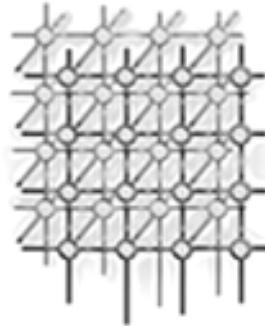


Checking Ownership and Confinement

Alex Potanin ^{*,†}, James Noble[‡], and Robert Biddle[§]

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



SUMMARY

A number of proposals to manage aliasing in Java-like programming languages have been advanced over the last five years. It is not clear how practical these proposals are, that is, how well they relate to the kinds of programs currently written in Java-like languages. To address this problem, we analysed heap snapshots from a corpus of Java programs. Our results indicate that object-oriented programs do in fact exhibit symptoms of encapsulation in practice, and that proposed models of uniqueness, ownership, and confinement can usefully describe the aliasing structures of object-oriented programs. Understanding the kinds of aliasing present in programs should help us to design formalisms to make explicit the kinds of aliasing implicit in object-oriented programs.

KEY WORDS: Aliasing, uniqueness, ownership, confinement, object-oriented programming

1. INTRODUCTION

There has been much recent research on managing aliasing in object-oriented programming languages through uniqueness, ownership, and confinement [NL01, BNR01, MPH99, NVP98, CNP01, CPN98, BV99, GPV01, AKC02, BLS03, CD02]. Unfortunately, it is not clear how applicable many of these approaches will be: that is, how well they can describe the kinds of aliasing used by programmers in practice. While there should be more to theories of object-oriented program design than rhetoric, we have found that there is surprisingly little solid evidence that programmers concern themselves with encapsulation, or adopt encapsulation disciplines in the programs they actually write.

In this paper we take an empirical approach to evaluating aliasing in object-oriented systems. By analysing snapshots of a corpus of running Java programs' heaps, we search for evidence of object uniqueness, ownership, and confinement. We hope that our results will enable researchers to verify their results and find new directions in the formal exploration of aliasing.

*Correspondence to: Alex Potanin, School of Mathematical and Computing Sciences, Victoria University of Wellington, PO Box 600, Wellington, New Zealand

[†]E-mail: alex@mcs.vuw.ac.nz

[‡]E-mail: kjax@mcs.vuw.ac.nz

[§]E-mail: robert@mcs.vuw.ac.nz

Contract/grant sponsor: Royal Society of New Zealand Marsden Fund; contract/grant number: 01-VUW-073

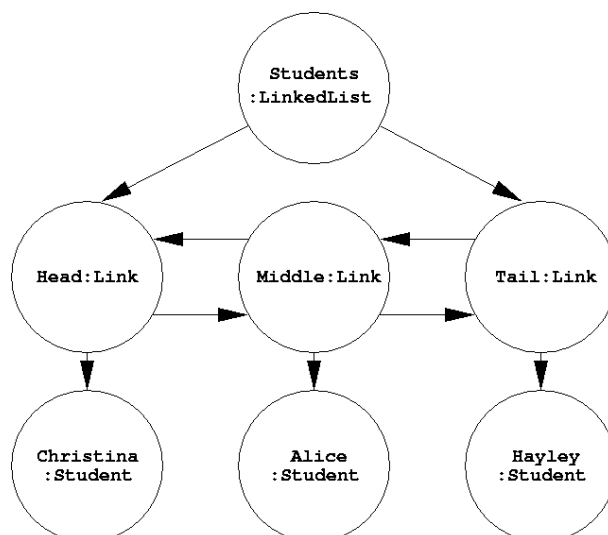


Figure 1. Simple object graph of a linked list

2. BACKGROUND

In this section we provide brief overviews of object graphs and research into aliasing in object-oriented systems.

2.1. Object Graphs

An object graph — the object instances in the program and the links between them — is the skeleton of an object-oriented program. 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, and a third reference to one of the `Student` objects contained in the list. Although the overall structure 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. In running programs, object graphs have one or more *roots* (such as threads, or global or stack variables) representing objects that are permanently accessible by the current computation: objects that become unreachable from the roots are considered garbage and any memory they occupy may be reused.



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.

2.2. Aliasing

An object is *aliased* whenever there is more than one pointer referring to that object [HLW⁺92]. Aliasing can cause a range of difficult problems within object-oriented programs, because one referring object can change the state of the aliased object, implicitly affecting all the other referring objects [NVP98, BV99]. Aliasing is endemic and unavoidable in object-oriented programming languages, as any assignment statement may cause an extra alias to be created.

As aliasing has been recognised as a significant challenge for object-oriented programming, there have been a number of proposals to mitigate its effects upon programs, ranging from alias protection schemes [MPH99, NVP98] to alternatives to the assignment statement in component-based software engineering [HW91]. In this paper, we are concerned with three major approaches to managing aliasing: uniqueness, ownership, and confinement.

2.2.1. Uniqueness

A *unique* object is guaranteed to have only a single incoming reference [Hog91]. Uniqueness is the most basic type of aliasing control: a unique object is encapsulated within its sole referring object [BNR01]. This implies that the enclosing object can depend upon the unique object for its private state without aliasing-related concerns. Note there are a number of alternative programming language *mechanisms* that ensure uniqueness (including swapping, destructive reads, alias-burying, and so on [Hog91, HW91, Alm97, Boy01]), however the *policies* that these mechanisms enforce are broadly similar.

Typical object-oriented programming languages do not impose any uniqueness constraints on objects. In order to determine how practicable such constraints would be, we have empirically examined the object graphs of a corpus of Java programs to determine the fraction of objects which were unique.

2.2.2. Ownership

Object ownership is in essence a generalisation of uniqueness [PNC98] that underlies many more sophisticated alias management schemes [NVP98, CPN98, CNP01, MPH99]. Where uniqueness requires that an object can only be accessed via a single unique referring object, ownership allows an object to be accessed by any number of incoming references, provided that the object is always contained within its *owner*. In other words, no object outside the owner a is allowed to refer to an subordinate object b .

Ownership allows us to structure an object graph into an implicit ownership tree. More formally, we posit a global root r through which all the roots of an object graph can be accessed. Then, ownership is based on graph-theoretic domination: a owns b if a is a dominator for b , that is, if all the paths from the global root r to the object b go through a .

By empirically examining the ownership tree within programs' object graphs, we can determine whether current object-oriented programs are written in a style which is conversant with object ownership, and thus whether ownership is likely to be a useful technique for managing aliasing.



2.2.3. Confinement

Confinement was explored by Bokowski and Vitek [BV99] and further by Grothoff, Paslberg, and Vitek [GPV01]. Where object ownership ensures a subordinate object can only be accessed via its owning *object*, confinement ensures a subordinate object can only be accessed within some larger protection domain, such as a Java package.

For example, every class in Java belongs to some package (if no package is explicitly specified then it belongs to a so-called default package). A class is *confined* if all its instances are only referred to by instances of classes in the same package: the referring classes may be confined or unconfined.

As with uniqueness and ownership, there has been only a little independent study of the practicability of confinement — that is, how much confinement subsists in object-oriented programs in practice, or alternatively how much existing programming style would have to change to make use of confinement. Our aim in this paper is to address this issue, by empirically analysing the degree of uniqueness, ownership, and confinement in a corpus of Java programs.

3. ANALYSING JAVA OBJECT GRAPHS

When we started our work on this project, our aim was to find a tool that would allow us to measure encapsulation aspects of real object graphs inside running programs. Although there was a large repository of visualisation and static analysis tools [ZZ01, GPV01, PNC98, HM01, Len00, HNP00], none of these tools could effectively estimate uniqueness, ownership or confinement parameters.

For this reason, we decided to develop our own object graph analysis tool. We decided to analyse Java programs, because there are a large number of Java programs available on the Internet, because Java is straightforward to analyse compared with hybrid object-oriented languages such as C++, and because Sun's Java Developer Kit (JDK) [SM02] includes a number of standardised interfaces and libraries to support tool development.

We first attempted to determine programs' object graphs via the Java Debug Interface (JDI) [Mic02a] that supports dynamic monitoring of any Java program as it executes. Unfortunately, this interface could not support sufficient performance to monitor the complete execution of large Java programs — a simple program, with 3,000 new object allocations, was slowed down by approximately 100 times. Thus, we decided that dynamically monitoring the ownership structure of a complete object graph was beyond our capability at the time, and we dismissed full memory simulation [DH99] for similar reasons. Program visualisation tools [HNP02] have avoided this problem by only monitoring the part of the program that is being displayed, for example, user-written objects. On the other hand, we needed to gather information about programs' entire object graphs.

3.1. HAT: The Heap Analysis Tool

The Java Virtual Machine Profiler Interface (JVMPi) [Mic02b] can be used as an alternative to the JDI to gather information about Java programs. Unlike the JDI (that communicates information between separate Java programs on individual virtual machines) the JVMPi provides low-level hooks directly into the structures of the virtual machine. Part of this interface allows the user to save a snapshot of the entire object graph of any Java program into a file.

These heap snapshots can be inspected using the Heap Analysis Tool (HAT) written by Bill Foote [Foo02]. Using a web browser as a user interface, the HAT supports simple interaction with an object graph — navigating links forwards and backwards between objects, enumerating all the instances of a given class, and so on.

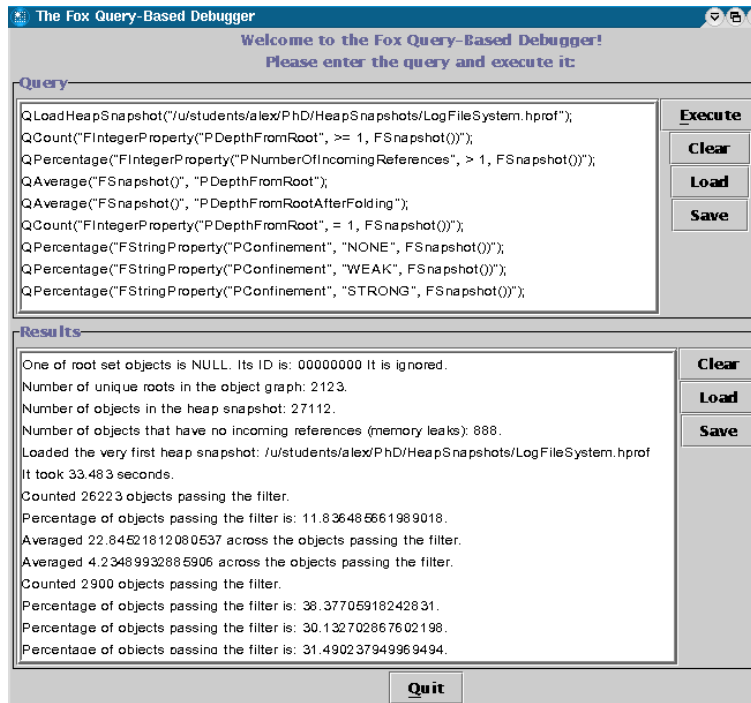


Figure 2. A screenshot of the Fox query-based debugger running on Solaris

3.2. The Rabbit That Came Out Of The HAT

We wrote a tool (named Rabbit) that used information from HAT about the objects to construct an ownership tree from the object graph. Rabbit used a well known dominator tree construction algorithm [LT79] to construct the ownership tree, and then calculated a range of uniqueness, ownership, and confinement metrics. Each calculation was specified in a separate class: a new calculation had to be implemented by providing a new calculation subclass and recompiling Rabbit.

3.3. The Fox That Came After The Rabbit

The need to recompile Rabbit, debug a new extension, and then reanalysing a corpus of object graph snapshots, meant that Rabbit was not particularly effective for exploratory analysis of object graphs. After examining the related research in object graph analysis using query-based debuggers, especially the work by Lencevicius [Len00], we decided to improve the flexibility of our tool by introducing a query language.



Figure 2 shows the final tool that we developed, called Fox, as a successor to Rabbit. Fox's query language called FQL (Fox Query Language) can express a range of ownership calculations without rebuilding the program or reloading object graph snapshots.

Once the information about a heap snapshot is loaded into memory, Fox views it as a single database table with each row containing information about a single object. We calculate a number of *properties* for each object and store them as columns in the table. FQL *filter* statements select objects meeting a set of constraints on their *properties*, and *queries* report information to the user.

FQL queries include standard 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 heap snapshot or when to save the results, and a set of interactive queries that visualise part of the current object graph.

For example, to determine the percentage of objects having more than one incoming reference in a snapshot, we would run the following FQL query:

```
QPercentage("FIntegerProperty(
  "PNumberOfDynamicIncomingReferences",
  > 1, FSnapshot())");
```

An important point about Fox (and Rabbit and HAT) is that it empirically analyses object graphs of running Java programs, as opposed to static source code or dynamic behaviour traces. We chose this method because we aimed to analyse metrics latent within a corpus of programs, rather than benchmark a particular algorithm or alias protection scheme. Compared to static confinement checkers like Kacheck/J [GPV01], for example, our approach is instantaneous: an object may be identified as unique, confined, or otherwise encapsulated, depending on the state of the object graph at the instant a heap snapshot is taken. Thus, our results overestimate the encapsulation data: an object which is encapsulated at one instant (or in one program run) may be aliased the next instant, or in the next run. On the other hand, more conservative approaches such as Kacheck/J cannot identify objects which are effectively encapsulated in practice unless they can prove the objects will be encapsulated across all potential program executions, most of which will never occur.

4. RESULTS

We have used Fox to calculate metrics across 58 object graph snapshots obtained from a corpus of around 35 programs. Half of the corpus was taken from the Purdue Benchmark Suite [GPV01] the other programs 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 is used in a normal way, and *heavy*, when the program is 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: the results over these snapshots did not differ markedly from the selected heavy snapshots. Full details of all our results are presented in the appendices: appendix A reports the majority of the metrics, while appendix B provides extra information on class confinement.

For each snapshot, we give the number of objects in the object graph (**NO**), the number of unique roots in the Java heap root set (**NUR**), and the number of objects accessible by reference traversal (**ART**) from the root set (other objects are presumed to be uncollected garbage). These numbers give an idea of the size of the program under consideration. We found that a typical Java program in the corpus had around 65,000 objects allocated on the heap, about 1,800 of these were part of the Garbage Collector's root set (this includes static



and global variables, metaobjects, and references from the Java stack) and 3,000 (around 5% or less) were garbage, i.e. not accessible from the root set.

4.1. Uniqueness

We analysed the object graphs to determine a number of aliased objects, i.e. objects with more than one incoming reference. We report this as the percentage of aliased objects (**PAL**). We found that on average only 13% of all objects had more than one object pointing to them, or in other words had an alias. This means that the number of aliases in the systems we looked at was not particularly 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. Given that the majority of objects are not aliased, we consider that it makes sense for formal techniques to support uniqueness, however, there are enough aliased objects that uniqueness alone will be insufficient for managing aliasing within an object-oriented programming style.

4.2. Object Ownership

Our primary metric of object ownership is the average depth (**AD**) 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* 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. Figure 3 shows an H3Viewer [Mun02] visualisation of the ownership tree for a small test program that constructed a large linked list.

To address this issue, we decided to fold up such structures by counting successive objects of the same class (e.g. `LinkedList$Node`) as having a depth of 1. This gave us a less biased figure, with the average depth after folding (**ADF**) being around 5.30 as opposed to 43.35 across all programs. Importantly the large outlying depths were greatly reduced (e.g. a simple linked list test program has AD 142.37, ADF 4.21). The median of the average depths is actually only 7.40, confirming that ADF estimates ownership depth better than AD. The resulting ownership metric does demonstrate, however, that there is a significant amount of object-based encapsulation in Java programs.

For a finer-grained picture of object ownership in particular programs, we also calculated the average ownership tree depth per class. The results for all classes cannot be presented here for space reasons (our study involved over 390,000 classes) but as an example, consider classes involved in implementing Java hash tables. Since most `java.util.Hashtable` instances should own one `java.util.Hashtable$Entry[]` array instance, we would expect the average depth of the hash table to be one greater than the entry array. Figure 4 shows a class diagram of `Hashtable` related interactions in the `java.lang`. Alongside the classes, we show the average depths in the ownership tree across all the instances of each class, as found by Fox in *BlueJ*. These results confirm our hypothesis and also imply that the actual hash table entries are contained within an entry array.

4.3. Confinement

We have also calculated a number of confinement metrics over the object graphs of the programs in our corpus. Since we are looking at object graphs, we take an instantaneous

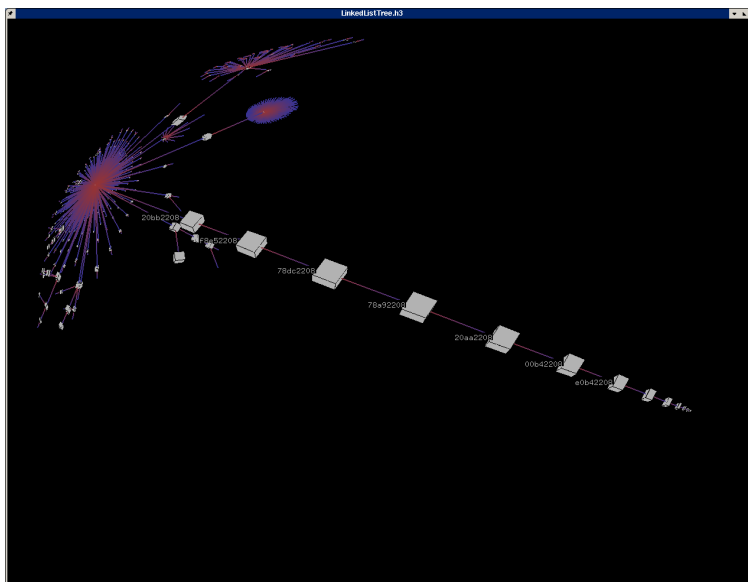


Figure 3. Linked List Ownership Tree

approach to class confinement. If the classes of all the objects which refer to an object of interest are in the same package as that 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 hierarchical packages in [BV99, GPV01]). If there are referrers from a different package, we call the object not confined. For example, if a `java.util.Vector` object is pointed at by a `hat.model.Snapshot` object, then it is not confined, while if all the classes of the referrers to a `java.util.Vector` are members of `java.*` but not necessarily of `java.util.*` then the vector instance will be weakly confined. Note that if Grothoff's static analyser [GPV01] determines that the class is confined, then all its instances should be confined from our point of view.

Over the whole corpus, our measurements have shown that around 46% of the objects were not confined (**NC**), 21% were weakly confined (**WC**), and 33% were strongly confined (**SC**). 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.

We further analysed confinement on a per-class basis, rather than per-object. The second table (Appendix B) gives the class confinement distribution. The first column, titled 0%, lists the number of classes that have no instances that are strongly confined, and then further columns list the number of classes that have a number of instances being strongly confined falling into a corresponding decile. The column named 100% counts those that have all their instances strongly confined. The last column is the Total Number of Classes (**TNC**).

One important feature of this table is that the 0% and 100% columns have by far the largest values: all the instances of a class tend to have the same confinement. We had hypothesised

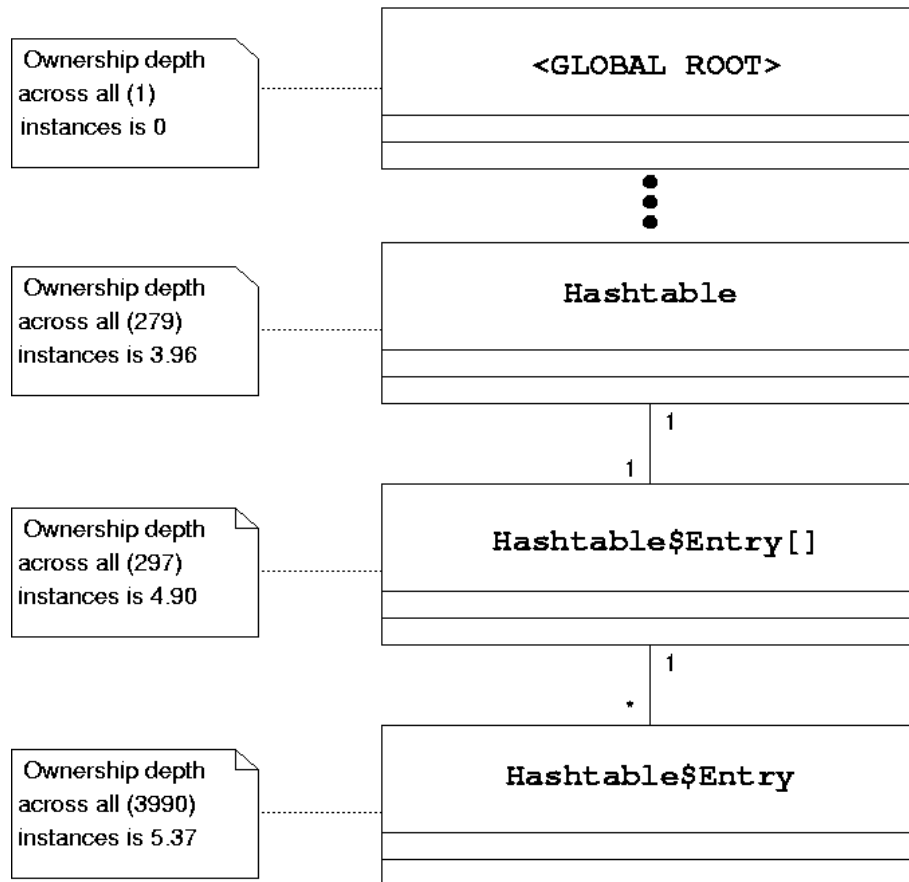


Figure 4. Hashtable Encapsulation Example

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.

We found that the vast majority of classes with instances present in our heap snapshot are not confined (88.48%). After comparing our results with the data obtained using static control flow analysis by Grothoff, Palsberg, and Vitek [GPV01] we noticed that the majority of classes that their tool detected as being confined at all times have not been instantiated in any of our heap snapshots. For those classes that were instantiated, we confirmed that they were confined from our perspective too.

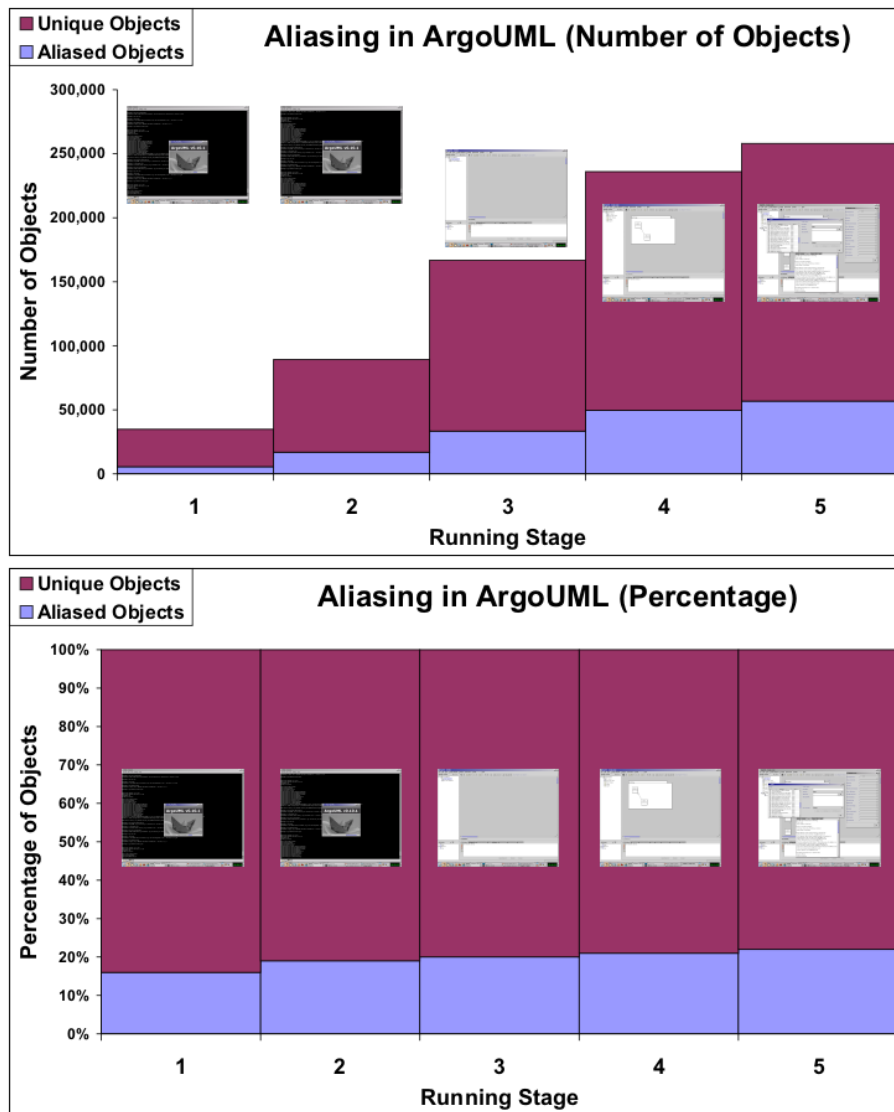


Figure 5. Aliasing at various stages of running ArgoUML, as shown in the screenshots.



4.4. Multiple Snapshots of a Single Program

All our results are based on the encapsulation information calculated from a single snapshot of a program's object graph. There is not enough information in the Java snapshots to allow us to trace individual objects or ownership across a sequence of heap snapshots of the same program run — individual objects are identified only by their memory address, which may be reused after garbage collection.

To demonstrate that our results are generally valid within programs, we have analysed a longitudinal sequence of snapshots taken during the same run of several of the larger programs in our corpus. Generally, we found that the metrics were stable once programs were heavily loaded: more specifically, Figure 5 shows a series of confinement results across snapshots of the ArgoUML program.

ArgoUML is a popular Java program for object-oriented modeling using UML. This program is open source and is available for free from <http://argouml.tigris.org/>. To generate the longitudinal results, we ran ArgoUML and took snapshots of its memory at different stages during its run. The first snapshot was taken while ArgoUML was just starting to load; the second snapshot was taken when it was about to finish loading; the third when it was running; the fourth after working on a class diagram and a use case diagram; and the fifth when ArgoUML was editing a large model and using a large amount of memory.

Figure 5 shows the results for object aliasing. The percentage of aliased objects (PAL) rose from 16%, up to 22% while at the same time the total number of objects (NO) rose from around 35,000 to over 255,000. This means that the number of aliased objects rose from 5,500 to over 55,000. Figure 6 presents the distribution of confined, weakly confined, and non-confined objects: the percentages are essentially stable throughout all the snapshots.

5. FURTHER WORK

We are now working closely with researchers in uniqueness and confinement to evaluate proposed confinement schemes in more detail. In particular, we plan to measure external uniqueness [CW03]. We are also improving our corpus of heap snapshots.

In the future, we plan to experiment with formal protection schemes which are implemented as extensions to Java [AKC02, BLS03, CD02] and to observe the effects adopting such schemes has on the object graphs of the programs they constrain.

6. CONCLUSION

We have presented metrics of uniqueness, ownership, and confinement by analysing snapshots of Java programs' object graphs. Using our tool, Fox, we measured 58 heap snapshots of a corpus of over 35 programs from test programs to larger systems with over 350,000 objects. We found that around twelve percent of objects (ranging from 5-20 percent) had more than one incoming reference; that objects were on average five layers deep within an object ownership hierarchy; and that around one third of all objects were strongly confined.

We consider that these results indicate that object-oriented programs do in fact exhibit symptoms of encapsulation in practice, and that formal models of uniqueness, ownership, and confinement, can usefully describe the aliasing structures of object-oriented programs.

More information about the Fox tool and the Fox Query Language is available elsewhere [Pot02, Pot03]. This work is supported by the Royal Society of New Zealand Marsden Fund.

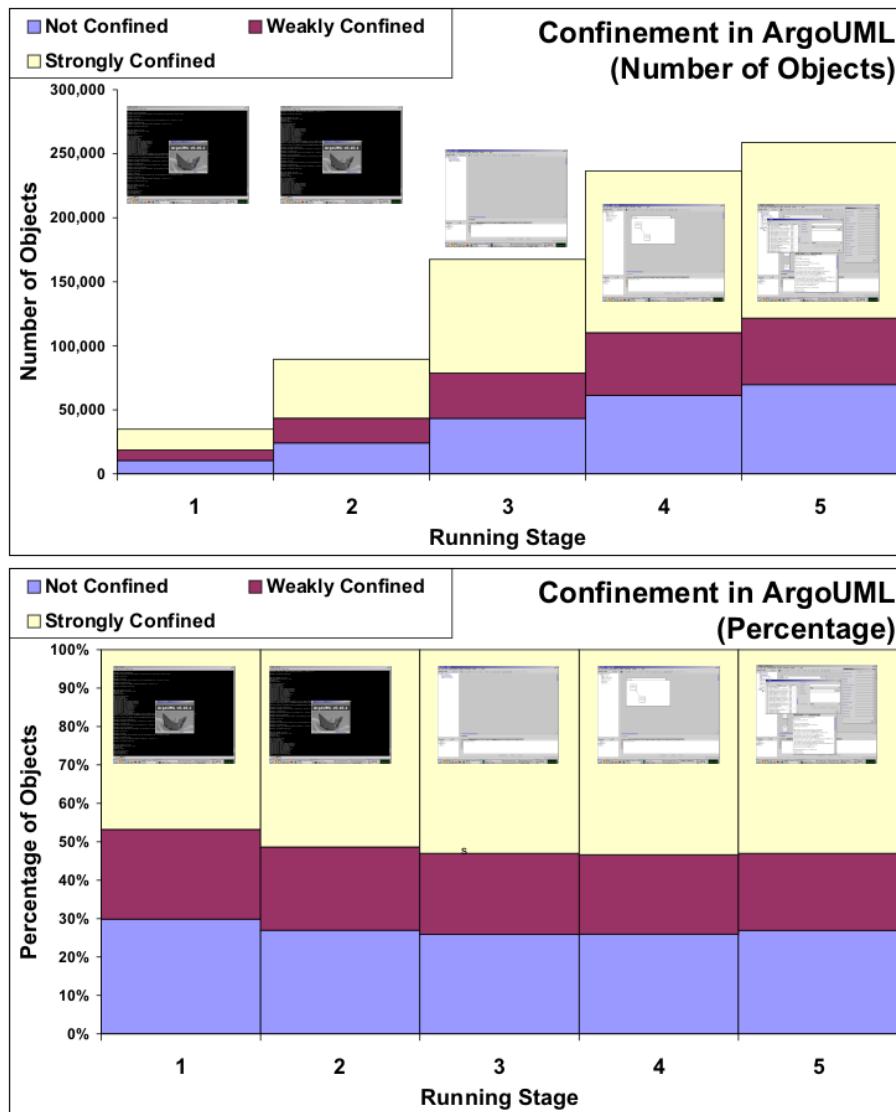


Figure 6. Confinement at various stages of running ArgoUML, as shown in the screenshots.



REFERENCES

- [AKC02] . J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA)*, November 2002.
- [Alm97] . Paulo Sergio Almeida. Balloon types: Controlling sharing of state in data types. In *ecoop*, jun 1997.
- [BLS03] . Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2003.
- [BNR01] . John Boyland, James Noble, and William Retert. Capabilities for sharing. In *Proceedings of European Conference for Object-Oriented Programming*. Springer-Verlag, 2001.
- [Boy01] . John Boyland. Alias burying: Unique variables without destructive reads. *Software — Practice and Experience*, 31(6):533–553, May 2001.
- [BV99] . Boris Bokowski and Jan Vitek. Confined types. In *Proceedings of Conference on Object-Oriented Programming, Languages, and Applications*. ACM Press, 1999.
- [CD02] . Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation, and the disjointness of type and effect. In *ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA)*, 2002.
- [CNP01] . David Clarke, James Noble, and John Potter. Simple ownership types for object confinement. In *Proceedings of European Conference for Object-Oriented Programming*. Springer-Verlag, 2001.
- [CPN98] . David Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of Conference on Object-Oriented Programming, Languages, and Applications*. ACM Press, 1998.
- [CW03] . Dave Clarke and Tobias Wrigstad. External uniqueness. In *FOOL10*, January 2003.
- [DH99] . Sylvia Dieckmann and Urs Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer-Verlag, June 1999.
- [Foo02] . Bill Foote. Heap analysis tool. <http://java.sun.com/people/billf/heap/>, 2002.
- [GPV01] . Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *Proceedings of Conference on Object-Oriented Programming, Languages, and Applications*. ACM Press, 2001.
- [HLW⁺92] . John Hogg, Doug Lea, Alan Wills, Dennis de Champeaux, and Richard Holt. The geneva convention of the treatment of object aliasing. *OOPS Messenger*, 3(2), April 1992.
- [HM01] . Chanika Hobatr and Brian A. Malloy. The design of ocl query-based debugger for c++. In *Proceedings of the 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, 2000. IEEE CS Press.
- [HNP02] . Trent Hill, James Noble, and John Potter. Scalable visualizations of object-oriented systems with ownership trees. *Journal of Visual Languages and Computation*, 13(3):319–339, June 2002.
- [Hog91] . John Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings of Conference on Object-Oriented Programming, Languages, and Applications*, volume 26, pages 271–285, New York, November 1991. ACM Press.
- [HW91] . D. E. Harms and B. Weide. Copying and swapping: influences on the design of reusable components. *IEEE Transactions of Software Engineering*, 17(5):424–435, 1991. IEEE CS Press.
- [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. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.
- [Mic02a] . Sun Microsystems. Java debug interface. <http://java.sun.com/j2se/1.4.1/docs/guide/jpda/jdi/index.html>, 2002.
- [Mic02b] . Sun Microsystems. Java virtual machine profiler interface. <http://java.sun.com/j2se/1.4.1/docs/guide/jvmpi/index.html>, 2002.
- [MPH99] . P. Müller and A. Poetzsch-Heffter. *Programming Languages and Fundamentals of Programming*, chapter Universes: a type system for controlling representation exposure. Fernuniversität Hagen, 1999.
- [Mun02] . Tamara Munzner. H3viewer. <http://graphics.stanford.edu/~munzner/h3/>, 2002.
- [NL01] . James Noble and Doug Lea. Special issue on aliasing in object-oriented systems. *Software – Practice and Experience*, 31(6), May 2001.
- [NVP98] . James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *European Conference on Object-Oriented Programming (ECOOP)*, 1998.
- [PNC98] . John Potter, James Noble, and David Clarke. The ins and outs of objects. In *Proceedings of European Conference for Object-Oriented Programming*. Springer-Verlag, 1998.
- [Pot02] . Alex Potanin. The fox - a tool for java object graph analysis. Technical Report 02/28, School of Mathematical and Computing Sciences, Victoria University of Wellington, <http://www.mcs.vuw.ac.nz/comp/Publications/>, November 2002.
- [Pot03] . Alex Potanin. A tool for ownership and confinement analysis of the java object graph. Technical Report 03/6, School of Mathematical and Computing Sciences, Victoria University of Wellington, <http://www.mcs.vuw.ac.nz/comp/Publications/>, May 2003.
- [SM02] . Sun-Microsystems. Java development kit. <http://java.sun.com/j2se/>, 2002.
- [ZZ01] . Thomas Zimmermann and Andreas Zeller. Visualizing memory graphs. In *Software Visualization*, volume LNCS 2269 of *Lecture Notes in Computer Science*, pages 191–204. Springer-Verlag, May 2001.



APPENDIX A: GENERAL COLLECTION OF METRICS ACROSS CORPUS

Program	NO	NUR	ART	PAL	AD	ADF	NRK	NC	WC	SC	TT
Aglets	23212	1588	22723	10.64	7.44	6.63	2502	37.62	29.95	32.43	20
AlgebraDB	3749	481	3506	10.58	6.08	4.32	568	36.85	26.64	36.51	4
Ant	364407	875	363382	2.15	233.14	12.51	7814	5.57	37.19	57.24	854
ArgoUML (H)	128332	3710	123061	20.45	20.20	4.95	12119	52.06	20.77	27.17	302
ArgoUML (I)	99969	3553	95174	19.01	9.25	4.93	9609	52.32	21.10	26.58	212
ArgoUML (N)	261617	4317	252211	19.95	16.65	3.56	24981	49.86	20.72	29.42	1072
ArgoUML	211625	4262	205325	19.50	12.84	5.08	17172	53.66	20.95	25.39	591
Bloat	3951	373	3834	4.72	5.01	4.86	408	41.26	23.32	35.42	4
BlueJ (H)	175674	3821	173267	12.16	817.25	6.71	12201	44.33	23.32	32.34	994
BlueJ (I)	35955	2435	34639	12.73	6.30	5.59	3688	41.52	28.53	29.94	39
BlueJ (N)	101848	3194	99693	13.42	718.47	6.22	8718	44.26	24.62	31.12	530
BlueJ	34944	2455	33630	12.37	6.16	5.66	3445	41.17	28.85	29.98	36
CViz	20237	1490	18252	20.78	5.70	3.24	2959	38.54	25.45	36.01	25
DINO (H)	30419	2364	29378	13.08	6.53	5.60	3602	39.45	28.23	32.31	30
DINO (I)	28813	2252	27824	12.68	11.42	5.68	3245	38.42	29.26	32.32	29
DINO (N)	29718	2287	28693	12.79	6.81	5.65	3425	38.66	28.34	33.00	29
DINO	29235	2205	28246	13.16	9.26	5.67	3210	37.83	28.88	33.29	28
DYNO	29183	2554	27806	16.41	43.11	4.75	3824	41.28	25.75	32.97	40
Denim	74044	3641	69550	14.57	9.88	6.08	6312	39.16	27.20	33.64	135
Doubly L. List	3744	310	3627	31.76	4.65	4.21	331	25.06	23.57	51.36	4
Forte	374450	7503	359666	19.17	40.77	6.15	37523	46.85	20.79	32.36	2154
GJ	3146	391	2961	8.14	5.08	4.53	422	34.38	30.87	34.75	3
HAT	258959	486	256290	15.64	5.83	5.74	1205	26.88	13.44	59.67	679
HelloWorld	2746	309	2633	5.73	4.90	4.68	330	34.68	32.59	32.74	3
HyperJ	3327	498	3194	6.64	4.65	4.46	538	37.54	31.65	30.81	3
JAX	33467	2600	32095	17.32	5.67	4.83	5753	43.26	21.60	35.14	39
JDI (H)	34166	2404	33024	13.72	12.44	5.76	3810	42.38	27.02	30.60	36
JDI (I)	33123	2391	32119	13.45	16.03	5.59	3734	39.95	28.19	31.86	33
JDI (N)	52540	2359	51396	14.51	9.00	6.06	4556	51.40	21.42	27.18	54
JDI	35114	2431	34098	14.82	59.21	5.51	3747	38.70	28.85	32.45	43
JEdit	75255	3221	73147	21.58	42.20	3.99	9581	48.06	16.85	35.09	127
JInsight (H)	172987	2437	172054	18.19	18.10	4.31	29253	68.09	18.94	12.98	356
JInsight (I)	24190	1912	23317	10.15	6.54	6.11	2410	38.41	30.90	30.69	23
JInsight (N)	101565	2363	100616	8.67	9.05	5.01	7256	67.04	16.08	16.88	123
JInsight	63053	2267	62136	18.34	15.92	4.82	10376	59.42	20.81	19.77	80
JTB	3261	434	3128	5.66	4.63	4.44	464	36.41	28.55	35.04	3
Jess	10685	580	10487	8.71	6.13	6.01	1082	45.04	13.72	41.24	8
Jython	27458	1406	26739	11.69	8.51	8.26	2573	35.35	16.23	48.41	24
Kacheck/J (H)	120286	425	119979	14.09	10.54	5.53	10837	31.52	10.01	58.47	121
Kacheck/J (I)	8043	392	7910	3.01	8.50	7.76	437	27.02	42.97	30.01	7
Kacheck/J (N)	24345	426	24058	10.97	6.73	6.05	2084	29.50	20.12	50.38	19
Kacheck/J	9372	413	9169	5.98	7.88	7.14	743	28.90	38.08	33.01	8
Kawa	9685	805	9423	11.80	4.81	4.14	1148	48.64	10.84	40.53	8
Linked List	3744	310	3627	4.16	142.37	4.21	331	25.06	23.57	51.36	6
Log FS	27111	2123	26223	11.84	22.85	4.23	2900	38.38	30.13	31.49	29
OpenVM (H)	183120	466	182241	39.87	5.09	3.04	72744	46.59	1.73	51.68	1988
OpenVM (I)	9373	413	9168	5.98	6.88	5.24	743	25.87	38.55	35.58	9
OpenVM (N)	97411	475	96532	30.89	5.29	2.98	24894	55.41	3.95	40.64	205
Ozone	16031	660	15474	11.65	7.05	6.42	2037	33.48	31.09	35.43	14
SableCC	21486	515	20989	16.68	5.73	4.67	1561	57.88	11.97	30.15	16
Satin	80415	1922	77055	18.52	8.83	4.40	13152	48.47	23.55	27.99	174
Schroeder	101659	1432	36371	17.55	9.04	5.29	3221	41.75	27.28	30.98	891
Skyline	2106	192	2071	5.12	4.87	4.62	206	34.09	33.61	32.30	2
Small L. List	18585	1688	4733	7.27	3.09	2.59	1746	26.24	23.76	50.00	26
Soot	11588	583	11309	16.82	4.90	4.75	1931	46.53	17.28	36.19	10
SwingSet	49343	2889	47855	16.64	7.35	5.48	5327	46.14	20.63	33.23	57
Toba	2920	328	2828	6.30	5.13	4.83	377	32.32	34.19	33.49	3
Tomcat	34688	1025	33873	9.38	6.41	6.07	2824	39.08	26.10	34.81	32
Medium	30069	1638	29036	12.93	7.40	5.16	3233	39.31	25.04	32.99	31
Average	66059	1752	63169	13.61	43.35	5.30	6965	40.61	24.32	35.06	214

NO	Number of Objects in the Object Graph
NUR	Number of Unique Roots in the Heap Root Set
ART	Number of Objects Accessible by Reference Traversal from the Root Set
PAL	Percentage of Aliased (Not Unique) Objects
AD	Average Depth from the Root in the Tree
ADF	Average Depth from the Root in the Tree After Folding
NRK	Number of Root Kids in the Resulting Ownership Tree
NC	Percentage not Confined
WC	Percentage Weakly (but not Strongly) Confined
SC	Percentage Strongly Confined
TT	Time Taken to Analyse a Given Heap Snapshot (In Seconds)



APPENDIX B: CLASS CONFINEMENT METRICS ACROSS CORPUS

Program	0%	10	20	30	40	50	60	70	80	90	99	100%	TNC
Aglets	5729	7	6	3	5	7	0	2	0	6	4	757	6526
AlgebraDB	1007	1	1	0	1	1	0	2	0	1	1	272	1287
ArgoUML (H)	9171	24	3	8	15	15	3	10	4	7	8	1620	10888
ArgoUML (I)	8763	22	5	6	13	14	3	10	4	7	5	1551	10403
ArgoUML (N)	9895	25	3	10	14	14	2	10	3	7	9	1629	11621
ArgoUML	9975	23	8	7	14	14	2	11	4	7	8	1677	11750
Bloat	1313	4	1	1	2	3	0	0	0	2	0	244	1570
BlueJ (H)	18908	28	10	8	17	17	3	10	8	9	14	1847	20879
BlueJ (I)	7528	18	4	7	12	16	4	5	6	2	6	1264	8872
BlueJ (N)	17382	25	6	7	12	17	3	13	9	8	12	1595	19089
BlueJ	7364	17	4	6	12	13	4	5	6	2	6	1263	8702
DINO (H)	5876	16	3	5	11	12	3	7	4	4	7	1306	7254
DINO (I)	5709	14	3	3	10	15	4	5	4	3	7	1271	7048
DINO (N)	5817	14	3	4	11	13	3	5	5	3	7	1291	7176
DINO	5732	14	3	3	9	18	2	6	5	3	7	1251	7053
DYNO	5277	15	7	5	11	14	2	7	5	1	8	1429	6781
Denim	10529	21	10	5	10	14	3	11	6	6	9	1731	12355
Doubly L. List	863	2	0	0	1	3	0	0	0	1	1	213	1084
Forte	48546	53	18	12	27	38	10	11	7	15	18	1831	50586
GJ	939	2	1	3	0	1	0	1	0	1	0	232	1180
HAT	10298	5	1	1	2	3	1	1	0	0	1	324	10637
HelloWorld	864	2	0	0	2	2	0	0	0	1	0	212	1083
HyperJ	1155	1	3	0	0	3	0	0	1	2	0	255	1420
JAX	6813	13	10	5	9	12	4	6	10	3	3	1415	8303
JDI (H)	6830	17	6	5	14	13	3	6	5	3	6	1333	8241
JDI (I)	5906	18	4	4	13	10	2	9	2	2	8	1345	7323
JDI (N)	9283	18	6	5	14	16	2	6	5	3	6	1322	10686
JDI	5918	18	7	4	9	9	2	8	4	2	7	1371	7359
JInsight (H)	11707	16	10	3	9	11	3	3	3	5	5	1081	12856
JInsight (I)	5566	10	6	2	9	11	1	3	2	2	6	997	6615
JInsight (N)	10710	17	6	5	11	12	3	2	2	3	4	1118	11893
JInsight	7676	16	9	3	9	13	2	5	2	4	5	1078	8822
JTB	1051	1	2	0	0	2	0	0	0	2	0	255	1313
Jess	1815	5	0	2	0	2	0	2	4	0	4	350	2184
Jython	4128	5	1	0	0	5	1	1	2	1	8	292	4444
Kacheck/J (H)	10180	6	1	3	1	0	0	0	0	0	6	467	10664
Kacheck/J (I)	1779	5	0	1	0	2	0	0	0	0	1	257	2045
Kacheck/J (N)	3414	6	1	3	0	1	0	0	0	0	6	468	3899
Kacheck/J	2281	6	1	0	2	1	0	0	0	0	1	432	2724
Kawa	2243	3	2	4	2	2	0	2	4	2	5	301	2570
Linked List	863	2	0	0	1	3	0	0	0	1	1	213	1084
Log FS	5517	12	5	4	11	11	1	5	3	4	8	1168	6749
OMV	2281	6	1	0	2	1	0	0	0	0	1	432	2724
Ozone	3787	6	2	1	4	6	0	3	1	1	2	395	4208
SableCC	2916	8	1	1	1	3	2	2	0	2	4	439	3379
Satin	7088	10	9	4	1	5	0	4	4	1	9	854	7989
Schroeder	6912	11	10	4	16	11	4	8	4	9	8	255	7252
Skyline	700	1	1	0	0	1	0	0	0	0	0	122	825
Soot	3471	5	3	2	1	5	0	4	0	1	2	317	3811
SwingSet	7179	16	10	10	14	13	2	6	1	3	7	1484	8745
Toba	833	4	1	3	1	1	0	1	0	0	0	213	1057
Tomcat	7771	10	8	3	3	11	0	6	5	1	4	482	8304
Total	345258	624	226	185	368	460	84	224	144	153	265	45321	393312
Percentage (%)	87.78	.16	.06	.05	.09	.12	.02	.06	.04	.04	.07	11.52	100

Strong/Weak/Not Confined Distribution Across the Snapshots

