculator implements BalancesOfMonthCalculator { 13 14 private final List<Transaction_API> transactions; 15 16 public PushingBalancesCalculator(List<Transaction_API> transactions) { 17 this.transactions = transactions; 18 } 19 20 @Override 21 public void fillData(List<BalancesOfMonth_API> balancesOfMonthList) { 22 ValuesOfMonth valuesOfMonth = new ValuesOfMonth(); 23 int balance = 0; 24 25 for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) { 26 LocalDate dateOfMonth = balancesOfMonth.getDate(); 27 List<Transaction_API> transactionsOfMonth = transactionsOfMonth(dateOfMonth); 28 29 balance = valuesOfMonth.calculateValues(balance, dateOfMonth, transactionsOfMonth); 30 31 balancesOfMonth.setBalance(valuesOfMonth.getBalance()); 32 balancesOfMonth.setAverageBalance(valuesOfMonth.getAverageBalance()); 33 } 34 35 36 private List<Transaction_API> transactionsOfMonth(LocalDate date) { 37 List<Transaction_API> results = new ArrayList<Transaction_API>(); 38 for (Transaction_API transaction : transactions) { 39 LocalDate dateOfTransaction = transaction.getDate(); 40 if (areSameMonthAndYear(date, dateOfTransaction)) { 41 results.add(transaction); 42 } 43 } 44 return results; 45 } 46 } </pre>
--	---

Step 6: Use New Instance of Intermediate Object for Each Execution of the Inner Loop Body (maximize object usage)

Observations:

- The for loop reuses the same `ValuesOfMonth` object in each loop iteration.
- We have a local variable that transports the balance from one iteration to the next.

Solution:

In this step, we first explicitly pass the preceding balance to the calculation. We have to fetch it from the previous result in order to pass it in.

Secondly, we want each execution of the loop body to create its own intermediate object. Because no values are read from the previous result any more, we can simply create a new object before calling the calculation function.

Modified:

`PushingBalancesCalculator.fillData()`

<pre> 04FromPushToPull/src/push/PushingBalancesCalculator.java 20 @Override 21 public void fillData(List<BalancesOfMonth_API> balancesOfMonthList) { 22 ValuesOfMonth valuesOfMonth = new ValuesOfMonth(); 23 int balance = 0; 24 25 for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) { 26 LocalDate dateOfMonth = balancesOfMonth.getDate(); 27 List<Transaction_API> transactionsOfMonth = transactionsOfMonth(dateOfMonth); 28 29 balance = valuesOfMonth.calculateValues(balance, dateOfMonth, transactionsOfMonth); 30 31 balancesOfMonth.setBalance(valuesOfMonth.getBalance()); 32 balancesOfMonth.setAverageBalance(valuesOfMonth.getAverageBalance()); 33 } 34 35 36 private List<Transaction_API> transactionsOfMonth(LocalDate date) { 37 List<Transaction_API> results = new ArrayList<Transaction_API>(); 38 for (Transaction_API transaction : transactions) { 39 LocalDate dateOfTransaction = transaction.getDate(); 40 if (areSameMonthAndYear(date, dateOfTransaction)) { 41 42 } 43 } 44 } </pre>	<pre> 05FromPushToPull/src/push/PushingBalancesCalculator.java 19 @Override 20 public void fillData(List<BalancesOfMonth_API> balancesOfMonthList) { 21 ValuesOfMonth valuesOfMonth = new ValuesOfMonth(); 22 23 for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) { 24 LocalDate dateOfMonth = balancesOfMonth.getDate(); 25 List<Transaction_API> transactionsOfMonth = transactionsOfMonth(dateOfMonth); 26 27 int precedingBalance = valuesOfMonth.getBalance(); 28 29 valuesOfMonth = new ValuesOfMonth(); 30 valuesOfMonth.calculateValues(dateOfMonth, transactionsOfMonth, precedingBalance); 31 32 balancesOfMonth.setBalance(valuesOfMonth.getBalance()); 33 balancesOfMonth.setAverageBalance(valuesOfMonth.getAverageBalance()); 34 } 35 36 37 private List<Transaction_API> transactionsOfMonth(LocalDate date) { 38 List<Transaction_API> results = new ArrayList<Transaction_API>(); 39 for (Transaction_API transaction : transactions) { 40 LocalDate dateOfTransaction = transaction.getDate(); 41 if (areSameMonthAndYear(date, dateOfTransaction)) { 42 43 } 44 } 45 } </pre>
---	---

`ValuesOfMonth.calculateValues()`

<pre> 04FromPushToPull/src/push/ValuesOfMonth.java 20 this.balance = balance; 21 this.averageBalance = averageBalance; 22 } 23 24 int calculateValues(int precedingBalance, LocalDate dateOfMonth, List<Transaction> 25 int balance = precedingBalance; 26 int ultimo = dateOfMonth.getDayOfMonth(); 27 28 double averageBalance = 0; 29 int dayOfLatestBalance = 1; 30 for (Transaction_API transaction : transactionsOfMonth) { 31 int day = transaction.getDate().getDayOfMonth(); 32 averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, d 33 balance += transaction.getAmount(); 34 dayOfLatestBalance = day; 35 } 36 averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, ult 37 38 setBalanceAndAverage(balance, averageBalance); 39 } 40 41 </pre>	<pre> 05FromPushToPull/src/push/ValuesOfMonth.java 20 this.balance = balance; 21 this.averageBalance = averageBalance; 22 } 23 24 void calculateValues(int precedingBalance, LocalDate dateOfMonth, List<Transactio 25 int balance = precedingBalance; 26 int ultimo = dateOfMonth.getDayOfMonth(); 27 28 double averageBalance = 0; 29 int dayOfLatestBalance = 1; 30 for (Transaction_API transaction : transactionsOfMonth) { 31 int day = transaction.getDate().getDayOfMonth(); 32 averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, b 33 balance += transaction.getAmount(); 34 dayOfLatestBalance = day; 35 } 36 averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, ult 37 38 setBalanceAndAverage(balance, averageBalance); 39 } 40 41 private double calculateProportionalBalance(int dayOfLatestBalance, int balanc </pre>
---	--

Part III: From Calculating the Values Up-Front to Calculating the Values On-Demand

Overall Goal:

Currently, all values are calculated up-front with an explicit command from the outside, no matter whether they are needed or not. It would be much nicer if our object could calculate the values on-demand when (and only when) they are needed.

In order to achieve this, we transform the overall calculation method into individual methods that return each value on demand.

Step 7: Remove the Parameters from the Value Calculation Method

Observations:

- We still need to explicitly invoke the value calculation on the intermediate object.
- We pass all required inputs into the calculation method.

Solution:

The first step to removing the calculation method is to pass the input values into the constructor of the result object, thus turning these inputs into state for the calculating object.

We also introduce a constructor with no arguments. Call the other constructor from it!

Modified:

ValuesOfMonth (fields, constructor, method signature)

<pre>05FromPushToPull/src/push/ValuesOfMonth.java 3 import java.util.List; 4 import org.joda.time.LocalDate; 5 import common.Transaction_API; 6 7 public class ValuesOfMonth { 8 private int balance; 9 private double averageBalance; 10 11 public int getBalance() { 12 return balance; 13 } 14 15 public int getAverageBalance() { 16 return (int) averageBalance; 17 } 18 19 public void setBalanceAndAverage(int balance, double averageBalance) { 20 this.balance = balance; 21 this.averageBalance = averageBalance; 22 } 23 24 void calculateValues(int precedingBalance, LocalDate dateOfMonth, List<Transaction_API> transactionsOfMonth) { 25 int balance = precedingBalance; 26 int ultimo = dateOfMonth.getDayOfMonth(); 27 28 double averageBalance = 0; 29 int dayOfLatestBalance = 1; 30 for (Transaction_API transaction : transactionsOfMonth) { 31 int day = transaction.getDate().getDayOfMonth(); 32 averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, transaction.getAmount()); 33 balance += transaction.getAmount(); 34 } 35 } 36}</pre>	<pre>06FromPushToPull/src/push/ValuesOfMonth.java 10 private double averageBalance; 11 private final LocalDate dateOfMonth; 12 private final List<Transaction_API> transactionsOfMonth; 13 private final int precedingBalance; 14 15 public ValuesOfMonth() { 16 this(0, new LocalDate(), new ArrayList<Transaction_API>()); 17 } 18 19 public ValuesOfMonth(int precedingBalance, LocalDate dateOfMonth, List<Transaction_API> transactionsOfMonth) { 20 this.dateOfMonth = dateOfMonth; 21 this.transactionsOfMonth = transactionsOfMonth; 22 this.precedingBalance = precedingBalance; 23 } 24 25 public int getBalance() { 26 return balance; 27 } 28 29 public int getAverageBalance() { 30 return (int) averageBalance; 31 } 32 33 public void setBalanceAndAverage(int balance, double averageBalance) { 34 this.balance = balance; 35 this.averageBalance = averageBalance; 36 } 37 38 void calculateValues() { 39 int balance = precedingBalance; 40 int ultimo = dateOfMonth.getDayOfMonth(); 41 } 42}</pre>
---	--

PushingBalancesCalculator (method and constructor invocation)

<pre>05FromPushToPull/src/push/PushingBalancesCalculator.java 23 24 for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) { 25 LocalDate dateOfMonth = balancesOfMonth.getDate(); 26 List<Transaction_API> transactionsOfMonth = transactionsOfMonth(dateOfMonth); 27 28 int precedingBalance = balancesOfMonth.getBalance(); 29 30 valuesOfMonth = new ValuesOfMonth(); 31 valuesOfMonth.calculateValues(precedingBalance, dateOfMonth, transactionsOfMonth); 32 33 balancesOfMonth.setBalance(valuesOfMonth.getBalance()); 34 balancesOfMonth.setAverageBalance(valuesOfMonth.getAverageBalance()); 35 } 36}</pre>	<pre>06FromPushToPull/src/push/PushingBalancesCalculator.java 23 24 for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) { 25 LocalDate dateOfMonth = balancesOfMonth.getDate(); 26 List<Transaction_API> transactionsOfMonth = transactionsOfMonth(dateOfMonth); 27 28 int precedingBalance = valuesOfMonth.getBalance(); 29 30 valuesOfMonth = new ValuesOfMonth(precedingBalance, dateOfMonth, transactionsOfMonth); 31 valuesOfMonth.calculateValues(); 32 33 balancesOfMonth.setBalance(valuesOfMonth.getBalance()); 34 balancesOfMonth.setAverageBalance(valuesOfMonth.getAverageBalance()); 35 } 36}</pre>
---	---

Step 8: Make the Value Calculation Implicit

Observations:

- So far, we still calculate all values at once, in the same method.
- This method invokes a setter that writes all values into the intermediate object's attributes in one go (we introduced this setter in Step 1).

Solution:

In this step, we move the value calculations to the getters. As a result, `calculateValues` can be private.

Caution: This is only possible with reentrant (i. e. side-effect-free) methods.

Modified:

`PushingBalancesCalculator.fillData()` (calculation method invocation is removed).

```

07FromPushToPull
@Override
public void fillData(List<BalancesOfMonth> balancesOfMonthList)
{
    ValuesOfMonth valuesOfMonth = new ValuesOfMonth();

    for (BalancesOfMonth balancesOfMonth : balancesOfMonthList)
    {
        LocalDate dateOfMonth = balancesOfMonth.getDate();
        List<Transaction> transactionsOfMonth = transactionsOfMonth(dateOfMonth);

        int precedingBalance = valuesOfMonth.getBalance();

        valuesOfMonth = new ValuesOfMonth(dateOfMonth, transactionsOfMonth, precedingBalance);
        valuesOfMonth.calculateValues();

        balancesOfMonth.setBalance(valuesOfMonth.getBalance());
        balancesOfMonth.setAverageBalance(valuesOfMonth.getAverageBalance());
    }
}

08FromPushToPull
@Override
public void fillData(List<BalancesOfMonth> balancesOfMonthList)
{
    ValuesOfMonth valuesOfMonth = new ValuesOfMonth();

    for (BalancesOfMonth balancesOfMonth : balancesOfMonthList)
    {
        LocalDate dateOfMonth = balancesOfMonth.getDate();
        List<Transaction> transactionsOfMonth = transactionsOfMonth(dateOfMonth);

        int precedingBalance = valuesOfMonth.getBalance();

        valuesOfMonth = new ValuesOfMonth(dateOfMonth, transactionsOfMonth, precedingBalance);

        balancesOfMonth.setBalance(valuesOfMonth.getBalance());
        balancesOfMonth.setAverageBalance(valuesOfMonth.getAverageBalance());
    }
}

```

`ValuesOfMonth.getBalance()` is modified

`ValuesOfMonth.getAverageBalance()` is modified

```

07FromPushToPull
    }

    public int getBalance()
    {
        return balance;
    }

    public int getAverageBalance()
    {
        return (int) averageBalance;
    }

    public void setBalanceAndAverage(int balance, double averageBalance)
    {
        this.balance = balance;
        this.averageBalance = averageBalance;
    }

    void calculateValues()
    {
        int balance = precedingBalance;
        int ultimo = dateOfMonth.getDayOfMonth();

        double averageBalance = 0;

        for (Transaction transaction : transactionsOfMonth)
        {
            balance += transaction.getAmount();
            ultimo++;
        }

        averageBalance = balance / ultimo;
    }
}

08FromPushToPull
    }

    public int getBalance()
    {
        calculateValues();
        return balance;
    }

    public int getAverageBalance()
    {
        calculateValues();
        return (int) averageBalance;
    }

    public void setBalanceAndAverage(int balance, double averageBalance)
    {
        this.balance = balance;
        this.averageBalance = averageBalance;
    }

    private void calculateValues()
    {
        int balance = precedingBalance;
        int ultimo = dateOfMonth.getDayOfMonth();

        for (Transaction transaction : transactionsOfMonth)
        {
            balance += transaction.getAmount();
            ultimo++;
        }

        averageBalance = balance / ultimo;
    }
}

```

Step 9: Inline the Value Calculation Method

Observations:

- The two getters do the calculations, set the fields and return the values.

Solution:

We can now inline and remove the value calculation method and also the `setBalanceAndAverage` setter.

Now we can remove all code from each of the getters that is not related to calculating the value for which the getter is responsible. Each of these getters immediately returns the calculated value, without writing it into a field.

Finally, we can remove the fields for balance and averageBalance.

As a result, we have getters for the individual values of the intermediate object which describe exactly how each value is being calculated.

Modified:

`ValuesOfMonth` (fields, setter and calculate... methods are removed)

08FromPushToFull

```
    this.transactionsOfMonth = transactionsOfMonth;
    this.precedingBalance = precedingBalance;
}

public int getBalance()
{
    calculateValues();
    return balance;
}

public int getAverageBalance()
{
    calculateValues();
    return (int) averageBalance;
}

public void setBalanceAndAverage(int balance, double averageBalance)
{
    this.balance = balance;
    this.averageBalance = averageBalance;
}

private void calculateValues()
{
    int balance = precedingBalance;
    int ultimo = dateOfMonth.getDayOfMonth();

    double averageBalance = 0;
    int dayOfLatestBalance = 1;
    for (Transaction transaction : transactionsOfMonth)
    {
        int day = transaction.getDate().getDayOfMonth();
        averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, day, ultimo);
        balance += transaction.getAmount();
        dayOfLatestBalance = day;
    }

    averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, ultimo + 1, 1);

    setBalanceAndAverage(balance, averageBalance);
}

private double calculateProportionalBalance(int dayOfLatestBalance, int balance, int day, int ultimo)
{
    double rate = ((double) day - dayOfLatestBalance) / daysInMonth;
    return (balance * rate);
}
```

09FromPushToPull

```
    this.precedingBalance = precedingBalance;
}

public int getBalance()
{
    int balance = precedingBalance;
    for (Transaction transaction : transactionsOfMonth)
    {
        balance += transaction.getAmount();
    }
    return balance;
}

public int getAverageBalance()
{
    int balance = precedingBalance;
    int ultimo = dateOfMonth.getDayOfMonth();

    double averageBalance = 0;
    int dayOfLatestBalance = 1;
    for (Transaction transaction : transactionsOfMonth)
    {
        int day = transaction.getDate().getDayOfMonth();
        averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, day, dayOfLatestBalance);
        balance += transaction.getAmount();
        dayOfLatestBalance = day;
    }

    averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, ultimo + 1, 1);

    return (int) averageBalance;
}

private double calculateProportionalBalance(int dayOfLatestBalance, int balance, int day, int ultimo)
{
    int countingDays = day - dayOfLatestBalance;
    if (countingDays == 0)
    {
        return 0;
    }
    double rate = (double) countingDays / daysInMonth;
    return (balance * rate);
}
```

Step 10: Build a chain of ValuesOfMonth objects

Observations:

- We extract the balance of one ValuesOfMonth object and pass that to the next ValuesOfMonth object
 - Our original idea was to build a chain of month objects

Solution:

Each `ValuesOfMonth` object should know its predecessor and ask it directly to obtain the preceding month's balance.

Modified:

PullingBalancesCalculator (directly passes the current `ValuesOfMonth` object to the `ValuesOfMonth` constructor that builds the next object)

Java Source Compare

08FromPushToPull/src/push/PullingBalancesCalculator.java	09_1FromPushToPull/src/push/PullingBalancesCalculator.java
10 List<Transaction_API> transactionsOfMonth = transactionsOfMonth(dateOfMonth);	11 LocalDate dateOfMonth = balancesOfMonth.getDate();
12 List<Transaction_API> transactionsOfMonth = transactionsOfMonth(dateOfMonth);	13 List<Transaction_API> transactionsOfMonth = transactions(dateOfMonth);
14 int precedingBalance = valuesOfMonth.getBalance();	15 valuesOfMonth = new ValuesOfMonth(precedingBalance, dateOfMonth, transactionsOfMonth);
16 valuesOfMonth = new ValuesOfMonth(precedingBalance, dateOfMonth, transactionsOfMonth);	17 valuesOfMonth = new ValuesOfMonth(valuesOfMonth, dateOfMonth, transactionsOfMonth);
18 balancesOfMonth.setBalance(valuesOfMonth.getBalance());	19 balancesOfMonth.setBalance(valuesOfMonth.getBalance());
19 balancesOfMonth.setAverageBalance(valuesOfMonth.getAverageBalance());	20 balancesOfMonth.setAverageBalance(valuesOfMonth.getAverageBalance());
20 }	21 }

`ValuesOfMonth` (now stores the preceding `ValuesOfMonth` object)

```

08FromPushToPull/src/push/ValuesOfMonth.java
11 private final LocalDate dateOfMonth;
12 private final List<Transaction_API> transactionsOfMonth;
13 private final int precedingBalance;
14
15 public ValuesOfMonth() {
16     this(null, new LocalDate(), new ArrayList<Transaction_API>());
17 }
18
19 public ValuesOfMonth(int precedingBalance, LocalDate dateOfMonth, List<Transaction_API> transactionsOfMonth) {
20     this.dateOfMonth = dateOfMonth;
21     this.transactionsOfMonth = transactionsOfMonth;
22     this.precedingBalance = precedingBalance;
23 }
24
25 public int getBalance() {
26     int balance = precedingBalance;
27     for (Transaction_API transaction : transactionsOfMonth) {
28         balance += transaction.getAmount();
29     }
30     return balance;
31 }
32
33 public int getAverageBalance() {
34     int balance = precedingBalance;
35     int ultimo = dateOfMonth.getDayOfMonth();
36
37     double averageBalance = 0;

```



```

09_1FromPushToPull/src/push/ValuesOfMonth.java
11 private final LocalDate dateOfMonth;
12 private final List<Transaction_API> transactionsOfMonth;
13 private final ValuesOfMonth precedingMonth;
14
15 public ValuesOfMonth() {
16     this(null, new LocalDate(), new ArrayList<Transaction_API>());
17 }
18
19 public ValuesOfMonth(ValuesOfMonth precedingMonth, LocalDate dateOfMonth, List<Transaction_API> transactionsOfMonth) {
20     this.dateOfMonth = dateOfMonth;
21     this.transactionsOfMonth = transactionsOfMonth;
22     this.precedingMonth = precedingMonth;
23 }
24
25 public int getBalance() {
26     int balance = precedingMonth == null ? 0 : precedingMonth.getBalance();
27     for (Transaction_API transaction : transactionsOfMonth) {
28         balance += transaction.getAmount();
29     }
30     return balance;
31 }
32
33 public int getAverageBalance() {
34     int balance = precedingMonth == null ? 0 : precedingMonth.getBalance();
35     int ultimo = dateOfMonth.getDayOfMonth();
36
37     double averageBalance = 0;

```

Step 11: Introducing an Initial ValuesOfMonth Object

Observations:

- The preceding ValuesOfMonth object can be null in our code (error-prone, nasty code)
- We already have an initial ValuesOfMonth object that is constructed differently, but it still uses the same code.

Solution:

Introduce an InitialValuesOfMonth object to solve this problem.

Delete the unnecessary ValuesOfMonth constructor.

Modified:

Created IValuesOfMonth (provides the getBalance() method)

```

1 package push;
2
3 public interface IValuesOfMonth {
4
5     int getBalance();
6 }
7
8

```

Created InitialValuesOfMonth (provides the initial value for the balance)

The screenshot shows a Java code editor with the following code:

```
1 package push;
2
3 public class InitialValuesOfMonth implements IValuesOfMonth {
4
5     @Override
6     public int getBalance() {
7         return 0;
8     }
9
10 }
```

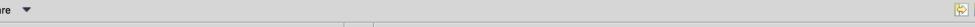
`ValuesOfMonth` (implements the interface, no need for conditionals any more)

The screenshot shows a Java Source Compare tool interface. On the left, the code for file 09_FromPushToPull/src/push/ValuesOfMonth.java is displayed. On the right, the code for file 09_2FromPushToPull/src/push/ValuesOfMonth.java is displayed. The right-hand code is annotated with numerous red and blue highlights and underlines, indicating differences between the two versions. Red highlights are primarily on the first few lines of the right-hand code, while blue annotations are scattered throughout, often appearing as underlines or small boxes around specific code snippets.

```
09_FromPushToPull/src/push/ValuesOfMonth.java
09_2FromPushToPull/src/push/ValuesOfMonth.java
```

```
09_FromPushToPull/src/push/ValuesOfMonth.java
09_2FromPushToPull/src/push/ValuesOfMonth.java
```

`PullingBalancesCalculator` (passes either the preceding `ValuesOfMonth` or an `InitialValue-
sOfMonth` object to the constructor)



The screenshot shows a Java Source Compare tool interface. It has two tabs at the top: '09_1FromPushToPull/src/push/PullingBalancesCalculator.java' on the left and '09_2FromPushToPull/src/push/PullingBalancesCalculator.java' on the right. The left tab's code includes annotations like '@Override' and methods like 'void fillData(List<BalancesOfMonth_API> balancesOfMonthList)'. The right tab's code is identical but contains several red rectangular highlights over specific lines of code, indicating differences between the two versions. The bottom status bar shows the file path 'C:\Users\...'. The toolbar at the top right includes icons for copy, paste, and other operations.

```
09_1FromPushToPull/src/push/PullingBalancesCalculator.java
18 j
19
20 @Override
21 public void fillData(List<BalancesOfMonth_API> balancesOfMonthList) {
22     ValuesOfMonth valuesOfMonth = new ValuesOfMonth();
23
24     for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) {
25         LocalDate dateOfMonth = balancesOfMonth.getDate();
26         List<Transaction_API> transactionsOfMonth = transactionsOfMonth(
27             dateOfMonth);
28         valuesOfMonth = new ValuesOfMonth(valuesOfMonth, dateOfMonth, tr
29         balancesOfMonth.setBalance(valuesOfMonth.getBalance());
30     }
31 }
```

```
09_2FromPushToPull/src/push/PullingBalancesCalculator.java
19
20 @Override
21 public void fillData(List<BalancesOfMonth_API> balancesOfMonthList) {
22     ValuesOfMonth valuesOfMonth = null;
23
24     for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) {
25         LocalDate dateOfMonth = balancesOfMonth.getDate();
26         List<Transaction_API> transactionsOfMonth = transactionsOfMonth(dateOfMonth);
27
28         valuesOfMonth = new ValuesOfMonth(valuesOfMonth == null ? new InitialValuesOfMonth() : valuesOfMonth, dat
29         transactionsOfMonth);
30
31     }
32
33     balancesOfMonth.setBalance(valuesOfMonth.getBalance());
34 }
```

Step 12: Wrap the Transactions

Observations:

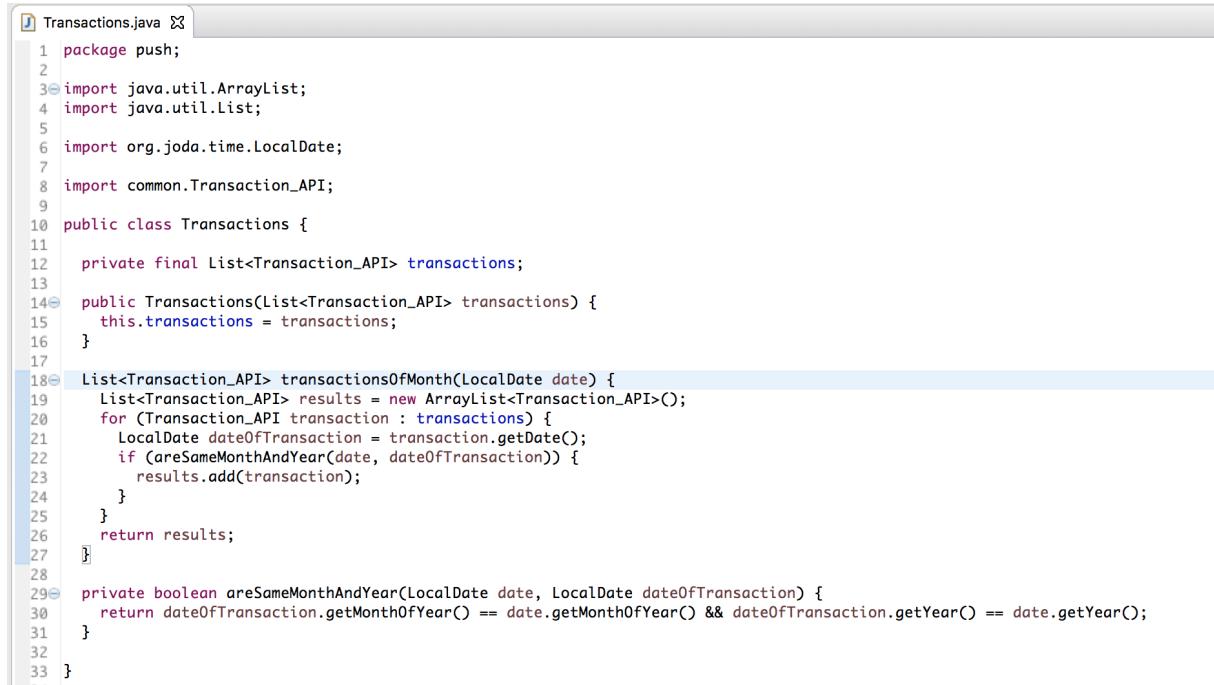
- Transactions API is directly used in the PullingBalancesCalculator

Solution:

Create a `Transactions` object that is wrapped around the list of transactions.

Modified:

Created Transactions (provides the filtering)

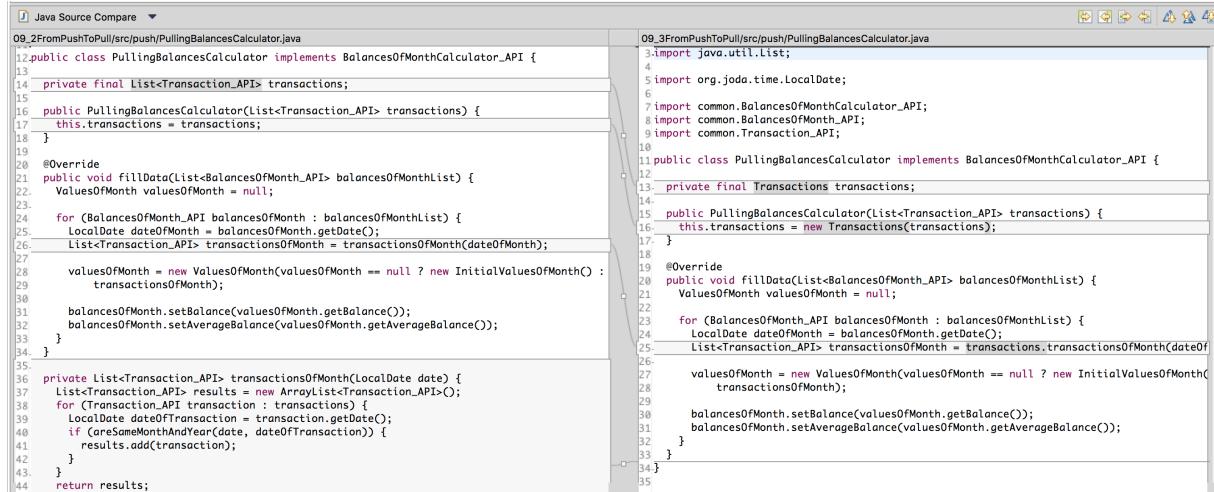


```

Transactions.java
1 package push;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 import org.joda.time.LocalDate;
7
8 import common.Transaction_API;
9
10 public class Transactions {
11
12     private final List<Transaction_API> transactions;
13
14     public Transactions(List<Transaction_API> transactions) {
15         this.transactions = transactions;
16     }
17
18     List<Transaction_API> transactionsOfMonth(LocalDate date) {
19         List<Transaction_API> results = new ArrayList<Transaction_API>();
20         for (Transaction_API transaction : transactions) {
21             LocalDate dateOfTransaction = transaction.getDate();
22             if (areSameMonthAndYear(date, dateOfTransaction)) {
23                 results.add(transaction);
24             }
25         }
26         return results;
27     }
28
29     private boolean areSameMonthAndYear(LocalDate date, LocalDate dateOfTransaction) {
30         return dateOfTransaction.getMonthOfYear() == date.getMonthOfYear() && dateOfTransaction.getYear() == date.getYear();
31     }
32
33 }

```

PullingBalancesCalculator (uses the Transactions object)



```

09_2FromPushToPull/src/push/PullingBalancesCalculator.java
9 package push;
10
11 public class PullingBalancesCalculator implements BalancesOfMonthCalculator_API {
12
13     private final List<Transaction_API> transactions;
14
15     public PullingBalancesCalculator(List<Transaction_API> transactions) {
16         this.transactions = transactions;
17     }
18
19     @Override
20     public void fillData(List<BalancesOfMonth_API> balancesOfMonthList) {
21         ValuesOfMonth valuesOfMonth = null;
22
23         for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) {
24             LocalDate dateOfMonth = balancesOfMonth.getDate();
25             List<Transaction_API> transactionsOfMonth = transactionsOfMonth(dateOfMonth);
26
27             valuesOfMonth = new ValuesOfMonth(valuesOfMonth == null ? new InitialValuesOfMonth()
28                                             : transactionsOfMonth);
29
30             balancesOfMonth.setBalance(valuesOfMonth.getBalance());
31             balancesOfMonth.setAverageBalance(valuesOfMonth.getAverageBalance());
32         }
33     }
34
35     private List<Transaction_API> transactionsOfMonth(LocalDate date) {
36         List<Transaction_API> results = new ArrayList<Transaction_API>();
37         for (Transaction_API transaction : transactions) {
38             LocalDate dateOfTransaction = transaction.getDate();
39             if (areSameMonthAndYear(date, dateOfTransaction)) {
40                 results.add(transaction);
41             }
42         }
43     }
44     return results;
45 }

```

```

09_3FromPushToPull/src/push/PullingBalancesCalculator.java
3 import java.util.List;
4
5 import org.joda.time.LocalDate;
6
7 import common.BalancesOfMonthCalculator_API;
8 import common.BalancesOfMonth_API;
9 import common.Transaction_API;
10
11 public class PullingBalancesCalculator implements BalancesOfMonthCalculator_API {
12
13     private final Transactions transactions;
14
15     public PullingBalancesCalculator(List<Transaction_API> transactions) {
16         this.transactions = new Transactions(transactions);
17     }
18
19     @Override
20     public void fillData(List<BalancesOfMonth_API> balancesOfMonthList) {
21         ValuesOfMonth valuesOfMonth = null;
22
23         for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) {
24             LocalDate dateOfMonth = balancesOfMonth.getDate();
25             List<Transaction_API> transactionsOfMonth = transactions.transactionsOfMonth(dateOfMonth);
26
27             valuesOfMonth = new ValuesOfMonth(valuesOfMonth == null ? new InitialValuesOfMonth()
28                                             : transactionsOfMonth);
29
30             balancesOfMonth.setBalance(valuesOfMonth.getBalance());
31             balancesOfMonth.setAverageBalance(valuesOfMonth.getAverageBalance());
32         }
33     }
34 }

```

Step 13: Separate Month Creation and Balances Filling

Observations:

- We create the months at the same time as we fill the balances, so we cannot reuse the months.

Solution:

Pull out the month creation and store the months in a map for random access.

Modified:

PullingBalancesCalculator (separates the creation and the usage of the ValuesOfMonth objects)

```

09_3FromPushToPull/src/push/PullingBalancesCalculator.java
09 import common.Transaction_API;
10
11 public class PullingBalancesCalculator implements BalancesOfMonthCalculator_API {
12     private final Transactions transactions;
13
14     public PullingBalancesCalculator(List<Transaction_API> transactions) {
15         this.transactions = new Transactions(transactions);
16     }
17
18     @Override
19     public void fillData(List<BalancesOfMonth_API> balancesOfMonthList) {
20         ValuesOfMonth valuesOfMonth = null;
21
22         for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) {
23             LocalDate dateOfMonth = balancesOfMonth.getDate();
24             List<Transaction_API> transactionsOfMonth = transactions.transactionsOfMonth(dateOfMonth);
25
26             valuesOfMonth = new ValuesOfMonth(valuesOfMonth == null ? new InitialValuesOfMonth() : transactionsOfMonth);
27
28             balancesOfMonth.setBalance(valuesOfMonth.getBalance());
29             balancesOfMonth.setAverageBalance(valuesOfMonth.getAverageBalance());
30         }
31     }
32 }
33

09_4FromPushToPull/src/push/PullingBalancesCalculator.java
16
17     private final Map<LocalDate, ValuesOfMonth> months = new HashMap<LocalDate, ValuesOfMonth>();
18
19     public PullingBalancesCalculator(List<Transaction_API> transactions) {
20         this.transactions = new Transactions(transactions);
21     }
22
23     @Override
24     public void fillData(List<BalancesOfMonth_API> balancesOfMonthList) {
25         ValuesOfMonth valuesOfMonth = null;
26         for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) {
27             LocalDate dateOfMonth = balancesOfMonth.getDate();
28             List<Transaction_API> transactionsOfMonth = transactions.transactionsOfMonth(dateOfMonth);
29
30             valuesOfMonth = new ValuesOfMonth(valuesOfMonth == null ? new InitialValuesOfMonth() : transactionsOfMonth);
31             months.put(dateOfMonth, valuesOfMonth);
32         }
33
34         for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) {
35             LocalDate dateOfMonth = balancesOfMonth.getDate();
36             balancesOfMonth.setBalance(months.get(dateOfMonth).getBalance());
37             balancesOfMonth.setAverageBalance(months.get(dateOfMonth).getAverageBalance());
38         }
39     }
40 }
41

```

Step 14: Extract Months Map Into an Aggregate

Observations:

- The months map and its code don't belong in the PullingBalancesCalculator but in their own object

Solution:

Pull out the month map and wrap it into its own object

Modified:

Created Months (builds up the chain of ValuesOfMonth objects)

```

Months.java
1 package push;
2
3 import java.util.HashMap;
4
5 public class Months {
6
7     private final Map<LocalDate, ValuesOfMonth> months = new HashMap<LocalDate, ValuesOfMonth>();
8
9     public Months(List<BalancesOfMonth_API> balancesOfMonthList, Transactions transactions) {
10         ValuesOfMonth valuesOfMonth = null;
11         for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) {
12             LocalDate dateOfMonth = balancesOfMonth.getDate();
13             List<Transaction_API> transactionsOfMonth = transactions.transactionsOfMonth(dateOfMonth);
14
15             valuesOfMonth = new ValuesOfMonth(valuesOfMonth == null ? new InitialValuesOfMonth() : transactionsOfMonth);
16             months.put(dateOfMonth, valuesOfMonth);
17         }
18     }
19
20     public ValuesOfMonth monthFor(LocalDate dateOfMonth) {
21         return months.get(dateOfMonth);
22     }
23 }
24
25

```

PullingBalancesCalculator (uses the Months object)

```

09_5FromPushToPull/src/push/PullingBalancesCalculator.java
14
15     private final Transactions transactions;
16
17     private final Map<LocalDate, ValuesOfMonth> months = new HashMap<LocalDate, ValuesOfMonth>();
18
19     public PullingBalancesCalculator(List<Transaction_API> transactions) {
20         this.transactions = new Transactions(transactions);
21     }
22
23     @Override
24     public void fillData(List<BalancesOfMonth_API> balancesOfMonthList) {
25         ValuesOfMonth valuesOfMonth = null;
26         for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) {
27             LocalDate dateOfMonth = balancesOfMonth.getDate();
28             List<Transaction_API> transactionsOfMonth = transactions.transactionsOfMonth(dateOfMonth);
29
30             valuesOfMonth = new ValuesOfMonth(valuesOfMonth == null ? new InitialValuesOfMonth() : transactionsOfMonth);
31             months.put(dateOfMonth, valuesOfMonth);
32         }
33     }
34
35     for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) {
36         LocalDate dateOfMonth = balancesOfMonth.getDate();
37
38         balancesOfMonth.setBalance(months.get(dateOfMonth).getBalance());
39         balancesOfMonth.setAverageBalance(months.get(dateOfMonth).getAverageBalance());
40     }
41 }

09_6FromPushToPull/src/push/PullingBalancesCalculator.java
1 package push;
2
3 import java.util.List;
4
5 import common.BalancesOfMonthCalculator_API;
6 import common.BalancesOfMonth_API;
7 import common.Transaction_API;
8
9 public class PullingBalancesCalculator implements BalancesOfMonthCalculator_API {
10
11     private final Transactions transactions;
12
13     public PullingBalancesCalculator(List<Transaction_API> transactions) {
14         this.transactions = new Transactions(transactions);
15     }
16
17     @Override
18     public void fillData(List<BalancesOfMonth_API> balancesOfMonthList) {
19         Months months = new Months(balancesOfMonthList, transactions);
20
21         for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) {
22             ValuesOfMonth currentMonth = months.monthFor(balancesOfMonth.getDate());
23             balancesOfMonth.setBalance(currentMonth.getBalance());
24             balancesOfMonth.setAverageBalance(currentMonth.getAverageBalance());
25         }
26     }
27 }

```

Step 15: Caching

Observations:

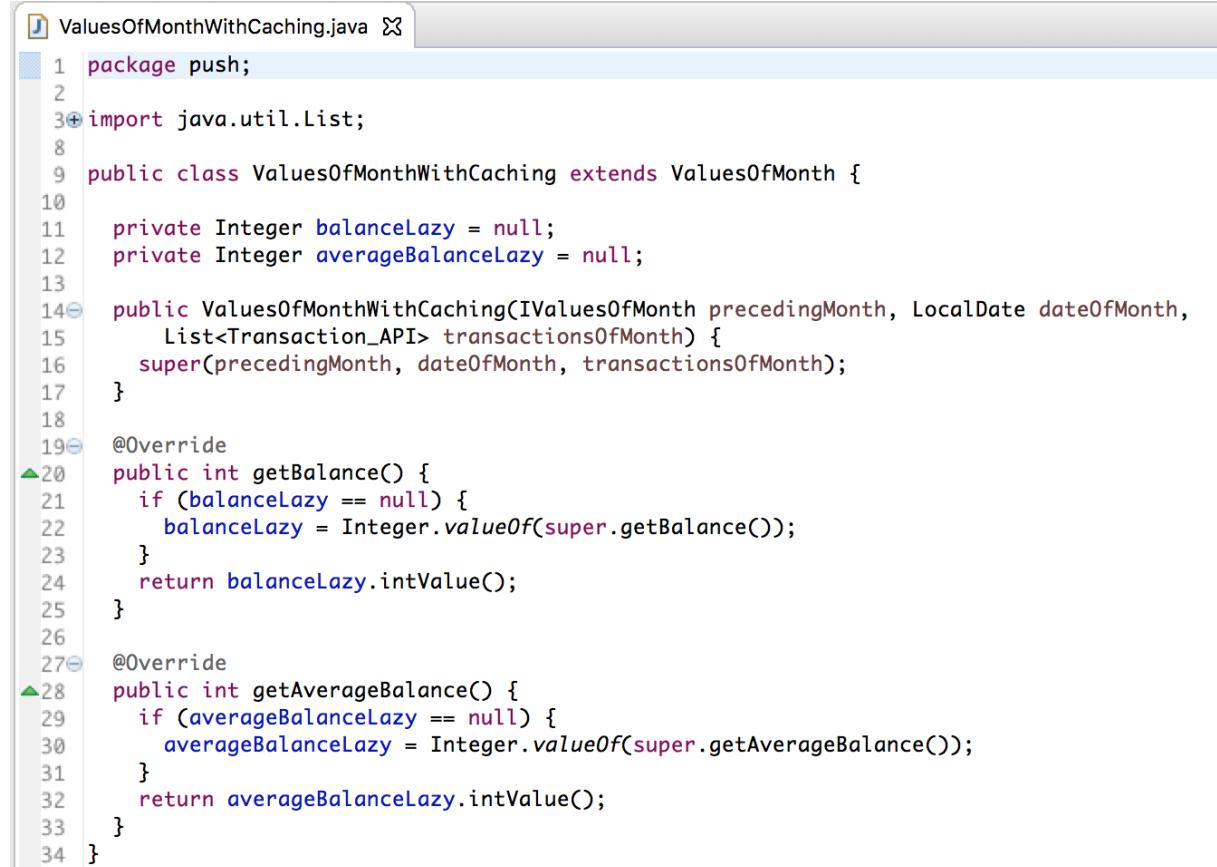
- For each value, the values of all preceding months have to be calculated

Solution:

Introduce caching.

Modified:

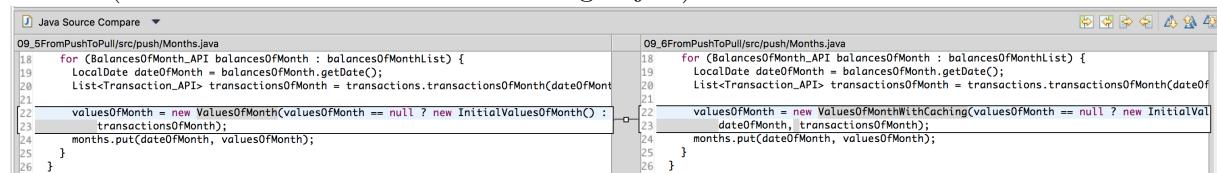
Created `ValuesOfMonthWithCaching` (provides caching for each method)
 (hint: also possible with delegation)



```

1 package push;
2
3 import java.util.List;
4
5 public class ValuesOfMonthWithCaching extends ValuesOfMonth {
6
7     private Integer balanceLazy = null;
8     private Integer averageBalanceLazy = null;
9
10    public ValuesOfMonthWithCaching(IValuesOfMonth precedingMonth, LocalDate dateOfMonth,
11        List<Transaction_API> transactionsOfMonth) {
12        super(precedingMonth, dateOfMonth, transactionsOfMonth);
13    }
14
15    @Override
16    public int getBalance() {
17        if (balanceLazy == null) {
18            balanceLazy = Integer.valueOf(super.getBalance());
19        }
20        return balanceLazy.intValue();
21    }
22
23    @Override
24    public int getAverageBalance() {
25        if (averageBalanceLazy == null) {
26            averageBalanceLazy = Integer.valueOf(super.getAverageBalance());
27        }
28        return averageBalanceLazy.intValue();
29    }
30
31 }
32
33 }
34 }
```

`Months` (uses the `ValuesOfMonthWithCaching` object)



09_5FromPushToPull/src/push/Months.java	09_6FromPushToPull/src/push/Months.java
<pre> 18 for (BalancesOfMonth API balancesOfMonth : balancesOfMonthList) { 19 LocalDate dateOfMonth = balancesOfMonth.getDate(); 20 List<Transaction_API> transactionsOfMonth = transactions.transactionsOfMonth(dateOfMonth); 21 22 valuesOfMonth = new ValuesOfMonth(valuesOfMonth == null ? new InitialValuesOfMonth() : 23 transactionsOfMonth); 24 months.put(dateOfMonth, valuesOfMonth); 25 } 26 }</pre>	<pre> 18 for (BalancesOfMonth API balancesOfMonth : balancesOfMonthList) { 19 LocalDate dateOfMonth = balancesOfMonth.getDate(); 20 List<Transaction_API> transactionsOfMonth = transactions.transactionsOfMonth(dateOfMonth); 21 22 valuesOfMonth = new ValuesOfMonthWithCaching(valuesOfMonth == null ? new InitialValuesOfMonth() : 23 dateOfMonth, transactionsOfMonth); 24 months.put(dateOfMonth, valuesOfMonth); 25 } 26 }</pre>

Step 16: Simplification

Observations:

- The calculation of the average balance is quite complicated.

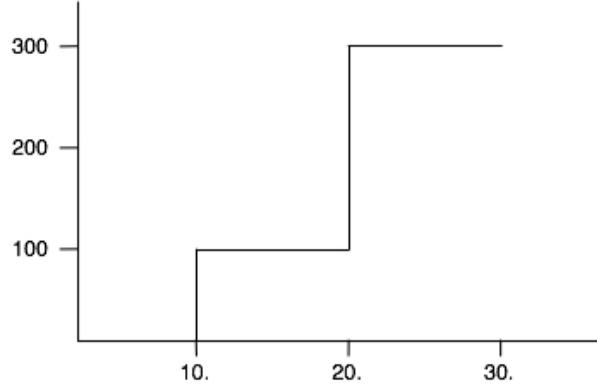
- The calculation of the average balance is isolated and easily testable.

Now that we can see each value calculation in isolation, we often notice that we can perform simplifications of the algorithms.

Solution:

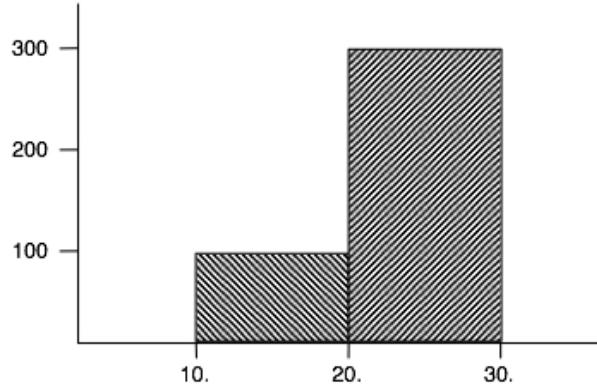
We can simplify the algorithm or even rewrite it (with TDD if we like). It is now easy to clarify the business aspects of this logic and to develop a new algorithm.

We demonstrate the change of algorithms with an example. Suppose we have a balance like this:



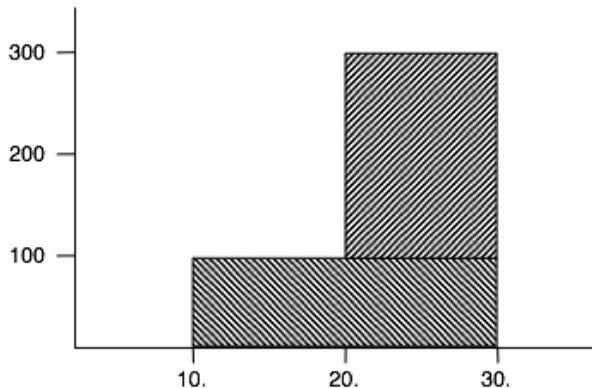
i.e. there is a transaction of 100 on the 10th and a transaction of 200 on the 20th to an initially empty account. We want to determine the average, which is the area under the graph divided by the number of days in the month.

The existing algorithm determines the average like this:



Whenever a transaction occurs, the algorithm calculates the area under the graph for the number of days between the previous transaction (or the start of the month) and the current transaction and divides it by the number of days that are between these two dates. The disadvantage is that we have many special cases (beginning and end of month) and that we need to keep track of the current balance and of when the previous transaction happened.

We prefer to implement the average like this:



Whenever a transaction occurs, this algorithm calculates its average up to the end of the month. This algorithm is much easier to implement because the average of each transaction can be determined in isolation.

The proof that this algorithm works correctly for transactions with negative amounts is left as an exercise for the reader.

Modified:

```
ValuesOfMonth.getAverageBalance()
```

`ValuesOfMonth.calculateProportionalBalance()` is removed

`ValuesOfMonth.rateOf()` is created

The screenshot shows a Java Source Compare tool interface with two panes. The left pane displays the code for `09_6FromPushToPull/src/push/ValuesOfMonth.java`, which contains logic for calculating average balance based on preceding month's balance and transactions for the current month. The right pane displays the code for `16FromPushToPull/src/pushpull/ValuesOfMonth.java`, which is similar but includes additional methods like `calculateProportionalBalance` and `daysInMonth`. Both snippets use the `Transaction_API` interface.

```
09_6FromPushToPull/src/push/ValuesOfMonth.java
16FromPushToPull/src/pushpull/ValuesOfMonth.java
```

The Final Situation

Observations:

- Calculation logic and result structure are now identical
 - We now have intelligent objects instead of stupid data records
 - The algorithms for each value are isolated and clearly visible

What is Missing?

We could also

- replace the LocalDate by YearMonth to indicate that the day is irrelevant

- Wrap the list of Transaction_API to further isolate it from our code (anticorruption layer)
- Wrap the list of BalancesOfMonth_API to further isolate it from our code (anticorruption layer)