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 work. 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 *Pattern Observer* in the *GLMPresentation* class. We thus conclude that *Glamour* was globally well-designed, despite the few code duplication.

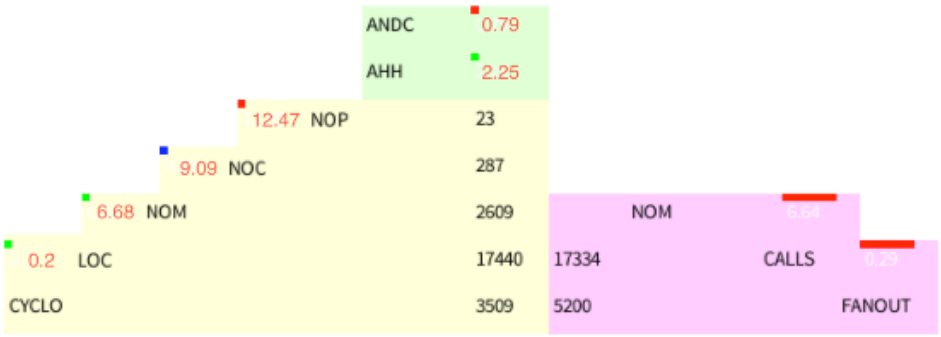
The report will be divided into two parts:

1. We start by analysing the code using criteria not identifiable by metrics to see
   - How well the code is commented.
   - How good the naming convention is.
2. Then, we develop a set of metrics we'll use later to analyse the code to identify the attributes causing the elephant classes.

2 Architecture

2.1 Overview Pyramid

With moose, we can easily have a pyramid visualisation of the system.



As you can see, three different colours are used in the figure.

1. The green part provides information about the depth of the inheritance. *ANDC* means "Average number of derived classes" and *Average Hierarchy height*.

2. The yellow part provides information about the complexity of the system. *NOP* means number of package, *NOC*, number of classes, *NOM*, number of methods, *LOC*, lines of code and *CYCLO*, cyclomatic number.

3. In purple part provides information about coupling. *NOM* means number of methods, *CALLS*, number of invocation and *FANOUT*, number of called classes.

On the left top of each pyramid component, there is a small coloured dot.

- Green means good.
- Blue means average.
- Red means bad.

From this picture, we made some observations.

- It seems that glamour behaves quite good in term of complexity.
- Their is some use of hierarchy, but the average number of derived classes is quite low.
- The coupling part is very bad.

This figure gave us some leads on where to begin our analysis.

2.2 Complexity

In the first report, we print the whole figure displaying the system complexity. We conclude they were some elephant classes with a lot of methods in the system.

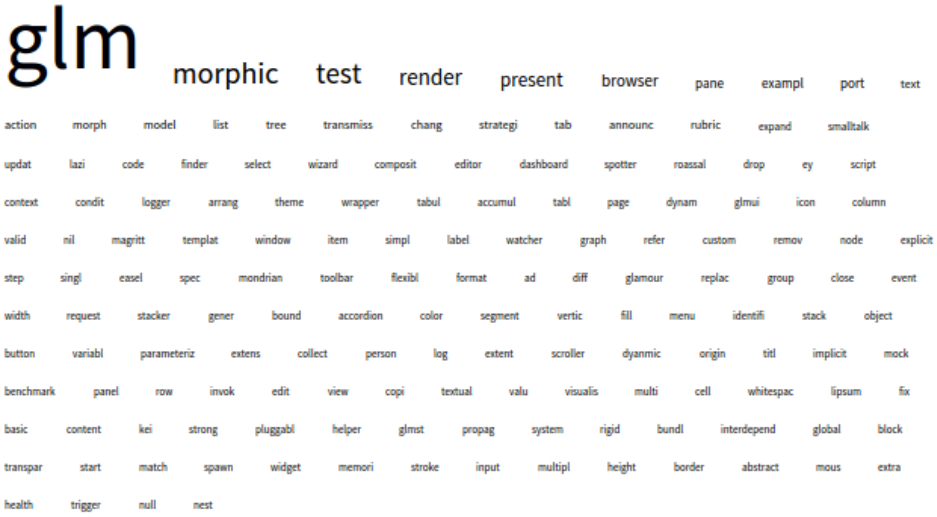
3 Comments

Commenting code is important to improve **readability** and the **re-usability**. It makes it easier for developers to continue a project or for student to analyse it.

There are so few comments in the code that we decide to not use a metric to analyse this part. In deed, there are 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 improve **readability** and **re-usability** 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 correctly 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 re-factor, we need first to identify where bad smells occurs inside the framework. Using *Moose*'s metrics set, we start by establishing criteria to identify those parts.

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

```
((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 comparison
- 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
- Large 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 program 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 re-factor the classes 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**.

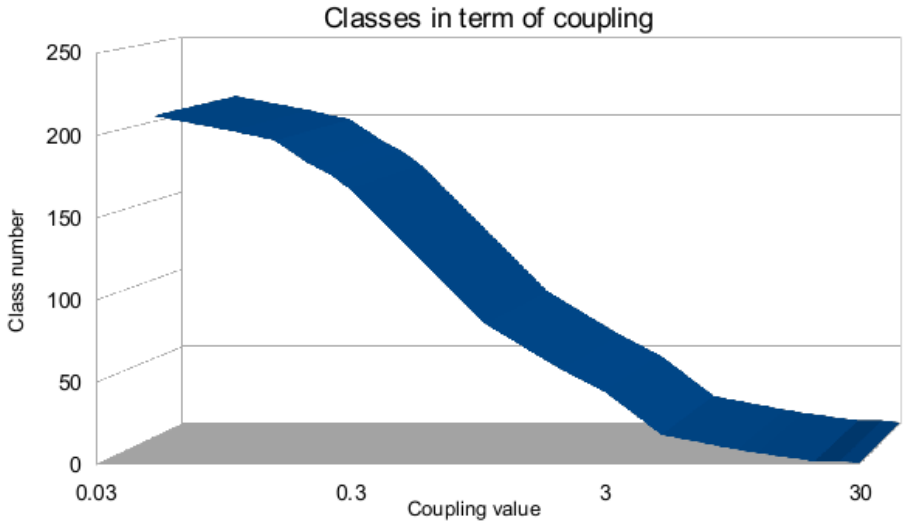
5.2.2 Cohesion

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 re-factor 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))
    ]
```

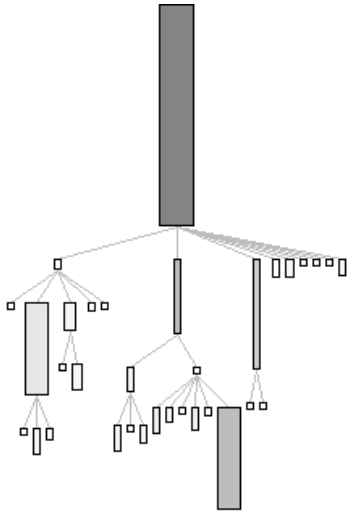
And compute the following graph, displaying the number the repartition of all the classes in term of their cohesion value.



- We see thaht the class having the highest cohesion value (1.47) is *GLMUpdateMorphicTest* 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):



Theses classes (*GLMPresentation*, *GLMCompositePresentation GLMListingPresentation*, *GLMWizard*, *GLMBrowser*, ...) where also identified in the first report as elephant classes.

5.2.3 Coupling

Coupling measures the interdependence of class with other classes. Tight coupling is a bad thing. It reduces flexibility and re-usability while increasing the difficulty to implements test. In order to measure coupling, we use the following metrics:

- *fanIn* : The number of classes referencing our class.
- *fanOut* : The number of classes referenced by our class.

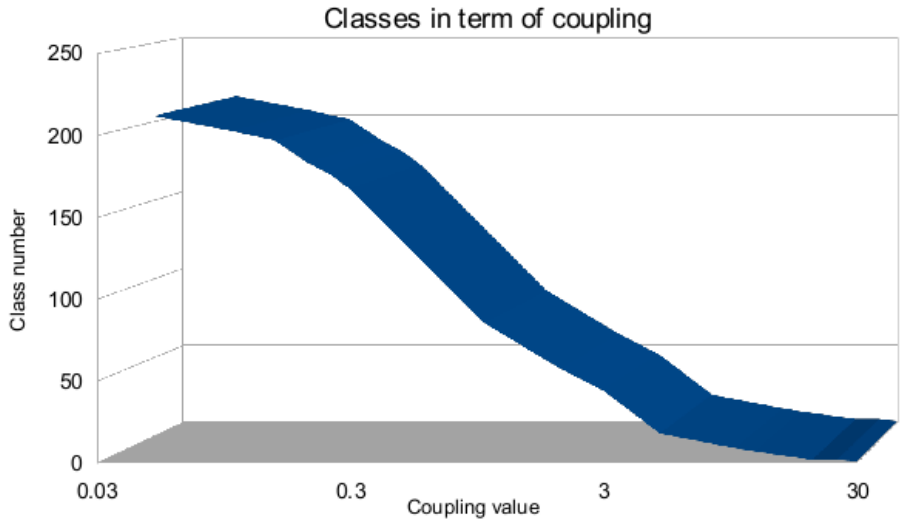
We

- Compute the following metric: $tightClassCoupling = \frac{fanOut}{fanIn}$
  It measure the tightness of a class's cohesion.
- Use the following query

```
(self flatCollect:[:each | each classes]) select:[:each |
        (
        (each fanIn > 0)
        ifTrue:
        [
        (each fanOut / each fanIn) < THRESHOLD_VALUE
        ]
        ifFalse: [ false ]
        )
]
```

Some classes have a *fanIn* attribute whose value is zero. To overcome this issue, we decide to ignore theses classes. In deed, after querying for them, we found out that all those classes were test ones. That explains why there aren't called by any other classes.

- Compute the following graph with the collected data.



Note : We use a logarithmic scale to display the x axis.

Immediately, it appears that, if we invert the x-axis, the curve has the same shape as the cohesion one (See the above section). Once again, all the big classes have high coupling numbers.

5.2.4 Large methods-classes

To pursue our elephant classes analysis, we can look into these metrics:

Metric	Threshold 1	Threshold 2
numberOfLinesOfCode	25	30
numberOfStatements	15	25
numberOfMessageSends	10	20

The idea here is to look for classes having large methods doing a lot of stuff. So we look for method respecting the following rule:

$$numberOfLinesOfCode > 25$$
$$\vee\ numberOfStatements > 15$$
$$\vee\ numberOfMessageSends > 10$$

We get 196 classes. If we look at the complexity graph, these are the same classes we get on the above section. To be certain, we increase our threshold (See

the second threshold of the table). We get 150 classes, among them, the one we identify in the first report and in the previous section.

5.2.5 Accessors

There are different ways to access an attribute:

- Using an accessor/mutator.
  This is the best way in the oriented-object programming. According to this principle, the attribute is made private to hide and protect it from other code. It can neither be used directly, nor e modified by other code, except by the accessor/mutator function.

- Directly from outside of the class.
  This is dangerous, and could cause some maintenance problems, because we are limited to modify the code

- Directly from inside.
  This is not good too, but this will not be a big problem if they are accessors/-mutators inside the class and we have just forgotten to use them. The reason is that, for the person who maintains the code, he just needs to take care of the code inside the class.
  However, if there isn't any accessors/mutators, it will clearly become and issue. The person maintaining the program must not only check this class, but also find out all the uses/modifications of the attribute.

We consider that, for a good class, every attribute has one *getter* method (accessor) and on *setter* (mutator) method. Therefore, there are **two** accessor-methods per attribute.

The Glamour system has 287 classes. We fill find out among them how well attribute are accessed.

At first, we should know how many classes have attributes by using the following query:

```
((self flatCollect:[:each|each classes])
        select:[:each|each numberOfAttributes>0]
```

Glamour has 138 classes with attributes.
Then we query for classes respecting the following rule:
$(numberOfAttributes) > 0)^(numberOfAccessorMethods >= (2*eachnumberOfAttributes))$

This query results in 36 classes that might correctly access their attributes. **But**, we have to be aware that they might be some attributes with 3 or more accessors-methods. As consequence, they might be some false positives results

with attributes having only 1 or 0 accessor method.

The solution is to query for classes having more than 3 accessor methods and to manually look into them. Luckily, there is only one class printed out: *GLMPanePort*). The maximum number of well designed classes is thus 36.

Therefore, we can compute the proportion of good classes as follow: $\frac{Classesrespectingtherule}{Classeswithattributes} = \frac{36}{138} = 26.09$

In order to going deeper, we try to get more information that might help us to further analyse the *Glamour System*. Thus, we query for classes having one or more attributes. There are 78 classes respecting this rule.

Therefore, we have 47 classes having attributes without accessor-methods. They could cause maintenance problems.

Now, we are going to look for classes accessing directly their attributes. First, we use the following query to get the total number of attributes.

```
((self flatCollect: [:each | each classes]) flatCollect: [:each |
    each attributes])
```

We get 364 attributes.

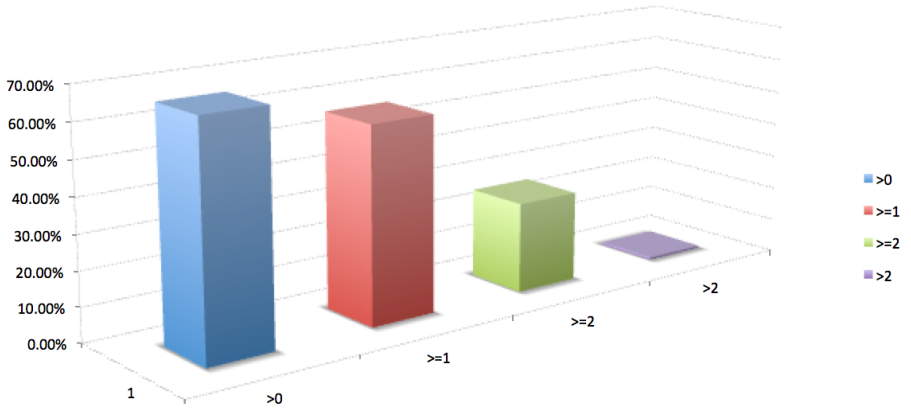
We can get the number of directly accessed variable using the textitnumberOfGlobalAccesses metrics. That represents only 2.20% of the total attributes of the system. That's very nice!

But, using the following query

```
((((self flatCollect: [:each | each classes])
        select: [:each | each numberOfAttributes > 0])
        select: [:each | each numberOfAccessorMethods > 0]))
flatCollect: [:each | each attributes])
select: [:each | numberOfGlobalAccesses > O ]
```

it appears that, among these 8 attributes, 4 of these globally acceded attributes have accessor-methods!

We can compute the following chart, showing information about class using accessor-methods:



- The blue column represents classes having accessor-methods (about 66%)
- The red one represents classes having at least one accessor-method (about 57%)
- The green one represents the *good* classes (about 26%)
- The purple one represents having 2 or more accessor-methods for only one attribute. (Only one class)

There are many classes with no accessor-methods, which will may cause maintenance problems in the future. Furthemore, we see that they might be some false-positives, as they might be an average of 2 accessor method per attribute if one atribute has only one accessor and an other 3. But, as there are only 8 directly acceded attributes, we see that the developer rarely use such bad smells.

5.2.6 Conclusion

If we query for classes having low cohesion, high coupling, high line number of code and large methods, we get always. similar results. Even more, we reach one of our first report conclusion (We identified a large number of big classes.) We come up here the first design problem of the framework: *Cohesion*. One must increase the cohesion of theses classes by re-factoring them into smaller ones.

Some of the concerned classes are:

Class Name	Lines of code	Messages sent	Cohesion	Coupling
GLMBasicExample	1370	1600	0.0	3.9565
GLMPresentation	659	502	0.076	2.6093
GLMWhiteSpaceTheme	778	524	0.0	4.25
GLMUITheme	797	552	0.0	4.7272
GLMCompositePresentation	255	287	0.0051	0.4659
GLMWizard	374	321	0.0029	1.6206
GLMBrowser	360	436	0.012	0.8041

6 Improvements

7 Conclusion