The Social Network of Java Classes

Diego Puppin, Fabrizio Silvestri
Institute for Information Science and Technologies, ISTI-CNR
Via Moruzzi 1
56100 Pisa, Italy
{diego.puppin, fabrizio.silvestri}@isti.cnr.it

ABSTRACT

Several works in literature have analyzed the link structure of programs in relation with software engineering: it has been observed that the programming standards caused small-world networks to emerge among classes in object-oriented programming. The need for coherent design and the coding conventions introduce regular patterns in the link structure of code.

In this work, we study the social network naturally emerging from unrelated software projects. We studied the links present among Java classes coming from different contexts. In this case, any observable patterns come from social behaviors, rather than software engineering practices.

In our analysis, we could observe a regular social network, organized according to a power-law distribution that is typical, for instance, of links among Web pages. We give a positive value to class links, which we consider a sign of relevance and acceptance. Out of this, we propose a way of ranking classes, and we present our prototype search engine for Java classes.

1. INTRODUCTION

Although first studies date back to late thirties [12], the analysis of the social behavior of complex systems, i.e. arising in natural phenomena, received a lot of attention only in these last years. Also known as *link analysis*, the study of social networks has connected very distant fields, such as biology and computer science, because networks emerging in very different contexts share, many times, the same properties. One of the most important property is the one that relates the number of individuals in a population and its connectedness. For instance, Stanley Milgram in 1967 [11] found that the number of intermediate acquaintances separating any two people in the United States is about 6. Networks presenting this kind of strong locality characteristics are called *small worlds*. For instance, it is well known that Web pages are organized into a small-world network with strong social properties [6]. Quite recently, Web search engines started to exploit this in order to offer relevant results to their users. For instance, the PageRank [14] algorithm, used by Google, gives higher rank to pages which are referred by other high-rank pages.

The interest of this work is to extend this analysis to the code structure of object-oriented (OO) programs.

The popularity of OO languages has made available a large number of independently developed objects and classes. In a very natural way, developers make use of the available software library to build on: they will use existing parsers for new compilers, existing graphic routines for new human interfaces. In other words, code gets re-used by developers. Moreover, it is very likely to find that the same functionality is offered by different software providers, e.g. Java graphic interfaces can be developed either with Swing or AWT. To this extent, developers have to choose one on the basis of some kind of quality measure.

This way, an ecosystem of software entities emerges: similar objects and classes are offered by different vendors or publishers; they will *compete* for visibility and adoption, and they will *co-operate* in larger applications.

Some interesting questions arise when we consider the ensemble of components as an ecosystem:

- *Can this ecosystem be considered a small-world?*
- *Is it possible to evaluate the quality of a software product considering the strength of its relations with the others?*

Answering these questions is very challenging and has very important applications. For instance, the measure of quality can help users to find the best software tools for their needs. Link analysis can help to provide an answer.

This paper uses Java as a test-case. Java classes are naturally organized into a social network: classes make use of each other (using objects of different classes as method arguments or return values), they are organized into a hierarchy of inheritance, they share interfaces. This social network has a central core represented by the Java Library, surrounded by all the classes making part of other Java projects. Some of these latter classes are commonly used by developers to complement the functionality offered by JDK. Some classes of Apache Tomcat, as an example, are very popular among programmers.

In order to perform link analysis over the Java class ecosystem, the full source code is not needed: we just need to analyze the interface of each class. Interfaces are always available for each piece of software released for public use (libraries, APIs...).

Also, the access to interfaces does not mean free access to the algorithms. Consider this: any person can browse the references to papers stored in an on-line digital library; on the other side, only paying users can download the full text of papers. This way, a reader can find if the paper is of interest (on the basis of the references) before buying it. Also, the list of references alone does not allow a generic user to discover anything about the content of the paper itself.

Similarly, a vendor of software components will have an interest in publishing the interface, in order to attract customers, but s/he may be unwilling to release the source code. Clearly, our analysis may be done on different frameworks, with the only constraint of having the program interfaces available.

This paper is structured as follows. In the next section, we present the relevant related works. Then we describe our analysis, followed by our initial experiments. Later, we present an application of our results. Finally, we conclude.

2. RELATED WORK

Link analysis allowed us to compare the structure of networks coming from very different areas of study. The growth of the Internet network, for instance, shows patterns similar to those observed in human biology [10] and automobile networks [5]. The structure of the social network of code, in particular Java classes, has also obtained attention in the literature.

A very recent work [16] analyzed the dynamic references to objects in Java applications. The authors examined the heap of large Java programs in order to measure the number of incoming and outgoing references to each object in the program. They verified that the number of references is distributed according to a power-law curve [1]. Their work is based on the *dynamic* realization of a program. We will show that this relationship exists also in the *static* links among classes.

In other words, code can be examined at different levels (see Table 1): the interface of objects, the full source code, or the dynamic realization of executable code. [16] analyzes the run-time behavior of code, while we are more interested in interfaces.

The main goal of our technique is different from the approach described in [16]. While our analysis clearly does not consider the references to objects and classes that do not appear in the interface (classes that are used within class code but not in the class interface), we do not need to run any code, and we can do data-mining over unrelated projects.

In [9], the authors cite an interesting technique for ranking components within a set of given programs. Ranking a collection of components simply consists of finding an absolute ordering according to the relative importance of components. The method followed by the authors is very similar to the method used by the Google search engine to rank Web pages: PageRank [3]. In ComponentRank, in fact, the importance of a component is measured on the basis of the number of references (imports, and method calls) other classes make to it within the given source code.

[18] analyzes the structure of the Java Development Kit (JDK) and of a computer game, and observes a scale-free network structure. The authors relate this to the *optimal design*, that is the software engineering practices about software modularity. Similarly, [20] analyzes the class structure of three big software projects (Ant, Tomcat and again JDK) and finds, for each project in isolation, interesting power-law curves in several properties.

Level	Used in
Program Interface	this work
Full Source Code	[9, 18, 20]
Dynamic Realization	[16, 7, 19]

Table 1: Different level of analysis of code

It is important to point out that the main difference with our work resides in the fact that we analyze the network of independently developed classes, rather than a single software project. For this reason our work resembles more the classic works analyzing social networks such as the Web or the Internet. Later in this work we present a possible application of this analysis: a ranking function for software components.

There has been a number of interesting works also in the field of *workflow mining*, i.e. the analysis of the implementation and the dynamic realization of an application, in order to measure the number of objects, their usage and so on.

This is a very flexible approach, suitable for software components in general. The approach we propose in this work is complementary to that followed in [7, 19]. In these works, the dynamic realization of an application is analyzed in order to discover profiling properties, frequently used activities and so on. Link analysis can be performed also by using profiling data, but this would require running and measuring all the objects in the repository. We believe that an approach focused on the static structure of the code has a stronger potential. The cited works [7, 19] are interested in the dynamic realization, while our interest lies on the interfaces.

3. MOTIVATIONS

We started our research with the goal of developing a ranking strategy for software components. We were interested in designing a search engine for components, so that a developer could easily find the solutions that best fits the problem at hand.

Today, with the emergence of the concepts of Grid [8] and Software-Oriented Architecture [15], the interest toward modular software solutions (known as software components or services) is growing.

We believe a market of competing services will emerge. Software components, packaged as services (Web or Grid Services), will be offered to the public by a number of vendors and developers, with different price, quality of service, trust, performance and other features. An economy-based competition [4] will drive the user into choosing one service over another. This vision is emerging as a key point for European and international projects [13].

This is why it is very important to study the dynamics of code: how different components relate to each other, how many references there are to a given component, what the shape of the social network is. We launched our study by using Java classes as our component model. This choice is shared by other projects: Apache Hivemind uses POJOs (Plain Old Java Objects) as building blocks for applications. This gave us the opportunity to analyze a big repository of code that is very well documented.

4. EXPERIMENTS

We performed our analysis on a large collection of Java classes found on the Internet. We collected the documentation about projects we were aware of, and then we searched (with a standard Web search engine) collections of Java documentation files across the Web. Within these files, we looked for references to external classes.

This way, we collected an initial set of 7700 classes, then grown to 49500. We were able to retrieve very high-quality JavaDocs for the following projects, among others: Java 1.4.2 API; Java 1.5.0 API; HTML Parser 1.5; Apache Struts; Globus 3.9.3 MDS; Globus 3.9.3 Core and Tools; Tomcat Catalina; JavaDesktop 0.5, JXTA; Apache Lucene; Apache Tomcat 4.0; Apache Jasper; Java2HTML; DBXML; ANT; Nutch; Eclipse Open Source IDE; ObjectWeb ProActive. The collection is large enough to show a big variety of programming patterns and to include competing classes (offering similar functionality).

We parsed the JavaDocs files, and we recorded a link between Class A and Class B every time a method in Class A used as an argument or returned as a result an object of type B. This way, we generated a directed graph describing the social network of the Java libraries.

We then counted and plotted the number of inlinks and outlinks from each class. The plot followed clearly a power-law curve [1] with $\alpha \approx 1$ (see Figures 1 and 2): the number of inlinks, i.e. the number of times each class is referred, is distributed following a power-law pattern. In other words, very few classes are linked by very many others, while several classes are linked by only a few other classes. This is true both for our initial sample of $7,700$, and for the bigger base.

In the Figures, the reader can see a plot representing the number of incoming links to each class, in log-log scale. Classes are sorted by the number of incoming links. The distribution follows closely a power-law pattern, a small exception given by the first few classes (Object, Class etc.) which are used by almost all other derived classes to provide basic services, including introspection and serialization.

This is a very interesting result: within Java, the popularity of a class among programmers seems to follow the pattern of popularity shown by the Web, blogs and so on.

5. AN APPLICATION

An interesting application of this study is, as anticipated, a theoretical support to a new ranking strategy for software components. Our goal is to offer a tool to find existing software components to developers.

We developed a very simple search engine, able to find high-relevance classes out of our repository. Classes matching the query can be ranked by $TF \times IDF$ [17] (a common information retrieval method, based on a metric that keeps into account both the number of occurrences of a term within each document and the number of documents in which the term itself appears) or by ClassRank, our version of PageRank for Java classes, based on the class usage links.

Usage links are sometimes considered negative features of source code, because they can lead to chain of dependencies in the process of developing code. In this work, on the contrary, we give a positive value to links: we see them as a sign of recognition and acceptance, as it happens among Web pages.

We called this tool (GRIDLE /'grɪ-dəl/: a *Google^{TM}-like Ranking, Indexing and Discovery service for a Link-based Eco-system of software components*) Figure 3 shows the first web interface of our tool.[1]

ClassRank is a very simple algorithm, that builds on the popular PageRank algorithm used in Google [3]. To determine the rank of a class C, we iterate the following formula:

$$rank_C = \lambda + (1 - \lambda) \sum_{i \in inlinks_C} \frac{rank_i}{\#outlinks_i}$$

where $inlinks_C$ is the set of classes that use C (with a link into C), $\#outlinks_i$ is the number of classes used by i (number of links out of i), and λ a small factor, usually around 0.15.

5.1 Interesting Observations

We could observe some very interesting results. The highest-ranking classes are clearly some basic Java API classes, such as `String`, `Object` and `Exception`. Nonetheless, classes from other projects are apparently very popular among developers: #7 is Apache `MessageResources`, #11 is Tomcat `CharChunk`, #14 is DBXML `Value` and #73 is JXTA `ID`. These classes are very general, and are used by developers of unrelated applications (see Table 2).

We could verify that in most cases ClassRank is more revelant than $TD \times IDF$, especially when the class name is not

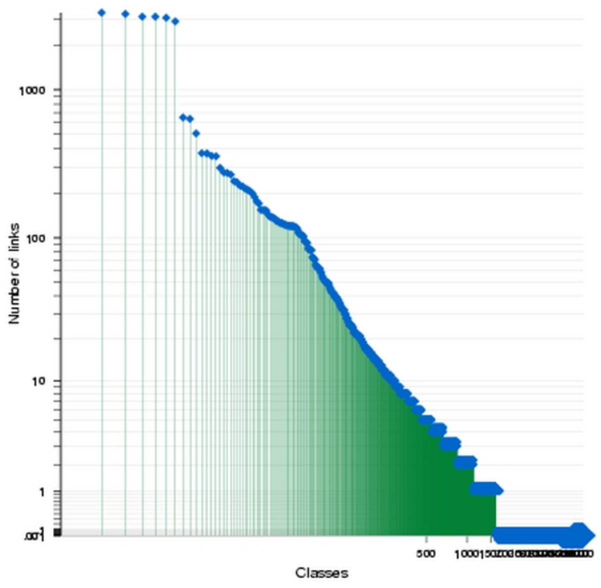


Figure 1: Distribution of inlinks, $7,700$ classes.

Rank	Class Name
TOP	String, Object, Class, Exception
7	Apache MessageResources
11	Tomcat CharChunk
14	DBXML Value
73	JXTA ID

Table 2: Some top ranking classes

[1] It is available on-line at: http://gridle.isti.cnr.it/.

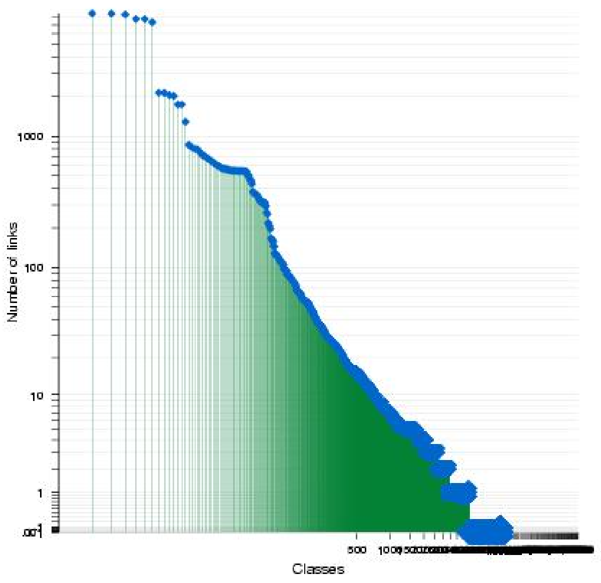


Figure 2: Distribution of inlinks, $49,500$ **classes.**

textually similar to the function we are looking for. For instance, if the developer is trying to write data to a file, and performs a query such as *"file writer"*, $TF \times IDF$ will return, in order: (1) javax.jnlp.JNLPRandomAccessFile, from JNLP API Reference 1.5; (2) javax.swing.filechooser.FileSystemView, from Java 2 Platform SE 5.0; (3) java.io.FileOutputStream, from Java 2 Platform SE 5.0; (4) java.io.RandomAccessFile, from Java 2 Platform SE 5.0. The second class is clearly unrelated with the problem under analysis, and only the third is probably what the user was looking for.

On the other hand, ClassRank will return four classes from the Java API (Java 2 Platform SE 5.0): (1) java.io.PrintWriter; (2) java.io.PrintStream; (3) java.io.File; (4) java.util.Formatter; which are all probably better matches. To verify this claim on result quality, we will need to test the search engine with Java developers.

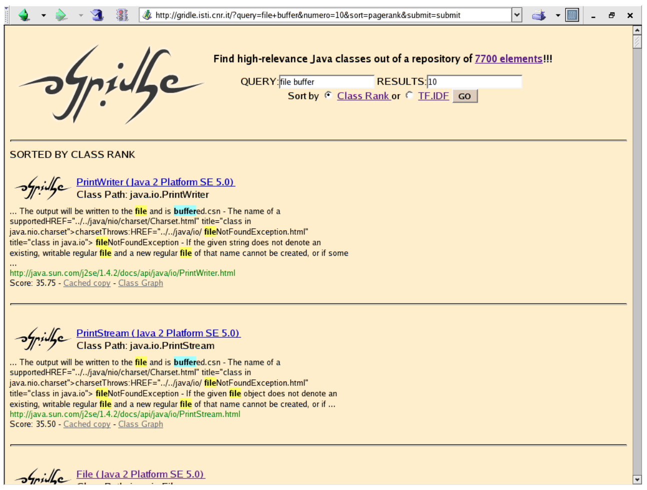
An interesting side-effect of a search engine is that popular results could become more and more popular over time (the well-known "rich get richer" effect [2]). We are interested in investigating the long term effects of this on software development and organization.

6. CONCLUSION

We observed a growing trend in the use of software components: more and more, modern programming platforms are oriented to software components (Microsoft .NET, Enterprise Java Beans, Apache HiveMind). This is the reason why we envision a market where software components are off-the-shelf commodities, which can be assembled to build the needed software solutions.

One of the open problems is to offer a way to search for components, out of this emerging service market. European projects such as NextGrid and SFIDA-PMI list this issue among the key research topics.

Power-law patterns have been found in the number of links



http://gridle.isti.cnr.it/?query=file+buffer&numero=10&sort=pagerank&submit=submit

Find high-relevance Java classes out of a repository of 7700 elements!!!

QUERY: file buffer RESULTS: 10

Sort by [✓ Class Rank] or TF.IDF GO

SORTED BY CLASS RANK

PrintWriter (Java 2 Platform SE 5.0)
Class Path: java.io.PrintWriter
... The output will be written to the **file** and is **buffer**ed.csn - The name of a supportedHREF="../../java/nio/charset/Charset.html" title="class in java.nio.charset">charsetThrows:HREF="../../java/io/ **file**NotFoundException.html" title="class in java.io"> **file**NotFoundException - If the given string does not denote an existing, writable regular **file** and a new regular **file** of that name cannot be created, or if some ...
http://java.sun.com/j2se/1.4.2/docs/api/java/io/PrintWriter.html
Score: 35.75 - Cached copy - Class Graph

PrintStream (Java 2 Platform SE 5.0)
Class Path: java.io.PrintStream
... The output will be written to the **file** and is **buffer**ed.csn - The name of a supportedHREF="../../java/nio/charset/Charset.html" title="class in java.nio.charset">charsetThrows:HREF="../../java/io/ **file**NotFoundException.html" title="class in java.io"> **file**NotFoundException - If the given **file** object does not denote an existing, writable regular **file** and a new regular **file** of that name cannot be created, or if ...
http://java.sun.com/j2se/1.4.2/docs/api/java/io/PrintStream.html
Score: 35.50 - Cached copy - Class Graph

File (Java 2 Platform SE 5.0)

Figure 3: Web-interface of .

between pages, in the number of contact to blogs, in many other social networks. In this work, we verified the existence of a social network among Java classes: the links among classes create a graph of social relations among them. In our experiments, we considered only classes with a social desire, i.e. classes documented according to the JavaDoc standard.

We observed that a very little number of classes are extremely popular and are used (i.e. used as an argument or returned as a result) by methods of several other classes, while many other classes are obscure and rarely used: a power-law appeared, in a majestic epiphany, out of our collection of almost $50,000$ classes. These results thus allow us to draw the conclusion that the space of Java classes form a *small-world* network.

We exploited this finding to rank our base with ClassRank, an algorithm that builds on the PageRank algorithm used in Google [3]. In our initial tests, ClassRank empirically proved to be more effective than $TF \times IDF$.

What if one day a programmer could find a class s/he needs as easily as s/he now can find the name of the greatest movie with Bela Lugosi? Surely, graduate students will be better off, and could spend more time with Ed Wood's masterpieces.

7. ACKNOWLEDGMENTS

We would like to thanks our colleagues at ISTI-CNR, in particular Domenico Laforenza and Salvatore Orlando, for the fruitful discussions about and software search.

8. REFERENCES

[1] L. A. Adamic. Zipf, power-laws, and pareto - a ranking tutorial. Available at http://www.hpl.hp.com/-research/idl/papers/ranking/ranking.html.

[2] A.-L. Barabasi. *Linked: How Everything Is Connected to Everything Else and What It Means.*

[3] S. Brin and L. Page. The Anatomy of a Large–Scale Hypertextual Web Search Engine. In *Proceedings of*

the WWW7 conference / Computer Networks, volume 1–7, pages 107–117, April 1998.

[4] R. Buyya. *Economic-based Distributed Resource Management and Scheduling for Grid Computing.* PhD thesis, Monash University, Melbourne, Australia, April 2002.

[5] C. Faloutsos and I. Kamel. Beyond uniformity and independence: Analysis of r-trees using the concept of fractal dimension. In *Proc. ACM SIGACT-SIGMOD-SIGART PODS*, pages 4–13, Minneapolis, MN, May 1994.

[6] G. Flake, S. Lawrence, and C. L. Giles. Efficient identification of web communities. In *Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 150–160, Boston, MA, August 20–23 2000.

[7] G. Greco, A. Guzzo, G. Manco, and D. Saccà. Mining frequent instances on workflows. In *Proceedings of PAKDD 03*, 2003.

[8] S. T. I. Foster, C. Kesselman. The Anatomy of the Grid: Enabling Scalable Virtual Organization. *Interational Journal Supercomputer Application*, 3(15), 2001.

[9] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component rank: relative significance rank for software component search. In *Proceedings of the 25th international conference on Software engineering*, pages 14–24, Portland, Oregon, May 2003. IEEE, IEEE Computer Society.

[10] B. Mandelbrot. *Fractal Geometry of Nature.* W.H. Freeman, New York, 1977.

[11] S. Milgram. The small world problem. *Psychology Today*, 2:60–70, 1967.

[12] J. Moreno. Sociometry in relation to other social sciences. *Sociometry*, 1, 1937.

[13] F. Nachira. Technologies for digital ecosystems - supporting growth and smes, December 2004. Opening for the workshop on Sector Digital Ecosystem.

[14] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[15] M. P. Papazoglou and D. Georgakopoulos, editors. *Service-Oriented Computing*, volume 46 (10) of *Communications of ACM.* October 2003.

[16] A. Potanin, J. Noble, M. Frean, and R. Biddle. Scale-free geometry in oo programs. *Communications of the ACM*, 48:99–103, May 2005.

[17] G. Salton. *The SMART Retrieval System – Experiments in Automatic Document Processing.* Prentice Hall Inc., Englewood Cliffs, NJ, 1971.

[18] S. Valverde, R. F. i Cancho, and R. Solé. Scale free networks from optimal design. *Europhysics Letters*, 60:512–517, 2002.

[19] W. van der Aalst and B.F. van Dongen and J. Herbst and L. Maruster and G. Schimm and A.J.M.M. Weijters. Workflow mining: A survey of issues and approaches. *Data & Knowledge Engineering*, 47:237–267, 2003.

[20] R. Wheeldon and S. Counsell. Power law distributions in class relationships. In *Proceedings of 3rd International Workshop on Source Code Analysis and Manipulation (SCAM)*, Amsterdam, September 2003.