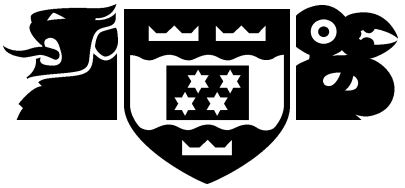
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



School of Mathematical and Computing Sciences
Computer Science

Scale-free Geometry in Object-Oriented Programs

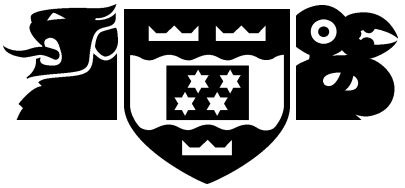
Alex Potanin, James Noble, Marcus Frean, Robert Biddle

Technical Report CS-TR-02/30
December 2002

School of Mathematical and Computing Sciences
Victoria University
PO Box 600, Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341, Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

Scale-free Geometry in Object-Oriented Programs

Alex Potanin, James Noble, Marcus Frean, Robert Biddle

Technical Report CS-TR-02/30
December 2002

Abstract

In this article we examine the graphs formed by object-oriented programs written in a variety of languages, and show that these turn out to be scale-free networks without exception. Apart from its considerable intrinsic interest, this unexpected facet of the geometry of real programs may help us optimize language runtime systems and improve the design of future object-oriented languages.

Publishing Information

Submitted for publication in the Communications of ACM

Author Information

Alex Potanin, James Noble, Marcus Frean, and Robert Biddle are based at Victoria University of Wellington

Scale-free Geometry in Object-Oriented Programs

Alex Potanin, James Noble, Marcus Frean, Robert Biddle
School of Mathematical and Computing Sciences
Victoria University of Wellington, New Zealand
{alex, kjx, marcus, robert}@mcs.vuw.ac.nz

Introduction

Object-oriented programs, when executed, produce a complex web of objects that can be thought of as a network with objects as nodes and references as links. In recent years interest has grown in the geometry of networks (or graphs), particularly those of human origin, many of which show a rather striking property: their structure has a characteristic that is *scale-free*. In the case of the World Wide Web, for example, the (vast) number of web pages with 1 incoming link is about twice the number with 2 incoming links, and *that* is twice the number with 4 links, and so on all the way up to Google and other massively referenced sites [1]. The phrase 'scale-free' relates to the fact that if we double the number of links n, the number of pages is always halved (or other fixed ratio) regardless of what n is. Compare this to what happens if a graph is constructed by simply adding links at random. Doing this leads to nearly all nodes having around the same number of links (i.e. the number of links divided by the number of nodes), and hence such random graphs have a 'typical scale' about them [4]. By contrast, the web has *no* typical scale to its connectivity — a remarkable and somewhat counterintuitive property closely related to that of fractals.

Other notable examples of scale-free graphs are the network formed by co-authors of papers in medical journals, the physical connections forming the Internet, the network of airports connected by airline flights, networks of sexual contacts, and even the patterns of connectivity between neurons in the human brain [2]. Well before being noticed in real-world graphical structures, scale-free distributions were found in other contexts, such as the relative frequencies of English words, the distribution of personal wealth, the sizes of cities, and the number of earthquakes of given strength [12].

In this article we examine the graphs formed by object-oriented programs written in a variety of languages, and show that these turn out to be scale-free networks without exception. Apart from its considerable intrinsic interest, this unexpected facet of the geometry of real programs may help us optimize language runtime systems and improve the design of future object-oriented languages.

Power Laws

The way to detect a scale-free phenomena is to see if it shows up statistically in the form of a *power law*. In power law distributions the frequency y_x of some event of size x is proportional to x raised to some fixed exponent. One drawback of this is that very rare events are by their nature "noisy" (there may be one node with, say 1000 connections, and another with 1005, but none with 1002), which distorts the estimation of the exponent. For this reason a second approach is often adopted in which we first rank the event sizes by how often they occur, and then look for a power law in the relationship between frequency y_x, and rank r_x of the form:

$$y_x \sim (r_x)^{-a} \tag{1}$$

The easiest way to see this is to take logarithms of both sides, or to plot y versus r on logarithmic scales — if the distribution follows a power law we expect to see a straight line whose slope is minus a.

For example, consider how often a particular word appears in any English novel. Common words like *"the"*, *"of"*, or *"and"* can be found many orders more times than the majority of other words, while at the other extreme there are a huge number of words that are used only rarely. In 1925, George Kingsley Zipf, a Harvard linguistics professor, conducted empirical studies [11] of word occurrences and made a remarkable observation that if we rank the words by the number of times they can be found in a text of a particular novel, then their rank will be inversely proportional to their number of occurrences. Hence, if you take your favourite novel and draw a logarithmic plot of the number of times you can find each word against the rank of such a word, then you will see a line! The slope a of this line is usually close to one, but more generally it should be some value above zero.

Object Graphs

An object graph — the object instances in the program and the links between them — is the skeleton of an object-oriented program during its execution. Because each node in the graph represents an object, the graph grows and changes as the program runs. It contains just a few objects when the program is started, gains more objects as they are created, and loses objects when they are no longer required. The structure of the graph (the links between objects) changes too, as every assignment statement to an object's field makes or changes an edge in the graph.

Figure 1 illustrates the object graph of a simple part of a program — in this case a doubly-linked list of `Student` objects. The list itself is represented by a `LinkedList` object which has two references to `Link` objects representing the head and tail of the list. Each `Link` object has two references to other `Link` objects — the previous and the next links in the list, plus a third reference to one of the `Student` objects contained in the list. Although the overall structure

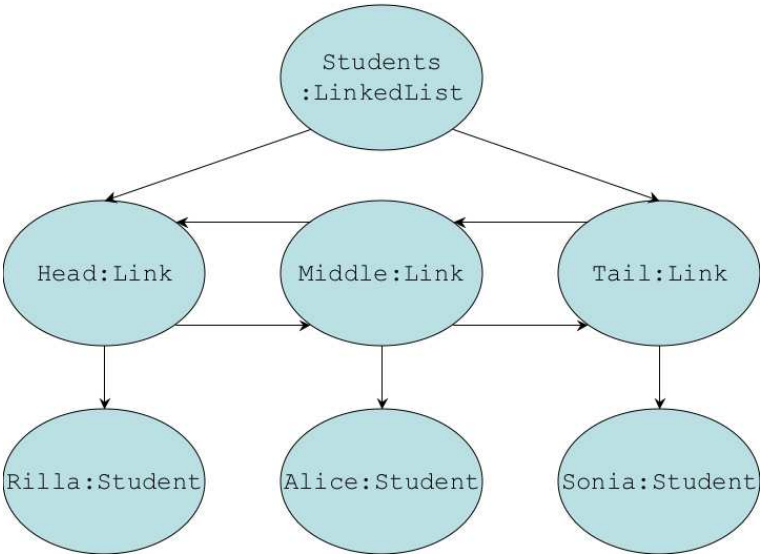


Figure 1: A simple object graph of a linked list. Each link object has two references to other link objects, except for the head and tail of the list. The student objects stored inside the list are pointed at by the link objects that store them.

is clearly a general directed graph with many cycles, rather than a tree or a directed acyclic graph, some objects (such as the `Student` "Alice") are accessed *uniquely* by only a single reference.

Object graphs are the most fundamental structure in object-orientation. The primary aim of object-oriented analysis is to model the real world in terms of communicating objects (that is, in terms of an object graph), while object-oriented design produces a description of an object graph that will eventually be embodied in a program. The artifacts and methods of object-orientation (classes, associations, interfaces, inheritance, packages, patterns, UML, CRC Cards, and so on) are ultimately techniques for defining object graphs by describing the contents of the objects and the structure of the links between them.

Given that object graphs are so basic to object-oriented programs, it is somewhat surprising that they have been paid little attention in the research community. Some temporal properties of object graphs have been analysed to support garbage collection - such as time performance of garbage collection algorithms and the distributions of object lifetimes [6]. Visualisation of object graphs has been used to support debugging [10] and programming language designers have been working on controlling object graph structures using type systems [7]. However, there has been very little work on the global structure of object graphs.

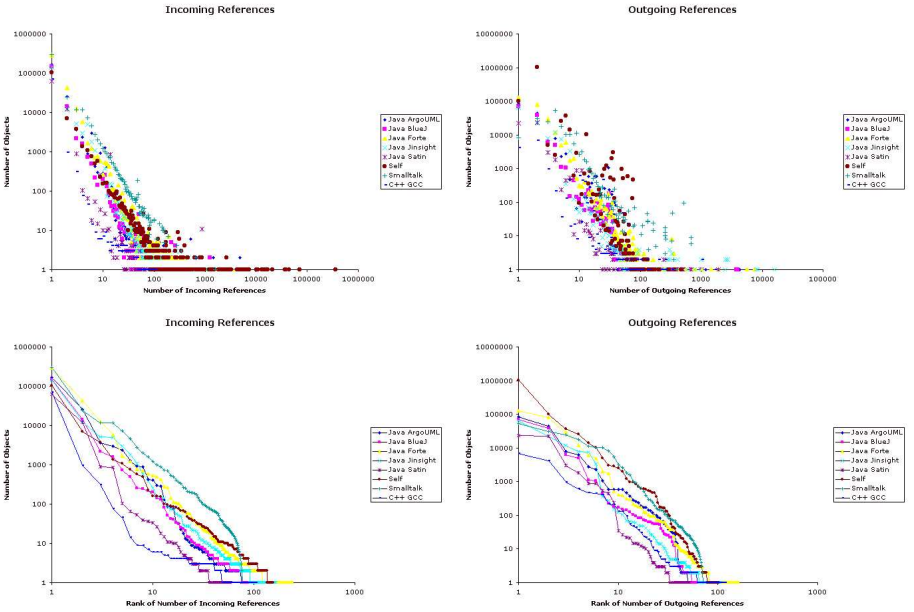


Figure 2: Power laws in object graphs. The upper two plots simply plot the number of objects with n references versus n, for incoming and outgoing references respectively. We can also use the alternative criterion in which the observed frequencies are rank ordered. The frequency is then plotted against its rank, as shown in the lower two plots. All the plots exhibit clear linearity on log-log scales, the characteristic feature of scale-free networks.

Concerning scale-free structure in programs, it has been shown [9] that class diagrams of the Java Development Kit 1.2 [8] have a scale-free aspect when *classes* are considered as nodes (not the actual instantiation of objects *of* those classes at run-time). Similar structure has also been observed in the distribution of pointers to list elements in LISP, allowing better optimisation of memory usage in long lists [3].

Power Laws in Object Graphs

We have analysed the geometry of object graphs in Java programs. Using the facilities of the Heap Profiler (HPROF) Library and the Java Virtual Machine Profiler Interface (JVMPI) [8] we collected a corpus of 60 object graphs from 35 programs, encoded as binary snapshots of the Java heap. These are instantaneous static snapshots of the objects in the programs, together with the topology of the references between them — exactly the kind of information shown in figure 1. To analyse this corpus, we extended the Java Heap Analysis Tool (HAT)

Program	Description	Objects >1 in-refs	Objects >1 out-refs
Java ArgoUML	A popular CASE tool.	203,875	153,106
Java BlueJ	Visual OO programming and learning environment.	171,666	123,701
Java Forte	A Java integrated development environment by Sun Microsystems.	358,279	267,755
Java Jinsight	A memory analysis tool by IBM.	76,312	118,272
Java Satin	A pen-based user interface research tool from Stanford University.	80,415	53,328
C++ GCC	GCC is a C++ compiler used by developers for UNIX platforms.	71,990	15,064
Self	Self is a prototype-based OO language and environment.	120,748	1,259,668
Smalltalk	Smalltalk is one of the original OO languages, self-contained in an environment developed using Smalltalk.	375,529	188,031

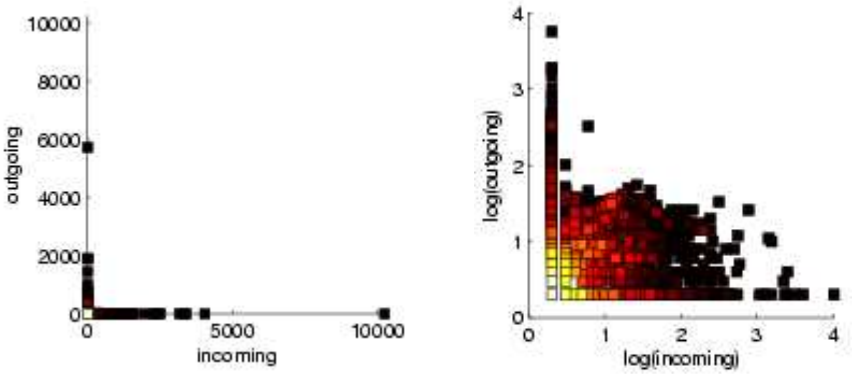
Table 1: The object graphs presented in the figures. The numbers above correspond to the objects in memory at the time that the snapshots were taken. We obtained them when the programs were most heavily used. While for C++ and Java we had to run particular programs, for Self and Smalltalk it was possible to obtain all objects for all the programs running inside these run-time systems.

[5] that parses these snapshots to determine the properties of the program's object graphs. Five large Java snapshots were taken, and three additional object graphs from programs in other object-oriented languages were also acquired, as described in table 1. These were selected for their size, popularity, and diversity.

For each graph, we first count the number of object with n references, for n from 1 upwards. If the object graph is scale-free we should see a straight line when this frequency is plotted against n itself (on log scales), or when frequencies are plotted against their rank ordering. Without exception, all the object graphs we examined show this phenomenon, as shown in figure 2. The same general effect applies to both incoming and outgoing references, and to a variety of smaller heap snapshots at our disposal. It appears that the world of object graphs is indeed scale-free, just like the World Wide Web, the Internet, and many other networks around us.

Perhaps the most intriguing aspect of the ranked graphs shown in figure 2 is that all eight plots have similar slopes. Such universality across samples is surprising given that the different samples were obtained from run-time snapshots of separate programs written for entirely different ends, and even in different languages! For incoming references the slope of the line is close to (minus) 2.5 while for outgoing references it is around 3.

The relatively large value of this slope reflects the fact that there are an exceedingly large number of objects with very few references. We might take



(a) (b)

Figure 3: Distribution of Incoming vs Outgoing references in Forte snapshot. Lighter squares correspond to a greater number of objects having that combination of references. (a) Data shown on linear scales. (b) The same data shown on log scales (data with zero incoming or zero outgoing references has been omitted since the logarithm diverges).

this to imply that programs tend to prefer simple objects to large ones, avoiding complexity just as software engineering guidelines would suggest. However the presence of a power law distribution indicates that this adage is not followed — instead large programs seem destined to contain objects that are much more highly connected than one might expect. For example, for the unranked power law the Java programs all have a slope of approximately minus 2, and it follows that for a given number of objects of size n there are about one quarter that number of size $2n$. Thus a program generating 10000 objects of size 1 will also involve about 2500 objects of size 2, 625 of size 4, 156 of size 8 and so on, leading to an expectation of one object of size roughly 100.

A natural question arises as to whether objects with many incoming references also have many outgoing ones. Figure 3 shows the number of objects having a given combination of incoming and outgoing references for the Forte data — the largest Java heap snapshot at our disposal. Notably there are no objects with both high in-degree and high out-degree: on the contrary, those with many incoming references have very few outgoing ones, and vice-versa.

Discussion

If object-oriented programs were constructed out of completely independent components, like Lego bricks, then we would expect the distribution of the size and popularity of objects to stay the same, no matter how large the program: the fixed-size Lego bricks can form buildings of any size. The rhetoric of object-oriented design is that large programs should be able to be constructed in just

the same way as small programs, by encapsulating complexity within objects at one level of abstraction, and then composing those objects together at the next. Thus all objects should appear to be the same size and complexity: larger programs merely use more objects and more levels of abstraction. This is not what we have found in our corpus of object-oriented snapshots however. In fact the presence of a power law indicates the reverse: there is no evidence of typical sizes (a Lego brick) to objects at all.

One aspect of scale-free networks is their robustness to damage. Because the vast majority of objects are poorly connected to the rest of the graph, deleting them has a negligible effect on the connectivity of those remaining [1]. On the other hand, a small number of "hub" objects are very highly connected, and deleting *them* is far more destructive. An implication of this is that by concentrating our debugging methodologies on such well-connected objects, rather than the small ones, we may be able to improve the reliability of code more efficiently: first eliminate bugs from the hubs, then deal with other objects.

Scale-free graphs can be generated by twin processes of growth and "preferential attachment", namely a tendency for new nodes to be linked to or from existing nodes that are themselves well connected [1]. In object-oriented programs objects are not independent of one another and may well show a form of preferential attachment to widely shared objects. This may also result in a *small world* effect whereby any two objects in the graph will be connected via a sequence of references going through one of the hubs.

Aside from their scale-free character, power laws are notable in that they have much longer "tails" than, say, exponential distributions. Thus larger programs will contain considerably larger and more popular objects than simpler models would predict. This may have consequences for both the design and implementation of object-oriented programming languages. Additionally garbage collectors can improve their performance in search and traversal of object graphs by assuming that in all probability objects will have only one or two outgoing references.

To summarise, we have found that distributions of incoming and outgoing references follow power laws. This unexpected result raises theoretical questions and has implications for debugging costs, program understanding, and garbage collection. More generally, it challenges the received view of OO design: unlike Lego bricks, objects within large programs have no characteristic scale.

References

[1] Albert, R., Jeong, H., and Barabasi, A.-L. The diameter of the world wide web. *Nature 401*, 130 (2000).

[2] Barabasi, A.-L. *Linked: The New Science of Networks.* New York: Perseus Press, 2002.

[3] Clark, D. W., and Green, C. C. An empirical study of list structures in lisp. *Communications of the ACM 20*, 2 (February 1977), 78–87.

[4] Erdos, P., and Renyi, A. On the strength of connectedness of random graphs. *Acta Math. Acad. Sci. Hungary 12*, 35 (1961), 261–267.

[5] Foote, B. Heap analysis tool. http://java.sun.com/people/billf/heap/.

[6] Jones, R., and Lins, R. *Garbage Collection*. Wiley, 1996.

[7] Noble, J., Vitek, J., and Potter, J. Flexible alias protection. In *Proceedings of European Conference for Object-Oriented Programming* (1998).

[8] Sun-Microsystems. Java development kit. http://java.sun.com/j2se/.

[9] Valverde, S., Cancho, R. F., and Sole, R. V. Scale-free networks from optimal design. http://www.santafe.edu/sfi/publications/wpabstract/200204019.

[10] Zimmermann, T., and Zeller, A. Visualizing memory graphs. *Software Visualization 2269* (May 2001), 191–204.

[11] Zipf, G. K. *Psycho-Biology of Languages*. Houghton-Mifflin, 1935.

[12] Zipf, G. K. *Human behavior and the principle of least effort : an introduction to human ecology*. New York : Hafner, 1965. Facsimile of 1949 edition.