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:

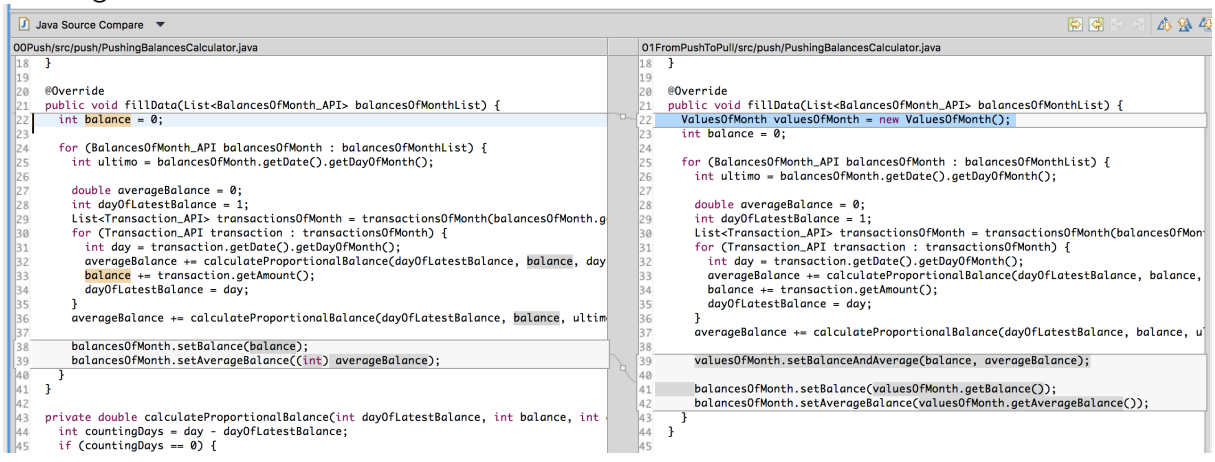
- The outer loop iterates over the list of result objects that need to be filled with the calculated data.
- The method calculates two values for each result object: `balance` and `averageBalance`.
- These two values cannot both be returned from a single method.

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()`



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`).

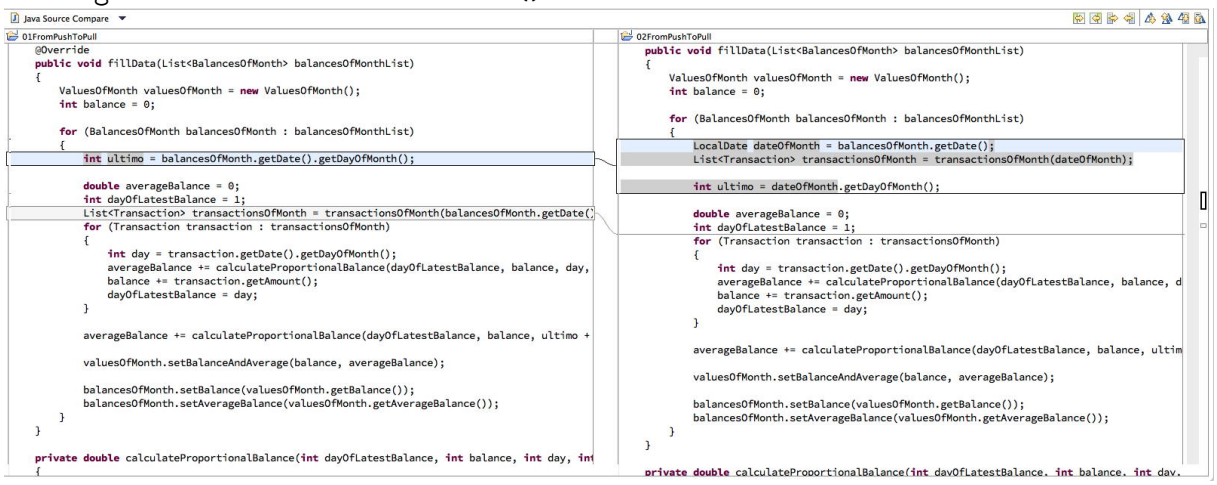
Both dependencies are undesirable.

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()`



Step 3: Separation of Calculation and Iteration

Observations:

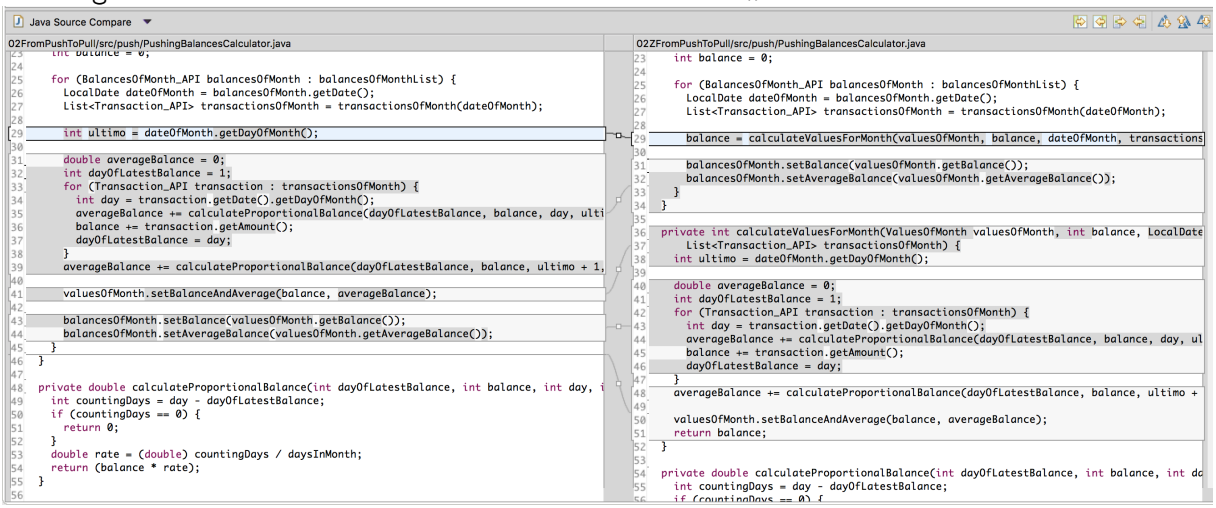
- We can now extract the major part of the loop body into a separate method.

Solution:

Let's do it!

Modified:

`PushingBalancesCalculator.fillData()`
`PushingBalancesCalculator.calculateValuesForMonth()` is created



Step 4: Improve Naming and Clean Up Code

Observations:

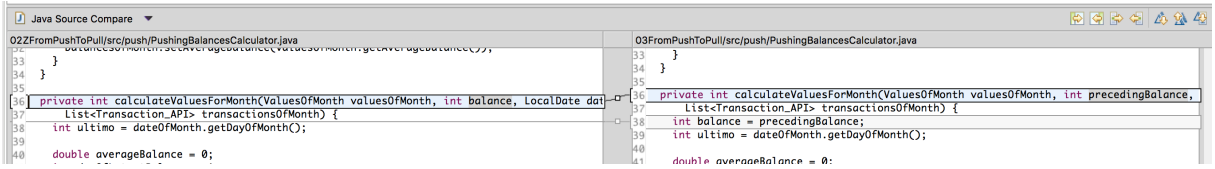
- The extracted method writes to a parameter
- The parameter name is rather unspecific

Solution:

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

Modified:

`PushingBalancesCalculator.calculateValuesForMonth()`



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()`

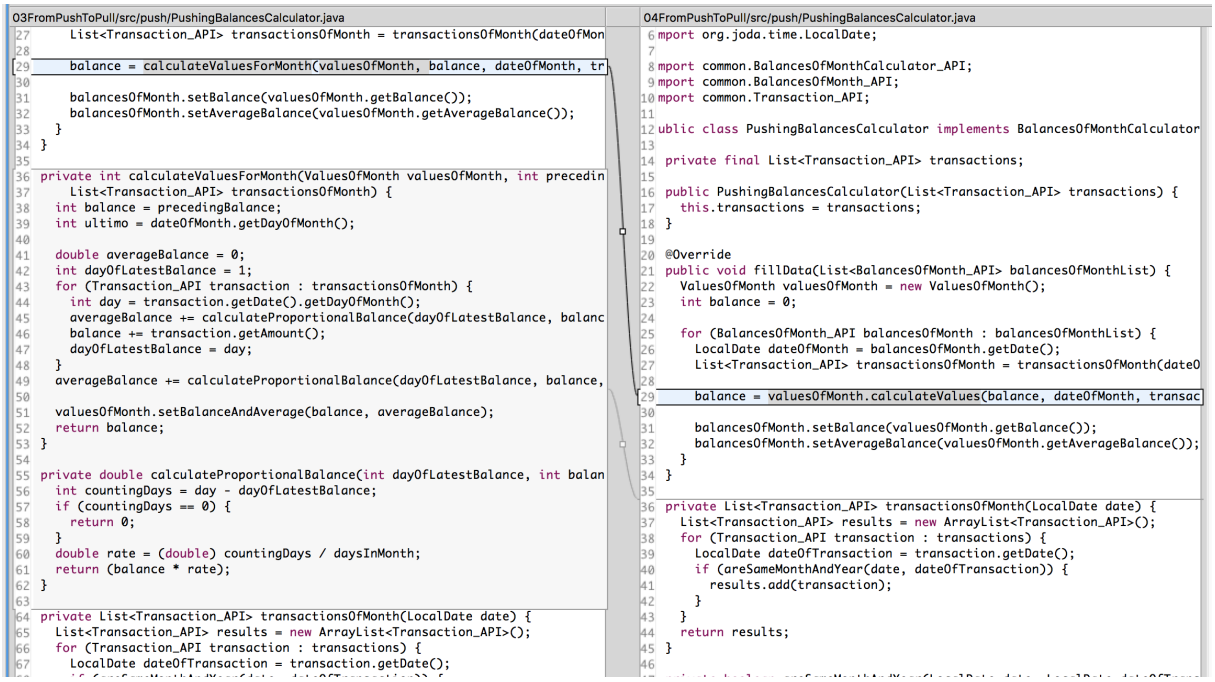
03FromPushToPull/src/push/ValuesOfMonth.java

```java
package push;

public class ValuesOfMonth {
  private int balance;
  private double averageBalance;

  public int getBalance() {
    return balance;
  }

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

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

04FromPushToPull/src/push/ValuesOfMonth.java

```java
    this.balance = balance;
    this.averageBalance = averageBalance;
  }

  int calculateValues(int precedingBalance, LocalDate dateOfMonth, List<Transac
    int balance = precedingBalance;
    int ultimo = dateOfMonth.getDayOfMonth();

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

    setBalanceAndAverage(balance, averageBalance);
    return balance;
  }

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

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



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

Observations:

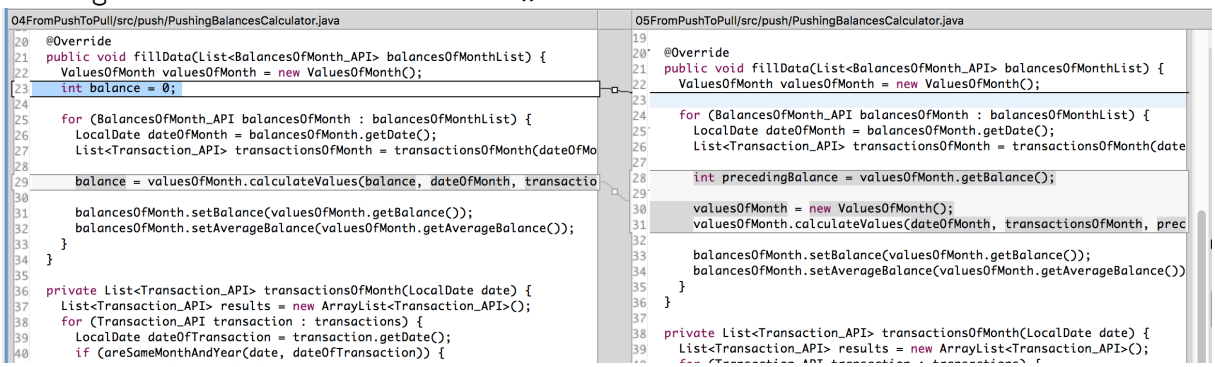
- The for loop reuses the same `ValuesOfMonth` object in each loop iteration.
- We have a local variable that transports the balance from one iteration to the next.

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()`



`ValuesOfMonth.calculateValues()`

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      return balance;
40    }
41
```

05FromPushToPull/src/push/ValuesOfMonth.java

```
20      this.balance = balance;
21      this.averageBalance = averageBalance;
22    }
23
24    void calculateValues(int precedingBalance, LocalDate dateOfMonth, List<Transa
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, balanc
33        balance += transaction.getAmount();
34        dayOfLatestBalance = day;
35      }
36      averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance,
37
38      setBalanceAndAverage(balance, averageBalance);
39    }
40
41    private double calculateProportionalBalance(int dayOfLatestBalance, int balan
```

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)

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<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, d
33       balance += transaction.getAmount();
```

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<Transa
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();
```

`PushingBalancesCalculator` (method and constructor invocation)

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 = valuesOfMonth.getBalance();
29
30     valuesOfMonth = new ValuesOfMonth();
31     valuesOfMonth.calculateValues(precedingBalance, dateOfMonth, transactionsOfMon
32
33     balancesOfMonth.setBalance(valuesOfMonth.getBalance());
34     balancesOfMonth.setAverageBalance(valuesOfMonth.getAverageBalance());
35   }
36 }
```

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, transacti
31     valuesOfMonth.calculateValues();
32
33     balancesOfMonth.setBalance(valuesOfMonth.getBalance());
34     balancesOfMonth.setAverageBalance(valuesOfMonth.getAverageBalance());
35   }
36 }
```

Step 8: Make the Value Calculation Implicit

Observations:

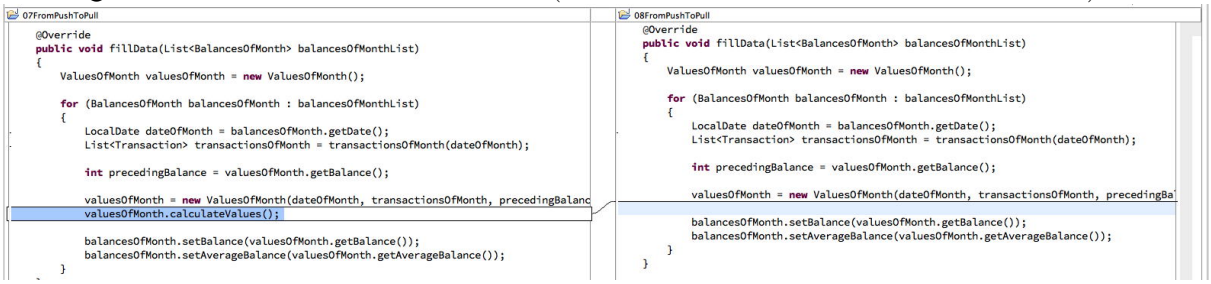
- So far, we still calculate all values at once, in the same method.
- This method invokes a setter that writes all values into the intermediate object's attributes in one go (we introduced this setter in Step 1).

Solution:

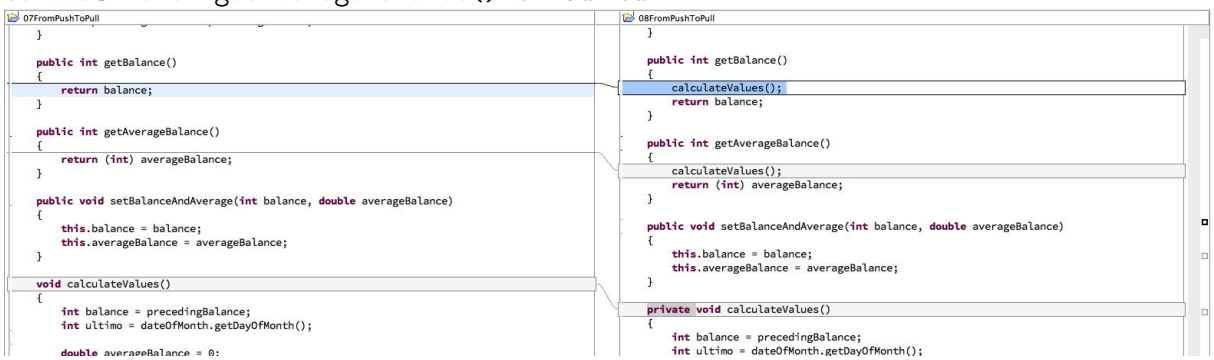
In this step, we move the value calculations to the getters. As a result, `calculateValues` can be private.
Caution: This is only possible with reentrant (i. e. side-effect-free) methods.

Modified:

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



`ValuesOfMonth.getBalance()` is modified
`ValuesOfMonth.getAverageBalance()` is modified



Step 9: Inline the Value Calculation Method

Observations:

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

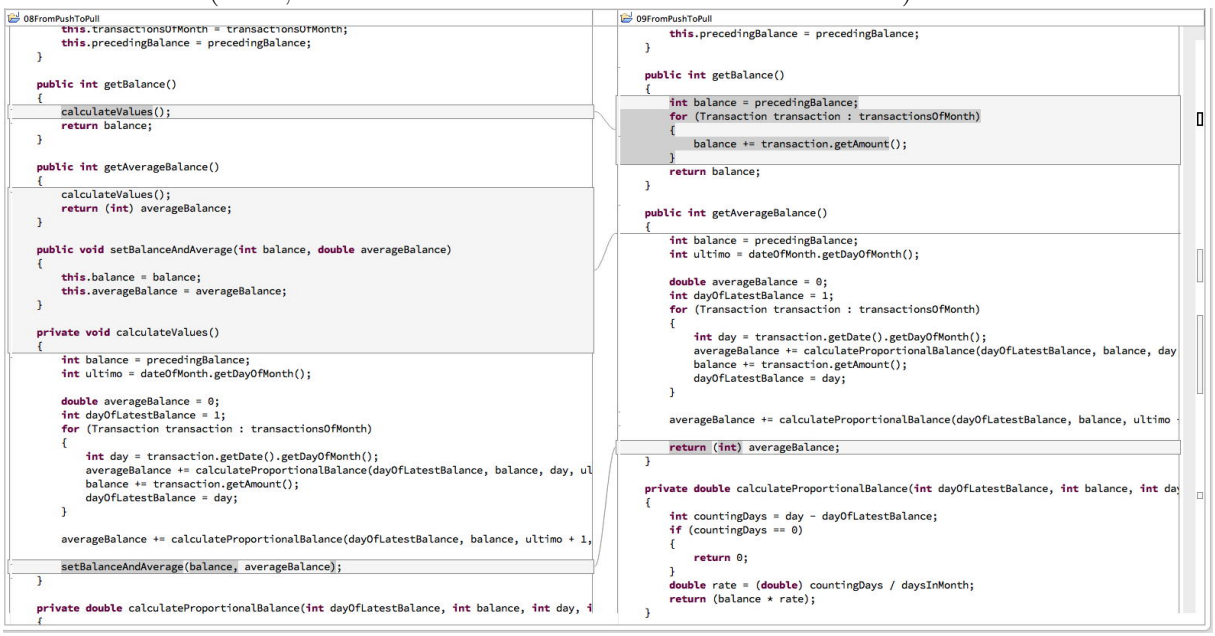
Solution:

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)



Step 10: Build a chain of ValuesOfMonth objects

Observations:

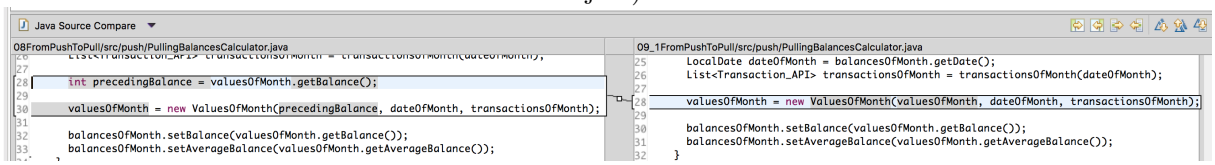
- We extract the balance of one ValuesOfMonth object and pass that to the next ValuesOfMonth object
- Our original idea was to build a chain of month objects

Solution:

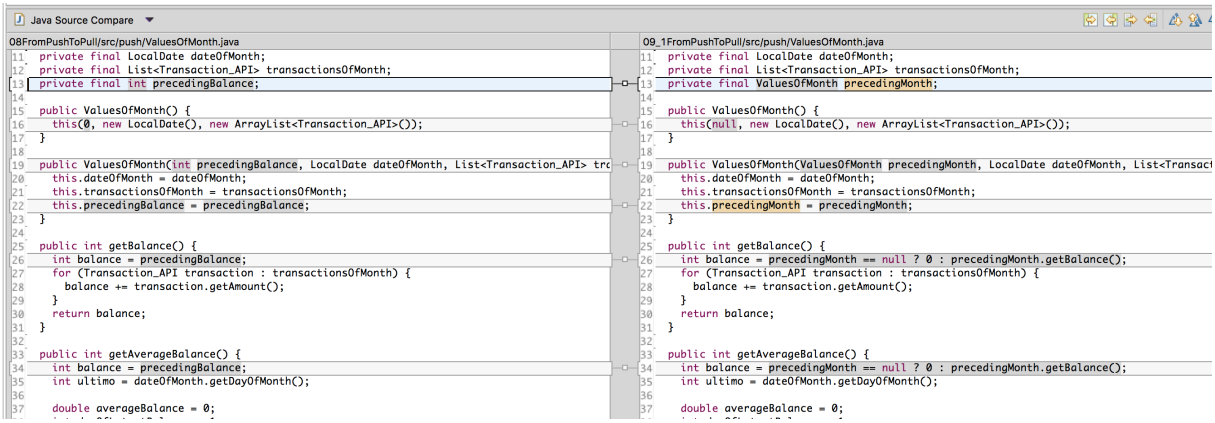
Each ValuesOfMonth object should know its predecessor and ask it directly to obtain the preceding month's balance.

Modified:

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



`ValuesOfMonth` (now stores the preceding ValuesOfMonth object)



Step 11: Introducing an Initial ValuesOfMonth Object

Observations:

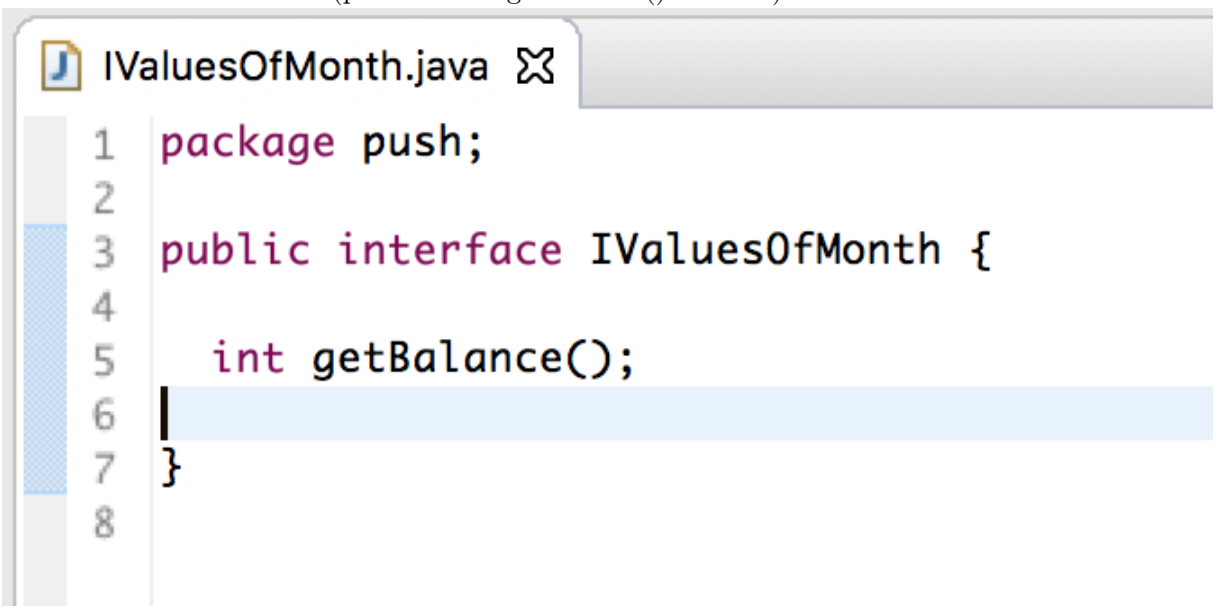
- The preceding ValuesOfMonth object can be null in our code (error-prone, nasty code)
- We already have an initial ValuesOfMonth object that is constructed differently, but it still uses the same code.

Solution:

Introduce an InitialValuesOfMonth object to solve this problem.
Delete the unnecessary ValuesOfMonth constructor.

Modified:

Created `IValuesOfMonth` (provides the getBalance() method)

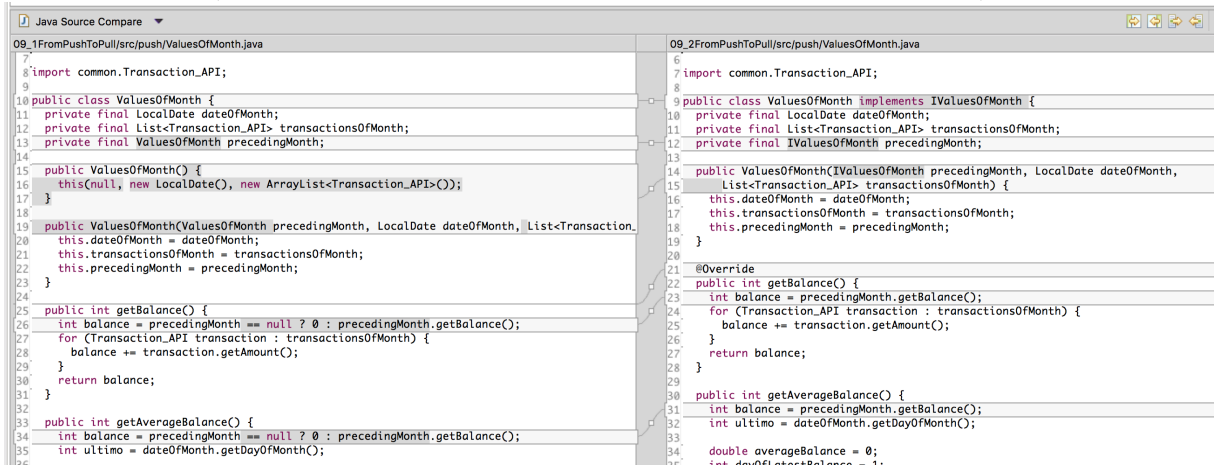


Created `InitialValuesOfMonth` (provides the initial value for the balance)

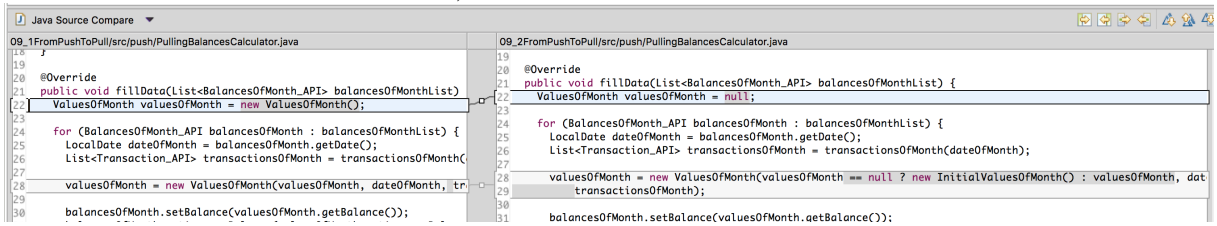
InitialValuesOfMonth.java

```
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)



`PullingBalancesCalculator` (passes either the preceding ValuesOfMonth or an InitialValuesOfMonth object to the constructor)



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

```java
package push;

import java.util.ArrayList;
import java.util.List;

import org.joda.time.LocalDate;

import common.Transaction_API;

public class Transactions {

  private final List<Transaction_API> transactions;

  public Transactions(List<Transaction_API> transactions) {
    this.transactions = transactions;
  }

  List<Transaction_API> transactionsOfMonth(LocalDate date) {
    List<Transaction_API> results = new ArrayList<Transaction_API>();
    for (Transaction_API transaction : transactions) {
      LocalDate dateOfTransaction = transaction.getDate();
      if (areSameMonthAndYear(date, dateOfTransaction)) {
        results.add(transaction);
      }
    }
    return results;
  }

  private boolean areSameMonthAndYear(LocalDate date, LocalDate dateOfTransaction) {
    return dateOfTransaction.getMonthOfYear() == date.getMonthOfYear() && dateOfTransaction.getYear() == date.getYear();
  }

}
```

`PullingBalancesCalculator` (uses the Transactions object)

Java Source Compare

09_2FromPushToPull/src/push/PullingBalancesCalculator.java

```java
public class PullingBalancesCalculator implements BalancesOfMonthCalculator_API {

  private final List<Transaction_API> transactions;

  public PullingBalancesCalculator(List<Transaction_API> transactions) {
    this.transactions = transactions;
  }

  @Override
  public void fillData(List<BalancesOfMonth_API> balancesOfMonthList) {
    ValuesOfMonth valuesOfMonth = null;

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

      valuesOfMonth = new ValuesOfMonth(valuesOfMonth == null ? new InitialValuesOfMonth() :
          transactionsOfMonth);

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

  private List<Transaction_API> transactionsOfMonth(LocalDate date) {
    List<Transaction_API> results = new ArrayList<Transaction_API>();
    for (Transaction_API transaction : transactions) {
      LocalDate dateOfTransaction = transaction.getDate();
      if (areSameMonthAndYear(date, dateOfTransaction)) {
        results.add(transaction);
      }
    }
    return results;
```

09_3FromPushToPull/src/push/PullingBalancesCalculator.java

```java
import java.util.List;

import org.joda.time.LocalDate;

import common.BalancesOfMonthCalculator_API;
import common.BalancesOfMonth_API;
import common.Transaction_API;

public class PullingBalancesCalculator implements BalancesOfMonthCalculator_API {

  private final Transactions transactions;

  public PullingBalancesCalculator(List<Transaction_API> transactions) {
    this.transactions = new Transactions(transactions);
  }

  @Override
  public void fillData(List<BalancesOfMonth_API> balancesOfMonthList) {
    ValuesOfMonth valuesOfMonth = null;

    for (BalancesOfMonth_API balancesOfMonth : balancesOfMonthList) {
      LocalDate dateOfMonth = balancesOfMonth.getDate();
      List<Transaction_API> transactionsOfMonth = transactions.transactionsOfMonth(dateOf

      valuesOfMonth = new ValuesOfMonth(valuesOfMonth == null ? new InitialValuesOfMonth(
          transactionsOfMonth);

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

Step 13: Separate Month Creation and Balances Filling

Observations:

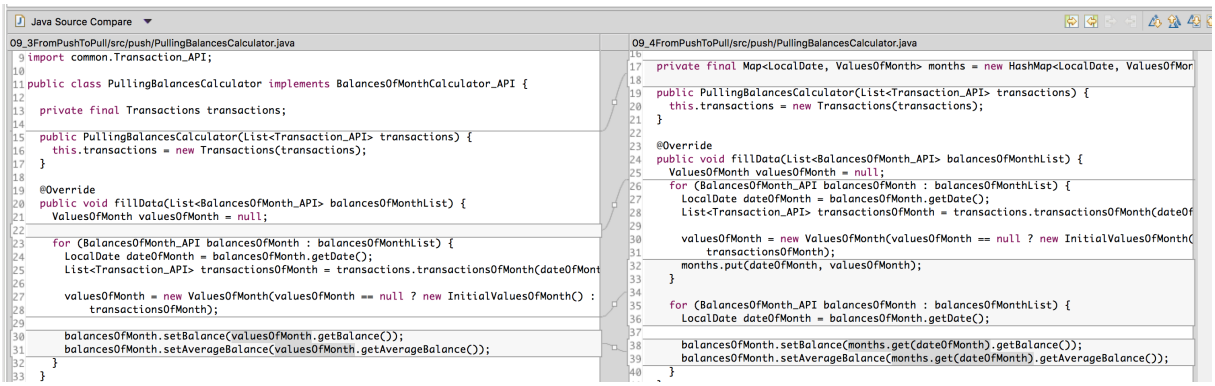
- We create the months at the same time as we fill the balances, so we cannot reuse the months.

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)



Step 14: Extract Months Map Into an Aggregate

Observations:

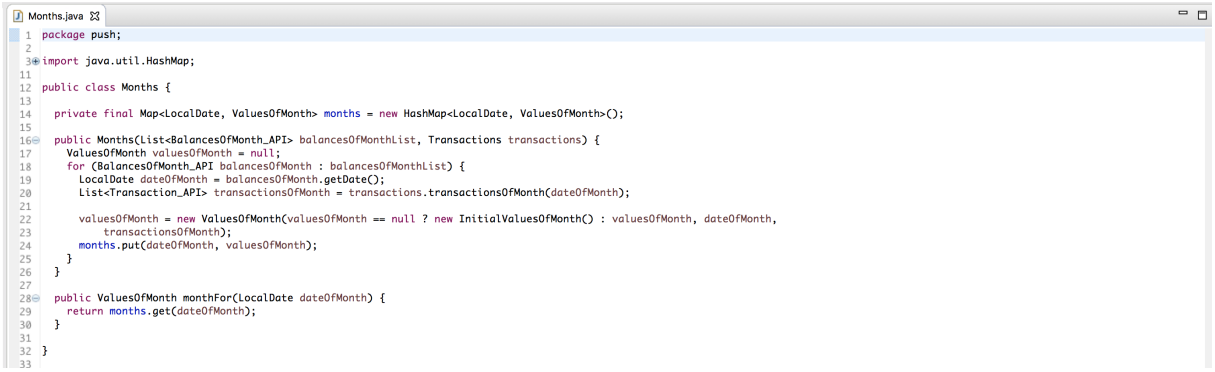
- The months map and its code don't belong in the PullingBalancesCalculator but in their own object

Solution:

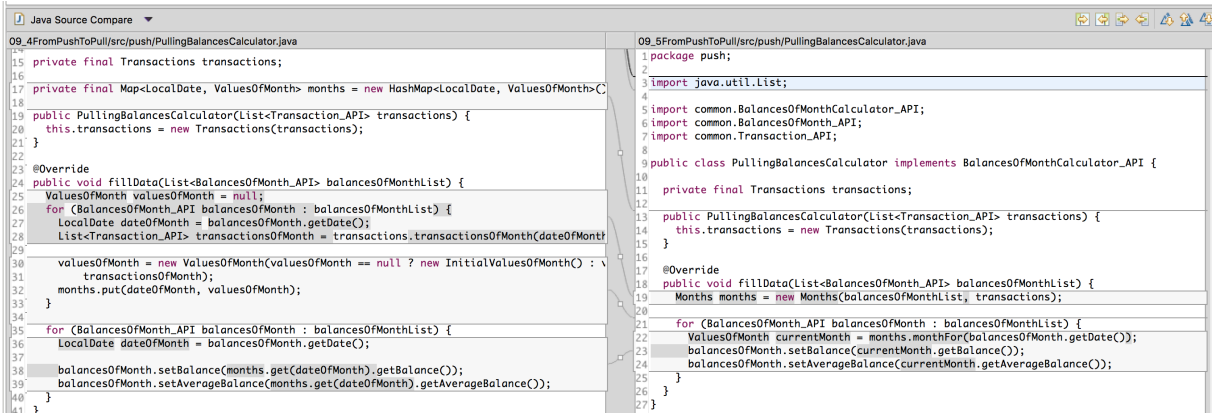
Pull out the month map and wrap it into its own object

Modified:

Created `Months` (builds up the chain of ValuesOfMonth objects)



`PullingBalancesCalculator` (uses the Months object)



Step 15: Caching

Observations:

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

Solution:

Introduce caching.

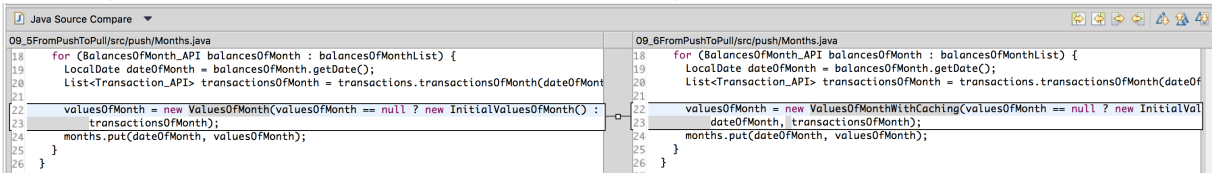
Modified:

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

ValuesOfMonthWithCaching.java

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

`Months` (uses the ValuesOfMonthWithCaching object)



Step 16: Simplification

Observations:

- The calculation of the average balance is quite complicated.

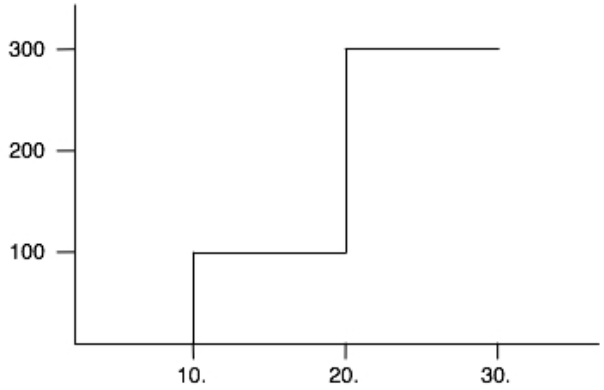
- The calculation of the average balance is isolated and easily testable.

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

Solution:

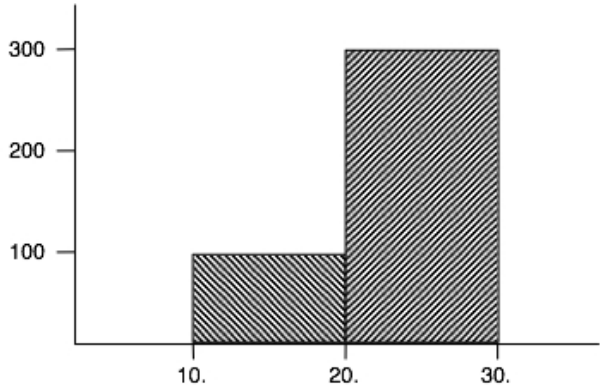
We can simplify the algorithm or even rewrite it (with TDD if we like). It is now easy to clarify the business aspects of this logic and to develop a new algorithm.

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



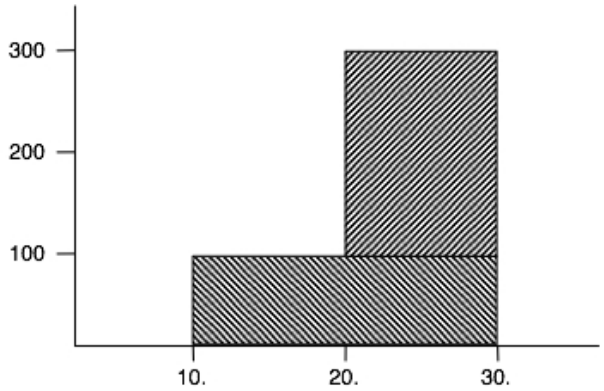
i.e. there is a transaction of 100 on the 10th and a transaction of 200 on the 20th to an initially empty account. We want to determine the average, which is the area under the graph divided by the number of days in the month.

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:

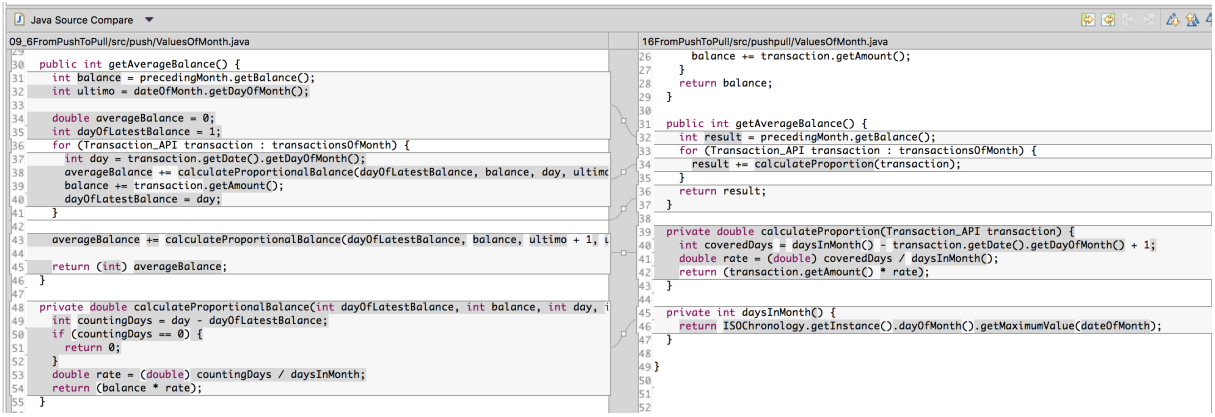


Whenever a transaction occurs, this algorithm calculates its average up to the end of the month. This algorithm is much easier to implement because the average of each transaction can be determined in isolation.

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




```
public int getAverageBalance() {
  int balance = precedingMonth.getBalance();
  int ultimo = dateOfMonth.getDayOfMonth();

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

  averageBalance += calculateProportionalBalance(dayOfLatestBalance, balance, ultimo + 1, u

  return (int) averageBalance;
}

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

```
    balance += transaction.getAmount();
  }
  return balance;
}

public int getAverageBalance() {
  int result = precedingMonth.getBalance();
  for (Transaction_API transaction : transactionsOfMonth) {
    result += calculateProportion(transaction);
  }
  return result;
}

private double calculateProportion(Transaction_API transaction) {
  int coveredDays = daysInMonth() - transaction.getDate().getDayOfMonth() + 1;
  double rate = (double) coveredDays / daysInMonth();
  return (transaction.getAmount() * rate);
}

private int daysInMonth() {
  return ISOChronology.getInstance().dayOfMonth().getMaximumValue(dateOfMonth);
}
}
```

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)