LINGI2252 Assignment 2 report

Group 17
Xiao Xu
Xavier Crochet

Kim Mens
Sergio Castro

Contents

1 Introduction

In the first report, we try to do a manual analysis of *Glamour*. This first step gives us a global overview of *Glamour* and helps us to understand how *Moose* and his tools works. Because of the size of the system (about 270 classes), we use some tools in order to get some hints on where to begin. It wasn't exactly what was asked in the statement of the first report but now that we have all the tools in our hand, we can start looking more effectively for the needles in the haystack that Glamour is.

Moreover, we found some code duplication in the source code (but relatively few compared to the size of the framework) and the utilisation of the *Patern Obeserver* in the GLMPresentation class. We thus conclude that Glmamour was globally well-designed, despite the few code duplication.

The report will be divded in two parts:

1. We start by analyse the code using crieria not identifiable by metrics to see

   - How well the code is commented.
   - How good the naming convetion is.

2. Then, we develop a set of thresolded metrics we'll use later to analyse the code.

2   Architecture

2.1   Introduction

### 2.2   Overview Pyramid

2.3   Complexity

3   Comments

Commenting code is important to improve **readability** and the **reusability**. It make it easier for developpers to contiue a project or for student to analyse it.

There is so few comments in the code that we decide to not use a metric to analyse this part. In deed, there is only 768 comments! That's about 3 comments for each class.

4   Naming Convention

A good naming convention is also an important point in order to imrpove **readability** and **reusability** of the code. Using the *Name Cloud* utility from moose, we can display the following figure, showing the most used words **for naming classes** in the framework.
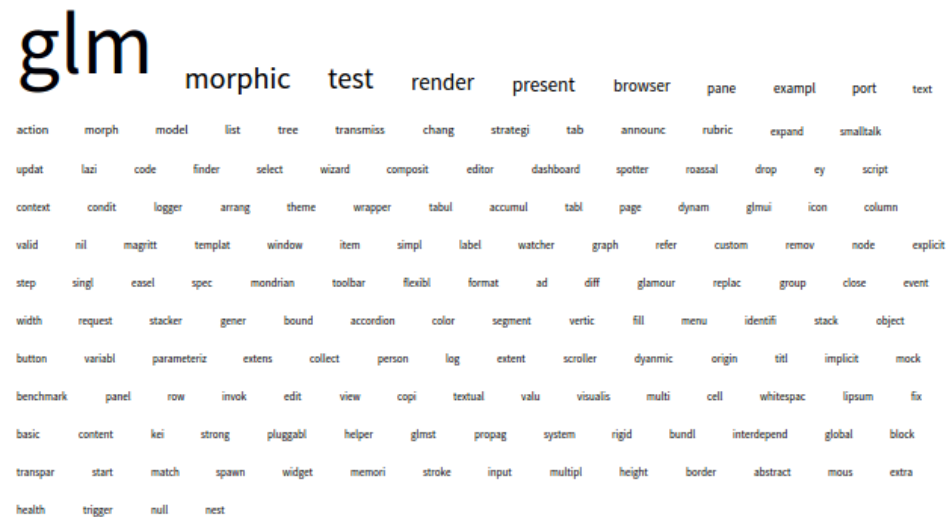
As we can see **GLM** is the suffix coming up the more often. Other suffix such as **test render** or **morphic** arise too. It seems that classes are named quite correcly in order to provide meta-information about their use.

We can perform the same analysis for methods or variable.Globally there is no bad smells coming up here. There is not randomly named classes variable or

methods. Anyway, naming convention is quite a controversial issue, so we are not going deeper in the analysis here.

5   Metrics-based analysis

5.1   Introduction

Before deciding when to refactor, we need first to identify where bad smells occurs inside the framework.Using *Moose*'s metrics set, we start by etablishing criteria to indentify those parts.

In practice, we'll use the following query in order to retrieve the classes based on our thresholded metrics:

```
((self flatCollect:[:each | each classes]) flatCollect:[:each |
    each methods]) select:[:each | (each ATTRIBUTE OPERATOR
    THRESHOLD)]
```

If we want only to look into classes, we can do:

```
(self flatCollect:[:each | each classes])  select:[:each | (each
    ATTRIBUTE OPERATOR THRESHOLD)]
```

Avec

- ATTRIBUTE - The name of the attribute

- OPERATOR - The operator for the comparaison

- THRESHOLD - The value of the threshold

When we wrote the first report, it appears clearly that they was some big classes in the code. We'll start by using *Elephant-identifiers* attributes such as:

- Cohesion

- Code Duplication

- Larg Method and Classes

to try to explain what goes wrong inside these classes (and propose solutions).

5.2   Elephant-identifier attributes

5.2.1   Code duplication

Having the same code structure in more than one place in the system is quite nasty. The programm will be better if we find a way to reunite them.The attributes we have

to handle are *NumberOfExternalDuplications* and *NumberOfInternalDuplications*.

Threshold - *SmallDude*, the tool used by moose to detect duplicated code have a threshold of 3 lines. We decide here to set the threshold of the two attributes to 0, and to analyse one at the time the results.

When querying for

- *NumberOfInternalDuplications* we get 27 classes

- *NumberOfExternalDuplications* we get 30 classes

Let's take for example the class *GLMBasicExample*.

- We have 3 external duplications, meaning that we have 3 block of code present in three other class of the framework. The duplication with *GLMUpdateInterdepententPanesTest* is the following:

```
browser := GLMTabulator new.
browser
        column: #one;
        column: #two;
        column: #three.
(browser transmit)
        to: #one;
        andShow:[:a] a tree display:[:x|1 to: x]].
(browser transmit)
        to: #three;
        from: #two;
```

here, we can extract a new class from *GLMBasicExample* and use it inside *GLMUpdateInterdepententPanesTest*.

- We have also 6 internal duplications. The following piece of code is present two times in the class.

```
|browser model|
        model := Dictionary new.
        model at: #some put: #(1 2 3 4).
        model at: #even put: #(2 6 8).
        model at: #odd put: #(3 7 9).

        browser := GLMTabulator new.
        browser column: #one.
        browser transmit to: #one; andShow: [ :a |
                a tree
                        display: [model keys];
```

We can refactor the classe by extracting the duplicated blocks as new methods, making the class more maintainable.

Basicly, the solution is to **extract** the duplicated block as **a new method** when an internal one occurs, and as a **new class** when it's an external. When a external duplicated block occurs between **two parent classes**, it's more appropriate to resolve it using **inheritance**.
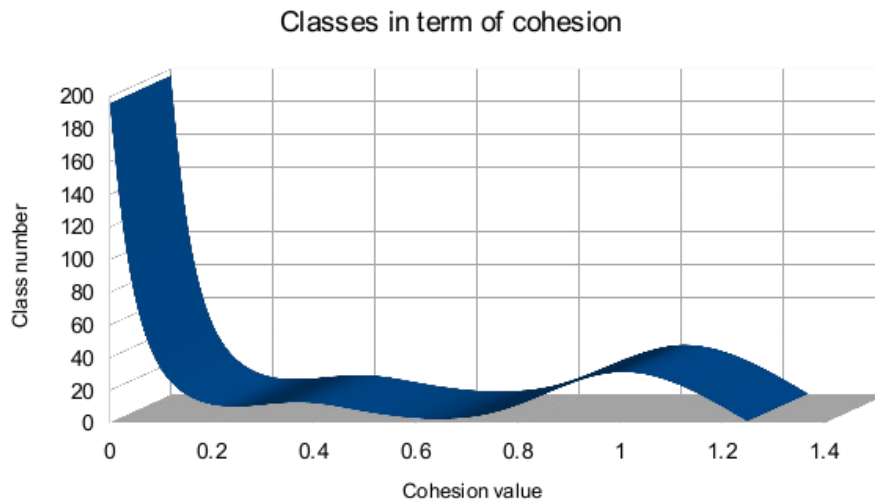
5.2.2   Cohesion

Cohesion metrics measure how well the methods of a class are related to each other. A cohesive class performs one function. A non-cohesive class performs two or more unrelated functions. The idea is to refactor the non-cohesive class into several smaller classes. Cohesion is thus a good thing.

The attribute to inspect here is *tightClassCohesion*. It's a ratio between number of connected methods and the maximum number of possibly connected methods. Two methods are connected if they access the same attribute. We modify our query a little bit:

```
(self flatCollect:[:each | each classes])   select:[:each | ((each
    tightClassCohesion  >= MIN) &( each tightClassCohesion < MAX))
    ]
```

And compute the following graph, displaying the number the repartition of all the classes in term of their cohesion value.

Classes in term of cohesion

- We see thaht the class having the highest cohesion value (1.47) is GLMUp-dateMorphicTest and has only **96 lines of codes**.

- About 200 classes have less than O.1 cohesion.

When we ask moose to display the concerned classes, we see that the concerned classes are the elephant one of the framework. Furthemore, the whole *GLMPresensation* Family appears on the picture (as you can see in the picture to the right):
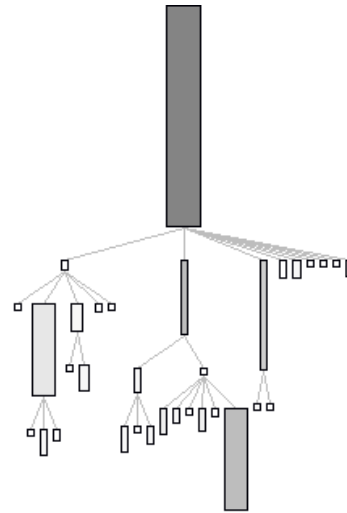
Thoses classes (GLMPresentation, GLMListingPresentaion, GLMWizard, GLM-Browser, ...) where also identified in the first report as elephant classes.

If we query for classes having low cohesion and high line number of code, we get similar results. We reach here one of our first report conclusion. They are a lot of big classes in the framework, there is work to do on this part. One must increase those classes's cohesion by refactoring them into smaller classes.

5.2.3   Large methods-classes

5.3   Accessors

5.4   Long Method

5.5   Long Class

5.6   Long Parameter List

5.7   Thresolded Metrics Summary

5.8   ...

6   Improvements

7   Conclusion

8   Annexes