Towards Visual Software Analytics

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

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

Abstract

Since its inception, a large amount of software has been written in Java and surprisingly little is known about the structure of Java programs in the wild. There are very few software visualization tools for analytical reasoning of Java software. We are creating a visual software analytics tool that will help to characterize our Java software corpus. Our tool will help to provide insight into a collection of Java programs, detect the expected, and discover the unexpected.

Keywords Visual Analytics, Software Visualization, Software Corpus, Java

1. Introduction

Large amounts of Java software have been written since the language was first created. We have almost no dependable data on what software coding standards such as Java class names are adhered to in practice (4). Creating visualizations will help to discover trends and commonalities in structure and behaviour of Java software.

We have a corpus of Java software[1] (10) used for conducting empirical studies to help understand how software engineers create code and the relationship between the code structure and quality attributes such as modifiability, reusability, maintainability, and testability. The corpus contains 91 distinct open-source Java applications. 22 of these applications have multiple versions, comprising 233 versions total. Our project requires better techniques for understanding and mining the software from the corpus.

We are interested in visualizing the evolution of the Java API, the characteristics of Java software, and the usage of the Java API within Java software. We believe creating visual analytic tools and techniques will help to derive insight about Java software.

This paper outlines our approach towards visual software analytics of Java software, our methodology for the thesis, and some preliminary results of creating software visualizations from the Java API and our software corpus.

[1] http://www.cs.auckland.ac.nz/~ewan/corpus/

2. The Need for Visual Analytics

Since the beginnings of software visualization research the field has focused primarily on algorithm animation (1980s), software architecture (1990s), and software evolution and mining from software repositories (2000s). As far as we are aware, to date no research has considered applying visual analytic techniques to provide insight into the structure and behaviour of large Java software corpora.

Visual analytics is a new research field and is defined as the science of analytical reasoning facilitated by interactive visual interfaces (14). The goal of visual analytics is the creation of tools and techniques to enable people to:

- Synthesize information and derive insight from massive, dynamic, ambiguous, and often conflicting data.
- Detect the expected and discover the unexpected.
- Provide timely, defensible, and understandable assessments.
- Communicate assessment effectively for action.

Visual analytics is a multidisciplinary field that includes the following focus areas (14):

- Analytical reasoning techniques that enable users to obtain deep insights that directly support assessment, planing, and decision making.
- Visual representations and interaction techniques that take advantage of the human eye's broad bandwidth pathway into the mind to allow users to see, explore, and understand large amounts of information at once.
- Data representations and transformations that convert all types of conflicting and dynamic data in ways that support visualization and analysis.
- Techniques to support production, presentation, and dissemination of the results of an analysis to communicate information in the appropriate context to a variety of audiences.

Some notable visual analytics systems include In-Spire (17), Jigsaw (13), and Improvise (16). However, none of these systems focus on the domain of software; they instead use document collections. Most software visualization systems (5; 12; 18) in the past have focused on visualizing just one piece of software at one time and using one or more visualization techniques.

We view visual analytics as a superset of information visualization, software visualization, and empirical software engineering. In order to conduct analysis about collections of software we need to visualize multiple data sets of software at once from our software corpus. The visualizations will help provide insight into a collection of programs 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, micro-patterns (7)).

3. Methodology

The objective of our Software Process and Product Improvement project[2] is to develop and apply a range of software productivity techniques and tools to enhance the performance of the New Zealand software industry. The key focus is software process and product improvement using advanced, model-based software visualisation methods and tools. Before we start to build our visual software analytics tools we plan to conduct a survey and various user studies with developers from industry. The survey and user studies will help guide us in building our visual software analytics tools.

The aim of the survey is to provide an overview of the New Zealand software development industry. The focus of the survey is on the use of software visualization techniques and tools for software comprehension. We want to discover what kind of software comprehension strategies participants undertake (e.g. bottom-up, top-down), what comprehension activities they are interested in (e.g. investigating the internal structure of an artifact), and what kind of questions they ask when comprehending a system (e.g. what is the class structure of the software system?). We want to find out what their definition of software visualization is and if they use any of the tools mentioned in the books on software visualization (5; 12; 18). It will be useful to find out what stages of the software development cycle they use the tools, what kind of artifacts they visualize, and what techniques they use. We also want to know if they are building their own software visualization tools or prototypes. Finally, we would like to know what tools and techniques they would like and where would they liked the software visualizations displayed (e.g. IDE, desktop, web).

After completing the survey we intend to do some observational studies of developers programming. The aim of these studies will be to observe what developers actually do on a daily basis and how much time they spend doing the various development activities. We plan to do some follow up in-depth interviews with participants of our survey and observation studies, in particular ones who use any software visualization techniques or tools. The questions for the interviews will be developed once we have completed the survey and observation studies. We plan on performing studies that track the eyes of developers when they are developing and maintaining software. Tracking the movement of the developer's eyes will help guide us to which parts of the code they are most interested in and which features of the IDE and other tools they use most. Finally, we plan to do some user studies on a selection of existing software visualization tools (5; 12; 18) with our participants and fellow students to highlight the good and bad points of each of these tools, again as a guide to building our own tools.

4. Java Software Visualizations

Our main platform focus for software visualizations to date has been to deliver visualizations over the web (1; 6; 9). There exist very few web-based software visualization tools (3). Our platform of choice may change due to the outcomes of our survey and user studies. In the meantime we have decided to explore using a successful web based visualization application called Many Eyes (15) to see how a system such as this could potentially work for a visual software analytics tool. Many Eyes is a web site provided by IBM research that provides collaborative visualization services. The site is set up to allow users to upload data in ASCII format, visualize it, and then talk about their discoveries with other people.

[2] https://wiki.auckland.ac.nz/display/csisppi

We have conducted experiments of visualizing the words used in class names of the Java API JavaDoc and from our software corpus (2) using Many Eyes. First, we have visualized the words in the class names from the Java API version 1.6. Second, we have visualized the ordering of words in the Java class names. Third, we have visualized the evolution of words in class names from Java version 1.1 and Java version 1.6. Finally, we have visualized the words in public class names from the 91 open-source Java applications in our software corpus. We do not distinguish between interfaces and classes.

Figure 1 shows a Tag Cloud visualization of the words used in the class names from the Java API version 1.6. For example the CamelCase class name AbstractColorChooserPanel becomes Abstract (position one), Color (two), Chooser (three), and Panel (four). The tag cloud shows the most common words used in Java class names are Exception (381 occurrences), UI (133), Helper (128), Type (120), Event (116), and Factory (99). The word Exception is used twice as many times as any other word in the Java API which shows that the design of Java relies quite heavily on capturing exceptions at run-time.

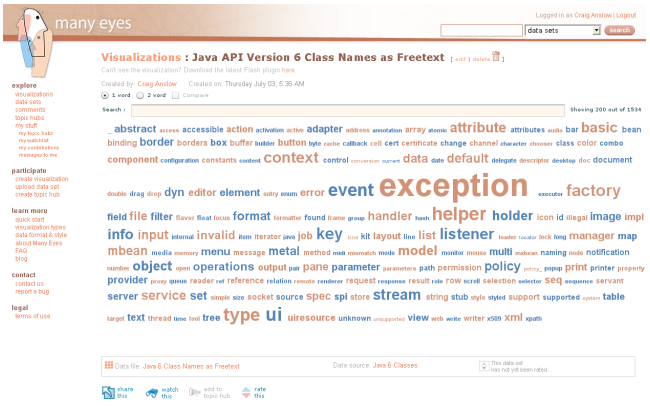

Figure 1. Tag Cloud Visualization of the words used in the class names from the Java API version 1.6.

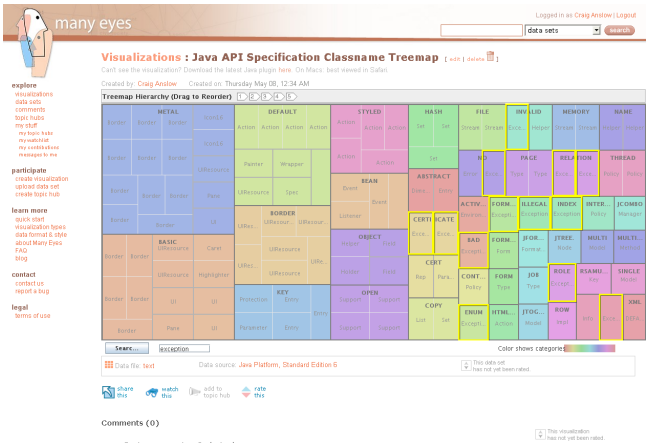

Figure 2. Tree Map Visualization of the ordering of words used in the class names from the Java API version 1.6.

Figure 2 is a Tree Map visualization which shows the ordering of the words in class names from Java 1.6. The current screenshot of the visualization shows the words in the class name ordered as in a Java class name (e.g. AbstractColorChooserPanel represented

as Abstract (position one), Color (two), Chooser (three), and Panel (four)) and the word Exception highlighted. There are 1217 unique words in position one, 761 in position two, 409 in three, 186 in four, and 70 in five. The tree map allows a user to change the order of the words in the class name to see which words are the most prominent in each position in the class name. The tree map shows that the most prominent word in position one is the word Metal followed by Basic, Default, Order, and then Key. However, the most prominent word in positions two, four and five are variants on the word Border. The word Exception is most common in positions four and five.

Figure 3 shows a comparison of the class names between Java version 1.1 (red colour) and 1.6 (blue colour). Java 1.1 contains 477 classes and Java 1.6 contains 3777 classes. This is an interesting visualization as it shows how the words used in the Java API have evolved over time. All of the words used in Java 1.1 have also been used in Java 1.6, there is no word that has been so called deprecated. There are, however, a number of additional words used in Java 1.6 which is to be expected being a more recent version. The word Exception is the most prominent word in both versions. There are no words associated with XML (e.g. XML, XPath) in the version 1.1 tags which suggests that this version of Java did not have any XML libraries at that time.

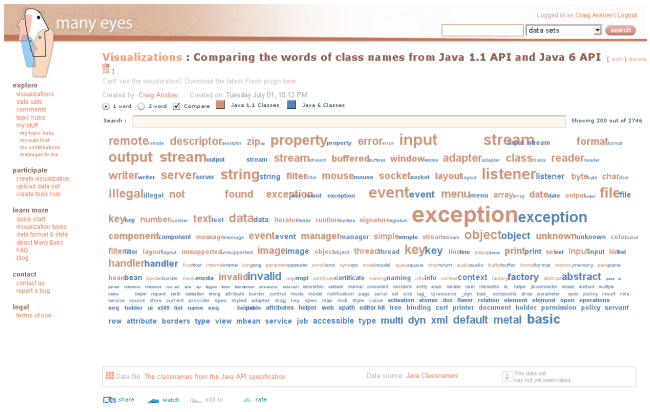

Figure 3. Comparison Tag Cloud Visualization of the words used in the class names of Java 1.1 (red) and Java 1.6 (blue).

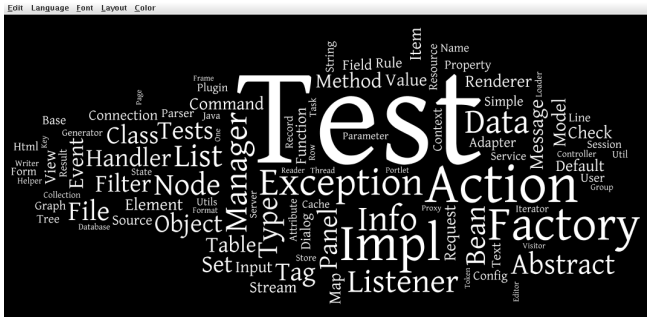

Figure 4. Word Cloud Visualization of the words used in the class names from our software corpus which contains 91 applications.

Figure 4 shows a visualization of the class names from the 91 open-source Java applications in our software corpus. We only considered public class names. The visualization contains approximately 51,000 classes. The most common words that are used are Test (3847 occurences), Action (1541), Impl (1451), Factory (1333), Exception (1089), and Data (948). This suggests that there is an emphasis of testing in these applications and perhaps a test driven development approach was followed.

Figure 5 shows a word tree visualization of the class names from the software corpus. Selecting a word shows all the different contexts in which it appears and displayed in a tree-like branching structure. Bean and Info, together appear 406 times, List and Model, appear 125 times, Test and Case, appear 98 times, and finally Factory and Bean, appear 79 times. This technique is useful for exploring which groups of words in class names are common and is similar to the tree map visualization. Note that Bean (appears 804 times) nor Info are among the top five most common words from Figure 4 but appear the most together. The word Test seems to be used together with a wide variety of words as opposed to Bean. Perhaps these applications make use of a lot of Java Beans or the Hibernate framework.

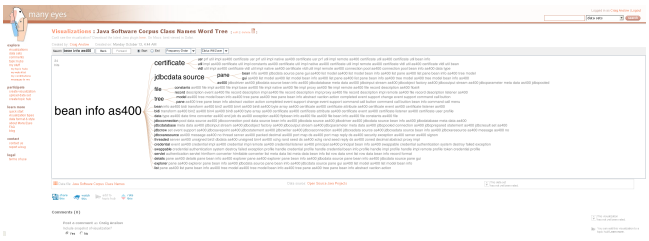

Figure 5. Word Tree Visualization of the words used in the class names from our software corpus which allows users to explore the ordering of words.

To create our own visual software analytics tools we need to either build our own visualization toolkits or use existing ones. We are currently exploring creating visualizations using existing information visualization toolkits incuding: prefuse (8), Large Graph Layout (LGL) and the Cairo graphics package, and Processing (11). Figure 6 shows Java 1.6 class to package relationships using prefuse (8). Packages are green, interfaces are red, classes are blue, annotations are grey, and enums are yellow. Relationships between packages are entities (e.g. classes) which belong to different packages but have the same name. This visualization shows that there are many entity names in Java 1.6 that have the same name but are located in different packages. No one entity name stood out more than the others.

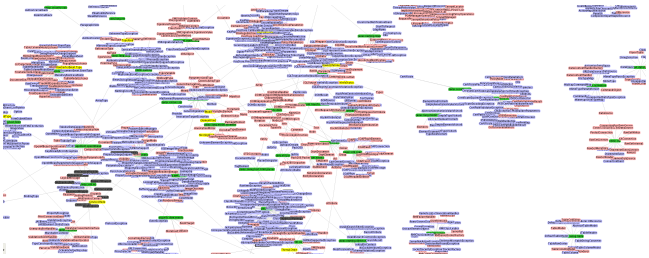

Figure 6. Java Class to Package Relationships in Java 1.6.

Figure 7 shows inherited class relationships in Java 1.4.2 implemented with Cairo and LGL. Nodes are classes and links represent relationships between inherited classes. This kind of visualization is very dense but it can give a quick overview to which are the most common classes that are inherited. When viewing this visualization in high definition greater details of the relationships are more visible. In this visualization the most inherited class is java.lang.Object and one would suspect this given that this class is the root of all

classes in Java. The java.lang.Object class node is located just to the left of the centre of the image with many outgoing purple links.

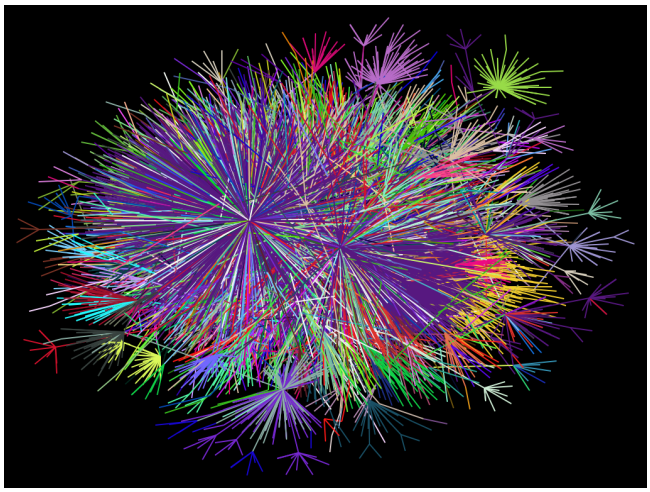

Figure 7. Use of Inheritance in Java 1.4.2.

Viewing these visualizations on standard desktop machines limits the amount of display screen space. We are beginning to use an OptIPortal visualization cluster to display multiple visualizations at once. The visualization wall has 12 screens arranged 4x3. Each individual display is 2560 x 1600 pixels 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, and video-conferencing and collaboration with other portals. Figure 8 shows some of our Java class name visualizations all displayed at once.

Figure 8. OptIPortal - Visualization Cluster displaying our visualizations.

5. Conclusion

Our approach to understanding and comprehending existing Java software is to apply visual analytics techniques. We intend to create a visual software analytics tool to understand the Java software in our corpus. We will be surveying software developers to find out what software visualization practices are used in industry; conducting user studies to see what they do on a daily basis, if any software visualization tools are used in practice, and what comprehension activities and strategies are followed. So far we have created some visualizations of the class names used in the Java API JavaDoc and form our software corpus which contains 91 open-source Java applications. Our visualizations detected that the most common words used in Java class names are Test, Action, Impl, and Exception. Knowing what the most common words in Java can help developers create coding standards for their class names.

Acknowledgements

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. Hayden Smith for implementing the inheritance visualizations.

References

[1] Craig Anslow, James Noble, Stuart Marshall, and Robert Biddle. Web Software Visualization Using Extensible 3D (X3D) Graphics. In Proceedings of *SoftVis*, pages 213–214. ACM, 2008.

[2] Craig Anslow, James Noble, Stuart Marshall, and Ewan Tempero. Visualizing the Word Structure of Java Class Names. In Companion to *OOPSLA*, pages 777-778. ACM, 2008.

[3] Craig Anslow, James Noble, Stuart Marshall, and Ewan Tempero. Towards End-User Web Software Visualization. In Proceedings of *Graduate Consortium at VLHCC*. IEEE, 2008.

[4] Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. Understanding the Shape of Java Software In Proceedings of *OOPSLA*, pages 397–412. ACM, 2006.

[5] Stephan Diehl. Software Visualization Visualizing the Structure, Behaviour, and Evolution of Software. Springer, 2007.

[6] Matthew Duignan, Robert Biddle, and Ewan Tempero. Evaluating Scalable Vector Graphics For Use in Software Visualisation. In Proceedings of *APVIS*, pages 127–136. ACS, 2003.

[7] Joseph (Yossi) Gil and Itay Maman. Micro patterns in Java code. In Proceedings of *OOPSLA*, pages 97–116. ACM, 2005.

[8] Jeffrey Heer, Stuart K. Card, and James A. Landay. prefuse: a Toolkit for Interactive Information Visualization. In Proceedings of *CHI*, pages 421–430. ACM, 2005.

[9] Stuart Marshall, Kirk Jackson, Robert Biddle, Michael McGavin, Ewan Tempero, and Matthew Duignan. Visualising reusable software over the web. In Proceedings of *APVIS*, pages 103–111. ACS, 2001.

[10] Hayden Melton and Ewan Tempero. The CRSS metric for package design quality. In Proceedings of *ACSC*, pages 201–210. ACS, 2006.

[11] Casey Reas, Ben Fry, and John Maeda. Processing: A Programming Handbook for Visual Designers and Artists. MIT Press, 2007.

[12] John T. Stasko, John B. Domingue, Marc H. Brown, Blaine A. Price. Software Visualization Programming as a Multimedia Experience. MIT Press, 1998.

[13] John Stasko, Carsten Görg, and Zhicheng Liu. Jigsaw: Supporting Investigative Analysis through Interactive Visualization. In *Information Visualization*, pages 118–132, Vol. 7, No. 2. Palgrave Macmillan, 2008.

[14] James J. Thomas and Kristing A. Cook. Illuminating the Path: The Research and Development Agenda for Visual Anaytics. IEEE, 2005.

[15] Fernanda B. Viegas, Martin Wattenberg, Frank van Ham, Jesse Kriss, and Matt McKeon. ManyEyes: a Site for Visualization at Internet Scale. In *Transactions on Visualization and Computer Graphics (TVCG)*, pages 1121–1128, Vol. 13, No. 6. IEEE, 2007.

[16] Chris Weaver. Building Highly-Coordinated Visualizations in Improvise. In Proceedings of *InfoVis*, pages 159–166. IEEE, 2004.

[17] J. A. Wise, J. J. Thomas, K. Pennock, D. Lantrip, M. Pottier, A. Schur, and V. Crow. Visualizing the non-visual: spatial analysis and interaction with information from text documents. In Proceedings of *InfoVis*, pages 51–58. IEEE 1995.

[18] Kang Zhang. Software Visualization: From Theory to Practice. Kluwer, 2003.