Does Size Matter? A Preliminary Investigation of the Consequences of Powerlaws in Software

Joshua Lindsay
School of Engineering and Computer Science
Victoria University of Wellington
Wellington, New Zealand

James Noble
School of Engineering and Computer Science
Victoria University of Wellington
Wellington, New Zealand
kjx@ecs.vuw.ac.nz

Ewan Tempero
Department of Computer Science
The University of Auckland
Auckland, New Zealand
e.tempero@cs.auckland.ac.nz

ABSTRACT

There is increasing evidence that many object-oriented software size metrics are characterised by scale-free, powerlaw distributions. This means programs will have arbitrarily large components, and the size of the largest component will increase as programs' overall size increases. This directly contradicts a crucial assumption of object-oriented design — that large programs can be build by combining many small components.

In this paper, we present a preliminary study of this contradiction. We illustrate the distribution of several size metrics over a corpus of 100 Java systems, and then investigate the largest classes (according to five size and complexity metrics) from one of those systems. We find that, while some large classes may be explained by code-generation or design patterns, most large classes were examples of poor object-oriented design.

Keywords

Size metrics, powerlaws, object-oriented design

1. INTRODUCTION

Many studies have indicated that many properties of software follow a powerlaw distribution [2, 4, 5, 9, 11, 15, 16]. Everyone would like to know why this is the case, that is, what the processes are that lead to such distributions (this is true outside software engineering as well). We are particularly interested in what the existence of powerlaw distributions in software structure says about the quality of its design. Is it the nature of software design that such distributions occur "naturally", that is, they are unavoidable, or does their presence indicate one thing (e.g. the design is "good") and their absence something else (the design is "bad")?

One software property where the presence of a powerlaw distribution may be cause for concern is "size". If the distribution of the size of classes in an object-oriented design is a powerlaw, then that means there must be some large classes in that design, and furthermore, the more classes there are in the design, the bigger the largest classes must be. Large classes are generally considered to be an indication of poor design (e.g., [7, 8]) so does this mean that powerlaw distributions for size are due to poor designs? Are the large classes a consequence of natural design principles, limitations of the language or just plain lazy programming? In this paper we present a preliminary study to attempt to answer this question.

The rest of the paper is organised as follows. In the next section we present our motivation for this study in more detail, including an initial discussion of possible consequences of powerlaws, and summarise the related work. Section 3 presents the details of how we carried out the study, the results of which are given in section 4. Section 5 discusses what our results might mean, and section 6 summaries our conclusions and discusses possible future work.

WETSoM'10, May 4, 2010, Cape Town, South Africa.

2. MOTIVATION AND RELATED WORK

Ultimately, software metrics by themselves are not very interesting. It is fairly easy to produce something that meets the definition of being a "metric", that is, "is a means to assign *numbers or symbols* to *attributes* of *entities* in the real world in such a way as to describe them according to clearly defined rules" [6]. However we also need to know what a particular measurement means, otherwise the fact that we can produce it is meaningless. There are any number of examples of software metrics that have been proposed but for which the meaning of the measurements they produce is at best guesswork. For example, Riel provides a very useful discussion about object-oriented design [13]. However he includes statements such as "In practice, inheritance hierarchies should be no deeper than an average person can keep in his or her short-term memory. A popular value for this depth is six" (Heuristic 5.5). The measurement "six" is apparently an important threshold however without considerably more in formation its difficult to evaluate the usefulness of this advice.

Of common interest is whether a particular measurement could be considered "typical" or not. A blood temperature of 37 °C would not excite much comment by a medical doctor because it is the "typical" value for humans, whereas a temperature of 40 °C would be cause for concern. The reason we can make this assessment is that the *distribution*

of measurements for blood temperature is both known and easy to characterise with a known mean and (small) variance. In order to evaluate a measurement for some software metric we need to achieve a similar level of knowledge about the "typical" values for that metric and most crucially its distribution. If we knew (for example) that the depth in inheritance hierarchies of designs that are generally agreed to be "good" had a mean of 4 and a variance of 1, then we would be right to consider those with depth 6 with some concern.

In fact we know very little about the typical values for most software metrics, and what little we do know suggests that evaluating measurements produced by these metrics could be problematic. The reason for this is the increasing evidence that many software metrics have powerlaw distributions. We believe that it is imperative that the software engineering community gains a better understanding of the consequences of such distributions. In this paper, we try to advance this understanding with respect to a particular class of software metrics, namely those measuring "size".

There seem to be two slightly different views as to what a powerlaw distribution is. The wider view is that a powerlaw distribution is one that is exponentially decreasing, and includes such distributions as those described by log-normal and stretched exponential functions. The more narrow view is that a powerlaw distribution has the form:

$$p(x) \propto x^{-\alpha} \tag{1}$$

where α is a positive constant and x is non-negative [10]. All distributions in the wider view have so-called "fat tails", that is, non-zero frequencies for very large values. As these large values correspond to (for size metrics) "big" classes, we will generally not distinguish between the different views.

There is, however, an important difference between the narrow definition and the wider one. Distributions characterised by equation 1 have the "scale-free" property, meaning that any part of the distribution has the same shape as the whole distribution. This has implications with regards to how software is designed. If a software system is designed by reusing classes, then we might expect that the shape of the size distribution vary over different sizes, the so-called "Lego Hypothesis" [11]. Such a distribution would *not* be scale free, which means that if the size distribution is a scale-free one then the Lego Hypothesis is not being met.

Another characteristic of scale-free powerlaws is that, for some values of α, they have infinite mean and/or variance. One consequence of this fact is that the central limit theorem doesn't hold for such distributions, so the mean and variance of a sample (which will always be finite) cannot be used as estimators for the population mean and variance. This means that using statistics such as "average class size" for any meaningful comparison is pointless. Or, to put it another way, in order to know whether "average class size" is useful, we need to know whether or not the distribution of class size is scale-free.

The difference between scale-free and non-scale free exponentially decreasing distributions is difficult to detect without a lot of data. For example, figure 1 shows the in-degree and out-degree distributions for aggregate (number of types used for field types) relationship for the `james` system[1]. This system has nearly 300 classes (plus another 30 or so interfaces). Even with the guidelines (the straight line shows the best-fit scale-free distribution, the other two are stretched-exponential and log-normal) it is difficult to tell which distributions the data fits best. In fact, the statistical tests show no significant difference between the scale-free and non-scale free distributions. Figure 2 shows the same distributions for the much larger `eclipse` system. In this case, we could not rule out the in-degree distribution being scale-free, but *could* rule out the out-degree distribution being non scale-free. The data shown in these two figures was gathered for an earlier study that we reported on previously [2] (although only the out-degree distribution for eclipse was shown in that paper).

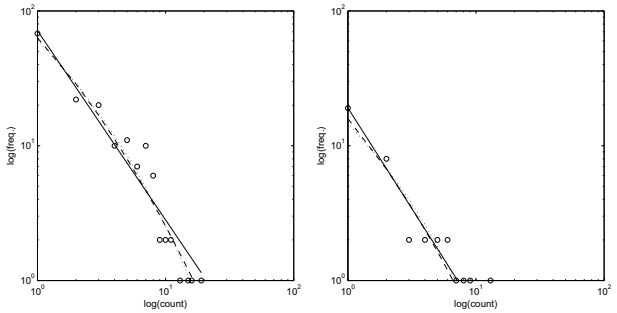

Figure 1: The in-degree aggregate (left) and out-degree aggregate (right) distributions for `james-2.2.0`

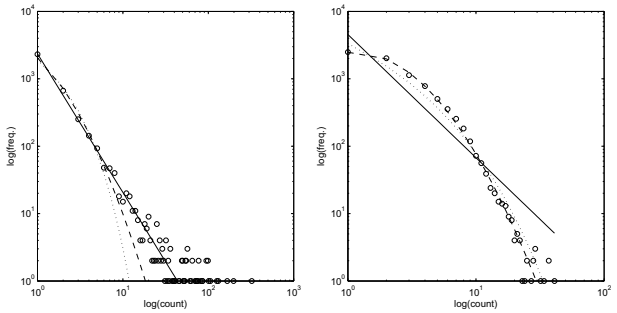

Figure 2: The in-degree aggregate (left) and out-degree aggregate (right) distributions for `eclipse-3.1.`

In practical terms, even when the difference between scale-free and non-scale free powerlaw distributions is significant, such as with `eclipse`, all distributions imply the existence of some quite large values.

The examples in figures 1 and 2 come from the relationship graph where vertices are types and edges exist between vertices when one type has a field of the other type. The out-degree is thus a lower bound on the number of fields a type has (there could be multiple fields with the same type). Number of fields is one possible candidate for a "size" metric, but it is not the only one. It is possible that the distributions in the figures are only seen in this relationship, but other possible size metrics may not exhibit such distributions.

[1] `http://james.apache.org`

Acronym	Common name	SciTools name	Description
LOC	Lines of Code	CountLineCode	the number of lines of code in a class
CS		CountSemicolon	the number of semi-colons in a class
CST		CountStmt	the number of declarative plus executable statements
CDCV		CountDeclClassVariable	the number of static fields in a class
CDCM		CountDeclClassMethod	the number of static methods in a class
NIM	Number of Instance Methods	CountDeclInstanceMethod	the number of non-static methods in a class
NIV	Number of Instance Variables	CountDeclInstanceVariable	the number of non-static fields in a class
CDM		CountDeclMethod	the number of local (not inherited) methods
CDMA		CountDeclMethodAll	the number of methods including inherited ones
CBO	Coupling Between Object classes	CountClassCoupled	the number of classes coupled to a given class
LCOM	Lack of Cohesion in Methods	PercentLackOfCohesion	the average percentage of class methods using a given class instance variable
WMC	Weighted Methods per Class	SumCyclomatic	the sum of the cyclomatic complexity of all methods in a class
SE		SumEssential	the Sum of essential complexity of all nested functions or methods in a class
NOC	Number Of Children	CountClassDerived	the number of immediate descendants of the class

Table 1: Size Metrics

All of the studies we are aware of that involved some kind of size metric appear to show some form of powerlaw distribution.

Wheeldon and Counsell examined a number of inter-class relationships in Java source code and found powerlaw distributions for all of them [15]. With Baxter et al. we repeated the Wheeldon and Counsell study on a larger corpus. As well as finding only fat-tailed distribution, we noted that out-degree distributions often did not appear to be scale-free [2]. Concas et al. found powerlaws for various relationships between types in Smalltalk and Java, and shows that the Yule model provided a good prediction for their data [5]. Zhang and Tan looked at a dozen Java systems and found that the sizes of classes are lognormally distributed. They derived a size estimation model based on this finding of the class size distribution that estimated system size with good accuracy [16]. Potanin et al. found that distributions of incoming and outgoing references in the *dynamic* object graphs follow a powerlaw [11].

Louridas et al. found that powerlaws appear in systems written in several different languages, including Java, C, Ruby, and Pascal, in use of libraries in various formats (e.g., DLL on Windows and ELF on Unix), in products developed and released by software organisations, and as a result of ad hoc contributions from around the world [9]. They confirmed our observation that out-degree distributions do not fit scale-free distributions as well as in-degree distributions. As noted earlier, they also discussed what the consequences of powerlaw distributions are with respect to such things as resource and effort allocation. Our interest is in what powerlaw distributions mean with respect to quality of design.

The evidence seems overwhelming that, whatever relationship we look at in software, it follows some kind of "fat-tailed" distribution, meaning there will be representatives with quite large values. We would like to know, in general, what the mechanism is that leads to such distributions, and specifically for this paper, whether the large classes implied by such distributions could be considered to be a natural consequence of software design, or some indication of quality of the design.

3. METHODOLOGY

Our goal is to understand why large classes exist, so we need to be able to identify such classes. This raises the question as to how size should be measured. There are a number of possible candidates metrics for "size" of a class, and it is possible that classes that are large according to one metric are not so large by another. It is also possible that some size metrics distributions are powerlaws and some are not. To deal with these issues we use multiple metrics. We check whether they are powerlaw distributions. We then identify the outlier classes and manually assess a sample of these to adjudge whether they represent a reasonable design choice or not, or to otherwise classify what decisions led to their creation.

3.1 Data Collection

We used the Qualitas Corpus [12], a collection of open source Java systems we have been using for such studies. The version we used was released on 2 February 2009 and consists of 100 distinct systems, with multiple releases of some systems. We only considered the most recent release of each system that was in that release of the corpus. The details, including the full list of systems studied, is available on the Qualitas Corpus website.

To take the measurements, we used an existing commercial tool, SciTools `Understand` 2.0 [14]. This provides measurements for a number of "common" metrics from the source

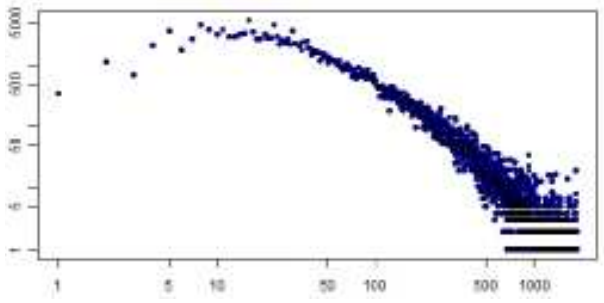

Figure 3: Frequency distribution of Lines of Code over corpus

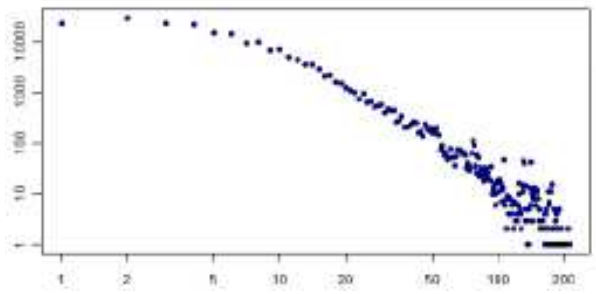

Figure 4: Frequency distribution of Number of Instance Methods

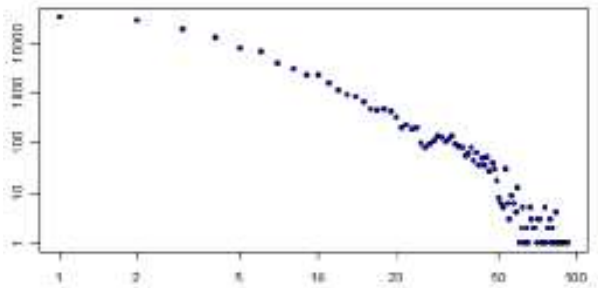

Figure 5: Frequency distribution of Number of Instance Variables

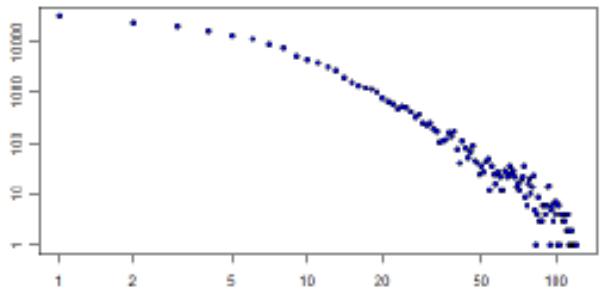

Figure 6: Frequency distribution of Coupling Between Object classes

code, including several metrics that are size related. The metrics we used where dictated by what the tool could produce (see below), a decision we discuss further in section 5. `Understand` analyses source code.

3.2 Metrics

While size metrics are perhaps the most important for finding, collating, and defining large classes, size metrics alone may not tell us why the classes are big. We also gathered measurements using several other classes of metrics — a cohesion metric (LCOM), a coupling metric (CBO), and two complexity metrics (SE and WMC). Table 1 lists the metrics we used (including the `Understand` identifier for ease of repeatability).

Lines of code is a common size metric, despite the fact that it has a number of issues [6]. `Understand` provides a number of variants of "lines of code," the one we chose (`CountLineCode`) seemed the most useful for our purpose as it provides a reasonable reflection of what a developer would have to deal with. For comparison we also consider two other variants (`CountSemicolon` and `ContStmt`).

4. RESULTS

Our study is not yet complete, however we have gathered all of the measurements and performed some of the qualitative assessment. In this section we give a summary of the measurements over the whole corpus and then do a case study on one system in the corpus for the qualitative assessment.

4.1 Corpus Measurements Summary

Figure 3 shows the distribution of LOC over all classes (recall this includes interfaces and other Java type declarations) in the corpus shown on a log-log scale. Its shape seems closer to the "truncated curve" shapes typical of log-normal and stretched-exponential distributions (right-hand chart of figure 2) rather than the straight line we would expect of a scale-free power-law. LOC has one of the largest variances of all the metrics, with no specific number of lines of code appearing in more than 2.5% of the corpus. Nevertheless, classes with lines of code ranging between 4 and 38 make up almost half of all classes LOC sizes and measurements outside of this range account for less than 1% each over all. 11% of the corpus has over 200 LOC, 5% over, 2.8% over 500, and 0.8% (or 1,750) classes have over 1,000 lines of code, with the largest class weighing in at an impressive 20,000 lines of code — which certainly qualifies for the "big" title! Inspection revealed that of the top 5 largest classes, 4 were parsers that had been generated with `javacc`, and one (human written) game server (12,183 LOC).

Figure 4 shows the results for Number of Instance Methods. This has one of the larger spreads of majority counts; with big percentages for having 1-6 instance methods. 80% of all classes have 10 or fewer Instance Methods, and half of all classes have less than 5. There are almost ten thousand classes that have 28+ instance methods, of which 1,000 have over 85 and 100 have over 170 Instance Methods, with the largest having 1,124. Parsers were once again the largest of the classes, taking the top 5 spots. Most were the same classes as those from the LOC inspection. The sixth to 10th largest classes however, were all implementing the facade pattern, which seemed reasonable except they also have over 2,000 LOC which is odd when the facade pattern is meant to provide a simplified interface to a larger body of code.

Figure 5 shows the results for Number of Instance Variables. Almost 40% of all classes had no instance fields.

Class	LOC	CDM	CDMA	CBO	WMC
net.sf.freecol.client.control.InGameController (c)	2604	94	94	51	528
net.sf.freecol.client.control.InGameInputHandler(c)	993	24	29	37	140
net.sf.freecol.client.FreeColClient	304	39	39	29	54
net.sf.freecol.client.gui.action.ActionManager	99	5	30	62	9
net.sf.freecol.client.gui.Canvas	1252	111	111	85	245
net.sf.freecol.client.gui.GUI	1770	77	77	38	403
net.sf.freecol.client.gui.ImageLibrary	952	65	80	8	249
net.sf.freecol.client.gui.InGameMenuBar	139	9	18	61	11
net.sf.freecol.client.gui.panel.ColonyPanel	1210	33	43	28	86
net.sf.freecol.common.model.Building	655	68	115	18	190
net.sf.freecol.common.model.Colony	1242	107	175	23	301
net.sf.freecol.common.model.IndianSettlement	858	64	132	17	210
net.sf.freecol.common.model.Map	862	52	99	17	136
net.sf.freecol.common.model.Player	1670	155	202	26	449
net.sf.freecol.common.model.Tile	1408	93	140	28	311
net.sf.freecol.common.model.Unit	3030	201	248	36	809
net.sf.freecol.server.ai.AIPlayer	1670	48	63	53	402
net.sf.freecol.server.control.InGameInputHandler(s)	2197	70	81	41	372
net.sf.freecol.server.model.ServerPlayer	125	20	222	9	30

Table 2: Measurements for CBO, CDM, CDMA, CBO, and WMC for classes that are one of the 10 largest for at least one of these metrics in alphabetical order of fully-qualified class name.

Class ID	Categories
InGameController (c)	CBO,CDM,CDMA,LOC,WMC
Canvas	CBO,CDM,CDMA,LOC,WMC
Unit	CBO,CDM,CDMA,LOC,WMC
Colony	CDM,CDMA,LOC,WMC
GUI	CBO,CDM,LOC,WMC
Player	CDM,CDMA,LOC,WMC
Tile	CDM,CDMA,LOC,WMC
InGameInputHandler (s)	CBO,CDM,LOC,WMC
AIPlayer	CBO,LOC,WMC
ImageLibrary	CDM,WMC
Building	CDM,CDMA
FreeColClient	CBO
InGameInputHandler (c)	CBO
InGameMenuBar	CBO
ActionManager	CBO
ColonyPanel	LOC
IndianSettlement	CDMA
Map	CDMA
ServerPlayer	CDMA

Table 3: `freecol` classes that are one of the 10 largest as measured by at least one of CBO, CDM, CDMA, LOC, or WMC. Parenthetic letters disambiguate classes in different packages with same name — "c" for client, "s" for server.

Those classes that did have instance fields were most likely to have between 1 and 6, as 50% of the corpus fell into that category. Less than 1,000 classes had more than 34 instance fields, 100 of which had over 55, and 10 of which had over 100. The largest of these classes were GUI dialogues or frames, often having well over 2,000 LOC and 100 instance methods.

For CBO (figure 6), 70% of classes have five or fewer coupled objects. Another 20% of classes have between 6 and 14 coupled objects, with the remaining percentage having anywhere from 15-141 coupled objects. It's not surprising then to see the same computer generated parsers seen in previous results appear at the top of the CBO list. The game server from the NIV results makes another appearance too, claiming the fifth spot with 124 coupled classes.

4.2 Case Study: freecol

The results above indicate that most notions of "size" in object-oriented software have fat-tailed distributions for classes taken from all systems in the corpus, and this remains true for individual systems (as the previous studies in section 2 hinted). In this section we take a closer look at one system in order to get a more refined understanding as to why large classes exist.

The system we have chosen is the game `freecol`[2]. Its website describes it as "is a turn-based strategy game based on the old game Colonization, and similar to Civilization." It is large enough to provide a useful amount of data to work with, without being so large as to make it difficult to carry out manual assessment. It is a single, complete, application, rather than a framework that is in intended to be incorporated into another application, such as `hibernate`, or a software development tool such as `ant` or `eclipse`. It covers a number of different domains, having a fairly sophisticated graphical user interface, a non-trivial architecture (client/server), with networking and some multi-media (au-

[2] `www.freecol.org`

Metric	1st	2nd	3rd	4th	5th
hline LOC	Unit	InGameController (c)	InGameInputHandler (s)	GUI	Player
CBO	Canvas	ActionManager	InGameMenuBar	AIPlayer	InGameController(c)
WMC	Unit	InGameController (c)	Player	GUI	AIPlayer
CDM	Unit	Player	Canvas	Colony	InGameController(c)
CDMA	Unit	ServerPlayer	Player	Colony	Tile

Table 4: `freecol` in top 3 positions for CBO, CDM, CDMA, LOC, WMC

dio) requirements. Another advantage with using a game is that one can quickly get an appreciation for the concepts involved by playing it.

Table 2 shows a sample of measurements for `freecol`. Due to space limitations we've only shown some of the metrics and some of the classes. The metrics we choose were those we thought would give us the most interesting subset, and the classes are those that appear as one of the 10 largest for at least one of the metrics (19 in all).

That 19 classes are one of the 10 largest classes according to one metric indicates a certain lack of agreement regarding "size". This can be further seen by the range of measurements shown. For example, LOC ranges between a high of 3030 to a low of 99, nearly 2 orders of magnitude. The class with measurement 99 (`ActionManager`) also has one of the smallest measurements for all the other metrics except CBO (measurement of 62), where it has the 2nd largest. While this might suggest that CBO is not as representative as LOC for size, `ServerPlayer` has the 2nd lowest CBO measurement (9) and the second smallest LOC measurement (126). The other metrics show similar variation in measurements.

Table 3 shows one summary of table 2, namely organising the classes with respect to which metric they have a measurement in the top 10. There are only 3 appearing in the top 10 for all 5 metrics, 5 appearing for 4 metrics, 1 for 3, 2 for 2 metrics, and 8 classes in the top 10 for just 1 metric. Table 4 shows a slightly different view, namely the positions of the top 5 classes for each metric. These tables show how much variation there is between the different metrics, indicating that none of these metrics can be easily ignored when trying to identify large classes.

Now we examine classes identified as being "large" by some metric in more detail. The class that stands out is `Unit` as it heads most of the top-10 lists. Interestingly it extends another (abstract) class, `FreeColGameObject`, so it's "real" size is bigger than the measurements given (other than CMDA) suggest.

`Unit` implements 4 interfaces, all specific to `freecol`, which, given these must be implemented, may explain why the class is so big. These interfaces contain 20 methods in total, only one of which is implemented in `FreeColGameObject`. Most of them have quite small implementations with only about 4 having LOC more than 4.

Another possible indication as to why `Unit` is so big is the number of fields it has (and so has to manage). It has the 4th largest measurement for NIV at 29 (all instance variables in its parent are private). For example if all of these had getters and setters then that would contribute to the number of methods (CMD), although not so much to LOC. In fact quite a number of methods are getters or setters (although many do some level of checking and so are more than just setting or returning the value of fields).

In fact it is in studying the use of instance variables that leads to some questions about the design to be raised. This class is meant to represent pieces that can move in the game, which seems like a reasonable abstraction to have. However, comments attached to the declaration of some fields and the kinds of checks that are being done when they are accessed make it clear that they manage state for only certain kinds of "unit". This and other evidence suggests that `Unit` is a conglomeration of several concepts corresponding to the different kinds of pieces, raising questions about the cohesiveness of its design. This suggests it could be replaced with a smaller class representing movable pieces and sub classes of that representing each different kind of piece. Whether the rest of the design would allow such a change requires considerably more study however.

Another large class by all measurements is `InGameController`. This class features mostly quite large, complex methods. In fact while its LOC measurement (2604) is as much as 85% of `Unit`'s (3030), it declares only 94 methods to `Unit`'s 201, meaning it has a much higher average method size. While it is difficult to identify any clear problem with its design, it does seem to have a number of somewhat diverse responsibilities, from renaming pieces to determining the destination of units (which involves a large switch statement) to managing in-game messages. Riel's Heuristic 3.2 says "Do not create god classes/objects in your system. Be very suspicious of a class whose name contains Driver, Manager, System, or Subsystem." To his list we could reasonably add "Controller", as `InGameController` does seem to have a lot of responsibilities.

Another prominent class in the top 5 is `Player`. Like `Unit` this class also inherits from `FreeColGameObject` however it only implements one (small) interface. This class represents entities that can control pieces in the game. There are two such entities — humans and those managed by the AI engine. That one class represents two different classes of thing immediately suggests possible issues with its cohesion. There certainly are aspects of its design that depend on which kind of entity a given object represents, again suggesting a redesign with smaller classes would be possible. Its CBO measurement is only just outside the top 10 for CBO.

As noted above, `ActionManager` is a small class by most metrics but has a large CBO measurement. In fact, inspection of its implementation shows that the CBO measurement comes from one method called `initializeActions`, which does as its name suggests. The initialisation involves a large number of "action" classes, hence the high CBO. As there is no way to avoid creating the action classes there is no obvious improvement in this case.

`Canvas` has the largest CBO measurement, the 3rd largest number of methods (CMD), and is in the top 10 for the other 3 metrics. This is a class that part of the management

of the user-interface, and Java user-interface classes always seem to be large due to their need to interact with the Swing framework. This is also true of the class `GUI`.

5. DISCUSSION

The results of the previous section indicate that not all large classes can be considered a consequence of good design. The implications of this is that if size distributions of classes are a powerlaw then that may indicate problems with its design. Of course close examination of almost any code is likely to identify some questionable decisions so the fact that we have found such decisions needs to be treated with caution and the evidence we have comes from only one system. Nevertheless, our experience with this study convinces us that this is a viable process for investigating the consequences of powerlaw distributions.

Not all large classes were obviously bad. At least some of the them can be explained by decisions that are perhaps not entirely under the control of the developer. Some of the large classes were generated from parser-generators and some were due to choice of a particular design pattern (Facade). Another category of large classes were those that play the role of hooking together the services provided by other classes. This is true of user-interface classes and classes such as `InGameController`.

In order to understand the consequences of powerlaw distributions it is worth considering the mechanisms that cause them to occur. A common explanation for them is what is referred to as "preferential attachment" [1]. Preferential attachment says that bigger things are more likely to get more than smaller thing. In class size terms, this says that the bigger classes get bigger faster than smaller classes. The work by Concas et al. shows there is some agreement using Yule processes to model some distributions, and a Yule process is a form of preferential attachment [5].

There is still the question as to why preferential attachment is a reasonable explanation for the cause of large classes. Is there any explanation for why this should be (or might be) true? We offer the following argument. When a system gets bigger, it offers more, and presumably different, functionality. If it is different functionality, then the existing classes shouldn't need to grow to provide this new functionality — that should be done by new classes. However these new classes need to be integrated with the rest, and that integration will cause the "integration-type" classes to grow. What's interesting about this argument is that classes that commonly connect other classes often seem to be those that provide the user-interface, and classes such as `InGameController`. Note that we are not suggesting that this is a good design, just that it might explain why the large classes we have seen got to be large. We would like to make this line of argument more precise and find more evidence to support (or refute) it.

As noted earlier our choice of metrics was mainly dictated by what `Understand` could produce. Since the tool provided variants of several fairly standard size metrics this was no real restriction. Of slightly more concern was the fact that, as is common with many metrics tools, complete definition of the metrics was not provided. The descriptions provided some ambiguous in some cases meaning we did not always have complete knowledge as to what was being measured. For example, the "CBO" metric does not appear to be either of those defined by Chidamber and Kemerer [3, 4], since it includes "use of type" as a form of coupling. We did do a small amount of checking with test cases but time limitations prevented us from checking all of the exotic combinations possible in Java designs. Since we were only interested in the *rank* of a class with respect to a metric rather than the actual measurement, we believe this uncertainty does not impact our conclusions, however this issue must be kept in mind for an replication efforts. That issue aside, `Understand` was a fairly easy-to-use and stable tool.

6. CONCLUSIONS

We believe that it is important to better understand what it means when the distribution of measurements of some software metric is a powerlaw. At the moment, our community cannot say with confidence even whether a powerlaw distribution corresponds to a "good" or "bad" design. We have presented the early results of a study whose goal is to examine one attribute that appears to have a powerlaw distribution, namely class size. What we've found does not support the suggestion that the large classes implied by a powerlaw distribution are a natural consequence of good design. While this is an important indicator, there is still much work to be done.

One question that naturally arises when considering multiple metrics for "size" is whether there is any difference between them. We have begun to look at the correlation between the different metrics that we hope to present in the near future.

7. REFERENCES

[1] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999. DOI: 10.1126/science.286.5439.509.

[2] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the shape of Java software. In W. Cook, editor, *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 397–412, Portland, OR, U.S.A, Oct. 2006.

[3] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 197–211, 1991.

[4] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[5] G. Concas, M. Marchesi, S. Pinna, and N. Serra. On the suitability of yule process to stochastically model some properties of object-oriented systems. *Physica A: Statistical Mechanics and its Applications*, 370(2):817–831, Oct. 2006.

[6] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. PWS Publishing Company, second edition, 1997.

[7] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] M. Lorenz and J. Kidd. *Object-oriented software metrics: a practical guide*. Prentice-Hall, 1994.

[9] P. Louridas, D. Spinellis, and V. Vlachos. Power laws in software. *ACM Trans. Softw. Eng. Methodol.*,

18(1):1–26, 2008.

[10] M. E. J. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46(5):323–351, Sept. 2005.

[11] A. Potanin, J. Noble, M. Frean, and R. Biddle. Scale-free geometry in OO programs. *Commun. ACM*, 48(5):99–103, 2005.

[12] Qualitas Research Group. Qualitas corpus, release 20090202. `http://www.cs.auckland.ac.nz/~ewan/corpus/`, Feb. 2009.

[13] A. Riel. *Object-oriented design heuristics*. Addison-Wesley, 1996.

[14] SciTools. Understand 2.0 build 489. `http://www.scitools.com/products/understand`, aug 2009.

[15] R. Wheeldon and S. Counsell. Power law distributions in class relationships. In *Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM03)*, 2003.

[16] H. Zhang and H. B. K. Tan. An empirical study of class sizes for large java systems. In *APSEC '07: Proceedings of the 14th Asia-Pacific Software Engineering Conference*, pages 230–237, Washington, DC, USA, 2007. IEEE Computer Society.