Analysis of Java Program Complexity - An Investigation of the Differences Between the GNU Classpath and Sun Java SE Implementations of the Standard Java Class Library Using Statistical Comparison

Jonathan Francis Roscoe
Second Year, Computer Science
Aberystwyth University
jjr6@aber.ac.uk

ABSTRACT

The GNU Classpath claims to be a viable alternative to the Sun Java SE class library. Using Java's Reflection API to analyse and compare the two libraries I have determined that the GNU Classpath library is not yet complete enough to replace the Java SE library, although there seems to be minimal functionality, many classes are missing a large number of members.

Statistical analysis has also shown that the intended purpose of a class seems to dicatate the number of members and the types of interfaces implemented. A class dealing with arithmetic will vary greatly in a simple count of members from a class dealing with I/O.

Most surprisingly, there seems to be little correlation between the number of fields a class has and the number of methods, one would think that a class with many fields would provide many get/set methods.

Keywords

Java SE, GNU Classpath, class library, package, reflection, statistical analysis, class member

1. INTRODUCTION

A Java class library is a set of files that can be used by Java applications at runtime to provide methods and structures for common requirements. This paper compares statistics on identical purpose Java classes from two different libraries in an effort to detect differences in design and structure. There are two main areas of focus:

- the competence of the GNU Classpath as an alternative to Sun's class library
- relationships between fields and members in classes
- uses for statistical data on classes

By comparing statistical data for both the GNU Classpath and Sun's Java SE classes (two different implementations of the same class libraries), it is hoped that the differences will demonstrate highlight possible failings of the GNU Classpath and possibly provide evidence of the alleged benefits and disadvantages of open source development. Whilst the investigation is by no means exhaustive, I believe a simple count of class members will help us gauge the abilities of the GNU Classpath and that statistical analysis can be used to look at: consistency, design and efficiency of class files.

I will also be comparing statistics for individual packages (such as 'java.util' and 'java.math') to determine if there are consistent differences in design for classes with a similar function when compared to those of a different nature. This should also provide more data for comparing the GNU library with the Java SE library.

2. HYPOTHESES

I hope that my data will prove a number of interesting hypotheses:

1. The average number of public members for classes within a specific package will differ greatly from the average for another package - I believe that the classes in java.math will have many more public fields (and probably methods) than java.util due to the different purpose of each package.
2. Classes in the GNU classpath will, on average have a greater/lower number of implemented interfaces, as a result of many different developers from different areas (professionally and geographically) there will likely be a lack of consistency when compared to Sun's implementation.
3. Classes in the GNU classpath will have fewer private members due to more efficient code (see "Assumptions" for more detail on this).
4. GNU classes will not have fewer public members than the same Sun classes, this is necessary to ensure compatibility with the standard class library - if it is less

then the current GNU implementation is insufficient as a replacement for Sun's own library.

5. Classes with more fields will have more methods, this will be even more pronounced in classes with a large number of private fields due to the need to provide get/set methods.

6. Classes with a larger number of fields will have a larger number of constructors, due to the need to feed in values as constructor parameter.

3. METHODOLOGY

To generate statistical data for analysis I have written a small Java application that makes use of the Java Reflection API to count the number of constructors, interfaces, methods and fields for a given list classes. The Reflection API is allows programs to examine Java classes and retrieve (as well as modify) class members, such as methods, fields and interfaces.

My application has two modes of operation:

- averages - prints out the most common and interface and average number of: constructors, interfaces, methods, private methods, fields and private fields for a given list of classes and all referred classes

- statistics - for each class given, prints out the number of constructors, interfaces, methods, private methods, fields and private fields, this is out put as a CSV list, it is not recursive

I will be using the data generated by my application to compare standard deviations and look for anomalous results.

For the purpose of comparing the structural differences between packages I will generate a set of averages for a selection of packages: java.util (data structures), java.math (arithmetic containers and functions) and java.io (input/output).

In order to look at the differences between the GNU classpath and the Sun Java class library I will be generating statistics in a non-recursive manner for all classes in the java.util, java.io and java.math packages. I chose these because they are well known and common parts of the class library and I think it will be interesting to see how the implementations vary.

3.1 Constraints

- Sun's Java SE 5 Update 15

- GNU Classpath 0.97.1

It should be noted that currently, the GNU Classpath is not a complete implementation of the Java 1.5 library, which may affect some of the averages, thus I have only taken values for classes that have been implemented by the GNU Classpath - a total of 174: 87 classes from each library.

3.2 Assumptions

In order to form conclusions I was forced to make certain assumptions, the points covered here require further investigation before my report could be considered conclusive.

To assess whether the GNU Classpath is "better" written/designed than the Java SE class library I will look at the number of private members used. Private methods are often used to break up complex public methods, a more efficient implementation may change the need for additional private methods. A reduced number of private fields would also be an indicator of more efficient code. For example, common implementations of the Sieve of Eratosthenes (an algorithm for finding prime numbers from 2 to n) use 2 arrays, yet it can be implemented with just one, reducing the amount of memory used by half. This is a potentially specious statement and requires more indepth study.

4. DISCUSSION OF THE GNU CLASSPATH VS. JAVA SE CLASS LIBRARY

I managed to generate some interesting and surprising data regarding differences between the GNU Classpath and Java SE libraries.

4.1 Totals and Averages

There is a large difference in the total counts of class members, as indicated by these tables of totals and averages:

Sun's Java SE Implementation:

	Constructors	Interfaces	Public Methods	Private Methods	Public Fields	Private Fields
Total:	255	83	1354	427	105	457
Average:	2.8	0.91	14.88	4.69	1.15	5.02
Standard Deviation:	2.57	1.19	16.19	8.05	5.3	6.47

GNU Classpath Implementation:

	Constructors	Interfaces	Public Methods	Private Methods	Public Fields	Private Fields
Total:	206	66	1079	258	94	263
Average:	2.34	0.75	12.26	2.93	1.07	2.99
Standard Deviation:	2.11	1.13	11.25	7.5	5.01	4.5

Whilst I originally hypothesised that the GNU Classpath would have fewer private methods as a result of more efficient programming I now believe this to be incorrect. The fact that the GNU Classpath has much fewer constructors, interfaces and public methods indicate that it is incomplete. Sun's java.util.Arrays has 105 public methods, but the GNU Classpath version has only 55, this is a distrubing fact for such a commonly used class (note, java.util.Arrays is not used to create arrays, but to manipulate them, this is still important though).

The standard deviation does no support the idea of "consistently incomplete", however, if we assume that minimal functionality for all classes has been implemented then this is to be expected. Once larger classes like java.util.Arrays are fully implemented the standard deviation will increase (up until the point that additional functionality for other classes is not implemented).

The results for the GNU Classpath and the Java SE libraries are similar, for example the class with the most public fields is Calendar in both cases, Vector and ArrayList have the most interfaces and Arrays has the most public methods. We can take this to mean that whilst the GNU Classpath is incomplete, it is consistently incomplete. That is, there would seem to be an equal amount of effort placed on all classes within each package so that they are all 50% (not accurate) complete.

4.2 Comparing Packages

The following table shows the most common interfaces and the average number of methods, fields, interfaces and constructors for java.util, java.math and java.io.

Sun's Java SE Implementation:

Package Name	Most Common Interface	Constructors	Interfaces	Public Methods	Private Methods	Public Fields	Private Fields
java.util	java.io.Serializable	3	2	17	6	1	5
java.math	java.lang.Comparable	6	1	36	25	5	13
java.io	java.io.Closeable	2	0	11	2	0	3

GNU Classpath Implementation:

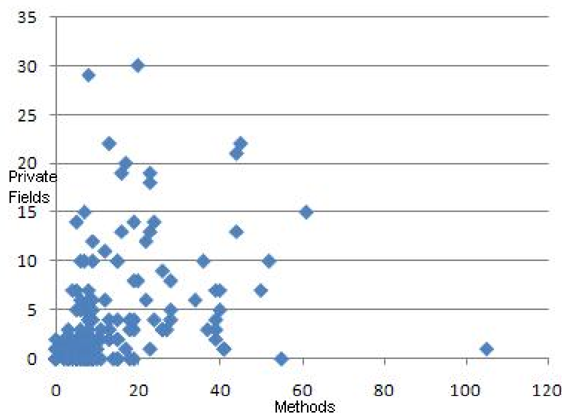
Package Name	Most Common Interface	Constructors	Interfaces	Public Methods	Private Methods	Public Fields	Private Fields
java.util	java.io.Serializable	2	1	14	2	1	3
java.math	java.lang.Comparable	11	1	37	20	6	14
java.io	java.io.Serializable	2	0	9	2	0	2

I believe these results back up the idea that the GNU Classpath is "consistently incomplete" as all of the averages for the GNU Classpath implementation are slightly lower than those for the Java SE.

We can also see that the average properties of class members varies a lot depending on package. For example, classes dealing with arithmetic have a high number of constructors (approximately 5 times more) and methods (approximately 10 times more) and classes that act as data structures have a hgih number of methods, low number of constructors and implement the class java.io.Serializable (used to convert Java objects into something transportable). This corresponds to my first hypothesis.

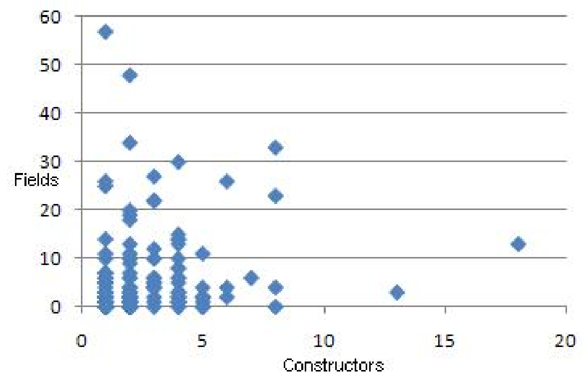
5. DISCUSSION OF THE RELATIONSHIPS OF CLASS MEMBERS

I met more unexpected results when comparing the counts of different sorts members to one another. *Private Fields vs. Public methods:*



The results show that contrary to my hypothesis, classes with a smaller number of private fields tend to have a smaller number of methods.

Fields vs. Constructors:



Once again, it seems there is little relationship between number of fields and number of constructors.

These surprising results are likely occuring because of to be because the high number of fields within methods of the classes that are not intended to be accessed from outside. It should also be considered that these results are anomalous as I have only focused on a small subset of classes.

6. USING STATISTICAL DATA TO DETERMINE CODE QUALITY AND CONSISTENCY

Analysis of the number of fields, methods, constructors and interfaces of classes may provide a means of identifying poorly written classes. If all the classes for a given project are analysed to create a set of averages then, any class that deviates greatly from this average should be considered as a high priority candidate for review. This method may not be consistent in identifying poorly designed classes, but it may be a good indicator in many cases.

Furthermore, such figures could be combined with other information such as number of lines of code, referenced classes, code blocks, etc for a more accurrate indicator of poor or inconsistent classes. It may be possible to create a standard scoring system, where various properties of a class are out together to make a single value, allowing for simpler comparison of related classes. For example, if a class had: 3 methods, 1 private methods, 6 private fields and 67 lines of effective code (not comments) then we might use those values in the formula "(lines/total method)/total fields" (please note, this formula is an example of a concept and is in no way correct) which would give a "score" of 2.79. If we applied this to many classes we might find that the average score is about 3, any classes with a score deviating significantly could then be considered inconsistent and in need of review.

However, I feel that in order to affirm my theories regarding using statistics to check for consistency further study is needed. This should be done with a greater range of classes, as well as loooking more closely at the source code of classes. I would also like to investigate a means of effectively "scoring" class source code based on properties such as number

of methods and fields as the example formula I gave has not been analysed effectively.

7. CONCLUSIONS

Unfortunately most of my hypotheses were contradicted by the results, these were due to assumptions I made regarding the GNU Classpath library, as I did not realised it contained incomplete implentations of the classes.

My most significant conclusion is that the GNU Classpath library is shockingly incomplete. Completion of major packages such as java.util should be a priority.

Individual analysis of Java packages has shown that the nature of a Java class seems to determine the average number of members. For example, classes related to arithmetic have many more fields, whilst classes used for storing data have much fewer.

I believe that there is a correlation of members between like purposed classes and with more research it should be possible to create a means of scoring classes for consistency.

It would be interesting to run these tests again once the GNU Classpath is formally completed. I suspect that there will be a greater number of members, particularly public methods when comapred to the Java SE class library. This will be because the GNU Classpath will be compatible with the Java SE library but will also contain additional members added by open source developers.

8. REFERENCES

http://java.sun.com/docs/books/tutorial/reflect/ - Sun Java Trail on the Reflection API