Visualizing the Size of the Java Standard API

Craig Anslow, James Noble,
Stuart Marshall
School of Engineering and Computer Science
Victoria University of Wellington, New Zealand
{craig,stuart,kjx}@ecs.vuw.ac.nz

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

ABSTRACT

The design of software should be made up of small packages and classes. The Java Standard API is now very large since Java's beginnings, and contains over 200 packages, nearly 5800 classes, and nearly 50,000 methods. We have conducted visual software analysis on the Java Standard API using existing software visualization techniques to identify large packages and classes in the API. Our analysis has identified that there exists a number of large packages and classes in the Java Standard API which leads to possible refactoring opportunities.

Categories and Subject Descriptors

D.3.0 [**Programming Languages**]: Standards
; H.1.2 [**User/Machine Systems**]: Human Factors

General Terms

Design, Languages

Keywords

Java Standard API, Software Metrics, Software Visualization

1. INTRODUCTION

The more we learn about past mistakes, the better are our chances to avoid them in the future – and build better software at lower cost [23]. The Lego Hypothesis states that the design of software should be made up of small packages and classes [3]. Having smaller packages and classes makes software easier to maintain. Our overall research project goal is to understand the quality of software design by collecting measurements of the software and then visualize the software metrics data.

To measure software design quality we use software metrics. A software metric measures some property of a piece of software such as the number of lines of code [6]. Applying software metrics can help determine the quality of software. Lanza et al. [9] claim that there is no magic software metric that has been found and consider the definition of a universal software design quality metric as the holy grail of software engineering.

To visualize the software metrics data we adopt existing software visualization techniques. Software visualization is defined as the use of computer graphics and application of information visualization techniques to facilitate the understanding of software [16]. The goal of software visualization is to help users comprehend software systems and to improve the productivity of the software development process [5].

To support our project we use the Java Standard API 1.6 and the Qualitas Corpus [12]. The current release of the Java Standard API 1.6 is now massive compared to the early beginnings of the language. There are now over 200 packages, nearly 5800 classes, 9500 variables, nearly 50,000 methods, and 523,500 lines of code in the API. The Qualitas Corpus is a corpus of 100 open-source Java applications collected from a number of sources and intended for use in empirical studies of software to understand how programs are structured and the relationships between code structure and design quality attributes.

This paper addresses the question are there any large packages or classes in the Java Standard API 1.6? Our approach for answering this question is as follows. We first collect software size metrics including: Number of Instance Variables (NIV), Weighted Methods per Class (WMC), and Lines of Code (LOC) from the Java API. We then create visualizations of the Java packages and classes using the System Hotspot View metaphor from the Polymetric View suite [8], and then conduct visual software analysis to find the large packages and classes.

The rest of this paper is organised as follows. In Section 2 we give an overview of related work. In Section 3 we discuss our methodology for creating software visualizations of metrics data. Section 4 outlines our results and illustrates our System Hotspot Views of the packages and classes in the Java Standard API 1.6. Section 5 discusses our results. Section 6 discusses directions for future work. In Section 7 we present our conclusions.

2. RELATED WORK

We have used the Qualitas Corpus [12] to study various software attributes such as how inheritance [19] and fields [18] are used in Java programs. With regard to using visualizations to display our results; we have conducted studies into the words most frequently used in class names

This paper was published in the proceedings of the New Zealand Computer Science Research Student Conference (NZCSRSC) 2010. NZCSRSC 2010, April 2010, Wellington, New Zealand.

and the most common class names used in the packages of the Java Standard API using Many Eyes, a web based visualization application [1]. We found that the most frequently used words in the Java Standard API 1.6 class names in descending order are Exception, UI, Helper, Type, Event, and Factory.

There exist a number of software metric tools (as standalone or plug-ins for IDEs) that allow analysing software. Some of these tools include SemmleCode[1] [21], SciTools Understand[2], Structure101[3], and Creole[4]. SemmleCode and SciTools have a number of built-in features to collect software metrics such as the Chidamber and Kemerer [4] metrics suite about software application(s) and allow further customisation. These tools employ basic visualization techniques such as graphs and tree structures, neither of them use any specific software visualization techniques. Structure 101 and Creole mainly focus on dependencies between entities and both have plug-ins to Eclipse.

Gill and Maman [7] created micro-patterns to classify Java class implementations, while Singer et al. [15] created nano-patterns to characterize and classify Java methods. They both have analysis tools for detecting these kinds of patterns in Java programs, but neither of them have been applied to large software corpora or the Java Standard API, nor integrated with commercial tools.

Lanza and Ducasse [8] describe Polymetric Views including System Hotspot Views which are visualization techniques to help understand the structure and detect problems of a software system in the initial phases of a reverse engineering process. The visualizations use metrics data about the size of classes, packages, and their dependencies. Wettel and Lanza [22] applied design dis-harmonies to their Code City tool which uses 3D Polymetric Views and disharmony maps to identify brain, god[5], and data classes.

Apatite and Jadeite are two tools part of the Natural Programming research project that make it easier for users to search the Java API [17]. The tools use visualization cues based on usage data of classes by users as opposed to other Java software such as our empirical study [11]. A user evaluation showed that programmers were about three times faster at performing common tasks with Jadeite than with standard Javadoc.

3. METHODOLOGY

Visual analytics is a new research field which has evolved from the fields of scientific visualization and information visualization. Visual analytics is defined as the science of analytical reasoning facilitated by interactive visual interfaces [20].

Our methodology is to use *visual software analytics* [2] which is a super-set of information visualization, software visualization, and empirical software engineering to understand the Java Standard API. The visual software analytics process will help provide insight into a collection of programs or software using multiple visualization techniques at once (e.g. tree maps, focus + context, node-link diagrams), as well as various data representations (e.g. metrics, revision history, class hierarchy).

We measure the size of a package as the total number of classes, methods, variables, and lines of code in that package. For classes we measure size as the Number of Instance Variables (NIV), Weighted Methods per Class (WMC), and Lines of Code (LOC) per class.

Figure 1 shows the suite of tools we use to create the visualizations. We first obtained the Java Standard API 1.6 source code from the Sun Microsystems web site[6]. Next we load the source code into a SciTools Understand Project and execute the metrics analyser feature which exports the selected metrics (NIV, WMC, LOC) into a CSV file. We filter the files and then load them into a Processing[7] [13] sketch where we implemented the System Hotspot View metaphor which is a software visualization technique from the Polymetric View suite [8]. Upon execution of the sketch large static images of the visualizations are then generated in any format that Processing supports (e.g. png, jpg, gif).

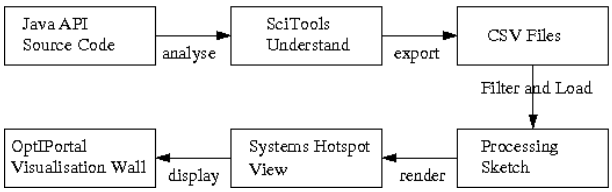


Figure 1: Tool suite to create our System Hotspot Views.

The Systems Hotspot View metaphor is described as follows. The packages are ordered alphabetically down the Y axis and classes in each package are ordered alphabetically along the X axis. For each class the width indicates the number of instance variables (NIV) and the height indicates the number of methods (WMC). Colour indicates the number of lines of code (LOC), the darker the class the more lines of code it contains (e.g. LOC < 100 = light grey, > 1000 = black).

We extended the Systems Hotspot View metaphor to contain package labels and coloured borders to represent the different kinds of classes. Concrete classes have blue borders, interfaces red borders, and abstract classes green borders. Figure 2 shows a snapshot of one of our visualizations that illustrates our extensions to the System Hotspot View metaphor.

Viewing the visualizations on paper or a desktop machine with 17-24 inch screens does not do them justice, one needs a large display to view them in their entirety or use techniques that allow the user to easily zoom in on the details. We conduct our visual software analysis of the visualizations using an OptIPortal visualization cluster to display multiple visualizations at once, see Figure 3. The visualization wall has 12 screens arranged 4x3. Each individual display is 2560 x 1600 pixels (at a cost of $3000 NZD) for a total display of 10240 x 4800 pixels. The wall is useful for visual analysis of multiple visualizations at once for discovering common trends, videoconferencing, and collaboration with other por-

[1]http://semmle.com/
[2]http://www.scitools.com/products/understand/
[3]http://www.headwaysoftware.com/products/structure101/
[4]http://www.thechiselgroup.org/creole
[5]An object that controls way too many other objects in the system and has become the class that does everything.
[6]http://download.java.net/jdk6/source/
[7]a programming language and IDE built for the electronic arts and visual design communities, http://processing.org

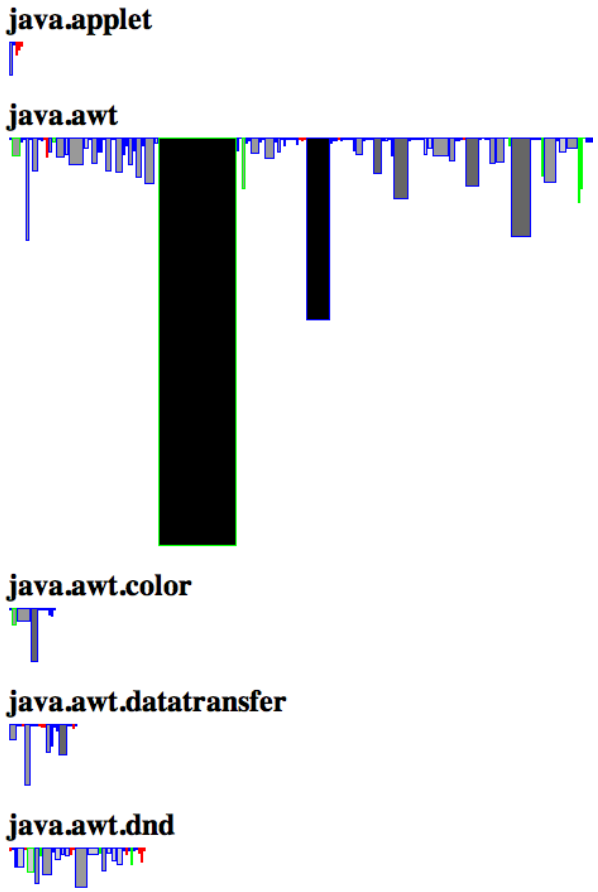


Figure 2: Our extensions to the System Hotspot View metaphor.

Figure 3: Conducting Visual Software Analysis using our OptIPortal Visualization Wall.

tals. The biggest visualizations we created were 3000 x 9000 pixels as that is the smallest resolution we could produce them to get fine enough granularity.

4. RESULTS

We are interested in identifying large packages and classes in the Java Standard API 1.6. Tables 1, 2, and 3 list the top 20 largest packages and classes in the Java Standard API 1.6 that we generated from the SciTools Understand application. Our approach is to use visual software analytics as described earlier with System Hotspot Views to understand this data from the Java API. We created three System Hotspot Views for the top level packages in the Java API as defined as java, javax, and org. We now present the System Hotspot Views for each of these packages.

Figure 4(a) shows a System Hotspot View of the top level packages from the *java package*. The two largest packages are java.awt and java.util. The smallest packages are java.applet and java.math. What is noticeable is that Component in java.awt is by far the largest class (NIV = 61, WMC = 323, LOC = 4410), which is also noticeable in Figure 2. In the java.awt package there are two other large classes (Container and Window) and two large interfaces (KeyboardFocusManager and Toolkit).

The java.awt.peer, java.security.interfaces, and java.sql packages are mainly made up of interfaces. The java.sql package has three interfaces that are represented by long narrow rectangles as they have a high WMC as evidenced by Table 3 (java.sql.ResultSet, java.sql.DatabaseMetaData, and java.sql.CallableStatement).

Each of the following packages has one very large class compared to the rest of the classes in the package, which could be god classes: java.lang (Character), java.net (URI), java.security.cert (X509CertSelector), java.text (DecimalFormat), and java.util.concurrent (ConcurrentSkipListMap). The java.awt.font and java.io packages have one very large class (TextLayout and ObjectStreamClass), a few other large classes, and then lots of smaller classes. The java.util.regex has one god class (Pattern), then another slightly smaller class, and then lots of very small classes.

Figure 4(b) shows the top level packages from the *javax package*. Clearly the largest package is javax.swing followed by javax.swing.plaf.basic, and then other javax.swing subpackages. The javax.swing package has two large classes (JTable and JTree) and two large abstract classes (JComponent and AbstractButton). The javax.swing.plaf.basic package has two large classes (BasicTreeUI and BasicTabbedPanelUI) and then nine other classes that have over 1000 lines of code.

There are a few other interesting packages. There are some packages that contain entirely interfaces including: javax.sound.midi.spi, javax.sound.sampled.spi, and javax.xml.bind.attachment. Then there are some packages that mainly contain interfaces with the exception of a few classes (or no classes) and these include: javax.sql, javax.imageio.event, javax.lang.model.element, javax.lang.model.type, javax.security.auth.api, javax.xml.stream.events, javax.xml.ws.handler, and javax.xml.ws.handler.soap

The javax.lang.model package contains no classes but has an enum instead. The rest of the javax.xml packages are relatively small having no more than 36 classes, but most much smaller than that.

Figure 5 shows the top level packages from the *org package*. The org.omg.CORBA package is the largest and contains many abstract classes. The org.w3c packages are made up of predominantly interfaces and they have one very large

interface dom.css.CSS2Properties which is represented as the very long red line and has the second most WMC, see Table 3. There are a couple of square like boxes in the org.jcp package where the number of instance variables and number of methods are near equal. Besides that there is not many other interesting patterns in this top level package.

Table 1: Top 20 largest Java API 1.6 Packages.

Packages (Totals)				
Name	Variables	Methods	Classes	Code
javax.swing	1041	5487	462	64242
javax.swing.plaf.basic	750	2666	326	47843
java.awt	674	3177	256	34712
java.util	448	2727	262	29914
javax.swing.text	461	1993	231	27449
javax.swing.text.html	361	1264	147	25073
javax.swing.plaf.synth	202	1639	105	18083
javax.swing.plaf.metal	203	953	164	16187
java.io	281	1202	124	12789
java.lang	146	1418	130	12651
java.awt.image	216	836	52	12015
java.util.concurrent	220	1063	123	11297
java.net	235	1010	119	11027
java.text	225	735	61	10349
java.awt.geom	165	732	49	9543
java.security	177	740	114	8485
javax.management	135	688	92	6237
org.omg.CORBA	83	1151	211	5220
java.beans	121	516	126	5853
javax.swing.tree	103	492	34	5848

Table 2: Top 20 largest Java API 1.6 Classes ordered by total of NIV, WMC, and LOC. # denotes A = Abstract Class, C = Class, I = Interface.

Classes				
#	Name	NIV	WMC	LOC
A	java.awt.Component	61	323	4410
C	javax.swing.JTable	38	204	4448
C	java.util.regex.Pattern	13	73	3312
C	javax.swing.plaf.basic.BasicTreeUI	46	136	3010
C	javax.swing.plaf.basic.BasicTabbedPaneUI	45	109	2916
C	javax.swing.JTree	30	154	2547
A	javax.swing.JComponent	21	186	2319
A	javax.swing.text.JTextComponent	28	100	2389
C	java.lang.Character	1	86	2127
C	java.awt.geom.AffineTransform	8	75	2115
C	java.awt.Container	18	144	2002
C	javax.swing.text.html.CSS	4	43	2115
C	javax.swing.text.html.StyleSheet	7	58	2092
C	javax.swing.text.html.HTMLDocument	7	58	2063
C	javax.swing.GroupLayout	12	50	2053
C	javax.management.relation.RelationService	12	58	2032
C	javax.swing.plaf.metal.MetalIconFactory	0	25	1965
C	java.io.ObjectInputStream	11	59	1920
C	javax.swing.plaf.basic.BasicListUI	21	49	1849
C	java.util.Collections	0	65	1812

5. DISCUSSION

We now discuss our results with respect to packages, classes, average size metrics of the classes, frequency of packages and classes used in Java software, and displaying our System Hotspot Views on our visualization wall.

5.1 Packages

The System Hotspot View metaphor easily allowed us to identify the largest packages in the Java Standard API 1.6 as they were packages that extended the farthest along the X axis. The javax.swing package is by far the largest package in the API. The javax.swing sub-packages make up a considerable amount of the largest packages. The package with the most amount of classes is javax.swing with more than 100 classes to the next largest package, javax.swing.plaf.basic, then almost 200 classes to the third and fourth largest packages, java.awt and java.util.

5.2 Classes

Most of the largest classes in the top 20 were from the javax.swing package. In most cases the classes in the Java Standard API 1.6 have more methods than variables. The java.awt.Component class is by far the largest class. The java.util.regex.Pattern is the third largest class but the actual shape of the class is a lot smaller than some of the other classes in the top 10 as it has a lot less NIV, WMC, but has a high LOC.

Table 3 contains the classes with the most WMC and these classes are represented as having a long height value in the System Hotspot Views. Some of these classes are also present in Table 2 as denoted by * in the name column in Table 3. The ones that aren't with the exception of java.util.Arrays and java.awt.Window have less than 1000 LOC, are most likely to be interfaces or abstract classes, and are likely to have zero NIV.

There are a couple of patterns related to the shapes of each class in the System Hotspot Views. There are are a few classes that have a similar number of variables and methods and closer inspection of the code reveals that they contain mainly accessor and mutator methods. The shapes of these classes are essentially squares. These classes include:

javax.swing.plaf.basic.BasicFileChooserUI
javax.swing.plaf.basic.BasicInternalFrameTitlePane
javax.swing.text.html.HTMLDocument.HTMLReader

Another pattern that is not obvious but is of interest with respect to large classes are ones that contain a low number (less than 20) of variables and methods and LOC greater than 1000. The shapes of these classes are small black lines. These classes include:

javax.swing.plaf.basic.BasicLookAndFeel
javax.swing.text.html.AccessibleHTML
javax.swing.text.DefaultEditorKit
javax.swing.text.RTFReader

5.3 Average Metrics for Classes

Table 4 lists the average metric values in the Java Standard API 1.6 for NIV, WMC, LOC for abstract classes, classes, interfaces and total average of all Java class entities. Table 4 also lists the same values for packages and also includes the average number of classes per package.

A recent project within our research group studied why Java classes are big using the Qualitas corpus [10]. The

Table 3: Top 20 largest Java API 1.6 Classes ordered by WMC. # denotes A=Abstract Class, C=Class, I=Interface.

Classes				
#	Name	NIV	WMC	LOC
A	java.awt.Component*	61	323	4410
I	org.w3c.dom.css.CSS2Properties	0	244	368
C	javax.swing.JTable*	38	204	4448
I	java.sql.ResultSet	10	187	245
A	javax.swing.JComponent*	21	186	2319
I	java.sql.DatabaseMetaData	61	172	271
C	javax.swing.JTree*	30	154	2547
C	java.util.Arrays	0	150	1588
C	java.awt.Container*	18	144	2002
C	javax.swing.plaf.synth.ImagePainter	8	144	772
C	javax.swing.plaf.synth.ParsedSynthStyle.AggregatePainter	1	138	936
C	javax.swing.plaf.synth.ParsedSynthStyle.DelegatingPainter	0	137	694
C	javax.swing.plaf.basic.BasicTreeUI*	46	136	3010
A	javax.swing.plaf.synth.SynthPainter	0	136	570
C	java.awt.Window	34	135	1482
A	javax.sql.rowset.BaseRowSet	23	131	748
C	java.nio.Bits	0	115	515
I	javax.sql.RowSet	0	115	161
I	java.sql.CallableStatement	0	111	157
C	javax.swing.plaf.basic.BasicTabbedPaneUI*	45	109	2916

study found that 50% of classes in the corpus have between one and six NIV, have less than five WMC, and between four and 38 LOC. The study did not explore the size of packages.

The averages in our study show that the Java API classes have about the same number of NIV, double the number of WMC for abstract classes but only slightly more for concrete classes and interfaces, and substantially more LOC for abstract and concrete classes.

Table 4: Averages for NIV, WMC, LOC for classes and packages.

Averages				
Type	NIV	WMC	LOC	Classes
Abstract Classes	1.65	12.96	120.35	N/A
Classes	1.98	8.57	106.07	N/A
Interfaces	0.24	6.22	11.36	N/A
Classes Average	1.64	8.60	91.05	N/A
Packages Average	51.67	270.08	2858.52	31.68

5.4 Java API Usage

A recent empirical study of ours explored the usage patterns of the Java Standard API 1.4.2 with an early version of the Qualitas Corpus which contained 39 open source Java applications [11]. They chose to use Java 1.4.2 API because most of the open source applications used in the study were originally compiled with Java 1.4.2. The results of the study showed that about 50% of the classes and 21% of the methods in the Java Standard API 1.4.2 are used at all. The java.lang package and String class are the most frequently used from the study.

We wanted to find out if there is any correlation between the *usage* of the Java API packages and classes in the Qualitas Corpus with the *size* of Java API 1.6 packages and classes. We do this by seeing where the largest packages and classes are ranked in the most frequently used lists and vice versa. Even though we are comparing the usage of the Java API 1.4.2 packages and classes with the size of the Java API 1.6 packages and classes, we still believe the frequency data is highly likely to remain the same. That is the java.lang package and String class are still the most frequently used.

The largest package in the API is the javax.swing package but it only appears at position six in the most frequently used Java packages. Two sub javax.swing packages (javax.swing.text and javax.swing.text.html) also appear in the top most frequently used packages. The most frequently occurring package java.lang has an even spread of different sized classes and appears at position 10 in the largest packages list. 12 of the top 20 most frequently used Java API packages appear in the top 20 of the largest packages. This would suggest that there no correlation between the usage and size of a package.

None of the most frequently used classes appear in the top 20 largest classes list. This would strongly suggest that there is no correlation between usage and size. This could also mean that the majority of the software in the early version of the Qualitas Corpus is unlikely to have graphical user interfaces built using the Swing package.

5.5 Visualization Wall

The issues we found with the visualization wall is that it lacked good interaction and collaborative capabilities. The only current way to interact with the visualizations on our visualization wall is from a control machine sitting a couple of metres back from the wall. The visualization wall only allowed repositioning and scaling the resolution of the images on the visualization wall using a mouse. However, these two important interaction tasks were quite cumbersome. The limited control screen doesn't actually show the details of an image only the outline, so making adjustments to the image had a delayed effect on the user.

6. FUTURE WORK

In the future we would like to redo the previous usage study [11] with the Java Standard API 1.6. We would then like to apply Spearman's rank correlation coefficient statistical technique[8] to statistically confirm if there is any correlation between the usage and size of packages and classes.

To help end-users we would like to create a more comprehensive software visualization tool. We would create a tool that allows users to explore the API by implementing more Polymetric Views such as System Complexity and Class Blueprint and augmenting them with Javadoc.

Ben Shneiderman [14] claims that gigapixel displays will be useful for some tasks, but innovative interface design is likely to have higher payoffs and wider usage. For our research we need better ways to be able to display the visualizations connected together, to interact and navigate within the visualization environment, and support co-located collaborative interaction. We would like to investigate how our visualizations could benefit from interfaces that are designed for collaborative work such as multi-touch tables which support multi-touch interaction gestures like zoom and scale.

7. CONCLUSIONS

The Java Standard API 1.6 is now very large and contains over 200 packages, nearly 5800 classes, and nearly 50,000 methods. We wanted to find out if there exists any large packages and classes in the Java API. We have created System Hotspot Views to identify large packages and classes in the API. Our visual software analysis has identified that there exists a number of large packages and classes in the Java Standard API 1.6. The javax.swing package is the largest package and java.awt.Component is the largest class. We found that the average number of WMC and LOC for abstract and concrete classes in the Java Standard API against the Java software in the Qualitas Corpus is much larger. We found that that there is no correlation between the usage and size of Java API packages and classes.

Acknowledgments

This work is supported by the New Zealand Foundation for Research Science and Technology for the Software Process and Product Improvement project, and a TelstraClear scholarship.

8. REFERENCES

[1] Anslow, C., Noble, J., Marshall, S., and Tempero, E. Visualizing the word structure of Java class names. In *Companion to the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2008), ACM Press, pp. 777–778.

[2] Anslow, C., Noble, J., Marshall, S., and Tempero, E. Towards visual software analytics. In *Proceedings of the Australasian Computing Doctoral Consortium (ACDC)* (2009), Australian Computer Society, Inc.

[3] Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., and Tempero, E. Understanding the shape of Java software. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (October 2006), ACM Press, pp. 397–412.

[4] Chidamber, S. R., and Kemerer, C. F. Towards a metrics suite for object oriented design. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (1991), ACM Press, pp. 197–211.

[5] Diehl, S. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer Verlag, 2007.

[6] Fenton, N. E., and Pfleeger, S. L. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing, 1998.

[7] Gil, J. Y., and Maman, I. Micro patterns in Java code. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2005), ACM, pp. 97–116.

[8] Lanza, M., and Ducasse, S. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering 29*, 9 (2003), 782–795.

[9] Lanza, M., and Marinescu, R. *Object-Oriented Metrics in Practice*. Springer Verlag, 2005.

[10] Lindsay, J. Why are classes big? Honours Report, Victoria University of Wellington, October 2009.

[11] Ma, H., Amor, R., and Tempero, E. Usage patterns of the Java standard API. In *Asia Pacific Software Engineering Conference (APSEC)* (2006).

[12] Qualitas Research Group. Qualitas corpus version 20090202, February 2009. The University of Auckland, `http://www.cs.auckland.ac.nz/~ewan/corpus`.

[13] Reas, C., and Fry, B. *Processing: A Programming Handbook for Visual Designers and Artists*. MIT Press, 2007.

[14] Shneiderman, B. Extreme visualization: squeezing a billion records into a million pixels. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)* (2008), ACM Press, pp. 3–12.

[15] Singer, J., Brown, G., Lujan, M., Pocock, A., and Yiapanis, P. Fundamental nano-patterns to characterize and classify Java methods. In *Proceedings of the Workshop on Language Descriptions, Tools and Applications (LDTA)* (2009).

[16] Stasko, J. T., Brown, M. H., and Price, B. A. *Software Visualization*. MIT Press, 1997.

[17] Stylos, J., Faulring, A., Yang, Z., and Myers, B. A. Improving API documentation using API usage information. In *Proceedings of the IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC)* (2009), IEEE Computer Society Press, pp. 119–126.

[18] Tempero, E. How fields are used in Java: An empirical study. In *Australian Software Engineering Conference (ASWEC)* (2009).

[19] Tempero, E., Noble, J., and Melton, H. How do Java programs use inheritance? an empirical study of inheritance in java software. In *Proceedings of European Conference on Object-Oriented*

[8] `http://en.wikipedia.org/wiki/Spearman's_rank_correlation_coefficient`

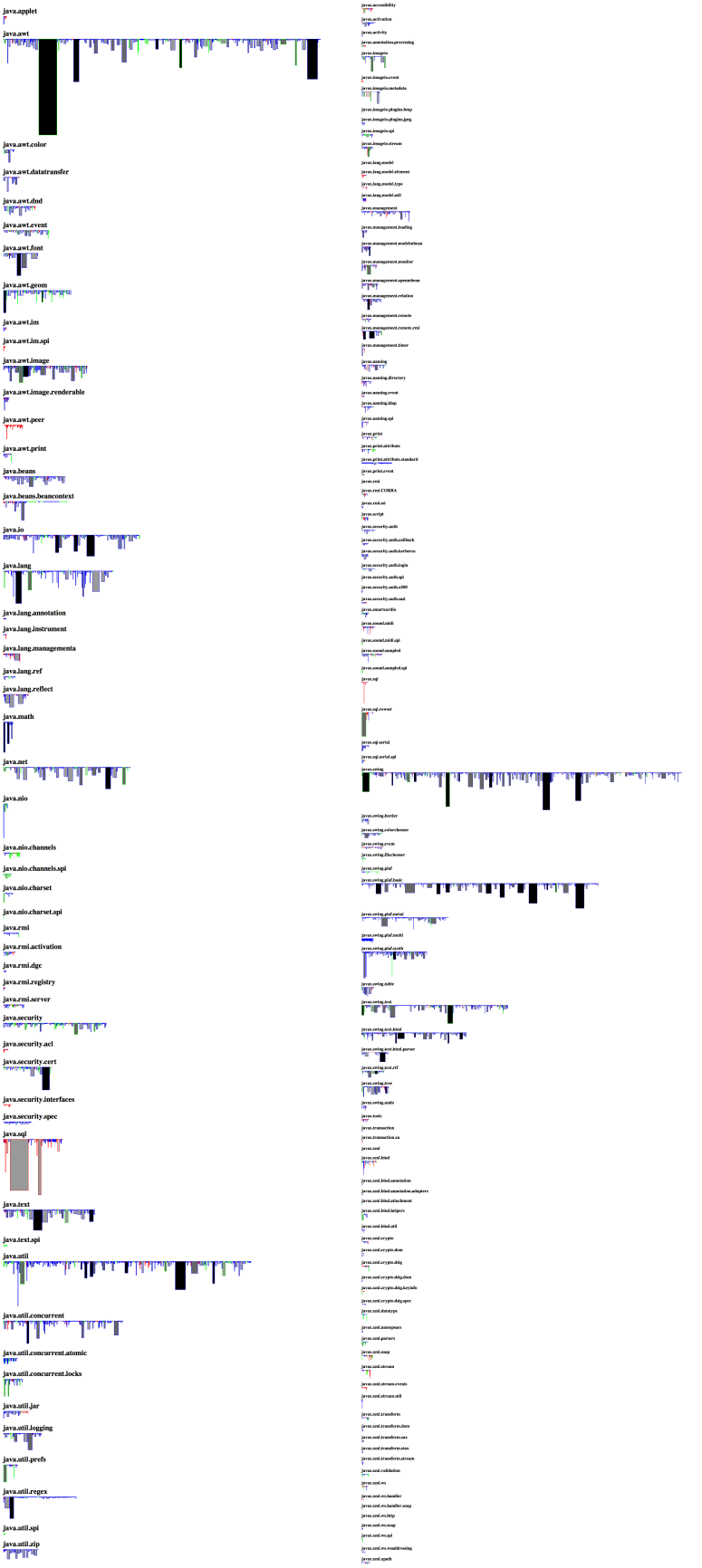
Programming (ECOOP) (2008).

[20] Thomas, J. J., and Cook, K. A., Eds. *Illuminating the Path: The Research and Development Agenda for Visual Analytics*. National Visualization and Analytics Center, 2005.

[21] Verbaere, M., Hajiyev, E., and Moor, O. D. Improve software quality with SemmleCode: an Eclipse plugin for semantic code search. In *Companion to the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2007).

[22] Wettel, R., and Lanza, M. Visualizing software systems as cities. In *Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)* (2007), IEEE Computer Society Press, pp. 92–99.

[23] Zimmermann, T., Premraj, R., and Zeller, A. Predicting defects for Eclipse. In *Proceedings of the International Workshop on Predictor Models in Software Engineering (PROMISE)* (2007), IEEE Computer Society Press.



(a) java packages (b) javax packages

Figure 4: The java and javax packages of the Java API 1.6

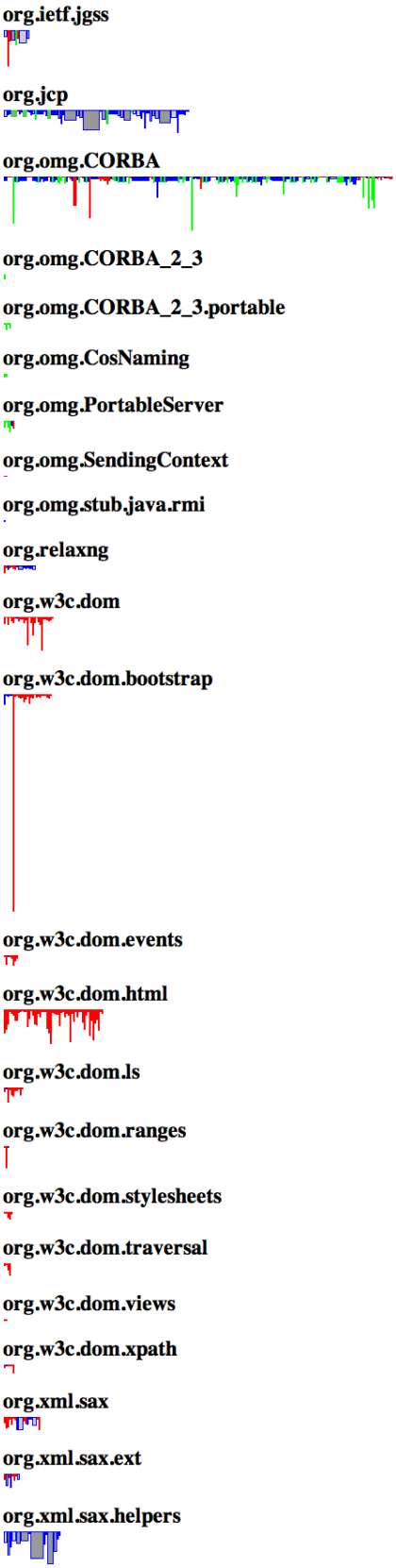


Figure 5: The org packages of the Java API 1.6