Refactoring

Kwinten Missiaen, Steven Thuriot, Koen Van den dries, Bart Vangeneugden

Methodologies for Design of Software

Tom Holvoet
Mario Cruz Torres

Academiejaar: 2009 - 2010

Contents

1 Introduction

This report describes the refactoring we did on our project for the course of *Methodologies for Design of Software*. We decided not to refactor the GUI of our project. We figured this was outside the scope of the assigment, and furthermore, no unit tests were available to support refactoring. Only one significant change (documented below) was made in the GUI, and otherwise, the code was left as it was before. The following then, is a short overview of the refactoring work done on the controllers and model section of our project. The reader will notice that this report is quite brief. We believe that our original code was already of high quality; relatively few bad smells were detected, and consequently, relatively little refactoring has been done.

2 Controllers

The refactoring of the controllers wasn't too much work as these were quite good as is. In other words, not too many bad smells were detected when reviewing them.

The biggest problem here were variables that had been given unclear names. This might cause uncertainties or problems when you, or especially when another member of the team, starts reworking the existing code. Changing these variable names also reduces the need to read the Javadocs of said methods, or in some cases removes it alltogether. As a result, the work that needs to be done can be done more efficiently and will also frustrate the programmer less. As you can see, even something as small as this is worth the trouble of doing right from the start. The controllers that had the most problems with this were the repository manager and the focus work classes.

In the XMLParser there was a large procedural algorithm for parsing the XML file and initializing the model. This however leads to a *Long Method* with comments explaining the different sections of the algorithm. This was solved using *Extract Method* to separate each section while also grouping related sections of the algorithm together.

The next problem was the usage of a few switch cases throughout the code. The focus factory had a switch case to determine which type of focus the user wants to create. The refactoring book clearly states that switch cases cause a bad smell. However, being a factory, there is no good way around this. Because this is a factory, it also solves the bad smell of the switch case

itself. The problem with switch cases is that these may appear more than once in your code. Since the whole point of the factory is to take care of the creation of the wanted focus object, this switch case will never appear again throughout the model. In other words, this piece of code should not be sensitive to duplicated code. The only other place this appears is in the GUI. The case in the GUI has therefore been adjusted in such a way that it now asks the factory for the possibilities and prints these on the screen, rather than the hardcoded print it used to have. This takes away the control the GUI has over this and gives it back to the factory. Because of this, the only occurrence of the switch case can be found here. We have thought about using polymorphism to solve this problem. This, however, was not the way to go. As a result of this, the GUI would become massive and almost impossible to change in the future. This is, of course, even worse than having a switch case there.

The last problem was also found in the focus factory. Depending on the type of focus you want, a certain amount of parameters needs to be passed. Earlier, the method asked to pass all the parameters. This resulted in the programmer having to pass more parameters than needed in some cases, some of them being null objects. The method has been reworked so that an array is now passed instead. The method is now much cleaner and easier to use.

3 Models

When looking at the models, we found small problems such as *Data Clumps* and *Long parameter list*. When creating a Task object, the GUI would call a controller, who would in turn call a Task constructor. All these methods would share the same few parameters. This would form a problem when we would want to change a method, or add a parameter. Eventually, having a long parameter list would simply slug the development process. We introduced a *Value Object*. In the GUI, a small object would be created that contains the parameters that are shared along all of these methods. It is passed along these methods, and only extracted when needed, in the Task Constructor. Having a value object makes changing parameters a very quick and easy task. It also keeps our code a little bit cleaner.

We also solved a few bad smells, all related to two issues: *Long Method* and *Duplicate Code*. Often, both issues could be solved using the same solution: *Extract Method*. A long method means a lot is going on in one

method. If a lot is going on, the method becomes bloated and more difficult to understand. Splitting it up makes it very clear to see what happens, and gives the developper a great overview so he can make faster and better changes. Smaller methods are usually very clear in exactly what they do, no matter how small their footprint. This is also the case for duplicate code. Often a small operation is used throughout several methods. However it's explicitly coded in all of these methods. If however that operation needs a change, that could create a large task for the developer. Extracting that operation to a seperate method takes all of these problems away. For example, in a few places, the code was changed to use the method Task.dependsOn(), a method that already existed, but wasn't always used.

Most of the refactoring in the model was done on the Task class. In other model classes, only little refactoring was necessary. A few small changes were made - most of them related to poorly chosen variable names. In the method Project.remove() a few lines of obsolete code have been removed.