

## Chapter 1

# Analyzing Software using Moose

Moose is a platform for data and software analysis. Moose is the result of a collaborative effort, initiated at the University of Bern, which now comprises several companies and research groups spread all over the world. The website of Moose is <http://moosetechnology.org>.

Visualization plays an important role in assessing the quality of a software. The source code defining a software may be very large and is characterized using a wide range of metrics and structural properties. Identifying anomalies, and more importantly, proposing actions to improve the quality is difficult. It is known that visualization greatly alleviates software maintenance.

Moose is often used to craft specific software engineering tools that guide engineers in making proper decisions about the course of a software development. Humane Assessment is a method designed by Tudor Gîba and largely described on the website <http://www.humane-assessment.com>.

*What Moose is good for:* Moose is made to carry out analysis on the source code structure. A whole range of analyses may be conducted by analysis source code. Moose is also frequently used to analyze configuration files, process descriptions, or mailing lists.

*What Moose is not good for:* Moose cannot directly be used to analyze the dynamic execution of an application. Dynamic analysis consists in recording relevant information during a program execution. Such information may then be used to spot particular behavior.

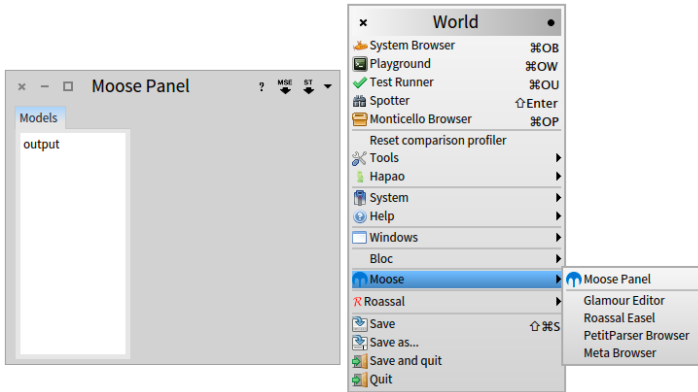


Figure 1.1: Opening the Moose Panel.

## 1.1 Moose in a nutshell

The core of Moose is composed of:

- *Panel*: An extensible tool to navigate, visualize, and formulate queries.
- *Parsers*: Two parsers are shipped with Moose. MSE is a file-format describe source code models. MSE is typically produced by externally provided parsers.
- *The Famix metamodel*: An abstract representation of source code. Famix is generic and it may describe applications written in Java, C, C#, and Pharo.
- *Chef*: A domain-specific language to query structural dependencies. Analysis the dependencies between software elements is particularly difficult. Chef allows one to precisely formulate queries

## 1.2 Loading a software application

Creating a model in Moose is probably the very first step to take. Such a model will contains all the software elements on which your analyse will be carried out. The standard way to create a model is to load an application. To do so, the Moose panel has to be open. The panel is accessible from the World menu (Figure 1.1).

Moose has two built-in parsers: one for Pharo, and another for MSE. The first parser is accessible by clicking on the **ST** icon, top-right of the Moose Panel. Clicking on it will open a package selector. You simply have to select the Pharo packages you wish to include in your model. The second parser takes as input a MSE file, accessible by clicking on the **MSE** button.

Consider the following Java code:

```
package agilevisualization.example;

public class HelloWorld {
    public static void main(String[] argv) {
        System.out.println("Hello World");
    }
}
```

The corresponding MSE file of that code is 48 lines long. The first few lines are:

```
(
  (FAMIX.Inheritance (id: 3)
    (subclass (ref: 10))
    (superclass (ref: 4)))
  (FAMIX.Class (id: 4)
    (name 'Object')
    (container (ref: 7))
    (isStub true))
  (FAMIX.FileAnchor (id: 5)
    (element (ref: 1))
    (endLine 6)
    (fileName './agilevisualization/example/HelloWorld.java')
    (startLine 4))
  (FAMIX.FileAnchor (id: 6)
    (element (ref: 10))
    (endLine 7)
    (fileName './agilevisualization/example/HelloWorld.java')
    (startLine 3))
  ...
)
```

Producing a MSE file from Java source code is done using VerveineJ. Unfortunately, at the time this chapter was written, VerveineJ has a proprietary license. Contact the book author or the Moose mailing list for more information.

Along this chapter, we will use a running example, obtained from the Java application checkstyle. Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. The source code of Checkstyle is available from its website: <http://checkstyle.sourceforge.net>.

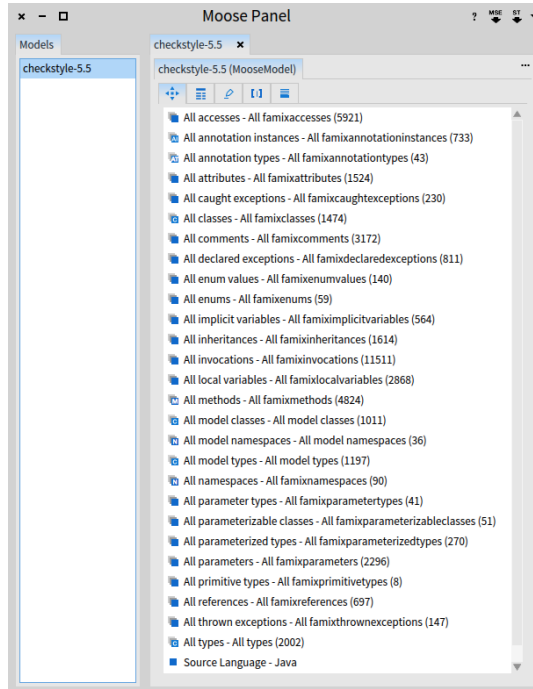


Figure 1.2: Loading checkstyle.

net. We have produced a MSE file, available on <http://bit.ly/checkstyle>.

### 1.3 Browsing the source code

After having loaded the checkstyle MSE file, a new model is listed in the Moose (Figure 1.2).

Selecting the checkstyle model in the panel lists the contained entities. The panel indicates a few interesting facts about Checkstyle, for example, it is composed of

- 4,824 methods (item *All methods*)
- 1,011 classes (item *All model classes*)
- 36 packages (item *All model namespaces*)

The item *All classes* lists all the classes defined in the application and all

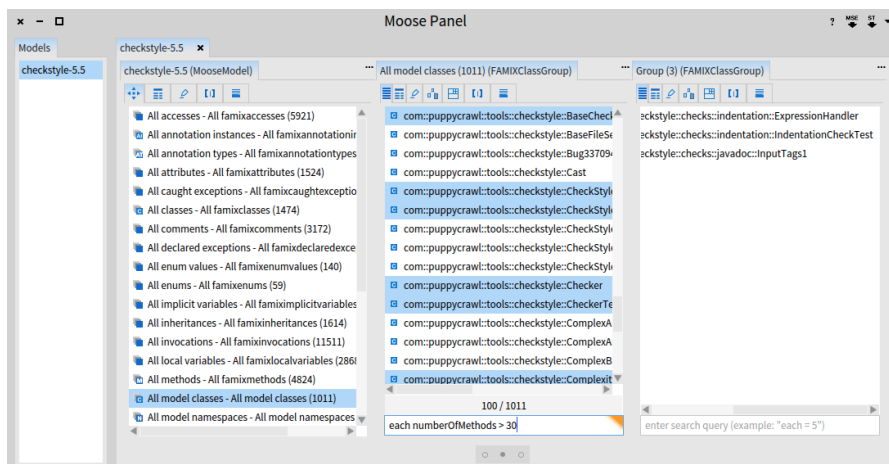


Figure 1.3: Querying in the panel.

the classes used by checkstyle that are not defined in it. Some examples of such classes are `Object`, `String`, `Integer`, which are defined in the Java runtime.

Select the *All model classes* item to list all the classes defined in the applications. Below the list of classes, a text input accepts filtering queries. Enter `each numberOfMethods > 30`. A third pane appears and lists the classes defined in checkstyle that have more than 30 methods (Figure 1.3).

Since the items displayed in the panel *All model classes* are instances of the class `FAMIXClass`, the each variable refers to an `FAMIXClass`. Browsing this class using the Pharo code browser will give you the list of methods that are understandable by the each variable.

Queries involving the class name are also often formulated. For example, the query

```
'*Log*' match: each name
```

designates classes defined in Checkstyle that have the word `Log` in their name. The string being an expression regular, matched using `match:`.

Another relevant query to type in is: `each isStub`. You need to select the item *All classes* instead. This simple query filters out all the classes that belongs to the checkstyle application, to leave the classes are used by Checkstyle. Said in other words, all the externally provided classes used by Checkstyle are remains. This is handy to assess external dependencies. In the Moose jargon, a stub class is a class that is used by the analyzed application but not defined in it. Checkstyle contains 398 stub classes, corresponding to classes defined in other libraries, not part of the `.MSE` file. For example,

Checkstyle depends on some libraries provided by the Apache foundation or Google. These libraries are not part of the .MSE file.

Query may be relatively complex. For example, consider:

```
each numberOfMethodsOverridden > (each numberOfMethodsInherited / 3)
```

This query returns the classes that override more than 1/3 of the classes that it inherits. Such classes are rather suspicious since it may be worth considering whether they belong to the adequate class hierarchy.

## 1.4 Visualizing the code distribution

Checkstyle contains more than one thousand classes, distributed over 36 packages. Understanding the code distribution over these packages may involve a dedicated visualization. After having selected the checkstyle model, select the packages by pressing *All model namespaces*. The new pane located on the right-hand side lists these packages. Oddly enough, the packages <Default Package> and blah are listed.

Pressing the [[]] tab on the right pane allows one to enter a script. The self variable is then bound to the collection of packages. Enter the script (Figure 1.4):

```
b := RTMondrian new.
b nodes: (self reverseSortedAs: #numberOfClasses).
b layout flow.
b normalizer
  normalizeWidth: #numberOfClasses max: 100;
  normalizeHeight: #numberOfLinesOfCode max: 100.
b
```

This simple visualization shows that not all the packages contain the same amount of code. The first package, the largest, is com.puppycrawl.tools.checkstyle. Clicking on the package opens up a new pane. The second tab gives the list of properties for that package. It indicates that it defines 142 classes, spread over 5,617 lines of code.

Clicking on the entry *Types* lists all the classes defined in the package checkstyle. The following script reveals a number of facts regarding these 142 classes (Figure 1.5):

```
b := RTMondrian new.
b shape rectangle
  width: #numberOfAttributes;
  height: #numberOfMethods.
b nodes: (self reverseSortedAs: #numberOfMethods).
b edges connectFrom: #superclass.
```

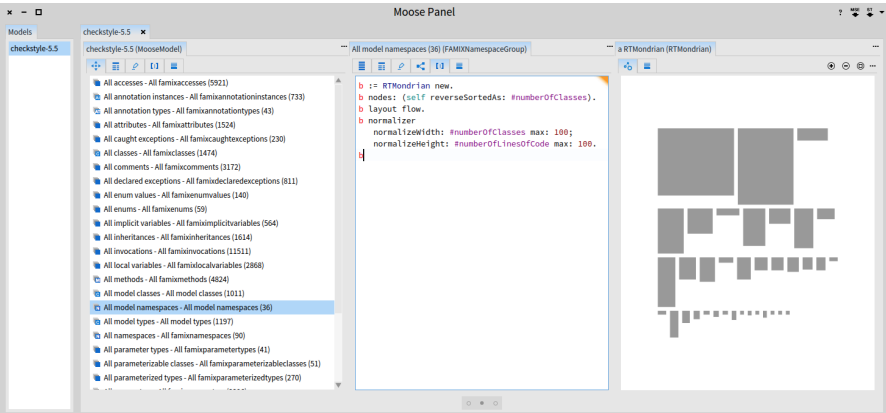


Figure 1.4: Querying in the panel.

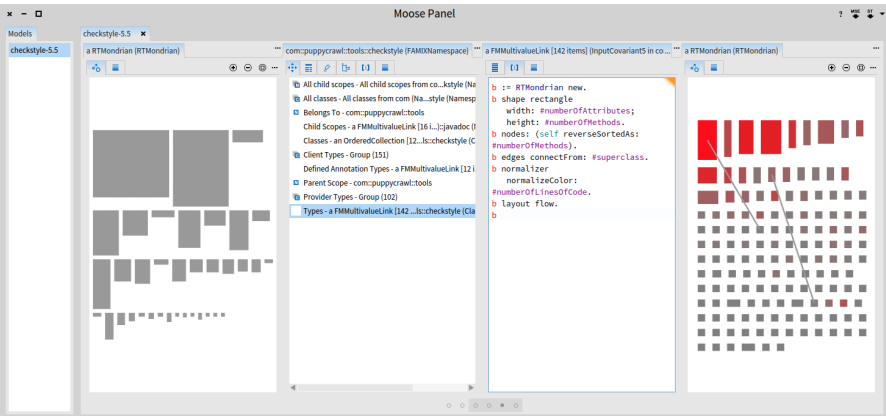


Figure 1.5: Classes in the largest package.

b normalizer  
    normalizeColor: #numberOfLinesOfCode.  
b layout flow.  
b

Most of the classes are small, in terms of lines of code, number of defined methods and attributes. Moreover, only two of these classes are subclasses.

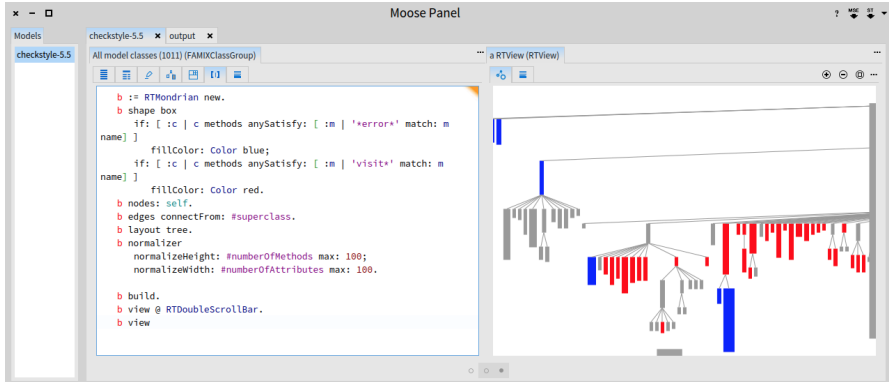


Figure 1.6: Visualizing cross-cutting concerns.

## 1.5 Visualizing cross-cutting concerns

Cross-cutting features are features that crosscut software structural entities, such as classes or packages. Due to their nature, cross-cutting are not explicitly modularized, and may therefore lead to some code anomalies. Consider the following script executed on the list of model classes (Figure 1.6):

```
b := RTMondrian new.
b shape box
  if: [ :c | c methods anySatisfy: [ :m | '*error*' match: m name ] ]
    fillColor: Color blue;
  if: [ :c | c methods anySatisfy: [ :m | 'visit*' match: m name ] ]
    fillColor: Color red.
b nodes: self.
b edges connectFrom: #superclass.
b layout tree.
b normalizer
  normalizeHeight: #numberOfMethods max: 100;
  normalizeWidth: #numberOfAttributes max: 100.

b build.
b view @ RTDoubleScrollBar.
b view
```

The script colors in blue classes that have at least one method with the word error in it and in red classes that are visiting some entities. The hierarchy of red classes indicates a specialization of visitors. Notice that that errors are rarely handled.



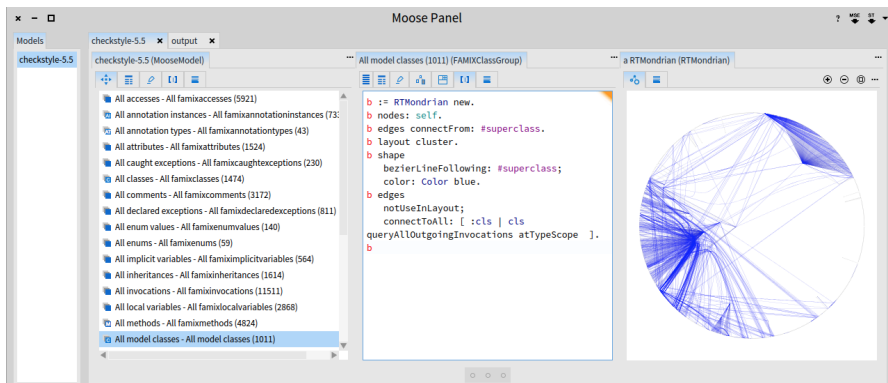


Figure 1.7: Dependencies between classes.

## 1.6 Dependencies using Moose Chef

Moose chef is an API allowing one to query dependencies between packages, classes, and methods. Consider the following script, running on the list of model classes (Figure 1.7):

```
b := RTMondrian new.
b nodes: self.
b edges connectFrom: #superclass.
b layout cluster.
b shape
  bezierLineFollowing: #superclass;
  color: Color blue.
b edges
  notUseInLayout;
  connectToAll: [ :cls | cls queryAllOutgoingInvocations atTypeScope ].
b
```

Sending the message `queryAllOutgoingInvocations` to a Famix class returns a list of Famix invocations. An invocation is an instance of the Famix class `FAMIXInvocation`. Invocations represent a low level abstract that often need to be transformed. Such transformation is called setting a scope by Moose Chef. The invocations may be scoped to the classes by sending `atTypeScope`. For a given Famix class `cls`, the expression `cls queryAllOutgoingInvocations atTypeScope` returns the list of Famix classes that `cls` depends on. A Chef query may be scoped using `atNamespaceScope`, `atPackageScope`, `atTypeScope`, `atMethodScope`.

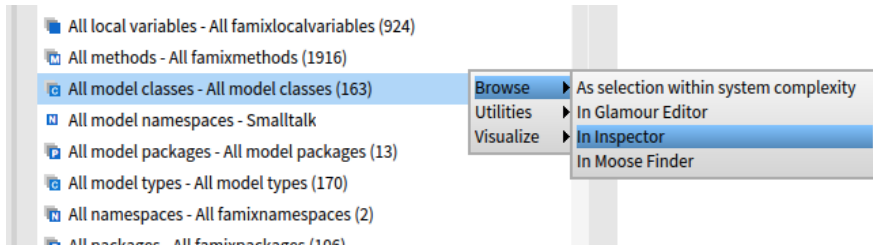


Figure 1.8: Inspecting a class group.



Figure 1.9: The pencil icon to open a system browser.

## 1.7 Reusing visualization

So far, we have defined some visualizations within the tab `[[`. This allowed us to play with and tune the visualization as we need. Once satisfied with a visualization, you may want to install it permanently, quickly launch the visualization for a different set of classes.

One easy way to install a visualization, is to create a method on the class describing the group you are interested in. For example, if you wish to make a visualization for a group of classes, then `FAMIXClassGroup` is probably the class you wish to install the visualization in.

First, you need to inspect the class behind the group you are interested in, as shown in Figure 1.8. This opens a new window titled `Inspector` on a `FAMIXClassGroup`. Click on the pencil icon (Figure 1.9).

Define the method on the class `FAMIXClassGroup`:

```
FAMIXClassGroup>>viewDependencies
<menulitem: 'Dependencies' category: 'Visualize'>
| b |
b := RTMondrian new.
b nodes: self.
b edges connectFrom: #superclass.
b layout cluster.
b shape
  bezierLineFollowing: #superclass;
  color: Color blue.
```

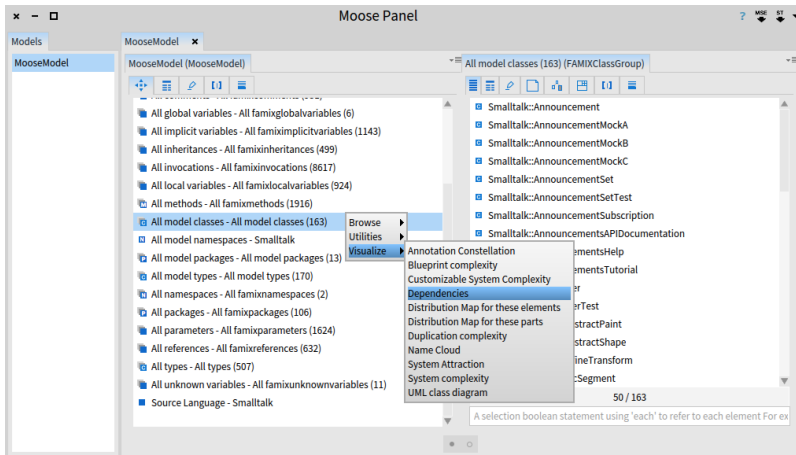


Figure 1.10: A new entry has been added in the sub menu Visualize.

```

b edges
  notUseInLayout;
  connectToAll: [ :cls | cls queryAllOutgoingInvocations atTypeScope ].
b inspect

```

After defining this method, right clicking on the All model classes item in the Moose Panel gives a new menu entry called Dependencies (Figure 1.10):

## 1.8 More on that topic

Moose is a large project that contains many exciting facets. This chapter only superficially highlights some of them. Another good source of documentation is the Moose Book (<http://www.themoosebook.org/book/>), project lead by Dr. Tudor Gîrba <http://www.tudorgirba.com>.