MooseGager, a Software Metrics Tool based on Moose

Student Project

Author

Thomas Bühler

October 2003

Supervised by:

Dr. Michele Lanza
Prof. Dr. Oscar Nierstrasz

Institut für Informatik und angewandte Mathematik
Universität Bern

The address of the author:

Thomas Bühler
Riedliweg 43
CH-3053 Münchenbuchsee
buehler@iam.unibe.ch

Abstract

Moose is a tool environment to reverse engineer and reengineer object-oriented systems. One feature of this environment is to compute software measurements based on the underlying FAMIX model.

A problem of this service was that many measurements were computed but could not be used in an efficient manner because they were not presented to the user.

The solution to this problem is a tool that displays the computed measurements using a graphical user interface.

In this project, we developed the tool MooseGager. This tool displays the computed measurements of the entities of the underlying model in a simple way and also offers the possibility to generate charts based on these measurements. These and other features of this tool provide an interface to the *Moose* reengineering environment that helps the user to use the available measurements efficiently.

Contents

Chapter 1

Introduction

The maintenance, reengineering, and evolution of software systems has become a vital matter in today's software industry. The law of software entropy dictates that most systems with time tend to gradually decay in quality, unless they are maintained and adapted to the evolving requirements [LANZ 03]. So, in a software development process, it is necessary to have a way to assess the software system and its quality. One approach to do so is the use of software metrics since they are, according to Fenton [FENT 96], a good means to control the quality and the state of a software system.

1.1 Software Metrics

Formally, metrics measure certain properties of a software system by mapping them to numbers (or to other symbols) according to well-defined, objective measurement rules. The measurement results are used to describe, judge, or predict characteristics of the software system with respect to the property that has been measured. Generally, software metrics can be divided in two groups [LORE 94]:

1. **Design metrics** are measurements of the static state of the project's design. Design metrics are used to assess the size and in some cases the quality and complexity of software. They tend to be more locally focused and more specific, thereby allowing them to be used effectively to directly examine and improve the quality of the product's components.

2. **Project metrics** deal with the dynamics of a project, with what it takes to get to a certain point in the development life cycle and how to know you're there. They can be used in a predictive manner, for example to estimate staffing requirements. Being at a higher level of abstraction, they are less prescriptive and more fuzzy but are more important from an overall project perspective.

In this work, we focus on design metrics. Furthermore, we focus on direct measurement metrics, *i.e.*, metrics that can be computed from the source code without using any other kind of information.

1.2 Good Software Design

The main goal of this project is to provide a software metrics tool that measures aspects of the design of object-oriented software and displays the measurements to help understand how the system is built and assess its design quality which is, as mentioned before, the first step in making an attempt at improving the quality of the system. Before we can assess the design quality of a system, we have to understand what makes a design good or bad. Coad defines a good design as follows:

> "A good design is one that balances trade-offs to minimize the total cost of the system over its entire lifetime." [COAD 91]

Thus, a good design is reflected by the minimization of costs, *i.e.,* the costs of creating the design, transforming it into a proper implementation, testing, debugging, maintaining, and improving the system. Coad also emphasizes the fact that from the formerly mentioned cost categories, the most substantial one is related to maintenance. Therefore, he concludes: "The most important characteristics of a good design is that it leads to an easily maintained implementation". Summarizing Coads points, we can say that it is important for a high-quality design that it is easy to understand, easy to implement, testable, and especially modifiable. But how can we recognize such a design and what can we do to make a good design or improve a design? In his PhD thesis "Measurement and Quality in Object Oriented Design"[MARI 02], Marinescu came up with the following two points:

- It is hard to comprehend and quantify the "goodness" of a design by itself; therefore we have to apply the biblical principle: "by their fruit you will recognize the", *i.e.,* we can get an understanding of the quality of the design only by regarding its "fruits": testing efforts, maintenance costs, and the number of reusable fragments.
- We need criteria for evaluating a design not in order to build "perfect" software but to help us avoid badness. Therefore, good design is a matter of avoiding those characteristics that lead to bad consequences [COAD 91].

These conclusions tell us that in order to evaluate an object oriented design, we can either use certain measurements such as maintenance costs, testing efforts or the number of reusable fragments or we can use heuristic knowledge to avoid badness. In this work, we focus on the use of heuristic knowledge.

1.3 Heuristic Knowledge

Marinescu [MARI 02] emphasizes that it is impossible to establish an objective and general set of rules that would lead automatically to high-quality design. On the other hand, heuristic knowledge reflects and preserves the experience and quality goals of the developers and also help the beginners evaluate and improve their design. So, heuristic knowledge is what finally allows us to assess the design quality of a software system by comparing certain measurements to measurements that are generally considered as good. The tool we implemented in this student project, MooseGager, provides measurements that then be compared and, together with heuristic knowledge be used to assess the quality of a system.

According to Erni and Lewerentz [ERNI 96], quality assurance in software design is achieved by exploiting design heuristics. For different design methods, guidelines for good design and programming style have evolved over time. Another term for these guidelines are design heuristics (used by Riel [RIEL 96]) or design rules (used by Erni and Lewerentz). There are different approaches to detect violations of design rules using object-oriented design metrics. In the classical approach, one single metric is applied to a complete software system to get statements about a certain quality criteria, *e.g.*, in combination with threshold values. Another approach is provided by Erni and Lewerentz. These authors model design-rules by putting together multiple metrics to multi-metrics. A multi-metric is a set of metrics that are all related to the same component. It describes a function that produces a value-set by applying the multi-metric to a component. Thus, various metrics are applied to one component (*e.g.*, a class or a method) of a system to get an extensive idea of its quality. In this work, we try to provide simple applications of both of these approaches: we provide the possibility that a user can inspect entities with all their measurements and select entities based on single or multiple metrics.

1.4 Structure of this document

In this student project, we developed the metrics tool MooseGager that is based on the *Moose* reengineering framework. A description of the features of MooseGager is provided in the next chapter. The following chapters describe the implemented software metrics and the implemented design flaw detection methods. In the last chapter, we present two applications of the tool MooseGager on two different case studies.

Chapter 2

Description of MooseGager

Moose is a tool environment to reverse engineer and reengineer object-oriented systems. It consists of a repository to store models of software systems using the FAMIX meta-model [DEME 01] (representing software artifacts such as classes, methods, etc.), and provides query and navigation facilities. Based on this environment we implemented the tool MooseGager. The main goal of this new tool is to provide an easy-to-use interface that allows the compute measurements for reverse engineering and reengineering purposes. The features of MooseGager are presented in the following Section 2.1. Section 2.2 describes the implementation of two central parts of MooseGager: Its core and its flaw detection framework.

2.1 Features

MooseGager has the following features:

Computation of Measurements. MooseGager computes several metrics (see Chapter 3). Most of these metrics are project summaries of metrics that are computed by the *Moose* reengineering framework. These metrics are presented in the overview window which is part of the *Moose* launcher now (shown in Figure 2.1). This overview canvas is also part of MooseGager's main window (shown in Figure 2.2).

Presentation of Classes and Methods. MooseGager presents the classes and methods in a table. The user can add and remove rows to this table by choosing a metric. Each row represents then the measurements of the entities of the selected metric. The table is sortable by its rows. In addition to the blank tables containing no measurements, there are several metric tables predefined in MooseGager. Each of these predefined tables shows one metric and is sorted by this metric by default.

Distribution Charts. MooseGager can generate distribution charts based on these tables. An example is shown in Figure 2.3. It shows the number of methods on the y-axis and the number of statements and the lines of code metrics on the x-axis. The user can chop these charts at a certain value. The chart contains then one bar above this value that shows the number of entities with a higher measurement than the value. The user can also summarize a certain amount of bars into one bar. As an example,

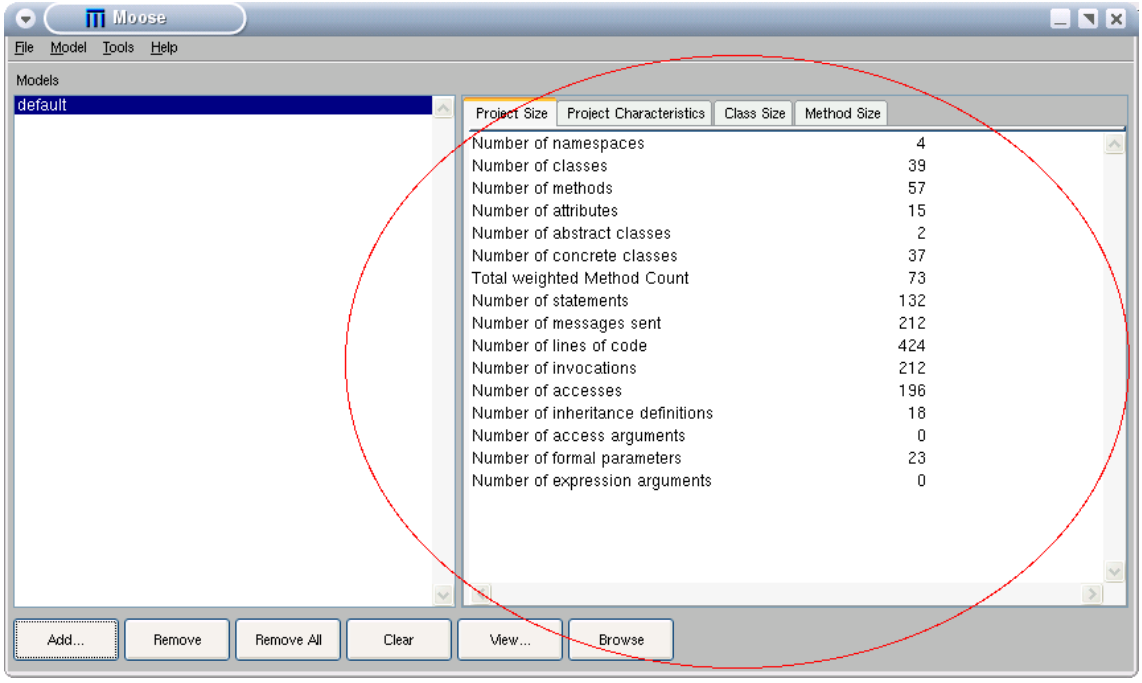

Figure 2.1: *Moose* Launcher with MooseGager's overview canvas on the right side.

the user chooses to summarize a chart by five. The first bar in the resulting chart represents then the first five bars, the second bar the bars number six to ten, and so on. The user can choose whether the values of the bars are displayed or not.

Correlation Charts. MooseGager can also generate **correlation charts**. The user can choose two metrics for each axis of the chart. One is then shown on the x-axis while the other is shown on the y-axis. An example correlation chart is provided in Figure 2.4.

Inspecting the Entities. To have a closer look at a certain entity, MooseGager provides several possibilities: First, the user can open a window which shows all computed measurements of the entity. The user can also directly open a Smalltalk-inspector on the underlying FAMIX-entity. The third provided possibility is to open a regular Smalltalk-browser on the selected entity. If the selected entity is a class, then the user can also directly open *CodeCrawler's* [1] *Class BluePrint* [LANZ 03] on that class.

Opening CodeCrawler's System Complexity View. MooseGager can directly open the *CodeCrawler's System Complexity View* on the entities in scope. This also works the other way round: The user can open MooseGager from a selection of entities in *CodeCrawler*.

[1] http://www.iam.unibe.ch/∼lanza/CodeCrawler/codecrawler.html

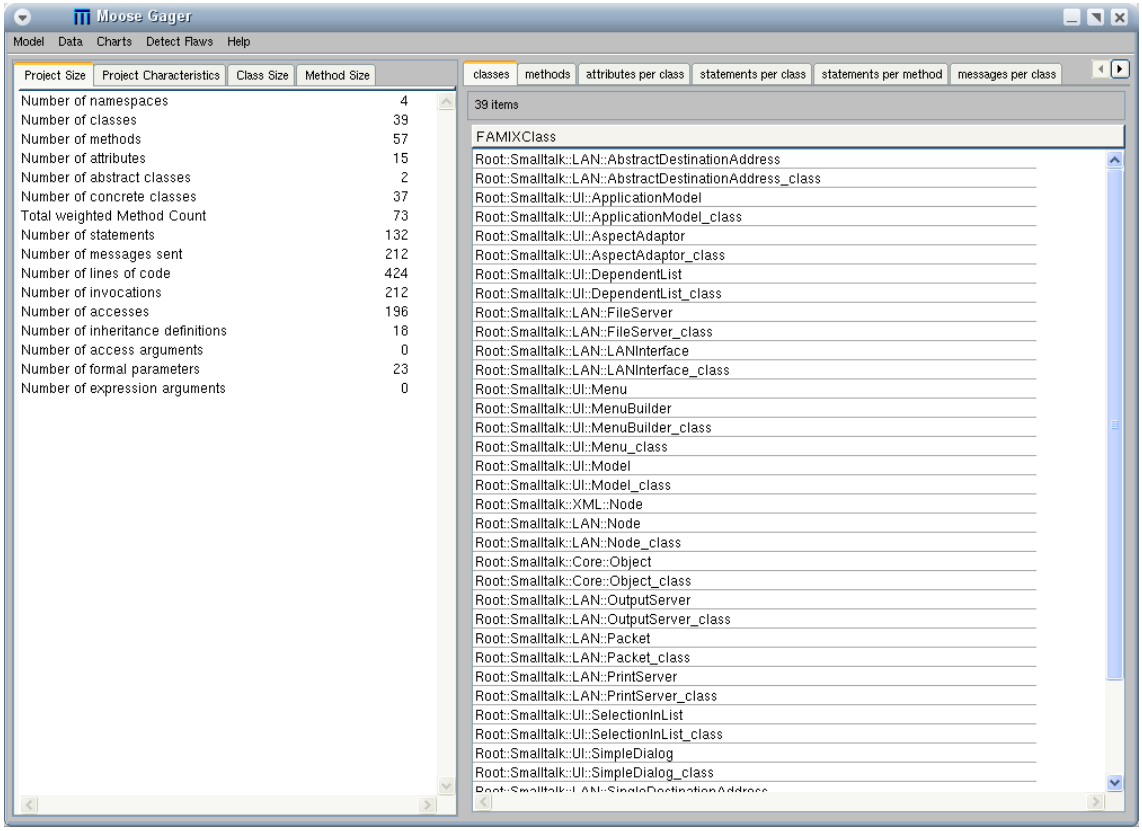

Figure 2.2: The Main Window of MooseGager.

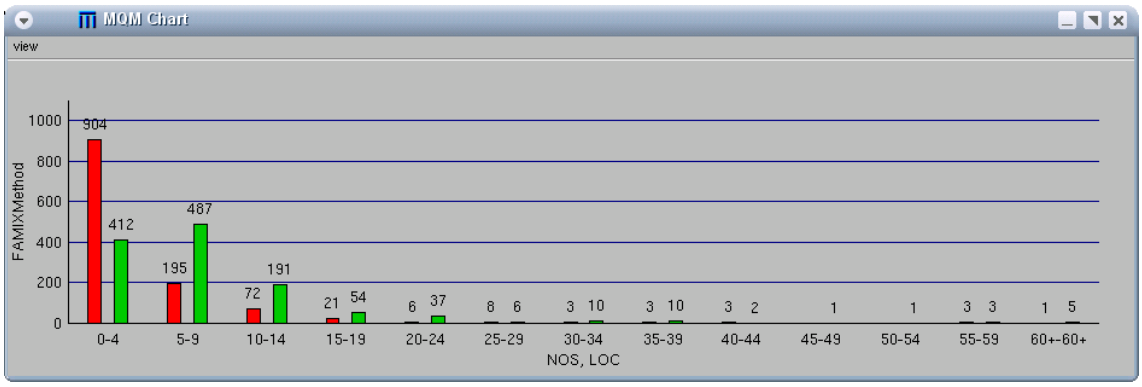

Figure 2.3: Sample distribution chart.

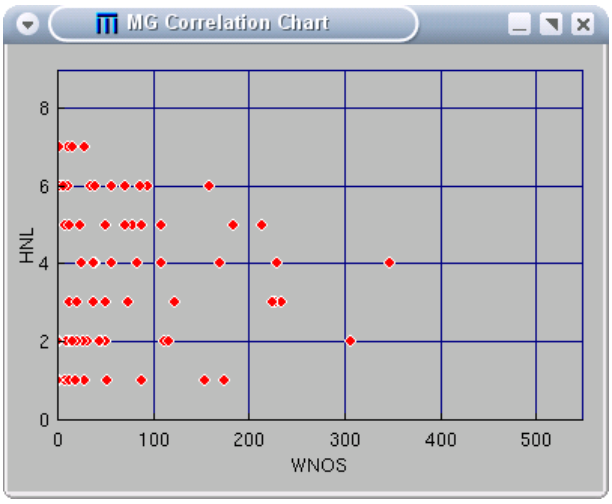

Figure 2.4: Sample Correlation Chart.

Queries. MooseGager offers a user interface to make queries using upper and lower threshold values of certain metrics. As an example the user can detect all classes that have less than four statements or more than twenty statements. To do this the user has to create a detector by choosing a metric and entering the lower and upper threshold values. The user can also combine created detectors with logical operators. As an example, it is possible to create a detector that detects methods that have more than seven lines of code and more than seven statements. This new detector that combines other detectors can again be combined with other detectors.

Flaw Detection. MooseGager also offers the possibility to detect design flaws using certain heuristics (see Chapter 4).

Metric Inspector. MooseGager provides also a **metric inspector**. This tool shows all defined metrics and displays information about the selected metric. A screenshot of this tool is provided in Figure 2.5.

Dropping Moose Stubs and Meta-Classes. The entities imported by *Moose* generally provide stubs and meta-classes. Often, these entities are not of interest. The user can simply remove them from the classes in scope.

2.2 Implementation

This Section contains a brief description of the implementation of two parts of MooseGager so the tool can be understood and further developed. Section 2.2.1 explains the core of MooseGager while Section 2.2.2 explains the flaw detection framework.

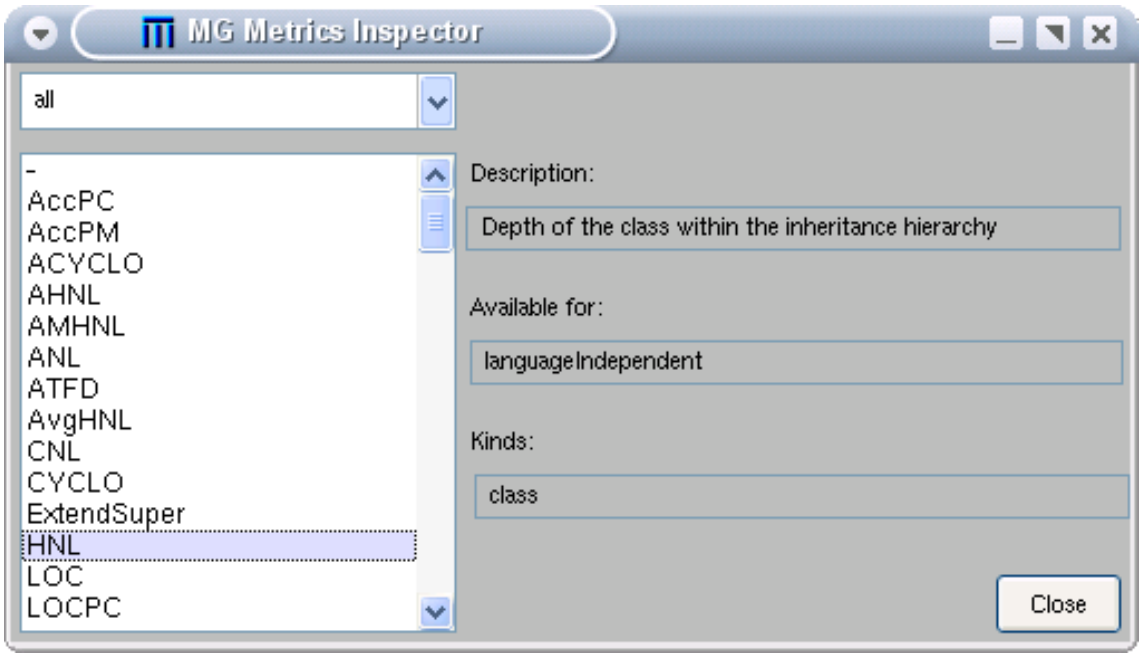

Figure 2.5: The Metric Inspector of MooseGager.

2.2.1 The Implementation of the Core of MooseGager

The UML diagram in Figure 2.6 shows the core classes of MooseGager. The most important is the *ContextualMetricOperator* class. It stores measurements computed by the *MGMetricsFacade* in its underlying model which can be either an *MSEModel* or an *MSEAbstractGroup*. This operator is executed each time the user imports a new model into *Moose* and each time a new *MGModel* is created to ensure that the necessary values are present.

The main responsibility of an object of the class *MGModel* is to hold a collection of FAMIX entities. To select specific kinds of entities such as classes, methods or attributes out of this model is the responsibility of the class *MGQueryFacade*. The class *MGMetricsFacade* is based on this feature. It computes the metrics that are defined in MooseGager (see Chapter 3) using a *MGMetricsFacade*. One of the MooseGager's benefits is that the displayed measurements are not computed each time they are displayed but are stored in the underlying model.

2.2.2 The Implementation of the Flaw Detection Framework

The flaw detection framework of MooseGager detects entities of a system that are possibly defective using object-oriented software metrics. Up to now this framework is not more than a basic approach but it could be enhanced further to a facility to detect design flaws using heuristics.

The UML diagram in Figure 2.7 shows the classes of the flaw detection facility of MooseGager.

The root class of all flaw detectors is the class **FlawDetector**. It has several subclasses that operate independently:

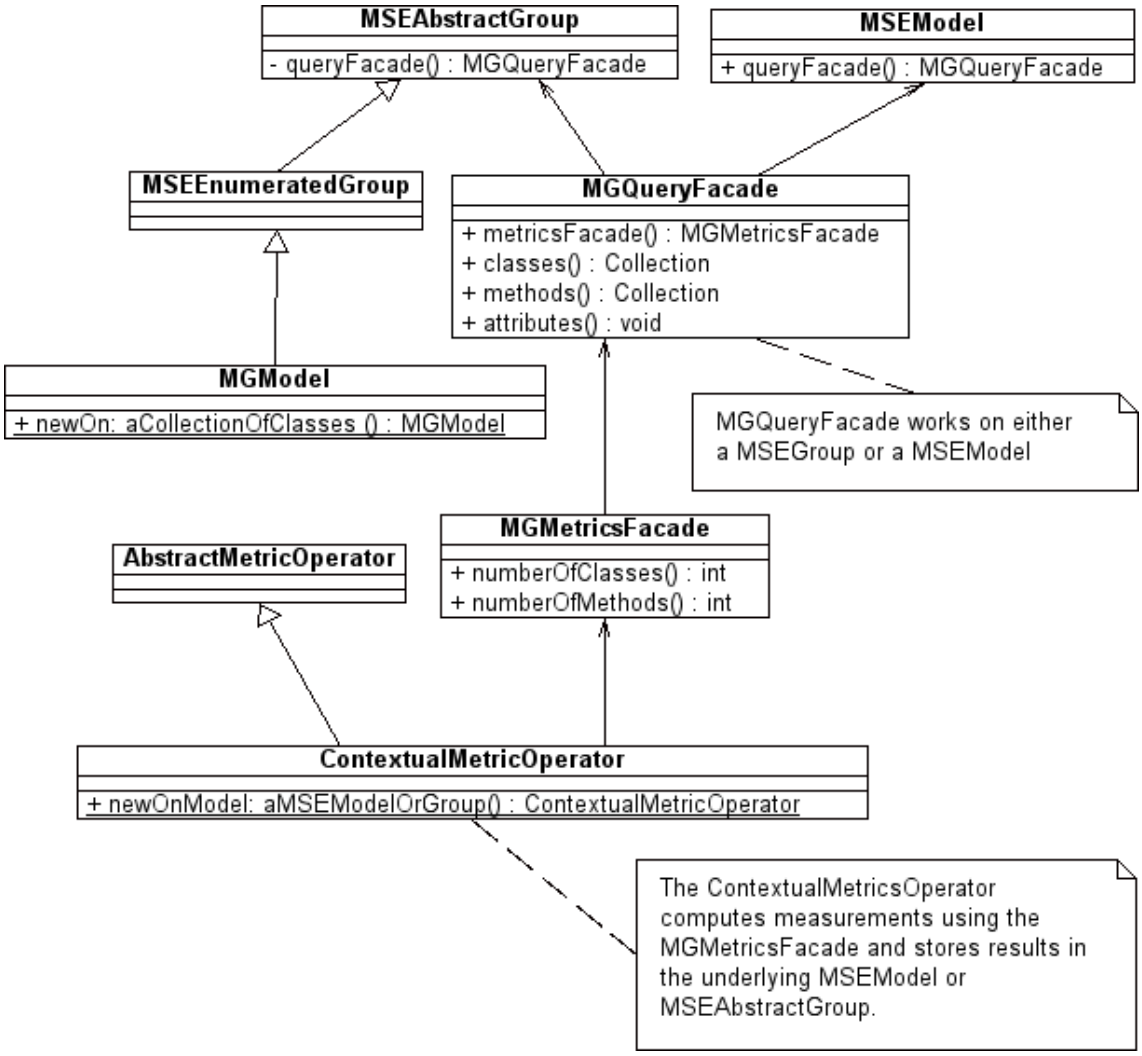

Figure 2.6: UML class diagram of the core of MooseGager.

- The class **BypassedAccessorDetector** computes accessor methods that are implemented but bypassed (see Chapter 4). It was not possible to implement this feature in a language independent way. Currently, this operator is based on the parse tree generated by the *Moose* class *VisualWorksImporter*.
- The class **MisplacedMethodDetector** detects methods that are with a high probability misplaced (see Chapter 4).
- The class **NOPOverrideDetector** detects method overrides that do not perform any operations.
- The class **SingleMetricFlawDetector** detects flaws using a single metric, a lower and an upper threshold. It has two concrete children: the class **SingleMetricMethodFlawDetector** that operates on methods and the class **SingleClassMetricFlawDetector** that operates on classes.

Instances of any of these detectors can be combined using either an **AndCombinedFlawDetector** or

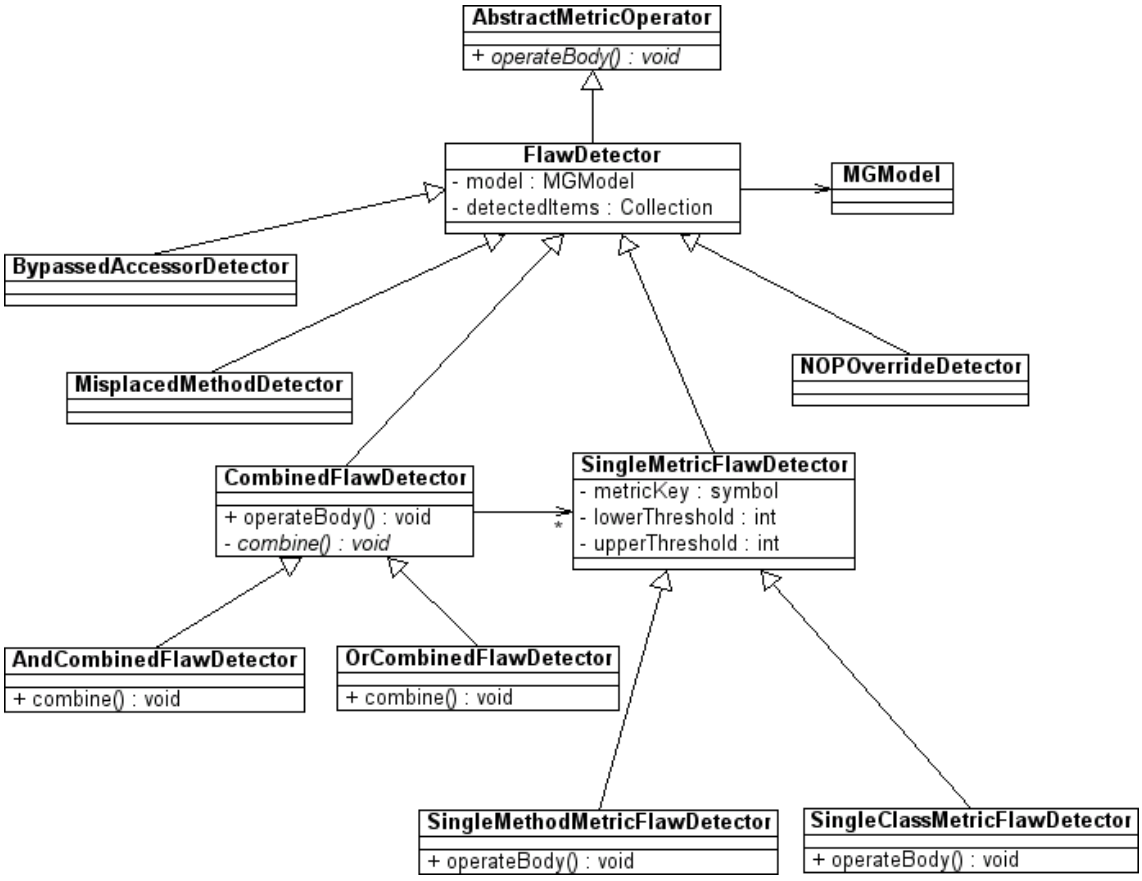

Figure 2.7: UML class diagram of MooseGager's flaw detector framework.

an **OrCombinedFlawDetector**. The *AndCombinedFlawDetector* combines detectors with a logical and while the *OrCombinedFlawDetector* combines detectors with a logical or. MooseGager provides a user interface that allows the user to create flaw detectors, combine them logically, and apply them on the current MooseGager model.

Chapter 3

Software Metrics in MooseGager

The reengineering environment *Moose* already computes a large number of metrics. Most of them are based on only single entities of the underlying model (such as classes, methods and attributes). For the tool MooseGager, we added a set of metrics. Most of them summarize measurements computed by the *Moose* framework of the entities (such as classes and methods) in order to provide information about the the whole system. The metrics we implemented in this project are grouped in four categories:

1. The metrics shown in Table 3.1 give the user an idea of the **size of the software system** in scope.
2. The second category (presented in Table 3.2) contains metrics to assess the **average class size**. The average class size gives, according to Lorenz and Kidd, information about how well the intelligence and workload is distributed in the system and how many relationships to other classes the classes have.
3. The third category contains metrics to measure the **average method size** (presented in Table 3.3). According to Lorenz and Kidd [LORE 94], the average method size is one indication of the quality of the design from an object-oriented perspective. Larger numbers indicate a higher likelihood that function-oriented code is being written. Smaller numbers indicate a higher likelihood that object-oriented code is being written. A growing average method size is another indicator that the design is not as good as it could be.
4. The metrics presented in Table 3.4 give a **characteristic** of the system.

MooseGager does not directly provide interpretations of the measurements.

Metric Name	Key	Description, Comments
Number of namespaces	NONS	
Number of classes	NOCl	
Number of abstract classes	NOACl	
Number of concrete classes	NOCCl	
Number of methods	TNOM	The total amount of methods of the whole analyzed system.
Number of attributes	NOAtt	The sum of all instance variables in all classes together. Note that in *Moose*, the number of attributes of a class does not include the attributes of its super-classes.
Total number of statements	TNOS	The total amount of statements summed over all methods of all classes in scope.
Total number of messages sent	TMSG	The total number of messages sent in the whole system.
Total number of lines of code	TLOC	
Number of inheritance definitions	NOID	The total number of inheritance relationships of all classes in scope.
Total number of invocations	NOI	
Total number of accesses	NOAcc	
Total number of access arguments	NOAA	
Total number of formal parameters	NOFP	
Total number of expression arguments	NOEA	
Total weighted method count	TWMC	Chidamber and Kemerer introduced in [CHID 91] the metric *Weighted Method Count* (WMC) as the sum of the cyclomatic method complexity (defined in *Moose*) of all the methods in a class. The metric *Total Weighted Method count* is the sum of the weighted method count over all classes of the system.
Number of constructor methods	NOCM	
Number of accessor methods	NOAccr	
Number of abstract methods	NOAM	

Table 3.1: Project size metrics

Metric Name	Key	Description, Comments
Number of methods per class	NOMPC	According to Lorenz and Kidd [LORE 94], this metric relates to the amount of collaboration being used.
Number of messages sent per class	MSGPC	
Number of lines of code per class	LOCPC	
Number of statements per class	NOSPC	
Number of attributes per class	NOAttPC	According to Lorenz and Kidd [LORE 94], the fact that a class has more instance variables indicates that the class has more relationships to other objects in the system. More instance variables, on average, indicate a possibility that the classes are doing more than they should. Due to this fact, the classes may have too many relationships to other objects in the system. They also say that a smaller number of instance variables lead to a higher level of reuse.
Average weighted method count	WMCPC	This metric is introduced by Chidamber and Kemerer [CHID 91].
Number of accesses per class	AccPC	

Table 3.2: Class Size Metrics

Metric Name	Key	Description, Comments
Number of messages sent per method	MSGPM	
Number of lines of code per method	LOCPM	Note that the metric "lines of code" is influenced by the formatting of code and that *Moose* includes the comments.
Number of statements per method	NOSPM	This metric has a more or less equivalent meaning to the metric "number of messages sent per method".
Average cyclomatic complexity	ACYCLO	
Number of implicit variables per method		
Number of accesses per method	NOImCPM	
Average method hierarchy nesting level	AMHNL	

Table 3.3: Method Size Metrics

Metric Name	Key	Description, Comments
Average hierarchy nesting level	AvgHNL	This metric indicates the average depth of the classes in the inheritance tree. Note the special definition of the underlying "hierarchy nesting level" class metric in *Moose*.
Maximum hierarchy nesting level	MaxHNL	This metric gives you the highest hierarchy nesting level in the system.
Number of accessors per attribute	NOAccPA	The number of accessor methods divided through the number of attributes. This number should normally be between zero and two. A value of zero means that there are no accessor methodsn at all. A value of two indicates that there's most likely a setter and a getter accessor method for every attribute.
Percentage of abstract classes	NOACl	According to Lorenz and Kidd [LORE 94], well designed projects typically have over 10% of abstract classes.
Percentage of concrete classes	NOCCl	
Number of parameters per method	NOPPM	
Number of conditionals per method	NOCondPM	

Table 3.4: Project Characteristics Metrics

Chapter 4

Design Flaw Detection

MooseGager provides a few simple possibilities to detect design flaws. In this chapter, each Section describes one design flaw detection method implemented in MooseGager. An outline is shown below:

- Section 4.1: Detection of NOP overrides
- Section 4.2: Detection of bypassed accessor methods
- Section 4.3: Detection of large classes
- Section 4.4: Detection of long methods
- Section 4.5: Detection of misplaced methods

4.1 Detection of NOP overrides

Riel's heuristic 5.17 [RIEL 96] says that it should be illegal for a derived class to override a method of a base class with a no operation (NOP) method, that is, a method that does nothing. The main reason for this heuristic is that it violates the nature of a inheritance relationship that should be a specialization relationship, *i.e.*, the subclass should extend the functionality of the superclass. Riel explains the appearance of the "NOPing" problem as follows:

> "It [the "NOPing problem"] has always occurred in designs where a derived class is already present and a base class is being added. For whatever reason, designers tend to consider any new class added to a design as being a derived class of the existing class. When the new base class is added, it is forced to inherit from something that should be its derived class. The result is to eliminate some of the functionality of the derived (acting as base) class via NOP methods. The correct design is found by flipping the hierarchy upside down, making the base class the derived class and the derived class the base class."

4.2 Detection of bypassed accessor methods

If a developer implements an accessor method for a certain attribute, it should be forbidden to bypass the accessor and use the attribute directly. The main reason is that the accessor method might provide

more than just accessing the variable. As an example it may send a “self changed” to inform all its dependent objects that its state has changed. If the attribute is both addressed using the accessor method and directly, the dependent object cannot be sure that it knows the current state of the object. Another reason is that maintaining a class where attributes are addressed both directly and through accessor methods is harder than if the attributes are only addressed in one way.

MooseGager detects methods that address an attribute directly although there is an accessor method. Read and write accesses are treated separately. Note that this feature is based on the accessor assessment of *Moose* which only detects “pure accessor methods”. This means that accessor methods that also do something else like “self changed” are not considered as accessor methods. Anther problem is that this feature is implemented using the class *VisualWorksImporter* and therefore only works for Smalltalk code.

4.3 Detection of large classes

According to different authors, classes should implement one and only one key abstraction. This can be simplified by saying that they should not exceed a certain size. In MooseGager, there are several predefined detectors to detect classes with a suspicious high amount of methods, attributes, and method overrides. These detectors are based on a single metric and detect classes with an upper threshold value. The predefined threshold values are taken from the book *Object Oriented Software Metrics* by Mark Lorenz and Jeff Kidd [LORE 94]:

- Number of methods: 20
- Number of attributes: 3
- Number of method overrides: 3

The threshold values of the predefined detectors can be set in the preferences menu.

4.4 Detection of long methods

Generally, in an object-oriented system, methods should be short and provide only one operation. A good explanation comes from Fowler [FOWL 99]:

> “I prefer short, well-named methods for several reasons. First, it increases the chances that other methods can use a method when the method is finely grained. Second, it allows the higher level methods to read more like a series of comments. Overriding also is easier when the methods are finely grained.”

In *Moose*, the method size can be measured in different ways. MooseGager provides predefined detectors for the number of statements, the number of messages sent, and the number of lines of code in a method. Each of these predefined detectors detects methods using a threshold value that can be set in the preferences menu of MooseGager. The default threshold values are proposed by Lorenz and Kidd [LORE 94]:

- Number of lines of code in a method: 7 (for Smalltalk)
- Number of messages sent in a method: 9
- Number of statements in a method: 0.8 * number of messages sent in a method

Lorenz and Kidd suggest not to use the LOC metric because it is highly dependent of the used programming language and the formatting of the code.

4.5 Misplaced methods

In the book *Smalltalk with Style* [KLIM 96], we can find the following statements:

> "Methods with several arguments can sometimes be implemented as methods in the class of any of its arguments. If a method does not send messages to the receiver or access its instance variables, then it should not be implemented in the class of the receiver. "

These statements induced us to implement a detector for such misplaced methods. What the authors of this book write is of course applicable not only to Smalltalk code but to any object-oriented system. But their definition of misplaced methods is for several reasons too blurry for the implementation:

- Hook methods fall into this category since they do not perform any operations at all.
- Methods that only return a value, such as default values, fall into this category. But since they mostly return an object that is used in the class they belong to, they are not misplaced.
- Methods that construct an object using *self* as an argument also fall into this category. An example is the method *asValue* in the class *Object* in the VisualWorks environment. These methods are almost always not necessary, since the objects being created could be created without this method. But still, these methods provide very useful services for other objects.
- Methods that only send messages to the class they belong also fall into this category.

So, we came up with the following definition of a misplaced method:

A **misplaced method** is a method

- that has a non-empty body,
- is not abstract,
- does not send messages to self,
- does not use instance variables,
- does not only return a value,
- and do not only create another object with *self* as the only argument.

MooseGager provides a detector for misplaced methods. But the lack of the type information in Smalltalk is a problem for the detection of misplaced methods because it is not possible to check if an object sends messages to itself. Another problem we could not solve up to now is to check if a method only sends messages to the meta-class of the invoking object(*e.g.*, the method *halt* in the class *Object* basically sends a message to the class object *Object*) since for the moment, this relationship is not represented in the *Moose* model. So, we had to use a simplified definition of misplaced methods for the implementation. Thus, MooseGager considers and detects a method as misplaced if it

- has a non-empty body,
- is concrete,
- does not send messages to self,
- does not access instance variables,
- and does not use self as an argument.

Note that if MooseGager says a method is misplaced, this does not necessarily mean that the method is really misplaced since the implementation does not totally fit the definition of misplaced methods and there are exceptions *i.e.*, methods that are considered as misplaced but are not. If MooseGager detects a method as misplaced, it means that it would be worth to have a closer look at this method and its placement.

An example for methods that are detected but not misplaced are utility methods: These methods are often written to avoid duplicated code or to increase the readability and maintainability of the code. Since these methods do not access the state of an object, MooseGager considers them as misplaced. They could be placed anywhere since these methods do not use the attributes of the class in which they are implemented.

Chapter 5

Case Studies

In this chapter we present two applications of the tool MooseGager. In Section 5.1 we present a comparison of the collections frameworks of two different Smalltalk-80 implementations. And the Section 5.2 we present the comparison of two different versions of Jun, an open source application framework and 3D graphics library.

5.1 Comparison of two Collections Framework

Squeak[1] and **VisualWorks**[2] are two different Smalltalk-80 [GOLD 83] implementations. Squeak is an open implementation meaning that everybody is allowed to modify and enhance the Squeak system as long as the modifications are made available on the Internet. VisualWorks on the other hand is a product of Cincom. There is a non-commercial version for free, but it is not open. So as a contrast to Squeak where a large community is responsible for the whole code, in VisualWorks there is basically only a core team responsible for the system.

In this case study, we would like to figure out if we can see differences in the implementation of the two products using software metrics. To do this, we compare two more or less equivalent parts of both systems: the collections framework. For our examinations, we dropped the meta classes and the *Moose* stubs.

5.1.1 Provided Measurements

Tables 5.1 to 5.4 show us the overview measurements of the frameworks. First of all, let us have a look at the project size measurements table.

Both system have 80 classes. But in the Squeak collections framework, there are more methods (130 more), more attributes (8 more), more statements (648 more), and a higher total weighted method count (184 higher). So, both systems basically provide the same classes but framework of Squeak is larger and more complex.

Having a look at the Table 5.2, we see that the classes of Squeak have in average about 1 method more each, are longer (about 8 more statements in average), plus they have a higher weighted method count

[1] http://www.squeak.org
[2] http://www.cincom.com/smalltalk

(ca. 3 higher).

All measurements of the method size (Table 5.3) are just about equivalent. This lets us conclude that the additional complexity of the Squeak's collections framework is not caused by more complex methods but by a higher amount of methods (per class).

In the last measurement Table (Table 5.4), we see that the average hierarchy nesting level and the amount of abstract methods and classes are lower in Squeak. According to Lorenz and Kidd [LORE 94], 10 to 15 percent of the classes should be abstract. Squeak is in contrast to VisualWorks below this threshold. Lorenz and Kidd say that the number of abstract classes is an indication of the successful usage of inheritance and the effort that has been spent looking for general concepts in the problem domain [LORE 94]. So, this could indicate a design flaw in the Squeak collections framework: the lack of abstraction.

	VisualWorks	**Squeak**
Number of classes	80	80
Number of methods	1293	1463
Number of attributes	102	110
Total weighted method count	2364	2598
Number of statements	4988	5636
Number of messages sent	6883	7804
Number of lines of code	10416	11166
Number of accesses	9337	9893
Number of formal parameters	1578	1651

Table 5.1: Project Size Comparison.

	VisualWorks	**Squeak**
Average number of methods per class	16.16	18.29
Average number of statements per class	62.35	70.45
Average number of messages sent per class	86.04	97.55
Average number of lines of code per class	130.20	139.58
Average number of attributes per class	1.28	1.38
Average weighted method count	29.55	32.48
Average number of accesses per class	116.71	123.66

Table 5.2: Class Size Comparison.

What we just have seen for the classes is also true for the methods (see Table 5.3). The methods of VisualWorks are a little bit longer, more complex and also take more parameters than the ones of Squeak.

Up to now, we can say that

- both systems have similar measurements,
- Squeak has slightly bigger classes,

	VisualWorks	**Squeak**
Number of abstract methods	30	9
Number of accessor methods	56	71
Average number of statements per method	3.86	3.85
Average number of messages per method	5.32	5.33
Average number of lines of code per method	8.06	7.63
Average number of parameters per method	0.99	0.82
Average cyclomatic complexity	1.83	1.78
Average number of accesses per method	7.22	6.76

Table 5.3: Method Size Comparison of VisualWorks and Squeak.

	VisualWorks	**Squeak**
Average hierarchy nesting level of the classes	3.56	2.90
Average hierarchy nesting level of the methods	3.49	2.82
Maximal hierarchy nesting level of the classes	7	5
Percentage of abstract classes	12.5	3.75
Percentage of concrete classes	87.5	96.25
Number of accessors per attribute	0.55	0.65
Number of parameters per method	0.99	0.82
Number of conditionals per method	0.65	0.58

Table 5.4: Characteristics Metrics.

- and that the biggest difference of both systems is that in Squeak, there is a low amount of abstract classes and methods.

Figure 5.1 shows the distribution chart of the **number of statements** metric for both frameworks.

Both of the charts look similar. But in Squeak, some classes with a high amount of statements stick out. This is not visible in the presented charts but in the main window of MooseGager which is not shown here. The class with the most statements is the class *String*: it has 943 statements. So let us have a closer look at this class to see what MooseGager can tell us.

The **String** class in Squeak also has a very high amount of instance methods (176). In VisualWorks, the equivalent class has only 53 methods and 184 statements, but it is one level deeper in the inheritance tree, *i.e.*, it is derived from a superclass which does not exist in Squeak. An important question is if the *String* class in Squeak is too large and should be split up. Using MooseGager, we came up with the following two points:

- In Squeak, the String class only has one child whereas in VisualWorks, the equivalent class has 13 children. These measurements let us assume that the services provided by the child classes in VisualWorks are put into the String class in Squeak.
- Another point is the number of attributes of a class. In Squeak, the String class has 10 attributes whereas in VisualWorks it only has 2. Riel's heuristic 2.8 says that a class should capture one and only one key abstraction [RIEL 96]. Such a high number of instance variables lets us guess that there might be more than one key abstraction in the Squeak *String* class.

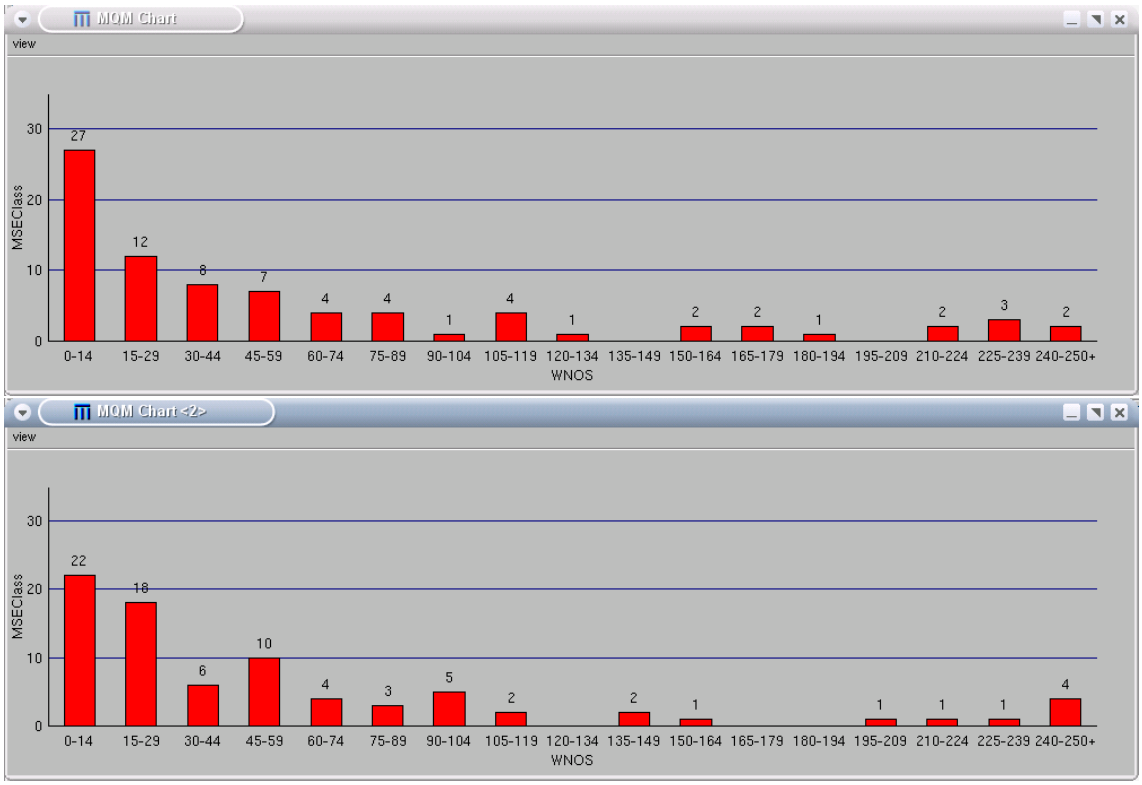

Figure 5.1: Number of statements distribution in VisualWorks (upper window) and in Squeak (lower window).

5.1.2 Interpretation

Up to now, we compared the measurements of both frameworks and found out that both are similar from the metrics point of view. But what do the measurements tell us about the systems? And what can we say about the quality of the systems? Lorenz and Kidd propose in their book *Object-Oriented Software Metrics* [LORE 94] threshold values for metrics they used in different projects. In the following paragraphs, we try to interpret our measurements using the threshold values proposed by these authors.

First, let us have a look at the average class size using the metric **number of instance methods**. According to Lorenz and Kidd, too many methods in a single class are a warning sign that too much responsibility is being placed in one type of object. This metric should only focus on public methods but since we are looking at Smalltalk code where all methods are public, we simply all methods. So, the average number of instance methods should give us a very simple indication if the intelligence and workload is distributed well across the whole system. Lorenz and Kidd propose an upper threshold of 12 for model classes. Both of our projects have higher values (16 in VisualWorks, 18 in Squeak) including private methods. If we subtract the amount of private methods, we end up with values below the proposed threshold value. This considerations let us conclude that in both frameworks the workload seems to be distributed well across the system. It is interesting to see that both systems have the same amount of classes that have more than twenty instance methods which is the threshold value that Lorenz and Kidd propose for a single class.

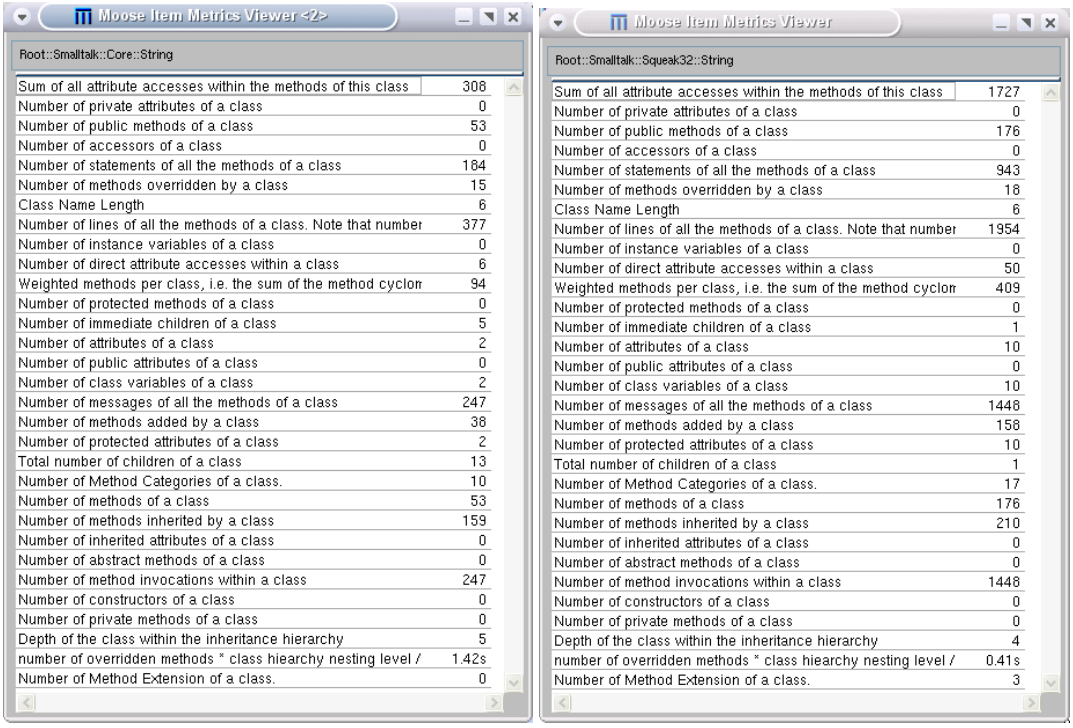

Figure 5.2: Measurements of the two String classes (VisualWorks on the left side, Squeak on the right.

Let us also have a look at the **average number of attributes per class**. According to Lorenz and Kidd, the fact that a class has more instance variables indicates that the class has more relationships to other objects. They also write that this measurement is an indication for the reusability of the system: Classes are generally more reusable when they have fewer instance variables. As an upper threshold value, they suggest three for model classes. Both of the systems we are looking at have a lower measurement (1.38 in Squeak v.s. 1.28 in VisualWorks).

Finally, let us have a look at the average method size measured with the metric **number of messages sent**. According to Lorenz and Kidd, larger numbers indicate a higher likelihood that function-oriented code is being written while smaller numbers indicate a higher likelihood that object-oriented code is being written. The authors write that it is certain that a large average method size is a problem. They experienced that the average method size for Smalltalk should be under nine messages sent per method [LORE 94]. In both the VisualWorks and the Squeak collections frameworks, there are only 5.3 messages sent per method. It is interesting to see that both systems have a very similar number of methods that send more than nine messages (239 in VisualWorks vs. 241 in Squeak).

Conclusion

To summarize, we can say that both frameworks are very similar (in terms of software metrics) and that we could not find flaws with the threshold values proposed by Lorenz and Kidd. One thing we found out is that one of the classes most programmers use, the *String* class, is extremely big in Squeak and seems to be doing too much work.

5.2 Case Study: Comparing Two Versions of Jun

In this case study, we compare two different versions of Jun. Jun [3] is an application framework and 3D graphics library that has been developed using object-oriented technology, and which supports both geometric and topographic manipulation of shapes. Our situation for this case study is the following: we have two models of Jun of two different versions (versions 5 and 195). We do not have the time stamps of these versions but we know that version 5 is compared to the 195 very old.

Tables 5.5 to 5.8 present the measurements computed by MooseGager. What we see is that the size of Jun changed: Version 195 has more than four times the amount of classes than version 5. We also see that the classes have grown too: The average number of methods per class increased by about five; the average number of statements and the average weighted method count multiplied by more than two. An interesting fact is that the average number of attributes per class stayed about the same. This could be a sign of a normal process of aging: The classes kept the same key abstractions, but over time, more requirements and functionality were built on the same key abstractions and therefore there were methods added to the classes.

The size of the methods did not change much. They contain more lines of code in the last version but the number of statements stayed at the same level. This probably means that the methods still contain the same amount of functionality but contain more comments in the newer versions.

	Version 5	Version 195
Number of classes	160	682
Number of methods	1161	8297
Number of attributes	224	974
Total weighted method count	1541	13819
Number of statements	4397	42661
Number of messages sent	6973	66847
Number of lines of code	11837	128718

Table 5.5: Project Size Comparison

	Version 5	Version 195
Average number of methods per class	7.26	12.17
Average number of attributes per class	1.4	1.43
Average number of statements per class	27.48	63.55
Average number of messages sent per class	43.58	98.02
Average number of lines of code per class	73.98	188.74
Average weighted method count	9.63	20.26

Table 5.6: Class Size Comparison

Figure 5.3 shows the distribution charts of the number of statements metric. Out of these figures we can read that the amount of classes with less than twenty statements shrunk from 75% to 45% while the percentage of the classes with a number of statements between twenty and 70 grew from 16% to

[3] http://www.sra.co.jp/people/aoki/Jun/

	Version 5	Version 195
Average number of statements per method	3.79	5.14
Average number of messages per method	6.01	8.06
Average number of lines of code per method	10.2	15.51
Average number of parameters per method	0.52	0.54
Average cyclomatic complexity	1.33	1.67

Table 5.7: Method Size Comparison of VisualWorks and Squeak

	Version 5	Version 195	
Average hierarchy nesting level of the classes	2.58	2.31	
Percentage of abstract classes	6.88%	6.51%	5.72%
Number of accessors per attribute	0.58	0.5	0.51
Number of conditionals per method	0.22	0.5	0.52

Table 5.8: Characteristics Metrics

35%. Another thing we can see is that there are more classes that have a high amount of statements: there are 20 classes that have more than 400 statements. These classes mostly have a relative low amount of methods. In the old version, this is true for all classes. In the new version, there are several classes with a high amount of methods. It is interesting that either they contain a lot of attributes or only a small amount of attributes. Analyzing the classes with a high amount of methods, we can distinguish between two categories:

- Classes with a high amount of methods and no attributes: In this category there are mostly meta-classes. With MooseGager we can figure out which meta-classes contain how much code. To analyze why this code is where it is and if it belongs there, we would have to dig deeper and figure out what this code does, which is beyond the scope of this case study.

- Classes with a high amount of methods and a high amount of attributes: The names of these classes (*e.g.,* JunPictImageStream, JunGifImageStream, JunOpenGLRotationModel, JunBmpImageStream, JunLispSmallCompiler, JunSourceCodeDifference) mostly tell us that they contain difficult algorithms that deal with source code or graphical files.

An outlier in this category is the class *JunEncyclopedia* (it contains 1292 statements, 107 methods, 3177 lines of code but only 2 attributes): The measurements tell us that this class contains a big amount of code but only two places to keep state. The methods of this class seem to be complex since they have in average more than 10 statements and more than 30 lines of code. The name of this class lets us assume that an object of this class acts similar to a dictionary: to save some sort of information and provide an interface to look for that information. The complex methods tell us something different; the class seems to act more like a place for functionality that does not belong in other classes or is used by many different classes. It would be interesting to check if the methods in this class really belong to this class.

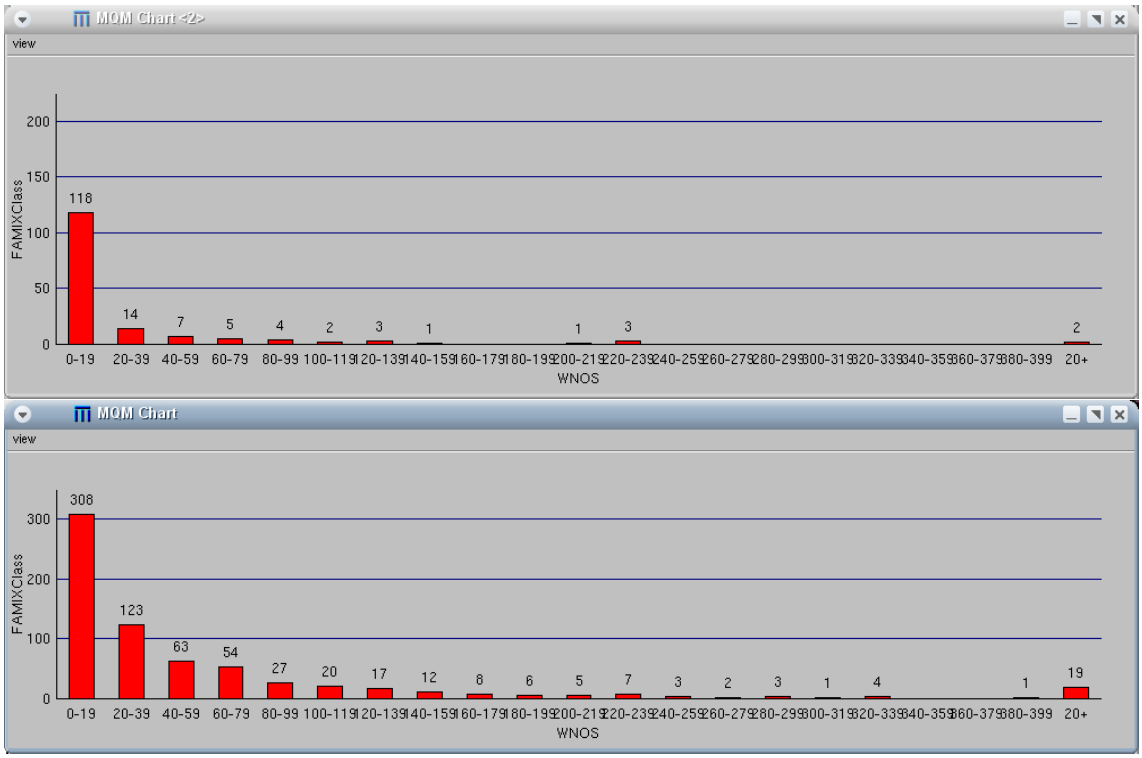

Figure 5.3: Number of statements distribution in the versions 5 (on top) and 195 (bottom) of Jun.

Conclusion

This case study gave us information about how the system evolved. We could figure out that the classes grew but the number of attributes per class stayed at the same level. We could also detect places that would be a good starting point for further inspections.

List of Figures

List of Tables

Bibliography

[CHID 91] S. R. Chidamber and C. F. Kemerer. *Towards a Metrics Suite for Object Oriented Design*. In Proceedings OOPSLA '91, ACM SIGPLAN Notices, volume 26, pages 197–211, November 1991. (p 12)

[COAD 91] P. Coad and E. Yourdon. Object Oriented Design. Prentice-Hall, 1991. (p 2)

[DEME 01] S. Demeyer, S. Tichelaar, and S. Ducasse. *FAMIX 2.1 — The FAMOOS Information Exchange Model*. Research report, University of Bern, 2001. (p 4)

[ERNI 96] K. Erni and C. Lewerentz. *Applying Design-Metrics to Object-Oriented Frameworks*, 1996. (p 3)

[FENT 96] N. Fenton and S. L. Pfleeger. Software Metrics: A Rigorous and Practical Approach. International Thomson Computer Press, London, UK, Second edition, 1996. (p 1)

[FOWL 99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. Refactoring: Improving the Design of Existing Code. Addison Wesley, 1999. (p 15)

[GOLD 83] A. Goldberg and D. Robson. Smalltalk 80: the Language and its Implementation. Addison Wesley, Reading, Mass., Mai 1983. (p 18)

[KLIM 96] E. J. Klimas, S. Skublics, and D. A. Thomas. Smalltalk with Style. Prentice-Hall, 1996. (p 16)

[LANZ 03] M. Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, may 2003. (pp 1, 5)

[LORE 94] M. Lorenz and J. Kidd. Object-Oriented Software Metrics: A Practical Guide. Prentice-Hall, 1994. (pp 1, 11, 12, 13, 15, 19, 21, 22)

[MARI 02] R. Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, University of Timisoara, October 2002. (p 2)

[RIEL 96] A. J. Riel. Object-Oriented Design Heuristics. Addison Wesley, 1996. (pp 3, 14, 20)