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 2-4)
- Encapsulating the calculation object (Steps 5-6)
- Separating calculation for each value (Step 7-8)
- Eliminating unwanted state (Step 9-10)
- Tidying up (Step 11)
- 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 01: 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. 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:

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

Step 02: Preparing for method extraction: Narrowing contexts

Observations:

- If we tried to extract the body of the outer loop into a method, we would end up changing a parameter inside the extracted method, namely `balance`.

Solution:

- Move the local variable into the loop body.
- Regarding `balance`:
  If we write `balance` to `balanceAndAverage` at the end of the loop, we can read `balance` from `balanceAndAverage` at the beginning of the loop.

Modified:

`PushingBalancesCalculator.fillData()`

Step 03: 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 04: Separation of Calculation and Iteration

Observations:

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

Solution:

Let's do it!

Modified:

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

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 05: 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.

Modified:

`PushingBalancesCalculator.calculateValuesForMonth()` is moved to
`BalanceAndAverage.calculateValues()`
`PushingBalancesCalculator.calculateProportionalBalance()` is moved to
`BalanceAndAverage.calculateProportionalBalance()`
`PushingBalancesCalculator.fillData()` (method invocation is adapted to moved method)

Step 06: Use New Instance of Intermediate Object for Each Execution of the Inner Loop Body

Observations:

- The for loop reuses the same `BalanceAndAverage` object in each loop iteration.
- The only value that needs to be transported from one iteration to the next is the balance of the preceding iteration, which is already retrieved anyway.

Solution:

In this step, we want each execution of the loop body to create its own intermediate object. If the value calculation uses values from a previous calculation, we need to take care of this and pass on the required values from the previous intermediate object to the calculation method, possibly widening its signature.
In our example, we explicitly pass the preceding balance to the calculation. We have to fetch it from the previous result before creating a new instance.

Modified:

`PushingBalancesCalculator.fillData()`
`BalanceAndAverage.calculateValues()`

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 07: 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.

Modified:

`BalanceAndAverage` (fields, constructor, method signature)
`PushingBalancesCalculator` (method and constructor invocation)

Step 08: 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 setters. 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).
`BalanceAndAverage.getBalance()` is modified
`BalanceAndAverage.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:

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

Step 10: Simplification

Observations:

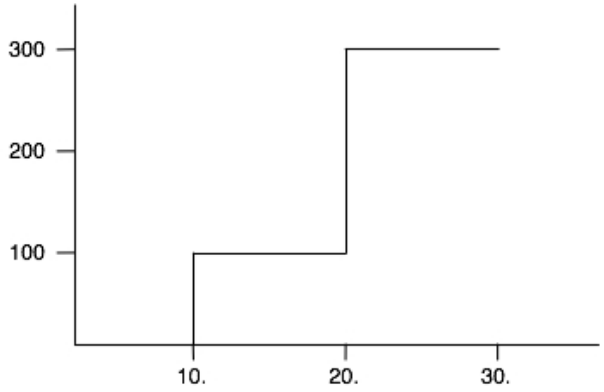
- The calculation of the average balance is quite complicated.
- The calculation of the average balance is isolated and easily testable.

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.

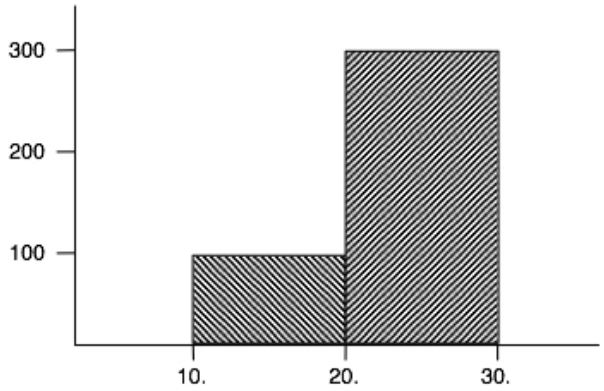
We demonstrate the change of algorithms with an example. Suppose we have a balance like this:



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

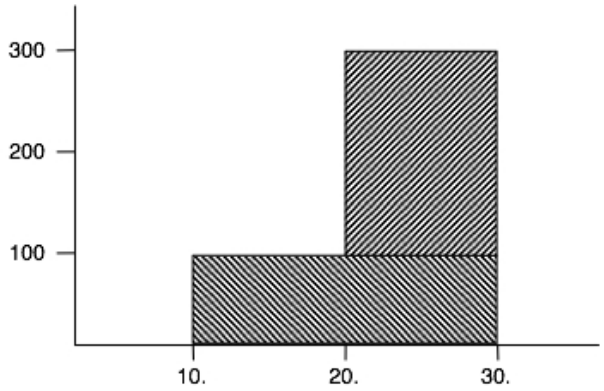
by the number of days in the month.

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:

`BalanceAndAverage.getAverageBalance()`
`BalanceAndAverage.calculateProportionalBalance()` is removed
`BalanceAndAverage.rateOf()` is created

Final Step: Achieving the "Pull" Structure (Workspace: Pull)

Observations:

- The outer for loop from the beginning still uses the same variable for the calculation objects.

- After one loop iteration, the calculation object is gone. We cannot access the results later on.
- If we want to access the results again at a later time, we need to recalculate everything again and again.

Solution:

Transform the calculation objects into a chained structure of calculation objects (called `Month`) where each month can access its predecessor's values.
Create this chained structure up-front (using a `Months`-object). Fill it with all relevant input data.
In the for-loop, simply access the calculated results of the months.
Introduce caching of the calculated results in order to avoid performance issues.

Modified:

`BalanceAndAverage` is converted to `Month` (with variants)
`Months` is extracted from `PullingBalancesCalculator`

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