Exploring the Inventor's Paradox: Applying Jigsaw to Software Visualization

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

ABSTRACT

Software visualization research has typically focussed on domain specific tools and techniques. In this paper, we evaluate applying a general purpose visual analytics tool Jigsaw to visualize the JHotDraw open source software system. We describe how Jigsaw can be applied to visualize software, and show how it can support some program comprehension tasks.

Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*evaluation/methodology*; H.4.3 [**Information Systems Applications**]: Communications Applications—*information browsers*; H.1.2 [**User/Machine Systems**]: Human Factors

General Terms

Design, Human Factors

Keywords

Software visualization, visual analytics

1. INTRODUCTION

Software visualization research has traditionally focussed on specialist tools and techniques. These tools and techniques are designed to provide users with certain insights regarding some (typically) pre-determined aspect of software systems. Ongoing research with these tools and techniques have also contributed insights into the inherent challenges of — and strategies for — visually representing various aspects of software systems. However these tools and techniques have generally not seen widespread use. Also, they are typically not capable of being generalized for issues they are not specifically designed for. This problem can be viewed in the context of the Inventor's Paradox:

> "The solution to a general problem is simpler than a solution to a specific problem, the solution of many problems is simpler than the solution to one problem." [16]

Recently, visual analytics research has resulted in the development of general purpose visualization tools. In this paper, we apply Jigsaw [2] — a general purpose visual analytics tool — to a software comprehension problem. Our case study shows that general purpose visual analytics tools can be used to meet some of the goals of software visualization and program comprehension.

The remainder of this paper is organized as follows. Section 2 discusses related work that motivates our study. Section 3 outlines our methodology. Our case study of the Jigsaw visual analytics tool and the open source JHotDraw software application is presented in Section 4. We evaluate and discuss the case study in Section 5. Lastly, we summarise the contributions of this research, and outline future work and directions in Section 6.

2. RELATED WORK

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). Most existing software visualization tools are special purpose and focus on visualizing just one piece of software at one time [6].

There are many special purpose software visualization tools. Code Crawler [10] visualizes software metrics using Polymetric Views [9]. CodeCity [26] is stems from Code Crawler, that uses a 3D city metaphor based on Polymetric Views. Code City displays disharmony maps which show the quality of the system design by focusing on design flaws using metrics. Code Swarm [11] organically visualizes the commit history of open source projects using animations and displayed as videos.

Some general purpose software visualization tools support a range of tasks. SeeSoft [7] and SeeSys [4] visualize various textual features of evolving large and complex software systems. The features include software metrics, number and scope of modifications, number and types of bugs, and dynamic program slices. Jinsight [14] is a tool for visualizing and analysing the execution of Java programs and is useful for performance analysis, memory leak diagnosis, and debugging. SolidFX [22] is an IDE for reverse engineering C/C++ programs and provides many advanced visualization techniques to explore attributes of a code base including call graphs, software metrics, and UML diagrams.

SOFTVIS'10, October 25–26, 2010, Salt Lake City, Utah, USA.

Sensalire and Ogao [18] suggest that the needs of target users should be identified then tools developed to meet their needs. Schafer and Mezini [17] argue that software visualization tools are either too focused to be applicable to a broad range of tasks, or too generic to support specific tasks. They proposed a flexible software visualization framework that allows users to customize views for given tasks. These studies have identified a common theme – software visualization tools need to be more human-centered if they are to have a higher adoption rate within industry.

Visual analytics is an emerging research field that has evolved out of information visualization. The goal of visual analytics [23] 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; and communicate assessment effectively for action. There are many general purpose visual analytics tools including In-Spire [27], WireVis [5], and Jigsaw [21]. None of these tools focus on the domain of software; rather they visualize structured and unstructured text document collections.

Diehl [6] claims that visual analytics has yet to reach software visualization. We view *visual software analytics* as the intersection of visual analytics, information visualization, software visualization, and empirical software engineering. Visual software analytics will help provide insight into 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, and micro-patterns).

Few researchers have evaluated existing general purpose information visualization tools for software visualization. A recent study of ours employed ManyEyes [25] a general purpose web visualization tool to explore the coding standards used for class names [3]. The case study looked into the words used in Java software and the Java Standard API 1.6. The results identitfied that the most common words used in Java class names are Test, Action, Impl, and Exception and that a test-driven development approach was used to build the Java software.

The contribution of this paper is to employ a general purpose visual analytics tool to explore Java software for program discovery and program maintenance tasks.

3. METHODOLOGY

Our objective is to evaluate how general-purpose visual analytics tools perform in visualizing software. We evaluated the Jigsaw visual analytics tool through exploration of a representative open source application, driven by two program comprehension activities. The observations and insights from the case study are presented in Section 4.

3.1 Apparatus

There are two software systems used in our case study: the Jigsaw visual analytics tool and the JHotDraw open source application.

3.1.1 Jigsaw

Jigsaw is a document-focused visual analytics system designed for investigative analysis, such as that found in the law enforcement and intelligence domains [2]. Jigsaw supports exploration of a large collection of documents by visualizing connections between entities across documents. Jigsaw provides built-in entity-identification that can recognize entity types such as people, places, money, organizations, dates, and user-defined entity types. Entities are connected to a document by appearing in that document; two entities are connected to each other if they both appear in the same document. The strength of entity pair connections is measured by the number of documents in which they both appear. Jigsaw assumes that documents are in natural language. We will apply Jigsaw to highly structured object-oriented source code.

3.1.2 JHotDraw

JHotDraw is an open source Java graphics framework, whose design is strongly based upon software design patterns [1]. JHotDraw has been used as the test application for a variety of software visualization evaluations [12, 13] as well as in case studies in the object-oriented design and programming literature [20, 24]. We used JHotDraw version 7.1 which contains 731 classes, 38 packages and 53,718 lines of code.

As an open source project and framework, users can choose to contribute to the source code base. Any open source developer who wishes to join the project (or contribute a modification) would first have to understand its source code – overcoming program comprehension problems that software visualization aims to mitigate. Therefore, evaluating Jigsaw by attempting to understand JHotDraw replicates real world problems faced by real developers.

3.2 Evaluator

The evaluation was undetaken by Haowei Ruan. We briefly describe the evaluator's pre-study knowledge, so as to provide context for the steps taken and the observations and insights gathered.

The evaluator had some prior knowledge of the Jigsaw system through having read Jigsaw's tutorials and research papers, and had applied Jigsaw to some small sample Java applications to gain some basic familiarity with the interface and visualization model. The evaluator has not however used Jigsaw on an ongoing basis in everyday work. The evaluator had a superficial knowledge of JHotDraw from having read of its use in other studies, however the evaluator had no experience in using JHotDraw in any project, and had no knowledge of the underlying source code base. The evaluator has six years of practical experience in using Java, and has a sound theoretical grounding in the language.

3.3 Entity/Document Concept Mapping

As mentioned earlier Jigsaw can identify entities in a document, and one common use is identifying entities such as people, organizations and places in natural language text. Natural language entities can appear in source code comments (e.g. authors, organizations, and modification dates). However for the purpose of this study we do not assume that all source code is extensively and consistently commented, and in this study we focus on understanding the program itself.

We need to define meaningful entities in the Java language domain. Past work on visualizing static aspects of object-oriented programs has mostly used entities such as package, class, method, and attribute. These entities represent key structural elements of an object-oriented program.

Similarly, we defined entity types as shown in Table 1. Note that we did not use attributes as entities in our case study, although the relevant entity types could easily be added if needed.

Many of Jigsaw's views are also based on the underlying information being divided into a collection of text documents. Logically, the Java source files are the set of text documents that are fed into Jigsaw. It is worth noting though that a consequence of this is that if the Java source file is huge, then it may potentially contain so many entities that the display is swamped. This would diminish the visualization's usability due to information overload [21]. We did not encounter this issue in our case study.

3.4 Procedure

Our evaluation was an exploratory process, and loosely followed the sorts of tasks used in existing software visualization evaluations [13, 15].

We first preprocess the JHotDraw source code based on the concept defined in section 3.3 by extracting entities for each entity types defined utilizing a Java parser and store them into entity dictionaries. The Java source files are then loaded into Jigsaw followed by executing the entity identification process that uses the entity dictionaries to identify the custom defined entities.

The evaluator took on the persona of an open source developer who is firstly exploring the JHotDraw project, and then who is secondly maintaining some aspect of the project. The evaluator analyzed the initial presentation of entity connections with the classes and packages. The evaluator then used different views supported by Jigsaw to answer whatever question was forefront in their mind. We avoided prescribing the order in which views were used.

4. CASE STUDY

In this section, we present the results of our case study applying Jigsaw to JHotDraw. Our case study follows two basic scenarios: program discovery and program maintenance. Developers carry out general program discovery activities in order to become familiar with an unknown program, and specific program maintenance activities to modify part of the program to meet a requirement. These scenarios and tasks are adapted from previous studies that evaluate software visualization tools [13, 15]; Sensalire et al.[19] categorize these tasks into program discovery and program maintenance.

4.1 Scenario 1 : Program Discovery

The first scenario used Jigsaw to get a general overview of the JHotDraw source code. Reading any non-trivial software system line-by-line is a time-intensive exercise, so quickly identifying relevant code snippets is important in exploring software.

In this first scenario, the exploration followed two tasks, taken from studies that evaluate software visualisation for open source developers [13]. These tasks are: to understand the code structure and relationships between entities within a package; and to understand the relationship and dependencies between the packages.

4.1.1 Initial Exploration

Jigsaw's *Document View* provides a *word cloud* that visualizes the frequency of entities within the supplied source code (Figure 1). Our evaluation starts here. Entities and

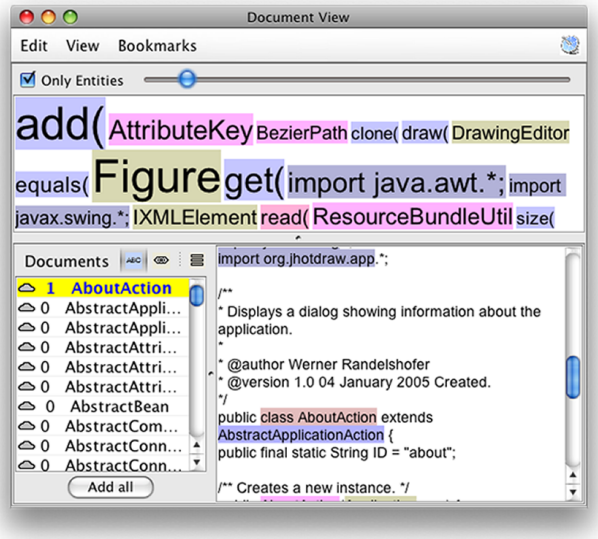

Figure 1: Overall document view.

words that occur regularly are shown in a large font, while those entities that occur infrequently are rendered in a small font. This is the best view to gain an overview of JHotDraw because the visualization naturally emphasizes the most frequently occurring entities. From the view shown in Figure 1 we observe that:

- *Figure* is the most frequently occurring interface entity in JHotDraw, followed by *DrawingEditor* and *IXMLElement* interfaces;
- *import java.awt.** and *import java.swing.** are the two most frequently occurring import entities;
- *add* is the most frequently occurring public method entity; and
- *AttributeKey* and *ResourceBundleUtil* are the two most frequently occurring class entities.

We can hypothesize some themes from this view:

1. JHotDraw uses Java AWT, Java Swing frameworks, and XML technologies;
2. JHotDraw mostly follows the "program to interface" practice because the most frequently used entity is an interface; and
3. it confirms our prior knowledge that JHotDraw's core function is a drawing editor.

4.1.2 Core Classes and Packages

Having identified the most frequently occuring entities, we then used Jigsaw's *List View* to explore the entities and their connections, especially in relation to their distribution across the source code files (Figure 2). A list view displays either documents, or entities of a specified type. Each entity in a list has an associated frequency visualized through a bar. The length of the bar represents the frequency, in the

Entity Name	Entity Definition
Def{Class|Abstract|Interface}	Declaration of a class, abstract class or interface respectively: e.g. *class Cycle* { ... } or e.g. *interface UnpoweredVehicle* { ... }
Def{Public|Protected|Private}	Declaration of a public, protected or private method
DefStatic	Declaration of a static method
Package	Declaration of a package
{Class|Abstract|Interface}	Reference to the use of a class, abstract class, or interface: e.g. *Cycle instance=new Cycle();* e.g. *class Moon extends Cycle;* — Cycle is used by Moon
{Public|Protected|Private}Method	Reference to the use of a method of the respective access level
StaticMethod	Reference to the use of a static method
Import	Reference to the use of the import keyword

Table 1: This table shows two different categories of entities. The top half of the table shows Definition-Type entities, while the bottom half shows Reference-Type entities. Note that the use of {}s indicates that there are related entities with a similar name structure.

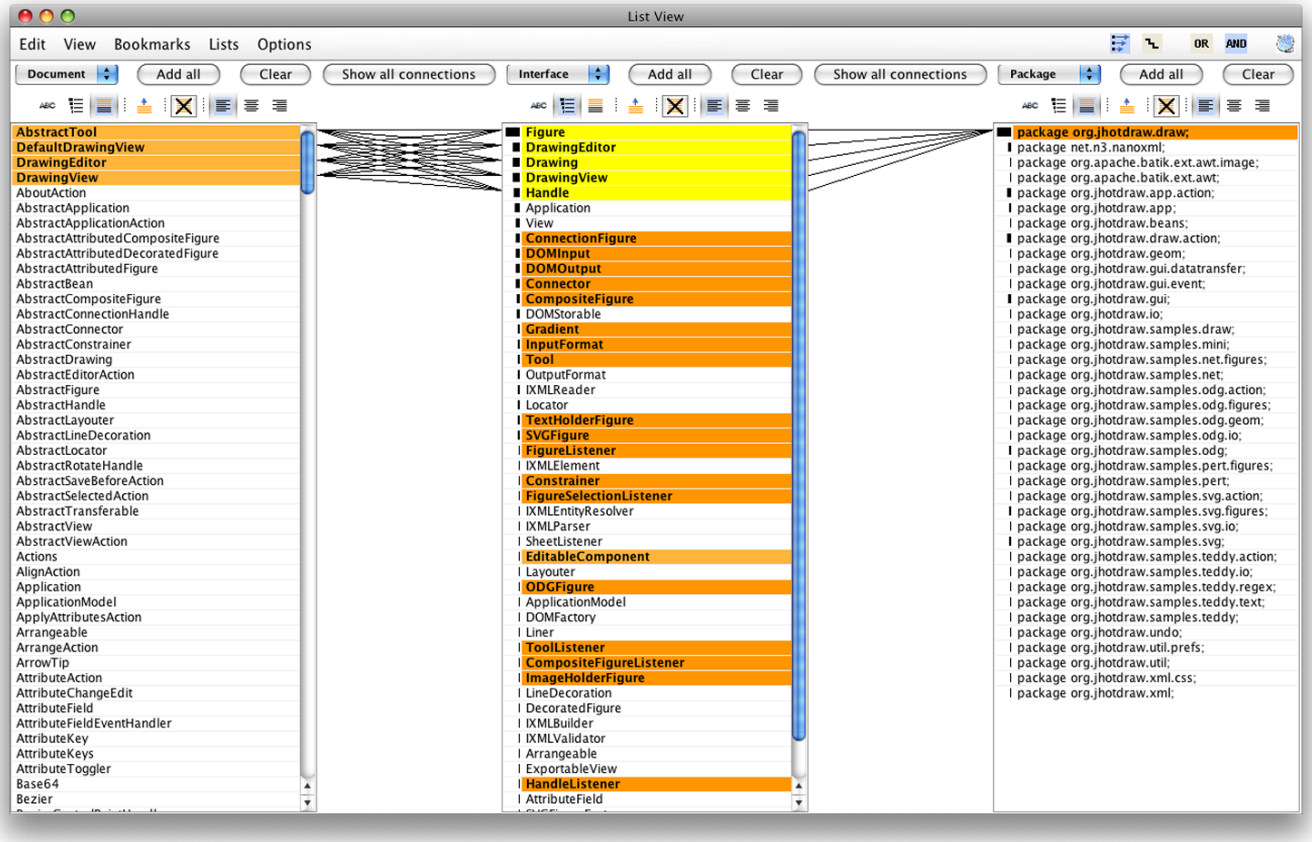

Figure 2: List view of Document, Interface, and Package entities.

number of documents the entity occurs in. The list view can be sorted by frequency. We have now identified which entities affect or are affected by the most number of source code files.

Figure 2 shows a list view with 3 lists: a list of all documents, a list of all interface entities, and a list of all package entities. The interface list is in descending order, and the *Figure* entity stands out as an interface that occurs in the most number of documents. Hovering the cursor over the bar reveals that the *Figure* entity occurs in 154 documents. This observation, together with the earlier observation from the word cloud, informs our understanding that the *Figure* interface is important to successful program comprehension. Similarly, the package *org.jhotdraw.draw* stands out with a lengthy frequency bar, and this shows that package *org.jhotdraw.draw* is the biggest (by document count) in JHotDraw. The package contains 146 separate source code documents.

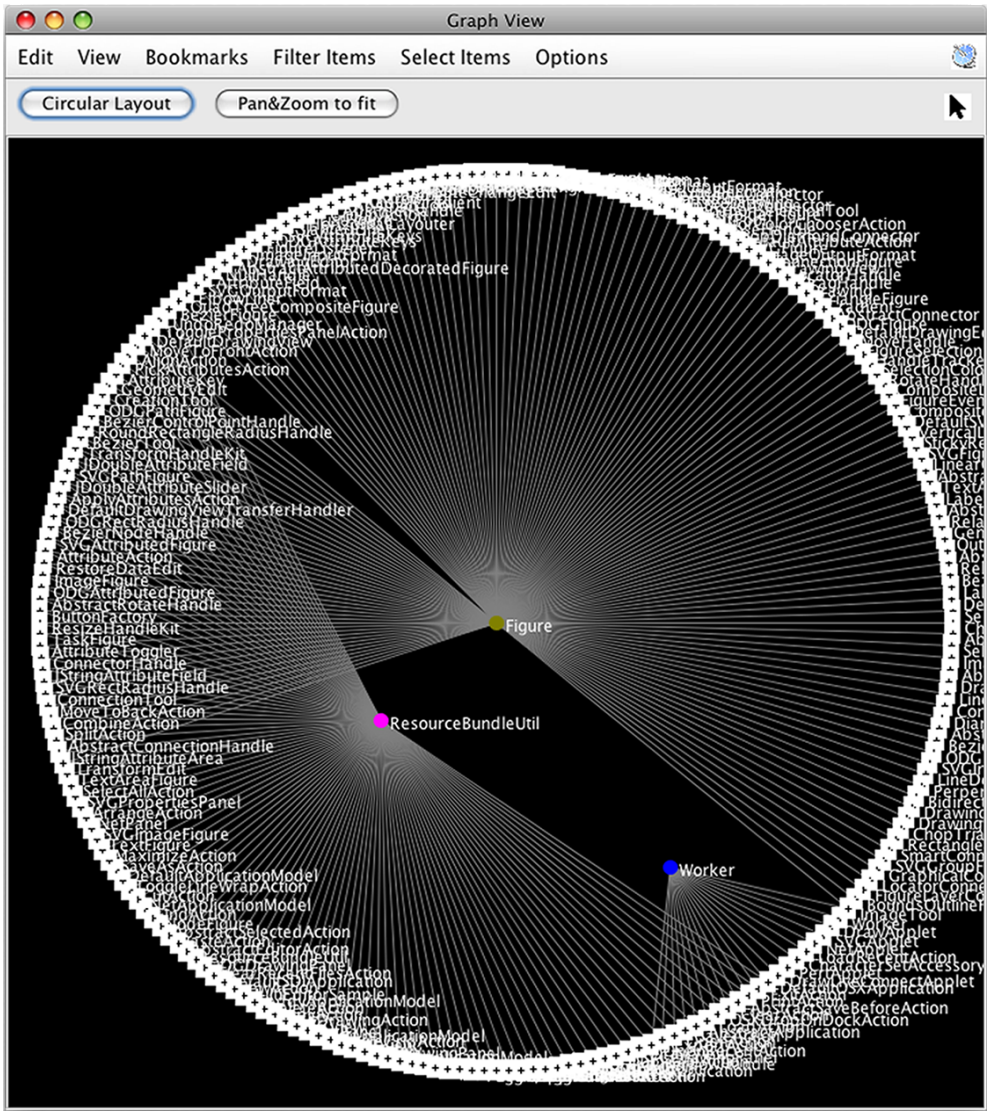

Figure 3: Graph view with Figure, ResourceBundleUtil, and Worker entities expanded.

Applying similar list view techniques revealed some other top occurring entities:

- class entity *ResourceBundleUtil* (113 occurrences),
- abstract class entity *Worker* (21 occurrences),
- import entity *import java.awt.** (262 occurrences),
- *import java.swing.** (180 occurrences), and
- *import org.jhotdraw.util.** (159 occurrences).

In this Jigsaw view, selecting an entity in a list highlights the entity in yellow. Connection lines are drawn from the highlighted entity to the entities of neighboring lists if the connected entities occur together in the same file or document. The connected entities are also color-shaded with dark orange representing high connection strength, and light orange representing low connection strength. The connection strength between two entities is measured by number of source code files where they both occur.

In Figure 2, we selected the top five interface entities from the list while sorted in descending order of frequency. The "and" operator (in the top right corner of the view) is selected. This operator ensures that only the entities connecting to all the five selected entities shall be shown. In contrast, the default "or" operator will show all entities that connect to any one of the selected entities. Figure 2 shows all of the five entities linked to the package *org.jhotdraw.draw* entity of the package list, representing their co-occurrence. This co-occurrence means package *org.jhotdraw.draw* uses all the top five interfaces. Each of the top five interface entities links to all four identified files in the document list. This means that all the top five interfaces are used by each of the files linked. These files are: *AbstractTool*; *DefaultDrawingView*; *DrawingEditor*; and *DrawingView*. Applying the same technique also leads to the discovery that the package entities: *org.jhotdraw.draw*; *org.jhotdraw.samples.svg.figures*; *org.jhotdraw.samples.odg.figures*; and *org.jhotdraw.samples.net.figures* all use the same top four class entities.

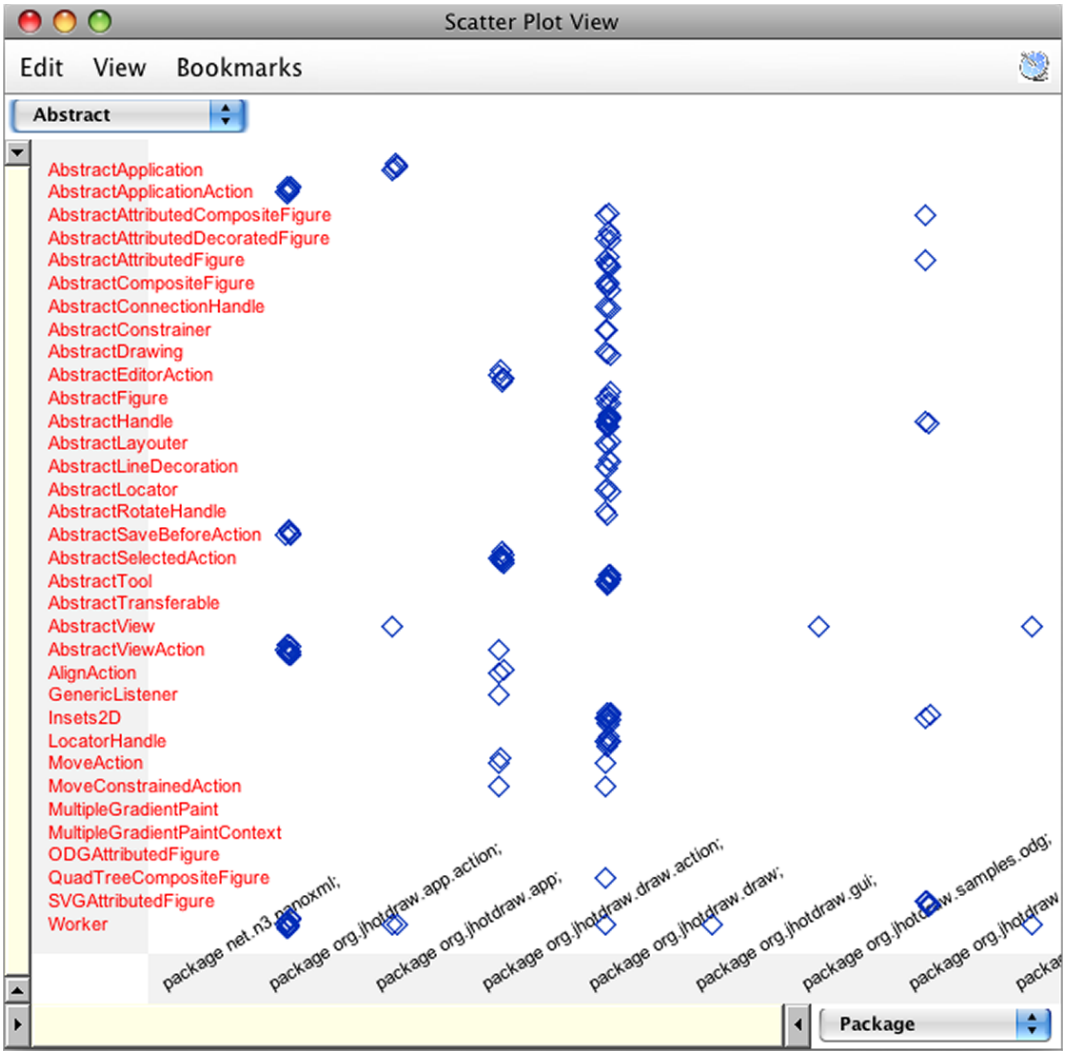

Figure 4: Scatter plot view with the top nine occurring packages and all abstract classes.

These observations lead to the insight that package *org.jhotdraw.draw* is a core package of JHotDraw because of it's size and dependencies on some highly used classes.

4.1.3 Correlating Classes and Packages

Jigsaw supports a *Graph View* that visualizes document and entity connections through a node-link representation (Figure 3). The nodes are documents (represented by white rectangles) or entities (represented by coloured circles). Connections are edges in the graph. We can add entities and documents to a graph view through a search query or display command. A node can be expanded or compressed to show or hide the linked nodes.

Figure 3 shows a graph view with three entity nodes expanded. Each entity node is connected to the document nodes that the entity occurs in. We have applied a circular layout to the view. Each of the entities shown is the top-occurring entity of their type. The brown-colored *Figure* entity represents an interface, the red *ResourceBundleUtil* represents a class, and the blue *Worker* represents an abstract class.

The overlay of links fanning out from the entity nodes shows that *Figure* and *ResourceBundleUtil* co-occur in a significant number of files. This suggests that *Figure* and *ResourceBundleUtil* are likely collaborators. An equivalent hypothesis can be derived from observing *Worker* and *ResourceBundleUtil* co-occurring. These two entities co-occur in a significant proportion of the total files that *Worker* appears in. *Figure* does not seem to collaborate with *Worker* because they have no common files. By zooming and dragging the view, it is possible to identify a single file that they co-occur in.

Following on from this, a scatter plot view allows the users to specify different entity types on the x and y axes. Particular entities can then be added along the axes through search queries or "display" commands from other views. A diamond will appear in the plot if an x-axis entity and a y-axis entity co-occur in the same file. Figure 4 shows the top nine occurring package entities on the x axis and all abstract class entities on the y axis. This figure provides the insight that the *org.jhotdraw.draw* package used the most number of abstract classes. The same insight can be ob-

served by mapping class entities to the y-axis and interface entities to the x-axis. These observations further confirm that *org.jhotdraw.draw* is a core package of JHotDraw.

The same technique provides further insights that both *net.n3.nanoxml* and *org.jhotdraw.draw* packages reference the most number of static methods. There is only one static method defined within the *org.jhotdraw.draw* package, and the rest of the static methods are defined within the *net.n3.nanoxml* package. This suggests that the core JHotDraw package *org.jhotdraw.draw* uses the *net.n3.nanoxml* package, mainly via static method invocations.

4.1.4 Summary

Using Jigsaw on JHotDraw for program discovery lead us to the following insights:

1. A smaller set of classes are recommended for a close read, such as *Figure*, *DrawingEditor*, *ResourceBundleUtil*, and *Worker* (Figure 1, 2, 3);

2. A good understanding of the Java Swing and AWT frameworks is crucial for anyone who wants join JHotDraw project (Figure 1);

3. Knowledge of a XML parser is quite important because JHotDraw uses one quite heavily.

4.2 Scenario 2: Program Maintenance

The second scenario requires the open source developer to make some changes to JHotDraw. They need to be able to answer some typical software maintenance questions in order to carry out this work.

4.2.1 Impact of Refactoring a Package

If we are interested in refactoring the *org.jhotdraw.gui* package, then knowing what other entities are likely to depend on this package is a good first task. We start by exploring connections between all the definition entities and the *import org.jhotdraw.gui.** entity.

Figure 5 shows the packages and classes that might be affected by this refactoring. The list view is suitable for this task as the view supports exploring connections through multiple lists. There are several entity lists to explore. An efficient way to do this is to have the *Import* list in the middle and two other definition entity lists on each side. Selecting the *import org.jhotdraw.gui.** entity from the middle list will link itself to entities on both sides. The linked entities may be affected by the refactoring of *org.jhotdraw.gui* package.

4.2.2 Find Significantly Interacting Classes

We can use the scatter plot view to measure the number of other classes a specific class interacts with. In this example, we need to understand the connections between the definition entity DefClass and the reference entity Class. We add these to the x and y axes of the scatter plot. The list view identifies the DefClass entities and Class entities from the same package *org.jhotdraw.gui*. A subsequent call to the "display" command adds them to the scatter plot.

Figure 6 shows that the *JDoubleAttributeSliderBeanInfo*, *JDoubleTextFieldBeanInfo*, and *JIntegerTextFieldBeanInfo* classes are the most heavily interacting with other classes, given the diamond count.

4.2.3 Dependencies Between Two Packages

We need to measure the dependencies between the two packages *org.jhotdraw.gui* and *org.jhotdraw.util*. The scatter plot view is the best view to answer this question. With all the DefClass entities of the *org.jhotdraw.gui* package added to the x axis, and Class entities of the *org.jhotdraw.util* package added to the y axis, counting and comparing the number of diamonds vertically gives the classes from the x axis that are most dependent on the class from the y axis. The scatter plot view shows none of the classes from the *org.jhotdraw.gui* package stand out from the others as being more dependent on the *org.jhotdraw.util* package. It also shows the *org.jhotdraw.gui* package is most dependent on the *Methods* class of the *org.jhotdraw.util* package.

4.2.4 Package Class Count

We need to measure package size, determined by the number of classes in that package. There are multiple Jigsaw views that can answer this question. For example, the list view supports hovering the cursor over the frequency bar. Alternatively, we can use a cluster view. A cluster view displays large number of Java source code documents as white rectangles, that can be clustered based on filters.

In Figure 7, the filter "package org.jhotdraw.draw" is applied, and Java source code documents are clustered into two piles. The red pile represents 146 documents that contain the package *org.jhotdraw.draw* entity. The white pile at the bottom represents the Java documents that do not contain the package *org.jhotdraw.draw* entity. Each pile is headed with the filter name and a file count. The file count shown by the cluster view makes answering the question easy.

5. DISCUSSION

Our case study has exercised Jigsaw in program discovery and maintenance activities. We discuss general findings based on our experience from the case study.

Visual analytical process. Jigsaw was better in the program discovery scenario than the program maintenance scenario. In the program discovery scenario, the evaluator explored JHotDraw by freely utilizing Jigsaw views that seemed the most useful and accessible at the time. The evaluator began to create hypotheses from some views that lead to goals for further exploration and verification in other views. This reflects that a visual analytical process is driven by a hypothesis generation and verification loop [8].

Low cognitive fit. One challenge we encountered was the need to refresh our understanding of the semantic connections between different types of entities especially when switching between views. We believe this is caused by the low cognitive fit between Jigsaw and the target source code domain. The low cognitive fit problem is an acknowledged problem that confronts generic software visualization tools [17]. We hypothesize that the cognition problem will ease as the evaluator becomes more familiar with Jigsaw views and the data set visualized.

Usability issues. We also experienced some usability issues in Jigsaw. For example, the colors of entity types changes randomly when a Jigsaw project is re-opened. We had to adapt to the color scheme for each session. Jigsaw should provide a consistent colour scheme for each project.

Multiple coordinated views. Jigsaw's multiple coordinated views helped the hypothesis generation and verification loop. Multiple views gave different perspectives on

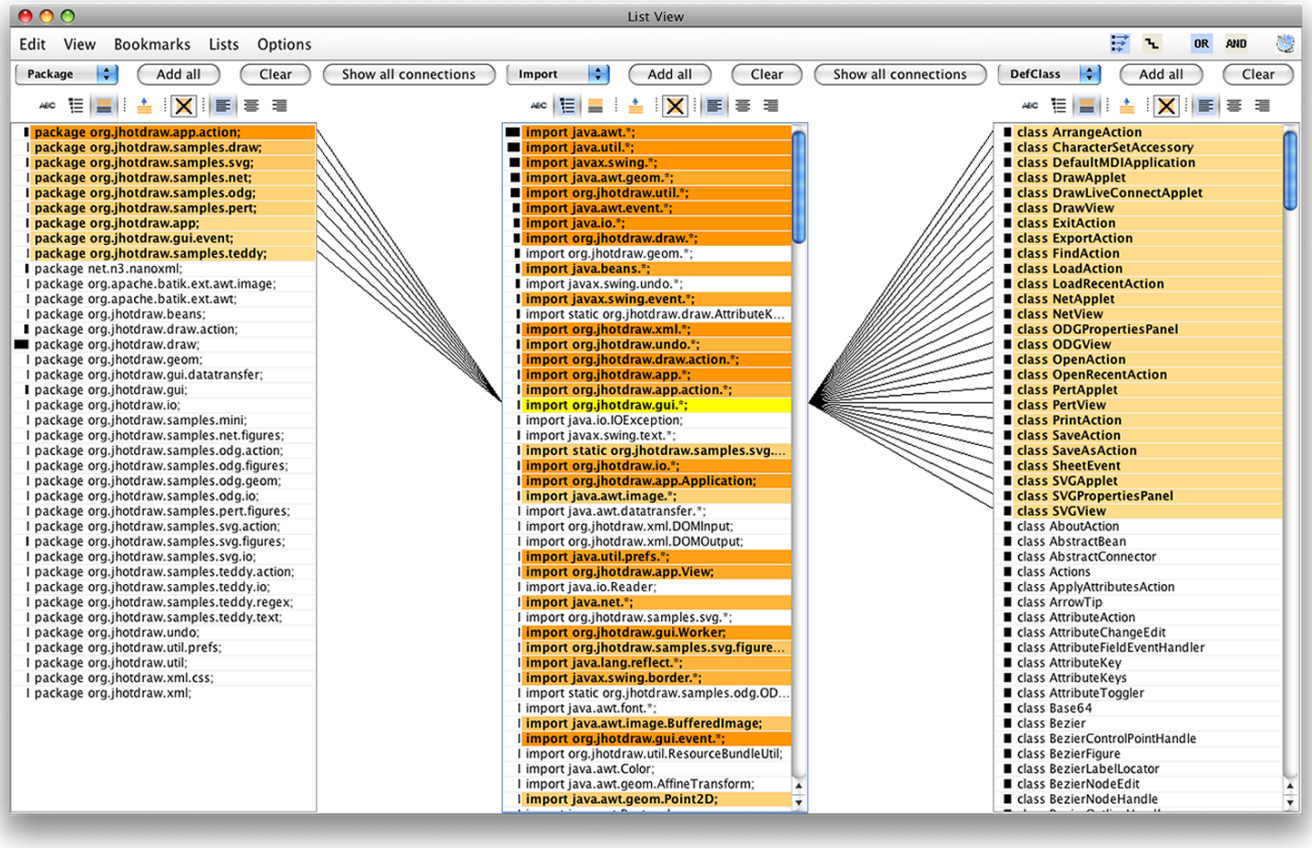

Figure 5: List view of Document, Interface, and Package entities.

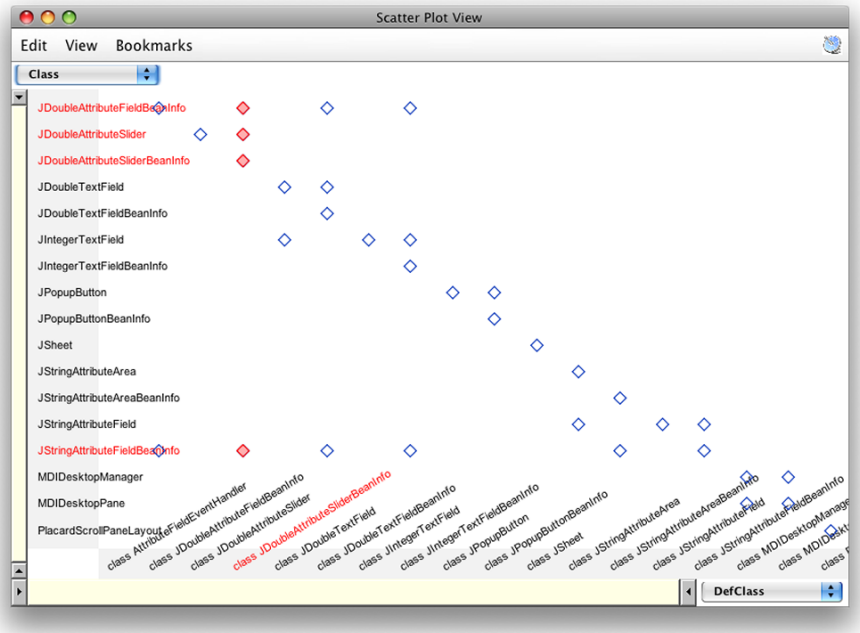

Figure 6: Scatter plot view of Class and DefClass entities from org.jhotdraw.gui.

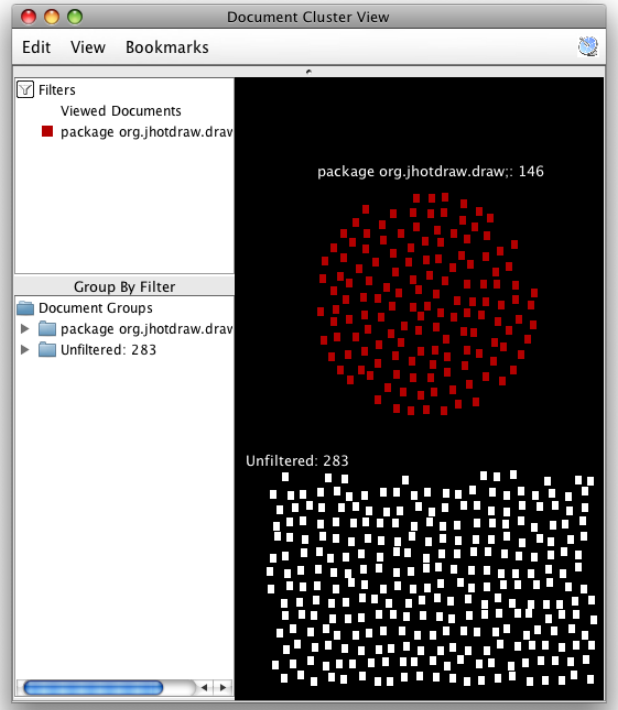

Figure 7: Cluster view with the package org.jhotdraw.draw filter applied.

the same data set, and interactions with one view are immediately reflected in other views. We found this highly useful in helping to form hypotheses when different views are compared. For example, the word cloud and the list view both visualize the entity frequency. If an entity frequency is visualized high in the word cloud but low in the list view, a hypothesis can be formed that the usage of the entity is relatively confined within a small set of files. Multiple coordinated views were limited by available display real estate. The evaluator was able to use up to three views on the same display, but mostly navigated between views by context switching.

Visualization noise. Jigsaw's entity identification is generic as it only performs lexical matching. An entity that represents a reference to a Java class is represented by the name of the Java class and nothing more. For example, Jigsaw matches a Java class name in Javadoc in the same way as in the code itself. Considering the connection between a definition and a reference class entity – our Jigsaw visualizations did not distinguish static relationships and dynamic relationships. It is not easy to identify solely from the visualizations whether it is a static relationship or dynamic relationship. The source code would have to be inspected to find out the exact meaning. We call these pseudo connection and ambiguous connection semantic problems *visualization noise*. The level of the noise depends on the characteristics of a program under exploration. For instance, the noise will be high if a program is well and extensively commented. We did not find noise a big problem in the case study. One way to eliminate the visualization noise is to strengthen Jigsaw's parsing capabilities, maybe in the form of plug-ins. The parsers could allow Jigsaw to identify programming entities whether they are source code or comments. The combination of lexical and semantical entity matching could greatly increase Jigsaw's effectiveness in exploring software data.

6. CONCLUSIONS

We have presented a case study using Jigsaw to explore the JHotDraw Java software system. We have demonstrated that a general-purpose visual analytics tool is capable of visualizing a Java program to support typical program comprehension tasks. By doing so, we have to some extent demonstrated that the Inventor's Paradox is applicable to the problem of software visualization. Therefore, we suggest that a general approach to software visualization problems is a promising direction in software visualization.

For future work, we plan to conduct a controlled experiment to assess the effectiveness of Jigsaw in helping users to explore and understand Java software. We hope to identify further usability issues the user might encounter in Jigsaw. The result of the experiments will help us understand users exploring a program with a general-purpose visual analytics tool. The usability evaluation will provide valuable feedback into the design and development of Jigsaw, and other visual analytics tools.

Acknowledgments

We would like to thank John Stasko and Carsten Görg from the Georgia Institute of Technology for access to the Jigsaw tool, technical support, and valuable feedback on this paper.

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

7. REFERENCES

[1] JHotDraw. http://sourceforge.net/projects/jhotdraw/, April 2010.

[2] Y. ah Kang, C. Görg, and J. Stasko. Evaluating visual analytics systems for investigative analysis: Deriving design principles from a case study. In *Proceedings of VAST*, pages 139–146, 2009.

[3] C. Anslow, J. Noble, S. Marshall, and E. Tempero. Visualizing the word structure of Java class names. In *OOPSLA Companion*, pages 777–778, 2008.

[4] M. Baker and S. Eick. Visualizing software systems. In *Proceedings of ICSE*, pages 59–67, 1994.

[5] R. Chang, M. Ghoniem, R. Kosara, W. Ribarsky, J. Yang, E. Suma, C. Ziemkiewicz, D. Kern, and A. Sudjianto. Wirevis: Visualization of categorical, time-varying data from financial transactions. In *Proceedings of VAST*, pages 155–162, 2007.

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

[7] S. G. Eick, J. L. Steffen, and J. Eric E. Sumner. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Soft. Eng.*, 18(11):957–968, 1992.

[8] D. Keim, G. Andrienko, J.-D. Fekete, C. Görg, J. Kohlhammer, and G. Melançon. *Visual Analytics: Definition, Process, and Challenges*, volume 4950 of *LNCS*, pages 154–175. Springer-Verlag, 2008.

[9] M. Lanza and S. Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Soft. Eng.*, 29(9):782–795, 2003.

[10] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer Verlag, 2006.

[11] M. Ogawa and K.-L. Ma. code_swarm: A design study in organic software visualization. *IEEE Vis. and Comp. Graph.*, 15(6):1097–1104, 2009.

[12] M. Pacione, M. Roper, and M. Wood. A novel software visualisation model to support software comprehension. In *Proceedings of WCRE*, pages 70–79, 2004.

[13] Y. Park and C. Jensen. Beyond pretty pictures: Examining the benefits of code visualization for open source newcomers. In *Proceedings of VISSOFT*, pages 3–10, 2009.

[14] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the execution of Java programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 151–162. Springer Verlag, 2002.

[15] M. Pinzger, K. Grafenhain, P. Knab, and H. Gall. A tool for visual understanding of source code dependencies. In *Proceedings of ICPC*, pages 254–259, 2008.

[16] G. Polya. *How to Solve It (Second Edition)*. Princeton University Press, 1973.

[17] T. Schafer and M. Mezini. Towards more flexibility in software visualization tools. In *Proceedings of VISSOFT*, pages 64–69, 2005.

[18] M. Sensalire and P. Ogao. Visualizing object oriented software: Towards a point of reference for developing tools for industry. In *Proceedings of VISSOFT*, pages 26–29, 2007.

[19] M. Sensalire, P. Ogao, and A. Telea. Evaluation of software visualization tools: Lessons learned. In *Proceedings of VISSOFT*, pages 19–26, 2009.

[20] N. Shi and R. Olsson. Reverse engineering of design patterns from Java source code. In *Proceedings of ASE*, pages 123–134, 2006.

[21] J. Stasko, C. Görg, and Z. Liu. Jigsaw: supporting investigative analysis through interactive visualization. *Information Visualization*, 7(2):118–132, 2008.

[22] A. Telea, H. Hoogendorp, O. Ersoy, and D. Reniers. Extraction and visualization of call dependencies for large C/C++ code bases: A comparative study. In *Proceedings of VISSOFT*, pages 81–88, 2009.

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

[24] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis. Design pattern detection using similarity scoring. *IEEE Soft. Eng.*, 32(11):896–909, 2006.

[25] F. B. Viegas, M. Wattenberg, F. van Ham, J. Kriss, and M. McKeon. ManyEyes: a site for visualization at internet scale. *IEEE Vis. and Comp. Graph.*, 13(6):1121–1128, 2007.

[26] R. Wettel and M. Lanza. Visualizing software systems as cities. In *Proceedings of VISSOFT*, pages 92–99, 2007.

[27] J. Wise, 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, 1995.