Measurement of Exception-Handling Code: An Exploratory Study

Keith Ó Dúlaigh, James F. Power
Dept. of Computer Science,
National University of Ireland, Maynooth,
Co. Kildare, Ireland.
Email: {keithod,jpower}@cs.nuim.ie

Peter J. Clarke
School of Computing and Information Sciences,
Florida International University
Miami, FL 33199, USA.
Email: clarkep@cis.fiu.edu

***Abstract*—This paper presents some preliminary results from an empirical study of 12 Java applications from the Qualitas corpus. We measure the quantity and distribution of exception-handling constructs, and study their change as the systems evolve through several versions.**

***Keywords*-exception handling code, empirical software engineering, software metrics**

I. Introduction

In this paper we present a number of empirical studies of Java applications in an attempt to quantify and visualise exception usage in code. We have examined a number of possible metrics and graphs, and present a sample here to stimulate discussion on two topics.

- What kind of metrics are most useful for measuring the level of exception-related code in a system?
- What kinds of graphs are most useful for visualising the metrics values calculated for a system?

The metrics discussed in this paper measure the quantity and distribution of exception-related constructs in Java code.

A. Related work

Size metrics for exception-related code have typically involved measuring the number of try blocks, catch blocks, finally blocks, and code involved in exception handling. These have been studied in more general exception-handling contexts [1], in theoretical frameworks [2] and in the context of Aspect Oriented Programming [3].

Work by Jo et al. on uncaught exceptions includes an empirical study of 13 "benchmark Java applications" that the authors chose as representative samples of Java applications [4, Table 1]. While this is similar to our approach, no version numbers are supplied for the applications, and, in our experience, it can be quite difficult to determine exactly which packages and classes from an application might be included in a study. We hope to enhance the empirical results from Jo et al. by rooting them in a publicly available and fixed corpus of Java systems, and by extending this work to include an evolutionary study.

System Name	*Initial Version*	*No. of Versions*	*Final Version*	*Description*
ant	1.1	20	1.8.1	Software build tool
antlr	2.4.0	18	3.2	Parser generator
argouml	0.16.1	10	0.30.2	UML diagramming application
azureus	2.0.8.2	51	4.5.0.4	BitTorrent client
freecol	0.3.0	23	0.9.4	Video game
hibernate	0.8.1	86	3.6.0-beta4	Object-relational mapping library
jgraph	5.10.0.0	39	5.13.0.0	Graph drawing tool
jmeter	1.8.1	18	2.4	Load testing tool
jung	1.0.0	23	2.0.1	Graph modeling and visualization framework
junit	2.0	21	4.8.2	Unit testing framework
lucene	1.2-final	21	3.0.2	Information retrieval library tool
weka	3.0.1	49	3.7.2	Machine learning software tool

Table I
SYSTEMS FROM THE *Qualitas Corpus* USED IN THIS STUDY. THIS TABLE LISTS THE TWELVE SYSTEMS, ALONG WITH THE INITIAL AND FINAL VERSION NUMBERS AND THE TOTAL NUMBER OF VERSIONS OF EACH.

Other measures not dealt with in this paper include the kind of code found in catch blocks [5], the precision of the match (in terms of the class hierarchy) between thrown exceptions and their handlers [1], [4], [6], [7], and the number of methods in the call-chain between throw instructions and the corresponding catch block [1], [5], [6], [8].

B. Experimental setup

An important aspect of our study is that we use applications from the *Qualitas Corpus* [9], a standardised, publicly-available "curated" set of Java programs. We have found it extremely difficult to compare or precisely replicate the results from previous empirical studies, since the numerical results can be significantly influenced by the application's version, the libraries and packages deemed to be included in the distribution, and any third-party libraries used. By fixing the *Qualitas Corpus* as a baseline for this study, we hope that our results will be repeatable and comparable with other studies using this corpus.

System Name	methods	catch blocks	throws clauses	throw insts	% meth with e-c
ant	10820	2322	1750	2823	37
antlr	2591	473	124	125	12
argouml	18268	1893	920	2690	19
azureus	36614	2982	2863	5381	28
freecol	7229	967	463	571	15
hibernate	18956	1917	3824	2665	28
jgraph	2163	65	29	33	5
jmeter	8116	1466	657	486	19
jung	4252	196	97	263	10
junit	1035	138	190	81	27
lucene	9611	1252	2786	1565	34
weka	17566	1781	2478	1801	24

Table II
SOME BASIC SIZE MEASURES FOR THE *final* VERSION OF EACH SYSTEM. THIS TABLE SHOWS A COUNT OF THE NUMBER OF METHODS AND EXCEPTION-RELATED CONSTRUCTS FOR EACH SYSTEM, ALONG WITH THE PERCENTAGE OF METHODS CONTAINING AT LEAST ONE EXCEPTION-RELATED CONSTRUCT.

Table I lists twelve systems from version *20101126e* of the *Qualitas Corpus*. Since this version of the corpus is intended for evolutionary studies, it contains a number of releases for each system, and we have listed the first and last version, along with the number of versions of each system in the second and third column of Table I. From these systems we have included all classes categorised in the corpus as belonging to "source packages", as being "distributed" and for which both binary and source code was provided. We have excluded one system, Eclipse, from our study as its classes were not distributed with source code in the *Qualitas Corpus*.

Our automated analysis was principally conducted by our own tool written with the aid of the *ASM* Java bytecode analysis framework (http://asm.ow2.org/), and augmented with visual inspection of the source code. This means that the data gathering and analysis was primarily carried out on Java bytecode, rather than the source code, but this should not impact the results presented here.

II. SIZE AND DISTRIBUTION COUNTS

In order to compare sizes between different systems, we normalise the counts based on the *size* of the system, which we measure as the number of methods (including constructors and class initialisers). Our analyses using Pearson's correlation coefficient (r) show that, taking all systems and versions, there is a strong correlation between the number of methods in a system and the number of classes ($r = 0.97$), the number of LOC ($r = 0.98$) and the number of NCLOC ($r = 0.99$), with a p-value in each case $< .001$ for a two-tailed test. In this context, we chose to use the number of methods as we felt it provided a more intuitive basis for the essentially behavioural constructs we are measuring.

The second column of Table II gives the number of methods in the final version of each system. Counting all 379 versions of all the systems studied, over 3.5 million methods were analysed, containing a total of just over 1.7 million exception-handling constructs.

In the following studies we measure the occurrence of exception-handling constructs in Java code, which we divide into three types: **catch blocks**, where we count each occurrence of either a catch block or a finally block (thus one try statement can correspond to multiple counts here); **throw instructions**, where we count each occurrence of a throw statement and **throws clauses**, where we count each exception named in the throws clause of a method signature. The total numbers of each of these constructs occurring in the final version of each system is given in the third, fourth and fifth columns of Table II.

A. Distribution of exception-handling constructs

When considering the number of exception-related constructs for each system on a per-method basis, it is important to remember that only a minority of methods actually contain exception-related constructs. The final column of Table II lists the percentage of methods that contain at least one throw instruction, catch block or throws clause. As can be seen from this data, most methods in each system do not contain any exception-related constructs at all, ranging from 63% in the case of ant to 95% in the case of jgraph. However, this still reflects a reasonably high proportion of methods being directly impacted by exceptions, with nine of the applications showing 19% or more (i.e. roughly one in five) of the methods containing exception-related code.

To assess the changes in distribution of exception-related constructs in the code, we have calculated the percentage of methods containing such code for each version of each system in the *Qualitas Corpus*. Figure 1 contains twelve bar-charts, one for each system in the study. Each individual bar-chart has one bar for each version of the system, with the height of the bar representing the percentage of methods in that version that contain exception-related constructs.

As can be seen from the height of the bars in Figure 1, the overall view is one of stability in terms of the percentage of methods directly using exception-related constructs. The major exception is antlr (row 1, column 2) where we see a sudden increase, followed by a sharp decrease. There was a considerable variance in the packages distributed with different versions of antlr, which for some versions included an entire copy of an earlier version of the system. This should be borne in mind when studying this and future results for the antlr system.

It can be seen in Figure 1 that the overall percentage of methods with exception-related constructs actually decreases for 6 of the 12 systems, and for most of the others the increase is relatively minor. The one exception is in the junit system, where from version 4.0 onwards there is a gradual increase in the percentage of methods with such constructs.

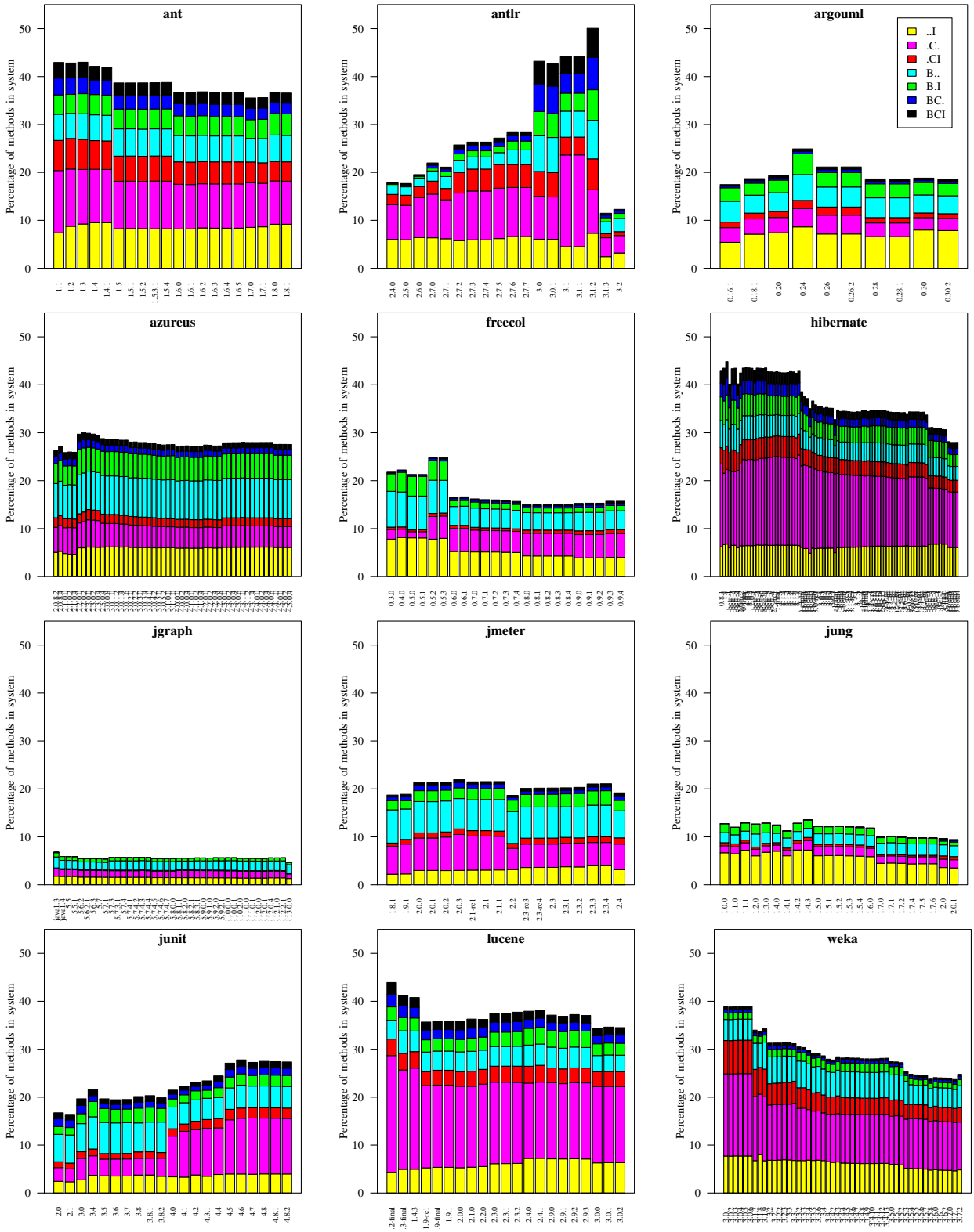

Figure 1. Evolution in usage of exception-handling code for each of the twelve systems in our study. Each bar represents a version of one system, and the divisions within a bar show the percentage of methods in each usage category.

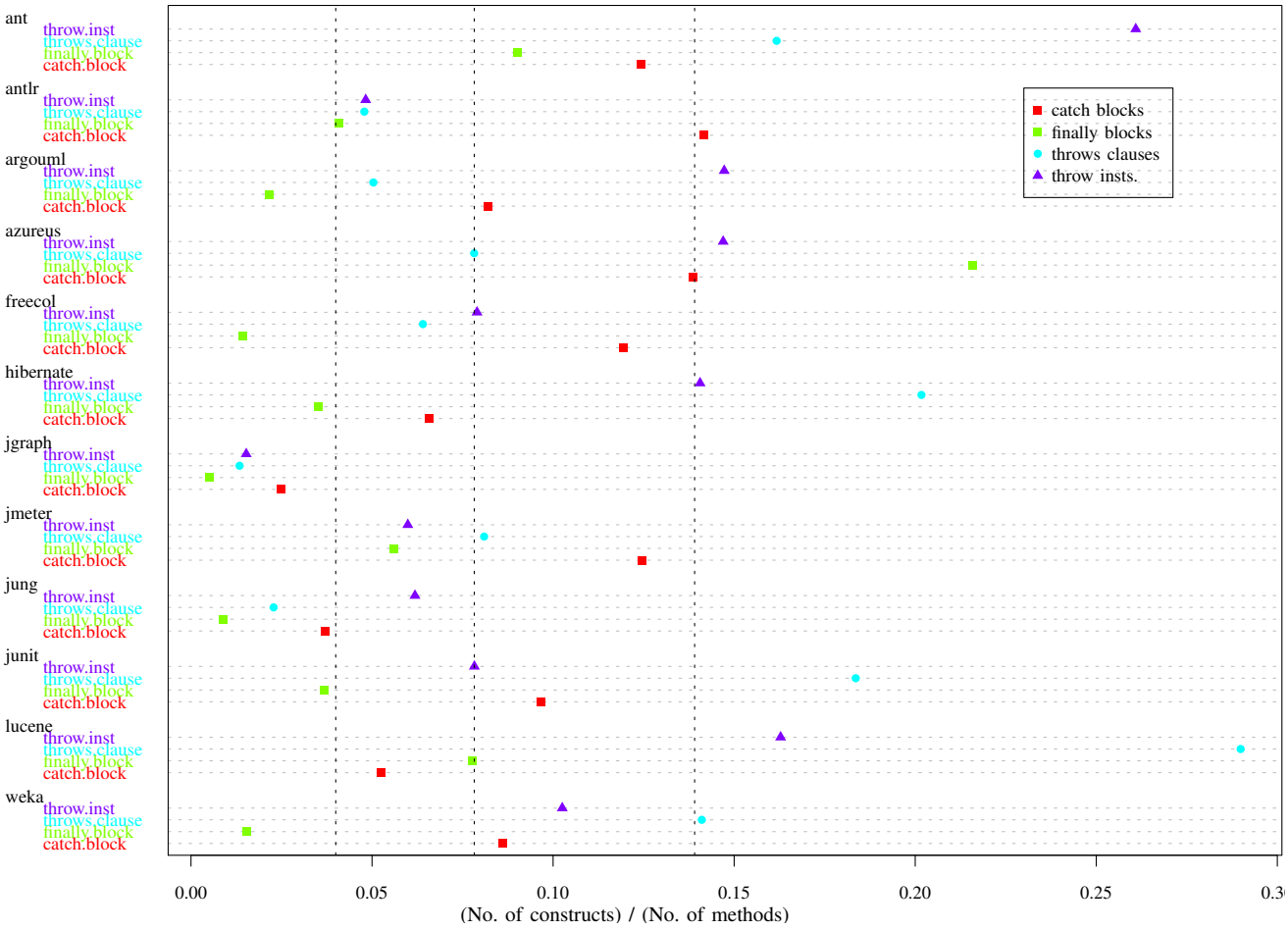

Figure 2. Comparison of exception constructs over different systems. This dot-chart shows the relative counts (divided by the no. of methods) of exception constructs for the most recent version of all systems. The vertical lines mark the quartile boundaries (25%=0.04, 50%=0.07, 75%=0.12).

To get a more detailed view of the changes, we have further broken up each bar in these charts into segments representing the kinds of methods in each of seven *categories*. These categories are coded using the following convention:

B	method has one or more catch or finally blocks
C	method signature has one or more throws clauses
I	method has one or more throw instructions

Since a method may be categorised as "true" or "false" under any of these three headings, we have eight ($=2^3$) categories in total, or seven when we do not count methods with no exception-related constructs.

Each bar in Figure 1 is partitioned into seven sections, representing the percentage of methods in each category. As might be expected, most methods make use of just one of the three kinds of exception-related constructs, but we can also see that ant, antlr and hibernate have a relatively high percentage of methods using all three. We can also distinguish differences between applications: for example, jmeter makes relatively low use of throw instructions and higher use of catch blocks and throws clauses, whereas for jung this situation is reversed. Finally, for junit we can see that the sudden increase of exception-related constructs is due to a decision to increase the percentage of methods with throws clauses from version 4.0 onwards.

III. Use of Exception-Related Constructs

Figure 2 contains a dot-chart which compares the number of exception constructs across the most recent version of each system. This dot-chart plots a count of the number of catch blocks, finally blocks, throws clauses and throw instructions, in each case divided by the number of methods in that version of the system. One of the most notable facts about the size distribution shown in Figure 2 is the relatively narrow range, as all but three of the data points fall in the range $0 - 0.2$. Thus, any discussion of the differences between the systems should be considered in the context of

this relatively low overall spread. Yet, even in this context, it is possible to observe some outliers, as mentioned above, in terms of exception construct usage.

For example, considering clusters on either side of the median line in Figure 2, it appears that jgraph and to a lesser extent jung make relatively little use of exception-related constructs overall, whereas both ant and azureus make relatively high use of all constructs. Considering individual kinds of constructs, we can see that ant, hibernate, junit and lucene all make relatively high use of throws clauses, suggesting a policy of propagating thrown exceptions. Both ant and lucene also have a high proportion of throw instructions, and further study of the code shows that they have a policy of using their own exception classes, rather than those from the Java API. The azureus system is distinctive in its relatively high use of finally blocks, presumably related to the need to release resources in this kind of application.

A. Evolutionary study

Studying the evolution of this construct usage over time helps to further pinpoint changes in the systems. Figure 3 also shows the number of exception constructs per method, but this time the ratios are plotted for all versions of each system in the *Qualitas Corpus*. Thus in Figure 3, each version of each system is represented by four vertically-aligned dots in the relevant plot. It should be noted that while all twelve plots in Figure 3 have the same scale on the vertical axis, the number of methods shown on the horizontal axis is different for each.

The graphs in Figure 3, organised by rows and columns, are essentially scatter plots, but in this case the dots have been joined by a line indicating the *temporal* sequence. In most cases this corresponds to a monotonic increase in both method size and construct occurrence, but there are some differences. For example, in the plot for lucene (row 4, column 2) we can observe a decrease in the total number of methods in the last version. As was the case for Figure 1, the picture for antlr (row 1, column 2) appears quite confused due to the variations in package contents, as noted previously in Section II-A.

The graphs in Figure 3 also give us some idea of how the changes in the ratio of exception constructs corresponds to overall changes in the system. This allows us to pinpoint changes that had a particular impact on exception-related code, or that may have denoted a change in the coding policy relating to exceptions.

Overall, the plots in Figure 3 show a gradual evolution towards the data given earlier in Figure 2. Both hibernate (row 2, column 3) and lucene show a marked decrease in the level of throws clauses as the code evolves, suggesting that the relatively high level overall is the result of legacy code, rather than new code. Similarly, the more modest decrease in the ratio of throw instructions for jung (row 3, column 3) and weka (row 4, column 3) suggests that the newer code added to later versions makes less use of these constructs.

The plots in Figure 3 allow us to pinpoint those versions where exception-related policy decisions were made. For example, in the centre of the plot for ant (row 1, column 1) we can observe a number of closely-clustered dots, indicating a series of versions with no great increase in size, most likely corresponding to rearranging or refactoring code rather than developing new features. We can see from the jump in the corresponding line that during this refactoring a decision was taken to increase the ratio of finally blocks in the application; the corresponding data shows an increase from 0.03 in version 1.5.2 to 0.07 in version 1.5.3.1. Similarly, the azureus (row 2, column 1) system also exhibits a clear policy shift towards increasing the ratio of finally blocks, from 0.11 in version 2.3.0.4 to 0.19 in version 2.5.0.4.

Looking at the plot for junit (row 4, column 1) in Figure 3 we can see a series of refactorings resulting in an increase in the use of throws clauses, from 0.05 in version 3.8.2 to 0.12 in version 4.0. A similar trend can be observed in freecol (row 2, column 2) from 0.02 in version 0.5.1 to 0.07 in version 0.5.2. However, it should be noted that junit is a relatively small application, and these changes were relatively early in its development, so only a small number of methods are involved in this case.

IV. Conclusion

This paper has presented a number of different quantifications and visualisations of the use of exception-handling constructs in twelve Java applications. We hope this exploratory study will help to provide a baseline for further analyses of such constructs, and for the development of useful measures of such constructs in Java code.

This paper has not addressed one of the central concerns that is common to most metrics, namely the *quality* of the code related to exception handling. We are currently investigating the nature of the code surrounding throw instructions, and the code used in catch/finally blocks (following [5], [7], [10]), and will present this in future work.

Acknowledgment

This material is based upon works supported by the Science Foundation Ireland under Grant No. 11/RFP.1/CMS/3068.

References

[1] M. P. Robillard and G. C. Murphy, "Static analysis to support the evolution of exception structure in object-oriented systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 12, pp. 191–221, April 2003.

[2] B.-M. Chang, J.-W. Jo, K. Yi, and K.-M. Choe, "Interprocedural exception analysis for Java," in *ACM Symposium on Applied Computing*, Las Vegas, NV, 2001, pp. 620–625.

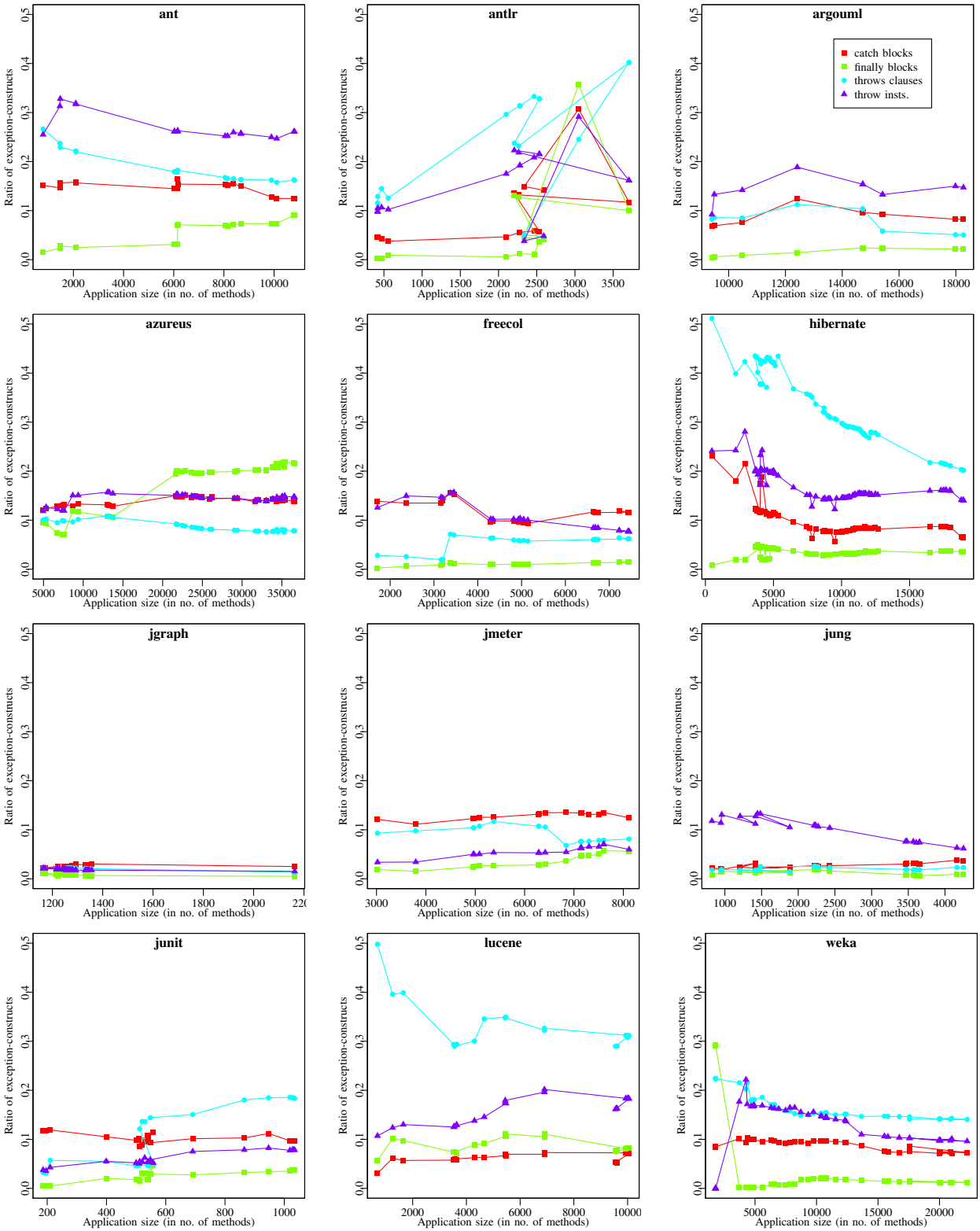

Figure 3. Changes in levels of exception-handling code for each of the twelve systems in our study. In the above plots, each dot represents a measure for one version of the system, and the dots have been joined to represent the temporal sequence of the version releases.

[3] J. Taveira, C. Queiroz, R. Lima, J. Saraiva, S. Soares, H. Oliveira, N. Temudo, A. Araujo, J. Amorim, F. Castor, and E. Barreiros, "Assessing intra-application exception handling reuse with aspects," in *Brazilian Symposium on Software Engineering*, Fortaleza, Ceara, October 2009, pp. 22–31.

[4] J.-W. Jo, B.-M. Chang, K. Yi, and K.-M. Choe, "An uncaught exception analysis for Java," *Journal of Systems and Software*, vol. 72, no. 1, pp. 59–69, June 2004.

[5] C. Fu and B. G. Ryder, "Exception-chain analysis: Revealing exception handling architecture in Java server applications," in *International Conference on Software Engineering*, Minneapolis, MN, 2007, pp. 230–239.

[6] S. Sinha, A. Orso, and M. J. Harrold, "Automated support for development, maintenance, and testing in the presence of implicit control flow," Georgia Institute of Technology, Tech. Rep. GIT-CC-03-48, September 2003.

[7] F. Filho, C. Rubira, R. de A. Maranhão Ferreira, and A. Garcia, "Aspectizing exception handling: A quantitative study," in *ECOOP Workshop on Advanced Topics in Exception Handling Techniques*, ser. Lecture Notes in Comp. Sci., C. Dony, J. L. Knudsen, A. B. Romanovsky, and A. Tripathi, Eds., vol. 4119. Springer, 2006, pp. 255–274.

[8] R. P. Buse and W. R. Weimer, "Automatic documentation inference for exceptions," in *Intl. Symposium on Software Testing and Analysis*, Seattle, WA, 2008, pp. 273–282.

[9] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "Qualitas Corpus: A curated collection of Java code for empirical studies," in *Asia Pacific Software Engineering Conference*, Dec. 2010.

[10] D. Reimer and H. Srinivasan, "Analyzing exception usage in large Java applications," in *ECOOP Workshop on Exception Handling in Object Oriented Systems: Towards Emerging Application Areas and New Programming Paradigms*, A. Romanovsky, J. L. Knudsen, C. Dony, and A. Tripathi, Eds., Darmstadt, Germany, July 21 2003.