# LINGI2252 Software Engineering : Measures and Maintenance
# Second Report : Automated code analysis


Group 12 : Mulders Corentin, Pelsser Francois
19/12/2011

# 1 Introduction

In this report we will try to go deeper in our assessment of the Monticello system's quality. The conclusions of our manual code analysis from the first report were that the system was of good quality and made good use of the object-oriented programming.
First we will identify Moose's metrics that we can use, then define thresholds for those to allow us to single out parts of the code that might be problematics. We will also explain how we used metrics visualizations tools.

# 2 First impression with the overview pyramid

Moose allows us to visualize the system as an overview Pyramid, this visualization is explained in the book "Object-Oriented Metrics in Practice" [1].
Here is the overview pyramid for the Monticello system :


**Figure 1**

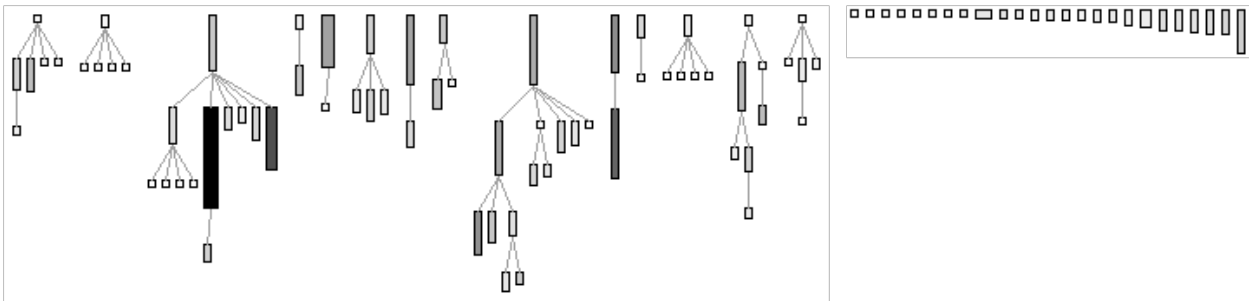
This representation of the system is separated in three different parts. The top concerns the inheritance measurements. The bottom-left concerns the size and complexity of the system and the bottom-right represents the coupling.
In the bottom-left part we have various metrics : NOP the number of packages, NOC the number of classes, NOM the number of operations, LOC the number of lines of code and CYCLO the cyclomatic number. In the bottom-right part we have CALLS the number of operations calls and FANOUT the number of called classes, and NOM is the same number of operations as in the left part of the pyramid. In the top part ANDC is the average number of derived classes and AHH is the average hierarchy height.
What is really interesting in this pyramid though is the values in the boxes. In the bottom parts of the pyramid these represent the division of the line below by the current line. And the most useful part about these boxes is their color, a red color means that the value in the box is considered good, a green box represents an average value, and a blue box a bad value. (These quality evaluations are based on comparison with an average value for smalltalk projects)

1. Lanza, M. and Marinescu, R. (2006) *Object-Oriented Metrics in Practice* Springer

Knowing this, we can see very easily with this pyramid both values are good for the inheritance part, all values but one are good for the size and complexity part of the pyramid. And the coupling has a good value and a bad one. This allows us to have a first glimpse of the strength and weaknesses of the system.

# 3 System complexity

The graph of the system complexity is a good help to have a first approach of the system. In this graphic each box represents a class, links between boxes are inheritance relations, height represent the number of methods, width the number of attributes and the color is darker when there is more line of codes in the class.



We directly see the inheritance is good, the maximum deep is 4 level. There is 25 lonely classes but a deeper look at them show they don't have easy easy relation to group them. This confirms the information given by the overview pyramid that the inheritance is well used in this system.

# 4 Tools used for metrics

## 4.1 Moose

We chose to use Moose for our automated code analysis. Moose offers a wide range of metrics some of which are pretty powerful like the duplicated code detection for example. One other reason that motivated our choice was that moose allows to generate various visualizations such as the overview pyramid and the system complexity graph that we used earlier to get a first glimpse of the system. And also Moose was already integrated in pharo so using it was the easiest thing to do.
Moreover as we will see later moose is perfectly adapted to query the code to apply the thresholds we define on metrics and get results.

## 4.2 Visualizations

### 4.2.1 Class blueprints

A visualization tool that we will use in this report is the class blueprint. These blueprints can be easily generated with moose. Here is a brief explanation on how to read those.

Here is as an example the blueprint of the class MCVersion from the Monticello system :

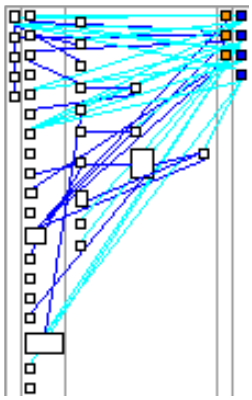


**Figure 2:** MCVersion blueprint

The boxes represent entities from the class whose type depends on the columns in which the box is placed. The columns are from left to right : initialization, public methods, private methods, accessors, attributes.
The dark blue links represent the invocation of methods and the light blue links represent access to attributes.

# 5 Find bad smells

## 5.1 Available metrics to use

Here is a list of the interesting metrics that we can get from moose. For each of those we need to know what the metric means. And we will also need to determine if we can use the metric to learn something about the system. And if it's the case what we can learn from it. These basic metrics will then later be combined to get informations about the system.
The last column in our tables will explain what informations we can get about the system from those specifics metrics if it is not obvious from the metric meaning.

First we consider the metrics about individual classes.

| **metrics for a class** | | |
|---|---|---|
| **name** | **meaning** | **informations about the system** |
| fanIn | Number of other classes that reference this class. | Can be used to measure coupling. |
| fanOut | Number of others classes referenced. | Can be used to measure coupling. |
| hierarchyNestingLevel | Depth of the inheritance tree at which this class is. | |
| numberOfAbstractMethods | Number of abstract methods. | |
| numberOfAccessorMethods | Number of accessors methods. | |
| numberOfAttributes | Number of instance variables. | |
| numberOfAttributesInherited | Number of instance variables from a mother class. | |

| | | |
|---|---|---|
| numberOfChildren | Number of class that inherit this one. | |
| numberOfComments | Number of comments. | |
| numberOfInteralDuplications | Number of duplicated lines of code in the class. | Will be used to find duplicated code. |
| numberOfExternalDuplications | Number of duplicated lines of code from this class in others. | |
| numberOfLinesOfCode | Number of lines of code of the class. | |
| numberOfMethods | Number of methods. | |
| numberOfMethodsInherited | Number of inherited methods. | |
| numberOfMethodsOverriden | Number of overridden methods. | |
| numberOfParents | Number of parents. | |
| numberOfPrivateAttributes | Number of private attributes. | |
| numberOfPrivateMethods | Number of private methods. | A low number of private methods compared to the total number of methods might be a sign of a lack of cohesion or strong coupling. |
| numberOfProtectedAttributes | Number of protected attributes. | |
| numberOfProtectedMethods | Number of protected methods. | |
| numberOfPublicAttributes | Number of public attributes. | |
| numberOfPublicMethods | Number of public methods. | A high number of public methods compared to the total number of methods might be a sign of strong coupling or low cohesion. |
| subclassHierarchyDepth | Depth of the subclass hierarchy. | |
| tightClassCohesion | Tight class cohesion metric to represent the cohesion of the class. [2] | |
| totalNumberOfChildren | Number of children of this class at every level. | |
| weightedMethodCount | Sum of the weight of methods from the class. | Can be used to get the complexity of the methods in the class. |

2. The tight class cohesion is computed by dividing the number of direct connections by the total number of possible connections. A connection is defined as two method accessing the same instance variable (even if this variable is accessed by any method called in the method). So the tight class cohesion is defined by $\frac{NDC}{NP}$ with $NP = \frac{N(N-1)}{2}$ with N being the number of methods.

And finally the metrics about methods and instance variables.

| **metrics for a method** | | |
|---|---|---|
| **name** | **meaning** | **informations about the system** |
| cyclomaticComplexity | Compute the cyclomatic complexity. | Can be used to find code with lot of conditional statements. |
| numberOfConditionals | Number of conditional statements in the method. | |
| nameLength | Length of method's name. | Can be used to locate variable with too long names. |
| numberOfComments | Number of comment statement in the method. | Method with a lot of comments is likely to be complicated, so we could use that to find too complicated methods. |
| numberOfLinesOfCode | Number of line of code in the method. | Can be used to find methods that does too much work. |
| numberOfStatements | Number of statements in the method. | |
| numberOfMessageSends | Number of messages sent by the method. | |
| numberOfParameters | Number of parameter taken by the method. | A method tanking too much parameters is either to vast trying to do too much or the parameters are related and should be grouped in an object. |

| **metrics for an attribute** | | |
|---|---|---|
| **name** | **meaning** | **informations about the system** |
| nameLength | The length of the attribute name. | Can be used to locate variable with too long names. |
| numberOfAccessingClasses | Number of class accessing this attribute (without counting the class containing the attribute). | Can be used to know if the attribute is used directly from other classes without using the accessors. |
| numberOfAccessingMethods | Number of methods accessing this attribute (without counting the methods of the class containing the attribute). | |
| numberOfGlobalAccesses | Number of access to the variable by external classes. | |
| numberOfLocalAccesses | Number of access to the variable by methods in the attribute's class. | Can be used to detect access to an attribute from inside the class. |

### 5.2 How we will use these metrics

In moose we are able to query a collection to filter elements out of it. Since we use moose's metrics and it's possible to query with moose we chose to use it for the queries as well.

Here is a typical query as we will use to apply all of our thresholds to get results :

```
((self flatCollect:[:each|each classes]) select: [:each |
   numberOfLinesOfCode > 50])
```

This query will get the list of classes from the current collection for which numberOfLinesOfCode is above 50. If we start from the Monticello packages we imported in moose we can use all the Monticello classes as basic collection.
This can also be used to get methods or attributes by replacing the "classes" keyword in the smalltalk query by "methods" or "attributes".

## 5.3 Finding bad smells with metrics

### 5.3.1 Code duplication

Finding duplicated lines of code with moose is pretty easy. We have to ask moose to compute the duplications. Then we can access the *numberOfInteralDuplications* and *numberOfExternalDuplications* metrics for each class. To find the duplicated code blocks moose uses SmallDude. Note that SmallDude will only detect duplicates of a blocks of at least 3 lines of code.
We can apply a simple filter on each of those at a time.

**Classes with numberOfExternalDuplications > 0** None.

**Classes with numberOfInternalDuplications > 0** Three classes have internal code duplications :

- MCClassDefinition, has 6 duplicated blocks of code.
- MCPackageManager, has 1 duplicated blocks of code.
- MCConfigurationBrowser, has 1 duplicated blocks of code.

We verified our findings by looking in the source code at the lines indicated by moose. For example for MCConfigurationBrowser we found the following code block at two different places in the code :

```
(self pickWorkingCopiesSatisfying: [:each | (self includesPackage: each
   package) not])
                do: [:wc |
                        wc ancestors isEmpty
                                ifTrue: [self inform: 'You must save ', wc
                                   packageName, ' first!
Skipping this package']
                                ifFalse: [
                                        self dependencies add: (
                                           MCVersionDependency
                                                package: wc package
                                                info: wc ancestors first)
                                                   ]].
        self changed: #dependencyList; changed: #description
```

Since the found duplications are all internal to classes. These classes can be refactored by extracting the duplicated codes blocks as new methods.

### 5.3.2 Accessors

Accessors allows to hide values in the object, it increase the abstraction. We are going to look for 4 things

**Direct access when accessors exists** This should be avoided the most. This implies the person who write the class prepare accessors who maybe do some computation, trigger actions or other things and there are just ignored.

**Direct access from outside the class** This is quite a problem because this implies the client using the class has to know how the class is working. This is a problem for maintenance because we are limited in changes. For example if we stored a date in a timestamp format, then decide to create a date object, with accessors retro-compatibility could be easily assured but with direct access not.

**Direct access from inside the class** This is approximately like the direct access from outside the class. But it may not be as bad because the person which is going to maintain the class only has to changes things in one class.

**No existing accessors** This could be the reason for direct accesses which can create some maintenance problems.

The presence and use of accessors can be verified by many metrics :
About the presence of accessors in a class, one simple idea would be to check if a class contains 2 times more accessors than attributes. But an attribute could have more than 2 accessors. MCAddition for instance has only 1 attributes and the initializeWithDefinition : is considered as a pure accessor of the definition attribute. And there might be attributes with accessors and others without in a given class.

Another way to do is to look if all attributes have a numberOfLocalAccesses of 2. For example this works better with classes having 3 accessors for an attribute and only one for another one. Direct accesses from outside (inside) the class are counted by numberOfGlobalAccesses (numberOfLocalAccesses). The thresholds for these should then be that numberOfGlobalAccesses must not be 0. And the value of numberOfLocalAccesses should always be 2 (if it's less than two and there is no direct access then the variable is unused and if it is more then the accessors are not the only way to access the variable).

The numberOfAccessingClasses and the numberOfAccessingMethods can show how complicated it would be to maintain the code. For instance if we want to trigger an action each time an attribute is modified, if we have a numberOfAccessingClasses of x, we'll need to change the code of x classes (the idea is the same for numberOfAccessingMethods).
Direct access when accessors exist can be detected by finding a class with accessors and then use numberOfGlobalAccesses and numberOfLocalAccesses to detect direct accesses locally or globally to the attribute.
Changing code to add accessors will make it easier to read since it will increase abstraction by making objects more responsible of their attributes. This will also increase the maintainability of the code because it allows to do pre or post-treatement.
In our previous assignment we said that accessors were used by some classes through Monticello. Let's go deeper and see how much it uses accessors.
First we will get the classes were we thing there is one getter and one setter per attribute while excluding the classes without any attribute. To do that we use the following code that we enter into a workspace in Moose after selecting all the Monticello packages.

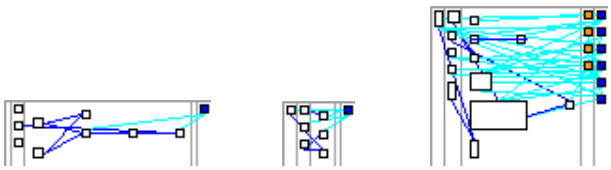
```
((self flatCollect:[:each|each classes]) select:[:each| each
   numberOfAttributes > 0 ]) select:[:each| each
   numberOfAccessorMethods >= ( 2 * each numberOfAttributes)]
```

This returned 17 classes let see some examples from those :

MCAddition MCModification MCPackage

Here we can notice something odd, MCAddition and MCModification both have more than two accessors by attributes, in fact they have two methods (targetDefinition and definition) returning the same attribute (definition). Another thing to notice is the bypass of the accessor, we will come to that later.
Then by replacing each numberOfAccessorMethods >= ( 2 * each numberOfAttributes) by each numberOfAccessorMethods < ( 2 * each numberOfAttributes) we get the 67 classes having less than 2 accessors per attribute. Here are few examples.

MCMerger MCPatcher MCVersionInfo



We are now interested about getting all attributes having direct local and global accesses.
First for direct global accesses we use this query :

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

It gave us 32 attributes that in fact are class accessing their parents classes attributes directly. Then for the direct local accesses we apply :

```
(((self flatCollect:[:each|each classes]) flatCollect:[:each|each
   attributes]) select:[:each| each numberOfLocalAccesses > 2 ])
```

Wich give us 178 attributes. There are some false positive like classes having more than two accessors for one attribute (see MCAddition or MCModification) but there are few.
Then finally we try to count the number of attributes having accessors but also accessed directly. For this we will use this query :

```
(((((self flatCollect:[:each|each classes]) select:[:each| each
   numberOfAttributes > 0 ]) select:[:each| each numberOfAccessorMethods
    >= ( 2 * each numberOfAttributes)]) )  flatCollect:[:each|each
   attributes]) select:[:each| each numberOfAccesses > 2 ]
```

This is not perfect because we are not getting classes that have accessors for some attributes and none for others. But we still found 19 attributes. Some of which were from MCAddition and MC-Modification as expected.
I think this is a very important flaw and this would be a great improvement to add accessors to attributes and to remove any direct access. This would improve the readability and the maintainability allowing to add pre or post-actions very easily.

### 5.3.3 Data classes

Data classes are classes that hold information but don't really offer any service. This contradicts the encapsulation and abstraction principle. It reduces the cohesion of classes and makes the code more difficult to read.
We can detect these classes by comparing the number of methods (without accessing methods) and the number of attributes.
This can be done by computing the ratio between the number of attributes and the number of methods. We wanted to use 2 as a threshold. That way a class is considered Data class if it has twice more attributes than methods (without counting the accessors). But there was no class in Monticello with twice more attributes than methods, so we lowered the threshold to 1.75.
This way we find two classes, here are their respective blueprints :

MCChangeSelectionRequest 

MCVersionNameAndMessageRequest 

MCPseudoFileStream 

We can rule the first two of these classes out as data classes since they have a public method and not just their two attributes and their accessors.
But the third class is composed only of an attribute and its accessors. This third class inherits from the smalltalk ReadWriteStream class so Monticello must use it to represent a file stream, but it is used only on MCConfigurationBrowser. So a possible refactoring would be to remove this class, store its only attribute directly in MCConfigurationBrowser with a ReadWriteStream.

### 5.3.4 Methods with high parameter number

Giving a lot of parameters to a class can be difficult to read. A method receiving a lot of parameters is either doing too much work or the parameters are related, in that case they should be grouped in an object. This would augment the cohesion, increase readability and increase maintainability. Because if we decide to change the way we represent the object passed in parameters, there will be less change to make. We haven't really talked about this in our precedent rapport.

Finding methods with a lot of arguments can be done easily with the numberOfParameters metrics.
We are going to get all methods with a big number of parameters, for this we will use this query :

```
((self flatCollect:[:each|each classes]) flatCollect:[:each| each
   methods ]) select:[:each| each numberOfParameters >=  10]
```

We used a pretty high threshold (10) what we are going to discuss may be used with a lower threshold either. This query get us 4 methods all from MCCassDefinition. Here is one of the four

```
initializeWithName:superclassName:traitComposition:classTraitComposition:
   category:instVarNames:classVarNames:poolDictionaryNames:
   classInstVarNames:type:comment:commentStamp:
```

We directly see the problem of readability this causes. A manual analysis of the code taught us that these are initializers. This could be refactored by creating an object containing some of these parameters, we could for instance group all parameters with a relation to the variables together. This would make the code more readable and would augment the cohesion because we could add methods about the variables directly into this object.

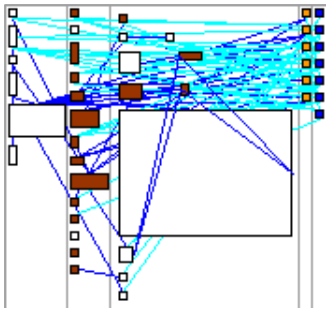
### 5.3.5 Multiple execution path methods

The cyclomatic complexity can represent the complexity of a method. The more paths through a method there are the more complicated it is. A method with high cyclomatic complexity is hard to read and then should be divided in smallest methods.
Reducing cyclomatic complexity could also avoid code duplication. For example if we have an if statement and the code executed is slightly the same for each case, moving it to a new method could remove duplicated (or almost duplicated) lines.
There is also a study that found a correlation between the cyclomatic complexity and the probability for the code to have bugs.
*"The results show that the files having a CC (cyclomatic complexity) value of 11 had the lowest probability of being fault-prone (28%). Files with a CC value of 38 had a probability of 50% of being fault-prone. Files containing CC values of 74 and up were determined to have a 98% plus probability of being fault-prone."*[3]
In conclusion the threshold can be set to 11.
This part wasn't analyzed in the previous report.
To find methods with too high cyclomatic complexity let us use this query :

```
((self flatCollect:[:each|each classes]) flatCollect:[:each| each
   methods ]) select:[:each| each cyclomaticComplexity  > 11]
```

That find us 3 methods. Analysing the code show it is not very readable. This is a flaw that could be arranged by separing big methods like this into smallest methods. Here is a blueprint of a class with on of these methods (MCMethodDefinition.scanForPreviousVersions()) that clearly show the too big method.

MCChangeSelectionRequest



### 5.3.6 Brain methods / classes

Brain methods are methods that do too much work. They are difficult to read and to modify and practically impossible to reuse. Theses methods have to be split in multiple other methods.
A brain class is a class with multiple brain method or a complex class with one brain method.
The technique to detect brain method consist of finding complicated methods, these are long methods and/or methods with a lot of branching.

**Long methods** We can use numberOfLinesOfCode here to find long methods

**Excessive branching** We can use numberOfConditionals to find theses

To find such methods we will use the fallowing query :

3. Rich Sharpe. "McCabe Cyclomatic Complexity : the proof in the pudding"

```
(((self flatCollect:[:each|each classes]) flatCollect:[:each| each
   methods ]) select:[:each| each numberOfLinesOfCode  > 20 ] ) select
   :[:each| each numberOfConditionals >= 10]
```

We here used threshold quite high but we should turn them quite small to find more results (numberOfConditionals set to 6 to get one more class). This query return one only method, the one discussed in the section "Multiple execution path methods". There is a clear correlation between these two techniques. Classes with high number of branchings will be detected with cyclomatic complexity and to have a high cyclomatic complexity it usually takes a high number of lines of code. This method was alredy found and analysed in our previous report.

### 5.3.7 God classes

God classes are classes that know too much and do too much. These classes should be separated into multiple smaller classes. This type of classes are very complex. That makes them hard to read and to maintain.

The detection technique we are going to use comes from the book "Object-Oriented Metrics in Practice" [4].
The principle of the god class detection is based on 3 ideas :

**High number of outer class accesses** A class that accesses a lot of other simpler classes is likely to do to much work and to be a god class.

**Complex** A god class is likely to be complex.

**Low class cohesion** The god class does work that should be done by others classes so its cohesion is low.

The 3 following conditions have to be matched :
- fanOut $>$ few
- weightedMethodCount $\geq$ very hight
- tightClassCohesion $< 1/3$

The query we use to find a god class is these :

```
(((self flatCollect:[:each|each classes]) select:[:each | each
   fanOut > 5]) select:[:each| each weightedMethodCount >= 200 ])
   select:[:each|each tightClassCohesion <(1/3)]
```

Here we used a fan out of 5 to represent few and 200 to represent a very high weighted method count. We didn't found any. To found a class we have do decrease the high weighted method count to 162 which is not so high. We'll then find MCWorkingCopyBrowser. An manual analysis of this class taught us that she has a lot of public methods but that is not a god class. We can say that Monticello doesn't include any god class.

### 5.3.8 Names

We can use metrics to detect names of attributes, methods or classes that are too long. It's interesting to seek these kind of items because a very long name is likely to convey too much information. We will filter the classes, attributes and methods by using a threshold on the value of their *nameLength*.

4. Lanza, M. and Marinescu, R. (2006) *Object-Oriented Metrics in Practice* Springer

If we set the threshold for class names to 25 characters we find 5 Monticello classes that have names longer than that :

- MCVersionNameAndMessageRequest : The "and" in the name might lead us to suspect this class from doing two different things, but its *tightClassCohesion* has a value of 0.6 so we can rule out that possibility.
- MCClassInstanceVariableDefinition
- MCRemovalPreambleDefinition
- MCInstanceVariableDefinition
- MCRemovalPostscriptDefinition

We can't easily refactor any these names to change them to shorter ones without losing in readability.

About attributes names, with a threshold of 20 characters we find the 3 following attributes :

- classTraitComposition in class MCClassDefinition
- classTraitComposition in class MCClassTraitDefinition
- unloadableDefinitions in MCPackageLoader

As for the class names these are long names but they are acceptable since we can't find any shorter name that conveys as much information.

About the methods *nameLength* can't really be used to filter the methods in the same way we used for the classes and attributes because it counts all the characters of the methods attributes as well since the method names include the parameters names. Because of this we can consider that the methods with long names have already been considered in the "Methods with high parameter number" section.

### 5.3.9 Cohesion

We can detect the classes with a lack of cohesion by using the *tightClassCohesion* metric from moose. From the definition of *tightClassCohesion* we know that it uses the relations between methods and attributes. So to be able to use it effectively we consider only classes that have at least one attribute and are not abstract.
We would like to use use as threshold of 0.33 which corresponds to the class having only one interaction out of three possible interactions between its methods. First we check that this threshold is relevant with the Monticello system. We made the following chart representing the distribution of selected classes depending on the value of their *tightClassCohesion* :

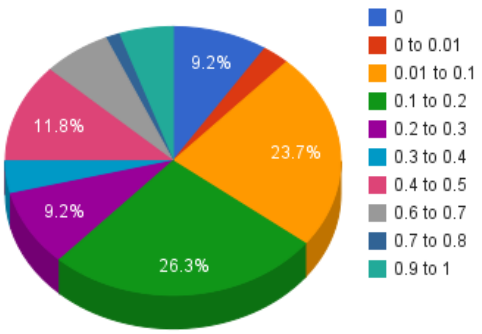



**Figure 3:** Distribution of classes for different ranges of tightClassCohesion values

We notice that we have a majority of the classes that will be under the chosen threshold of 0.33. But there are still 29% of the classes that can be eliminated by applying a filter that only keeps the classes with $tightClassCohesion < 0.33$.

# 6 Conclusions

From our analysis on the bad smells using moose metrics we found two major flaws.
The first one is the fact that there is some duplicated code. It makes the code both more difficult to read and less maintainable. However at least the appearance of duplicated code was very scarce. The second flaw is the high number of direct accesses to attributes. These happen mostly internally to classes but it is still a problem in regards to maintainability. It might really improve the quality of the Monticello system if every attribute from classes was used only through accessors. This would improve the maintainability.

The finding of such flaws is quite a step from the manual code analysis. In the first report our conclusion about the system were that it was nearly perfect. Off course we only were able to analyze random fractions of the code. The use of automated analysis through metrics allowed us to cover the whole code with a set of filters which allowed us to spot flaws more efficiently.