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

School of Mathematical and Computing Sciences

# Computer Science

# A Tool for Ownership and Confinement Analysis of the Java Object Graphs

Alex Potanin

***Technical Report CS-TR-03-7***

5 May 2003

School of Mathematical and Computing Sciences
Victoria University of Wellington

Tel +64-4-463 5666
Fax +64-4-463 5045

PO Box 600, Wellington
New Zealand

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 5666, Fax +64-4-463 5045
Email Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

# A Tool for Ownership and Confinement Analysis of the Java Object Graphs

Alex Potanin

***Technical Report CS-TR-03-7***

5 May 2003

## Abstract

This poster presents a tool for the analysis of Java heap snapshots. The tool supports a flexible query language to measure various aspects of the object graph related to the studies of uniqueness, ownership, encapsulation and confinement. One of the applications of our tool was the verification of the power law dependency in the distribution of incoming and outgoing references to objects.

This poster abstract was presented in the ACM Student Research Competition and came second in OOPSLA 2002 SRC, it was also one of the three winners of the ACM Grand Finals held in 2003.

# A Tool for Ownership and Confinement Analysis of the Java Object Graphs

Alex Potanin (alex@mcs.vuw.ac.nz), ACM Member Number: 8158933
School of Mathematical and Computing Sciences, Victoria University of Wellington, New Zealand
Advisors: James Noble (kjx@mcs.vuw.ac.nz) and Robert Biddle (robert@mcs.vuw.ac.nz)

## Introduction

*Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects...* **[Boo91]**

One of the characteristics of object-oriented programs is that there are a large number of objects present inside the program's memory during its execution. Objects are connected to each other via *references* and objects utilise these references to issue commands or send messages to other objects with which they cooperate. Thus, the contents of a typical program's memory can be thought of as a directed graph of objects, known as an *object graph*. Software engineering, among many things, is concerned with software reliability. Given that the object graph lies at the foundation of most modern software products, understanding its behaviour and guaranteeing its reliable operation forms one of the essential tasks for researchers. Our project contributes to this effort.

The study of inter-object relationships in object-oriented systems requires the analysis of large data structures corresponding to object graphs. The Java programs we looked at while working on this project contained between 3,000 and 400,000 objects. To be able to examine such large systems, researchers and programmers alike use tools that allow object-to-object traversals and visualisation of various parts of the graph **[Foo, HNP00]**. Other approaches to analysing the object graphs include automated examination of possible executions of the program's source to be able to predict all possible memory structures that can arise **[GPV01]**, or careful inspection of complete traces of all object creations and destructions that happen during a given run of a program **[IBM]**. Although comprehensive and precise, both of these methods are computationally intensive and do not allow for quick calculation of properties that require examination of the whole graph at a particular instant.

Our approach involves a tool to look at small heap snapshots of various programs running in real life. Because we deal with objects inside program's memory at one particular instant, we can examine each snapshot in great detail without taking a long time to execute. Our original motivation was to examine the corpus of heap snapshots that we accumulated with respect to aliasing-related properties. But due to the simplicity of our approach, we were able to expand our interests and provide evidence of structure in object graphs that is different from what most object-oriented researchers expect. In this abstract we present our tool and the results obtained using it. *Please note that although this report is written in first person plural, all the work was performed by me as part of my BSc Honours project.*

## Aliasing

Aliasing occurs when there is more than one pointer referring to one object. This causes the state of the object with respect to its referrers to be compromised because one referrer can change the state of the object without others knowing about it. A number of real world problems caused by aliasing are described by Noble, Potter, and Vitek **[NPV98]**, and Bokowski and Vitek **[BV99]**. They include several unsafe implementations of `Hashtable` and a security breach in Sun's JDK 1.1.1. In popular programming languages aliasing is endemic and unavoidable, as even something as simple as the assignment statement causes an extra alias to be created. There has been much research done that addressed the problems caused by aliasing, including alias protection schemes **[MH99, NPV98]** and, in the case of assignment, alternatives to the assignment statement in component-based software engineering **[HW91]**.

Aliasing-related defects and errors, cause concern for people who are trying to implement secure and reliable systems in reference languages such as Java. To address these issues a number of characteristics of the object graph are being explored **[NPV98, BV99]**. These include the uniqueness of objects (when they only have a single incoming reference), object encapsulation (guaranteeing that

things such as `Hashtable` elements will not be referred to from outside), object ownership (exploring the encapsulation of objects on the scale of the whole of the object graph), and confinement (restricting the object access rights to those objects that are in the same domain).

**Uniqueness and Encapsulation.** Uniqueness is the most basic type of aliasing control: a unique object is encapsulated within its sole referring object **[BNR01]**. This implies that the surrounding object can depend upon the unique object for its private state without the aliasing-related concerns. When examining an object graph we look at the percentage of objects that are unique versus those that are not. We can examine what classes of objects tend to stay unique when instantiated versus those that are rarely unique. All of these properties can be examined using our tool and some of the results are presented in a later section.

All unique objects are encapsulated within their referrers. Encapsulation is about preserving the object boundaries. We were always taught in the software engineering courses that information that is private to the object should not be exposed. A similar concept applies to collections of objects. A private pointer to an object that constitutes the state of another object should not be shared or given to anyone. These relationships can be checked using query-based debuggers such as those by Lencevicius **[Len00]**. Our tool is also capable of doing this via the introduction of an appropriate query **[PN02]**.

**Ownership.** The concept of encapsulation of one object within another can be extended to the whole of the object graph. Modern memory managers, such as the one in Java Virtual Machine (JVM), provide *garbage collection* that ensures that those objects that are no longer needed by the program are deleted so that memory that they occupy can be reused by new objects. To be able to find unused objects, the garbage collector maintains a special selection of objects called the *root set*. The garbage, or no longer used objects, are then defined as follows: "*An object is **garbage** if and only if there are no reference chains from some object in the root set to it.*"

Ownership uses the concept of root set to structure object graphs. We posit a global "fake" root *r* through which all the objects in the root set of an object graph can be accessed. Then, *ownership*, which is based on the notion of *dominators* from graph theory, can be defined as follows: "*An object **a** **owns** another object **b** if all the paths from the root **r** to the object **b** go through **a**. In this case **a** is called the **owner** of **b**.*" The implication of ownership is that no object outside the owner *a* is allowed to have a reference to *b*. Ownership allows us to structure an object graph into an implicit ownership tree. To clarify this concept, consider an example on the left in figure 1. The nodes form a memory graph with root *R* and dotted lines as edges, while the red lines show the resulting ownership tree. Note how it has exactly the same number of nodes but a lot fewer edges.

**Confinement.** Every class in Java belongs to some package (if none is specified then it belongs to a so-called default package). A class is called *confined* if all instances of it are only referred to by instances of classes in the same package. The idea of confinement was explored by Bokowski and Vitek **[BV99]** and further by Grothoff, Paslberg, and Vitek **[GPV01]**. The package need not be the unit of confinement. For example, objects can also be confined to some general domain defined by the programmer (for example by explicitly enumerating the classes in each domain).

Both ownership and confinement relate to an object's encapsulation with respect to aliasing. They restrict the amount of aliasing possible within the object graph. Ownership examines a real picture of the object graph and states which objects are within other object's "shadow" in a sense that noone outside those objects underneath the owner in the ownership tree is allowed to have references to those inside. Confinement looks at the collection of classes which will produce object graphs when executed. From this static code, confinement checkers can derive instances of which classes will be guaranteed to stay within the "shadow" of their defining package in a sense that noone outside those objects whose classes are in the same package will have references to them throughout all possible lifetimes of a program.

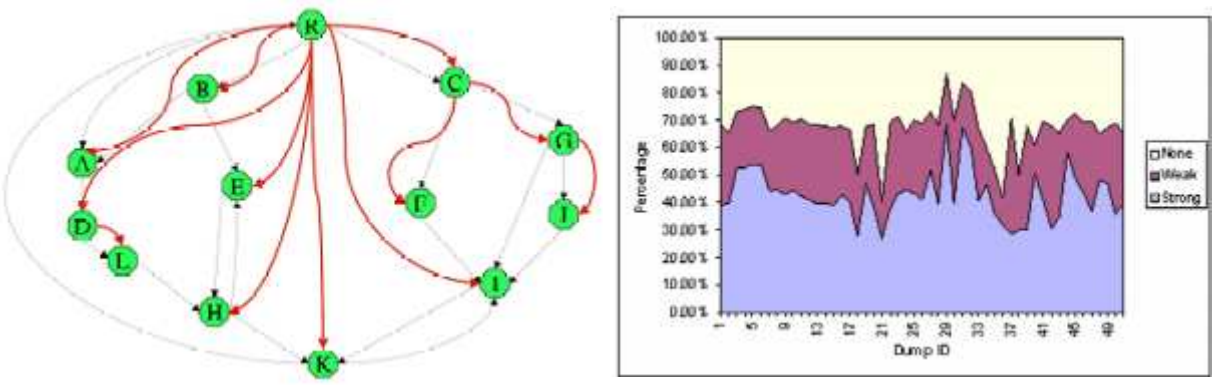



**Figure 1: Ownership Tree Example (left) and Three Kinds of Confinement Across the Corpus (right)**

## The Fox Query-Based Debugger

Analysing object graphs has always been a popular topic in object-oriented software engineering research **[ZZ01, GPV01, PNC98, HN01, Len00, HNP00]**. One of the motivations is that they can provide a valuable insight into the real behaviour of a given program, sometimes different from the intended behaviour created by programmers who wrote the underlying classes. Such analysis can help us with understanding, debugging, and maintaining object-oriented programs.

Among the tools that interested us were Lencevicius' Query-Based Debuggers **[Len00]**. They allow a programmer to either dynamically (as the program is running) or statically (as the program's memory snapshot is considered) verify relationships between objects. Lencevicius' work clearly demonstrated the advantage of having a flexible query language that can be used to specify the relationship that a programmer wishes to verify. For example, the following query will verify if all the nodes in a linked list point to different elements:

```
LinkedListNode* l1, l2;    // Types required by the expression below
l1.element != l2.element;  // Relationship to verify
```

**The Tool.** The initial aim of our project was to see if there is any evidence of the presence of ownership or confinement in Java programs in common use. We tried and failed to find a tool that would allow us to perform these measurements (early in 2002). The majority of tools that dealt with ownership or confinement either analysed potential states a program can be in, or visualised without analysis parts of the real state of a program.

At first, we tried to utilise the Java Debug Interface (JDI) **[JDI]** to track all object creations and destructions to get a complete picture of the changes to the object graph. Unfortunately, on even the smallest `HelloWorld` program, the slowdown factor was around 100. This rendered the approach unusable, especially for large programs. This motivated us to take a different approach. We used the Java Virtual Machine Profiler Interface (JVMPI) **[JVMPI]** to obtain snapshots of memory of Java programs at our disposal during their run. We accumulated a large corpus of such snapshots so that we can (a) have all the information about the state of a program at a particular intant, (b) have little effect on our ability to execute the program in question, and (c) be able to process information about the current state of the object graph in one go, without having to figure out its current state from, say, traces of object creations and destructions. This instantaneous approach worked well for us because it allowed us to measure all the properties we set out to find, and it provided us with a mechanism to experiment with a number of different aspects of object graphs.

The latest version of Fox works with heap snapshots obtained using a Heap Profiler library (HPROF) **[JVMPI]** that comes with Sun Microsystems Java Developer Kit 1.2 or higher. When instructed to load a file with the heap information, Fox utilises a Heap Analysis Tool (HAT) library by Bill Foote **[Foo]** to parse the snapshot. It then uses a well-known dominator algorithm by Lengauer and Tarjan **[LT79]**

to construct an ownership tree from an object graph. From these two main data structures and a number of additional observations, the Fox calculates as much information about each object on the heap as necessary for the queries that can be asked using the Fox Query Language (described in the next section).

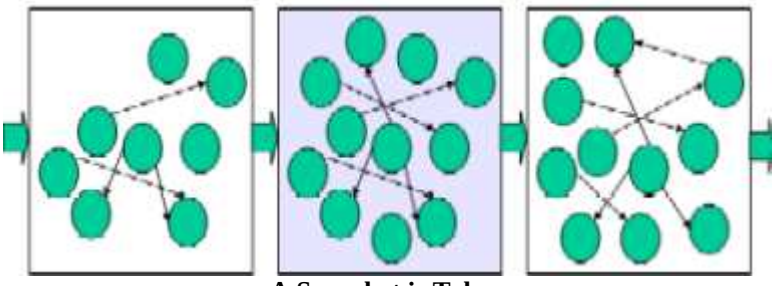
**The Fox Query Language.** The study of object graphs is about objects, hence the central concept underlying the Fox Query Language is the *heap object*. Once the information about a single heap snapshot is loaded into memory, we take a view of it as that of a single database table with each row containing information about a single object and each column denoting a particular property of the object (e.g. object's ID, object's class name etc.). We access information about each object by accessing its *properties*. Thus, for each heap object we calculate a number of properties and store them together inside a large table so that it can be closely examined by a user. We extend our analogy with a common database by allowing selection of objects from the table corresponding to a heap's snapshot in a manner similar to the `SELECT ... WHERE` statement in the Structured Query Language (SQL). We refer to the selection part of our query language as *filters*. Filters allow us to restrict the objects to those meeting a number of constraints on their *properties*.

Some properties, such as `PID` (object's unique ID), and `PClassName` (object's class name) come from the heap itself as they are intrinsic to the objects when used by the garbage collector inside the Java Virtual Machine. These are simply read from the heap snapshot file and recorded appropriately. Most other properties need to be calculated by Fox. While object graph specific properties, such as the number of links between the objects, can be calculated by a closer examination of the memory graph, more complex properties that relate to ownership and confinement require Fox to first construct an appropriate representation of the heap information. To allow the filters to compare the property values with parameters supplied by a user, each property has a type. The types include *integer*, *boolean*, *string*, and *list of integers*. This lets the parser check if the values supplied to the filters are compatible with those stored inside the properties.

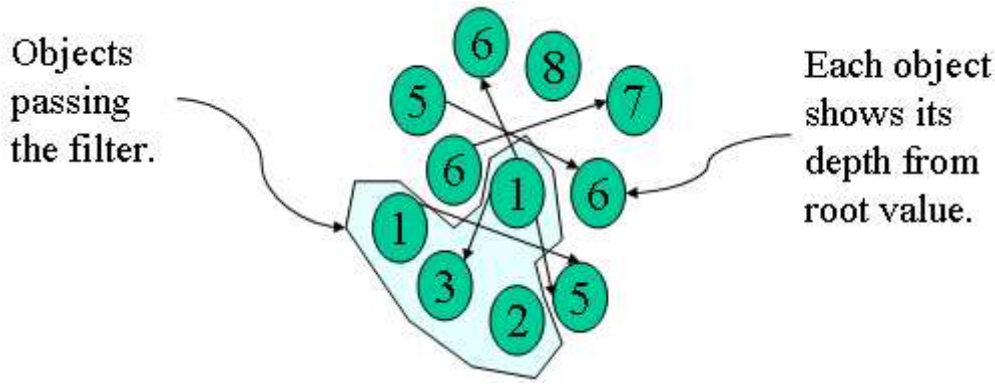
Finally, for a user to utilise the information available to them, we provide a number of *queries* that can be run upon different selections of objects returned by filters. Queries include a standard set of operations such as counting the objects or finding a minimum or a maximum value of a particular property, a set of control queries that are designed to be inserted into scripts to tell Fox when to load another snapshot or when to save the results, and a set of interactive queries such as `QVisualiseTree` that visualises an ownership tree of the current memory graph. Figure 2 gives an illustration of how a typical query works together with filters and properties. Put together, the syntax of FQL works as follows:

```
<query> :== query_name([<fltr_combination>] [, query_parameters]*);
<fltr_combination> :== fltr_name([fltr_parameters,]*
<fltr_combination>);
<fltr_combination> :== fltr_name(<fltr_combination>
[,<fltr_combination>]*);
<fltr_combination> :== FSnapshot();
```

**Object Graphs in Some Program as Time Goes By**



**A Snapshot is Taken**




**FIntegerProperty("PDepthFromRoot", < 4, FSnapshot());**

**Then We Run a Query over the Filtered Objects: QAverage(*filter*, *"PNumberOfOutgoingReferences"*);**
**Giving:** ***Averaged 0.5 across the objects passing the filter.***

**Figure 2: How Queries, Filters, and Properties Fit Together**

## Corpus Analysis

In the sections that follow, we briefly go over the results obtained using our tool. For readers interested in more details, we recommend to read Potanin and Noble **[PN02]**, Potanin, Noble, Frean, and Biddle **[PNFB]**, and Potanin **[Pot02]**. We calculated encapsulation metrics across our corpus of 58 snapshots obtained from around 35 programs - half of which were taken from the Purdue Benchmark Suite **[GPV01]**, the others are freely available over the Internet. The snapshots for large programs were taken at different stages: *initial*, when the program is initialised but hasn't been used yet, *normal*, when the program was used in a normal way, and *heavy*, when the program was driven to use as much memory as possible. To determine the sensitivity of our analysis to the precise instants when snapshots were taken, we made several additional snapshots for some of the large programs in our corpus. So far, no discrepancies were detected in this way.

**Uniqueness and Encapsulation.** We analysed the object graphs to determine a number of non-unique objects, in particular, percentage of objects with more than one incoming reference. We found that on average only 13% of all objects had more than one object pointing at them, or in other words had an alias. This means that the number of aliases in the systems we looked at was not too high. By comparing different stages of the program run, we found that the percentage of aliased objects in the system often increased. For example, `Kacheck/J` rises from 3.01% of aliased objects during the initial stage to 14.09% of objects being aliased in the heavy load stage.

**Ownership.** Our primary metric of object ownership is the average depth of an object in the ownership tree, that is, the average number of levels of encapsulation around any object. In most programs in the corpus, this was around 5 or 6; however some large programs had substantially larger values (e.g. 817.25 for `BlueJ` - a Java Programming Learning Environment - under heavy load). Given these figures, we hypothesised that large data structures such as linked lists could have a very significant effect when calculating the average depth of the node in the ownership tree: the average depth of a list node would be half the length of the list. To address this issue, we decided to fold up the reference paths by counting chains of objects of the same class (e.g. `LinkedList$Node`) as having the length of 1. This gave us a less biased account, with the average depth after folding being around 5.30 as opposed to 43.35 across all programs, with the large outlying depths being greatly reduced (e.g. a simple linked list test program has average depth 142.37 and folded depth 4.21). Furthermore, we were able to observe that the median of the average depths is usually only 7.40, thus confirming that folded depth estimates ownership depth better than the average depth itself. The resulting ownership metric does demonstrate, however, that there is a significant amount of object-based encapsulation in Java programs.

**Confinement.** The final aspect of encapsulation we analyse is object confinement, which is an instantaneous approach to class confinement **[BV99]**. If all the referrers to an object are in the same package as the object's class, we call the object *strongly confined*. If all the referrers are in the same top level package, we call the object *weakly confined*, this is similar to the idea of hierarchical packages in Grothoff, Palsber, and Vitek **[GPV01]**. If there are referrers from a different package, we call it *not confined*. For example, if `java.util.Vector` object is pointed at by `hat.model.Snapshot` object, then it is not confined. If all the referrers of `java.util.Vector` are members of `java.*` but not necessarily of `java.util.*` then it is weakly confined. Our measurements have shown that around 46% of the objects were not confined, 21% were weakly confined, and 33% were strongly confined. At first glance, these numbers are roughly what static analysis by Grothoff, Palsberg, and Vitek **[GPV01]** leads us to expect, even though we are looking from a different perspective. Figure 1 on the right showed the percentage distribution between three kinds of confinement across our corpus. We further analysed confinement on a per-class basis, rather than per-object. An interesting observation we were able to make was that classes having 0% or 100% of their instances strongly confined completely dominated in numbers. We had hypothesised that there may be a significant fraction of the instances of some classes which were "almost" confined (90%), but this did not appear to be the case.

## A Scale-free Nature of Object Graphs

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 **[AJB00]**. 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 **[ER61]**. By contrast, the web has *no* typical scale to its connectivity - a remarkable and somewhat counterintuitive property closely related to that of fractals **[AJB00]**.

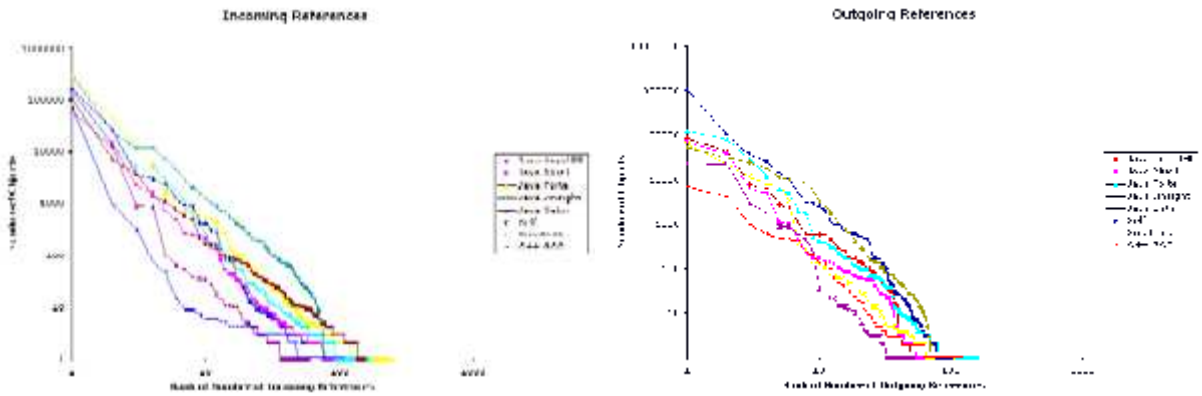



**Figure 3: Demonstrating the Distribution of the Rank by the Number of Incoming and Outgoing References versus the Number of Object of Each Rank**

Although the Fox was designed to analyse ownership and confinement, the flexibility of its query language allowed us to observe the distribution of incoming and outgoing references inside the object graph. Following the standard way used in literature as summarised by Barabasi **[Bar02]**, to demonstrate that the graphs in question were scale-free, we ranked the objects by the number of incoming and outgoing references and then plotted them on a LOG-LOG scale against the number of objects of each rank. Both incoming and outgoing reference graphs shown in figure 3 produce neat lines, thus demonstrating that object graphs are indeed scale-free.

Our tool allowed us to examine the graphs formed by object-oriented programs written in Java and show that these turn out to be scale-free networks without exception. Inspired by these results we expanded our attention to other object-oriented languages such as C++, Smalltalk, and Self. We examined object graphs of large Smalltalk and Self images, as well as a memory snapshot of GNU C++ compiler to find that these too are scale-free. As one final side-note, since the amount of results we can include in this abstract is limited, we would like to point out that when we attempted to plot the number of incoming references versus the number of outgoing references, we failed to find a correlation, at least in large Java programs like `Forte`. 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.

## Conclusion

We presented a tool called Fox that we created and the results of an analysis of a large number of object graph snapshots that we accumulated: our corpus. We showed evidence of both encapsulation and aliasing within modern object-oriented programs and we demonstrated that they have a non-hierarchical scale-free structure - a large number of rarely referenced and a small number of highly referenced objects - just like the sites in the World Wide Web **[AJB00]**. More information of these findings and more observations can be found on my web site **[Pot].**

## References


**[AJB00]** Reka Albert and Hawoong Jeong and Albert-Laszlo Barabasi. *The Diameter of the World Wide Web*. Nature 401(130), 2000.

**[Bar02]** Albert-Laszlo Barabasi. *Linked*. Perseus Publishing, April 2002.

**[BNR01]** John Boyland, James Noble, and William Retert. *Capabilities for sharing*. In Proceedings of ECOOP'01, Springer-Verlag, 2001.

**[Boo91]** Grady Booch. *Object-Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, 1991.

**[BV99]** Boris Bokowski and Jan Vitek. *Confined types*. In Proceedings of OOPSLA'99, ACM Press, 1999.

**[ER61]** P. Erdos and A. Renyi. *On the strength of connectedness of random graphs.* Acta Math. Acad. Sci. Hungary 12(35): 261-267, 1961.

**[Foo]** Bill Foote. *Java Heap Analysis Tool*. Available at: http://java.sun.com/people/billf/heap/index.html.

**[GPV01]** Christian Grothoff, Jens Palsberg, and Jan Vitek. *Encapsulating objects with confined types*. In Proceedings of OOPSLA'01, ACM Press, 2001

**[HM01]** Chanika Hobatr and Brian A. Malloy. *The design of an OCL query-based debugger for C++*. In Proceedings of 16th ACM SAC2001 Symposium on Applied Computing, 2001.

**[HNP00]** Trent Hill, James Noble, and John Potter.*Scalable visualisations with ownership trees.* In Proceedings of TOOLS Pacific 2000, Sydney, Australia, IEEE CS Press, 2000.

**[HW91]** D. E. Harms and B. Weide.*Copying and swapping: influences on the design of reusable software components*. In IEEE Transactions of Software Engineering, 17(5): 424-435, IEEE CS Press, 1991.

**[JVMPI]** Sun Microsystems. *Java Virtual Machine Profiler Interface*. Available at: http://java.sun.com/j2se/1.4/docs/guide/jvmpi/index.html.

**[JDI]** Sun Microsystems. *Java Debug Interface*. Available at: http://java.sun.com/products/jpda/doc/jdi/.

**[IBM]** IBM AlphaWorks. *Jinsight*. Available at: http://www.alphaworks.ibm.com/tech/jinsight/.

**[Len00]** Raimondas Lencevicius. *Advanced debugging methods*. Kluwer Academic Publishers, August 2000.

**[LT79]** Thomas Lengauer and Robert Endre Tarjan. *A fast algorithm for finding dominators in a flowgraph*. In ACM Transactions on Programming Languages and Systems, 1(1):121–141, July 1979.

**[MH99]** P. Muller and A. Poetzsch-Heffter. *Universes: A type system for controlling representation exposure*. In A. Poetzsch-Heffter and J. Meyer editors, Programming Languages and Fundamentals of Programming, Fernuniversitat Hagen, 1999.

**[Mun]** Tamara Munzner. H3Viewer. Available at: http://graphics.stanford.edu/~munzner/h3/.

**[NPV98]** James Noble, John Potter, Jan Vitek. *Flexible alias protection*. In Proceedings of ECOOP'98, Springer-Verlag, 1998.

**[PN02]** Alex Potanin and James Noble. *Checking Ownership and Confinement.* In proceedings of the Workshop on Formal Techniques for Java-like Programs, ECOOP'02.

**[PNC98]** John Potter, James Noble, and David Clarke. *The ins and outs of objects*. In Proceedings of Australian Software Engineering Conference (ASWEC), IEEE CS Press, 1998.

**[PNFB]** Alex Potanin, James Noble, Marcus Frean, and Robert Biddle. *Scale-free Geometry in Object-Oriented Programs.* Submitted for publication in Communications of ACM.

**[Pot02]** Alex Potanin. *The Fox - A Tool for Java Object Graph Analysis.* BSc Honours Report, Victoria University of Wellington, 2002.

**[Pot]** Alex Potanin's Web Site. `http://www.mcs.vuw.ac.nz/~alex/`.

**[ZZ01]** T. Zimmermann, A. Zeller. *Visualizing memory graphs*. In Proceedings of the Dagstuhl Seminar 01211 "Software Visualization", Lecture Notes in Computer Science, Dagstuhl, Germany, Springer-Verlag, May 2001.