

# Visualizing Reference Patterns for Solving Memory Leaks in Java

Wim De Pauw and Gary Sevitsky

IBM T. J. Watson Research Center, P.O. Box 704,  
Yorktown Heights, NY 10598 USA  
{wim, sevitsky}@watson.ibm.com

**Abstract.** Many Java programmers believe they do not have to worry about memory management because of automatic garbage collection. In fact, many Java programs run out of memory unexpectedly after performing a number of operations. A memory leak in Java is caused when an object that is no longer needed cannot be reclaimed because another object is still referring to it. Memory leaks can be difficult to solve, since the complexity of most programs prevents us from manually verifying the validity of every reference.

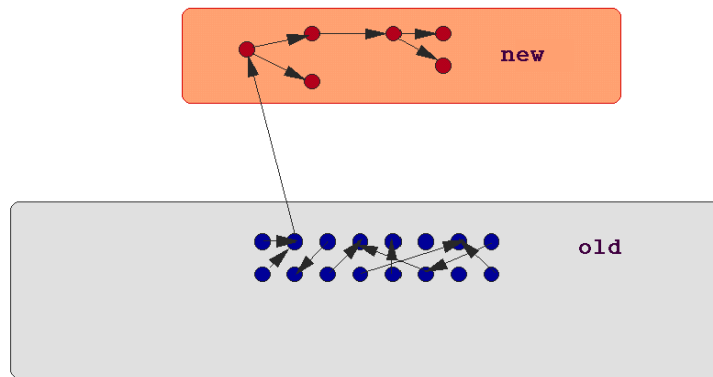
In this paper we show a new methodology for finding the causes of memory leaks. We have identified a basic memory leak scenario which fits many important cases. In this scenario, we allow the programmer to identify a period of time in which temporary objects are expected to be created and released. Using this information we are able to identify objects that persist beyond this period and the references which are holding on to them. Scaling this methodology to real-world systems brings additional challenges. We propose a novel combination of visual syntax and reference pattern extraction to manage this additional complexity. We also describe how these techniques can be applied to a wider class of memory problems, including the exploration of large data structures. These techniques have been implemented and have been proven successful on large projects.

## 1 Introduction

Complexity in software systems is still growing significantly. Modern languages like Java take away some of the burden of memory management by offering automatic garbage collection [1,2]. This feature can be a double-edged sword, however. Programmers may get the false impression that they don't have to worry about memory at all when using a garbage-collected language. In fact, a very common problem in Java is to inadvertently maintain a reference to a temporary object long after it is needed, preventing it from being reclaimed, and in effect causing a "memory leak". Thus, even in Java, programmers need a way to identify leaking objects, and to tackle the more difficult task of discovering who is holding on to these objects and why. In this paper we propose a new methodology to uncover these memory leaks and, more important, to identify their causes.

Our framework for solving memory leaks begins with the observation that memory leaks often occur according to a simple scenario, which we illustrate below. In this scenario, there is a self-contained operation (for example, the display of a temporary dialog window) in which temporary objects are created. By the end of the operation (in this example when the dialog window is closed) we expect all of these temporary objects to be released. Only some, however, are actually released. Many programs with memory leaks fit this pattern in one way or another. Later in the paper we discuss more complex variations of this scenario.

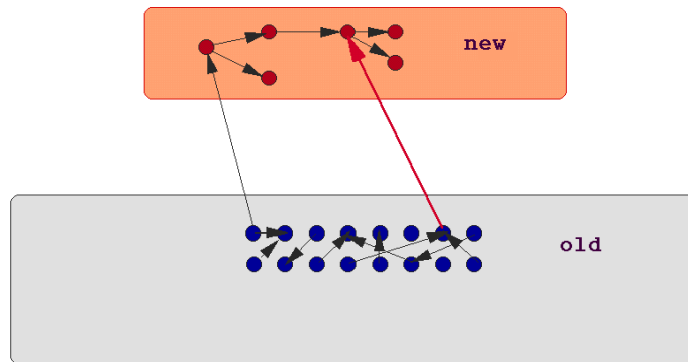
Figure 1 illustrates this scenario in more detail. A program reaches a stable state; its object population is shown in the lower area in the figure. Then we perform a temporary operation, for example opening a window. The new objects allocated for this operation are shown in the upper area. When the user closes the window, the program will typically set the reference between the old objects (lower) and the new objects (upper) to null. Ideally, the next garbage collection will then clean up all the new objects which were associated with the window.



**Fig. 1.** Ideal scenario: reference from old (lower) to new (upper) objects is removed

What happens very often, however, is that during such a temporary operation, other old objects may get to know about some of the new objects. This situation is shown in Figure 2. A typical case is a registry (in the lower area of the figure) that acquires a reference to a new object (as shown by the solid red arrow on the right). The programmer may not be aware of this hidden reference, and may fail to set this reference to null at the end of the operation. As a result the garbage collector will not reclaim some objects that were meant to be collected after the temporary operation has finished.

In section 2 of the paper we describe our approach for identifying and solving memory leaks which fit this basic scenario. The key idea is to have the programmer mark a period of time corresponding to the expected lifetime of



**Fig. 2.** Memory leak: vestigial reference (right arrow) prevents garbage collection of some of the new objects

some temporary objects (for example, the lifetime of the window in the above example). Using this information we are able to identify objects that persist beyond this period and the references which are holding on to them.

Scaling this methodology to real-world systems brings additional challenges. Even when we have identified leaking objects and references to them, some further exploration will usually be required by the programmer in order to fully understand the results. For example, the programmer may want to dig deeper and understand why an object was created in the first place, or distinguish true leaks from artifacts like cached objects which are retained intentionally. Searching through individual object references is not practical in anything but the simplest program, given the large number of objects and the richness of their interconnections in most programs. In section 3 we propose a novel combination of a new visual syntax and reference pattern extraction to address these needs.

In section 4 we explain the practical use of these techniques to solve memory leaks that fit the basic scenario we have described above. In section 5 we discuss how these techniques can be applied to a wider variety of memory problems.

All of the work described here has been done within the context of Jinsight [3], a research tool for visualizing and exploring many different aspects of a Java program’s run-time behavior. To use Jinsight, the programmer first runs the target program using an instrumented Java virtual machine, which can produce traces of various types of information about the program’s execution. The Jinsight visualizer can then be used to explore the program’s behavior from various angles.

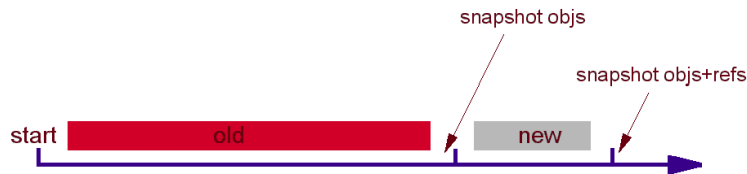
In section 6 we explain our implementation, the integration of the memory leak finder within Jinsight, and some practical results. In section 7 we discuss related work. We conclude with possible future directions.

## 2 Basic scheme for finding and solving a memory leak

Java provides the programmer with automatic garbage collection. The garbage collector will periodically free all the objects that can no longer be reached from the running program. Problems arise when an object that is more permanent still refers to a temporary object after the temporary object is no longer needed (as shown in Figure 2).

Our approach to memory leaks is based on the observation that many memory leaks occur during well-defined operations which are supposed to release all of their temporary objects upon completion. If we let the programmer tell us the boundaries in time of such an operation, we can use this information to greatly simplify the discovery and diagnosis of memory leaks.

We have two goals: to identify leaking objects, and then to find out who is erroneously referring to these objects, thereby causing the leak. Given the user-supplied boundaries of a “critical” operation, we can differentiate between “old” objects which existed before the operation, and “new” objects which were created during it (see Figure 3 below). To identify the leaking objects, we find the new objects that were created during this critical operation but cannot be reclaimed at the end. To identify the cause of the leak, we find the more permanent objects which are referring to them. These more permanent objects include the old objects which existed before the critical operation, as well as class objects, which are the holding place for static data members in Java. We also note other types of references, such as native code and local variables on the stack which could be referring to the leaking, new objects.



**Fig. 3.** Timeline of program execution: old objects are created before, and new objects during a critical operation

Using the Jinsight instrumented Java virtual machine, the user takes two snapshots of the program’s object population: one just before, and one after the critical operation. Each snapshot will have an inventory of all the objects in the heap at that point in time. Each of these objects has a unique identifier. When taking a snapshot, we only include objects that are not garbage collectable; we exclude objects that could be reclaimed if the garbage collector were to run at this time. To accomplish this, we traverse the heap using the Java virtual machine’s internal garbage collection routines.

The objects that appear new in the second snapshot when compared to the first snapshot are the ones that could not be collected after the critical operation completed - the leaking objects.

The next step is to find out who is pointing to these objects. In order to do this, we capture some additional information during the second snapshot. In addition to recording the inventory of the objects present in the heap, we also capture all the references between these objects, as well as references to these objects from variables on the stack and native code. We can then find out if an old object, class object, Java local variable, or native code reference is pointing to a new object.

### 3 Managing complexity

The approach described so far works well for simple programs. Many programs, however, have data spaces that are large and densely interconnected. Although this basic approach goes a long way toward narrowing the problem, the results are often still too large and complex to explore manually, for a number of reasons.

Very often a program with a memory leak will have thousands of objects unexpectedly remaining active because of a chain-like effect, where an object that is inadvertently retained will in turn hold on to more objects, preventing them from being reclaimed, and so on. The real cause may be buried deep in an immense pile of uncollected objects. In addition, not only the size but also the complexity of the information may prevent the programmer from solving the problem. Ultimately the programmer's goal is to understand why a reference was created and not severed at the right time, and this will often require understanding the context in which these objects and references exist.

We propose a novel methodology, combining a new pattern extraction technique and visual syntax, to allow the user to work with this high level of complexity. These techniques will enable the user to find and solve memory leaks which follow the basic scenario we have described, and also to solve more general types of memory problems, which we describe in section 5.

#### 3.1 Reference patterns

Fortunately, most programs with large data spaces also have much repetition in their data structures. We take advantage of this fact and extract reference patterns, which are a concise way to represent the interconnections among large numbers of objects. Reference patterns allow us to work with the essential structure of a data space in a simplified, aggregated form. Reference patterns are analogous to execution patterns [4], which make repetitive execution sequences more understandable by eliminating redundancy and bringing out their inherent structure.

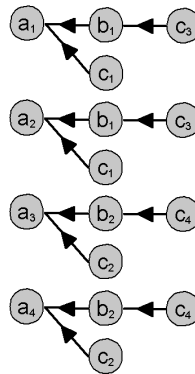
The goal of the reference pattern extraction scheme we present here is to produce a concise representation which highlights how groups of objects refer to one another. We begin with a user-defined starting set of objects. We would like

to understand the patterns of reference from other objects directly or indirectly to these objects. The result of the pattern extraction process is a forest of trees, which we extract from the directed graph of object references leading in some way to the starting set. We describe each tree below:

- The root of each tree represents all objects of a single class in the starting set.
- Each node of a tree represents all objects of a single class which refer to at least one of the objects represented by the parent of this node.

In other words, we are grouping objects by a combination of class and what actual objects they refer to. The details of the pattern extraction algorithm are given in the appendix.

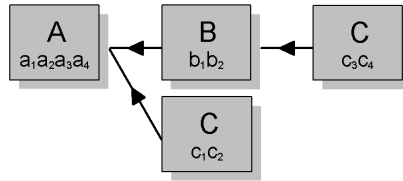
The figures below illustrate an example of reference pattern extraction. Figure 4 shows the actual structure of the data, where  $a_1, \dots, a_4$  are instances of class  $A$ ,  $b_1, \dots, b_4$  are instances of class  $B$ , etc. In this example, the user would like to see the pattern of references to objects of class  $A$ , and specifies these objects as the starting set.



**Fig. 4.** Actual objects and references

Figure 5 shows how the structure of the data would be summarized. Since the starting set in this case has all objects of the same class, the result of the pattern extraction is a single tree. Note also that  $c_1$  and  $c_2$  are grouped together, but segregated from  $c_3$  and  $c_4$ , since these two groups of objects have differing patterns of references even though they are of the same class.

To apply this to the basic memory leak scenario, the user would typically specify the starting set to be all the new objects which cannot be reclaimed, and then extract the pattern of references to these objects, in order to understand



**Fig. 5.** Result of pattern extraction

who is pointing to them. There are also other choices of starting set that are useful for working with this and other memory scenarios; we discuss these further in sections 4 and 5.

Rather than looking at patterns of references to objects of a given starting set, it is also possible to reverse the sense of the pattern extraction, to bring out the structure of references emanating from a starting set. Each node would then contain all the objects of a given class that are referred to by any object in the parent node. Applications of this type of reference pattern extraction are discussed in section 5.

### 3.2 Visual syntax

By extracting patterns of reference relationships among objects, we are shifting our focus from individual objects to groups of objects, allowing the user to work with a greater degree of complexity. Now we present a new visual syntax that makes this information easy to understand and to explore interactively. This presentation of the information, beyond helping the user understand patterns of object references, has additional features specifically for solving memory leaks of the type we have been describing. Figure 6 shows an example illustrating the basic elements of this syntax.

In this example, our starting set is the set of new objects that were created but not collected during the period of the critical operation. We apply the reference pattern extraction algorithm to this set, so that we can understand who is referring to these objects. The gray area on the left contains the starting set of objects, grouped by class. Each icon in the gray area is the root of a reference pattern tree, which expands to the right.

Each icon in the view represents a group of objects of the same class, with the same pattern of references. In other words, each is a node in the reference pattern tree. The color denotes the class of objects in the group. Twin squares represent a group with two or more objects. A diamond represents a class object, which contains the static members of a class. A diamond with a square behind it shows a group that includes the class object for that class. Flying over an icon with the cursor will show more detailed information about that group of objects, in the status line below.

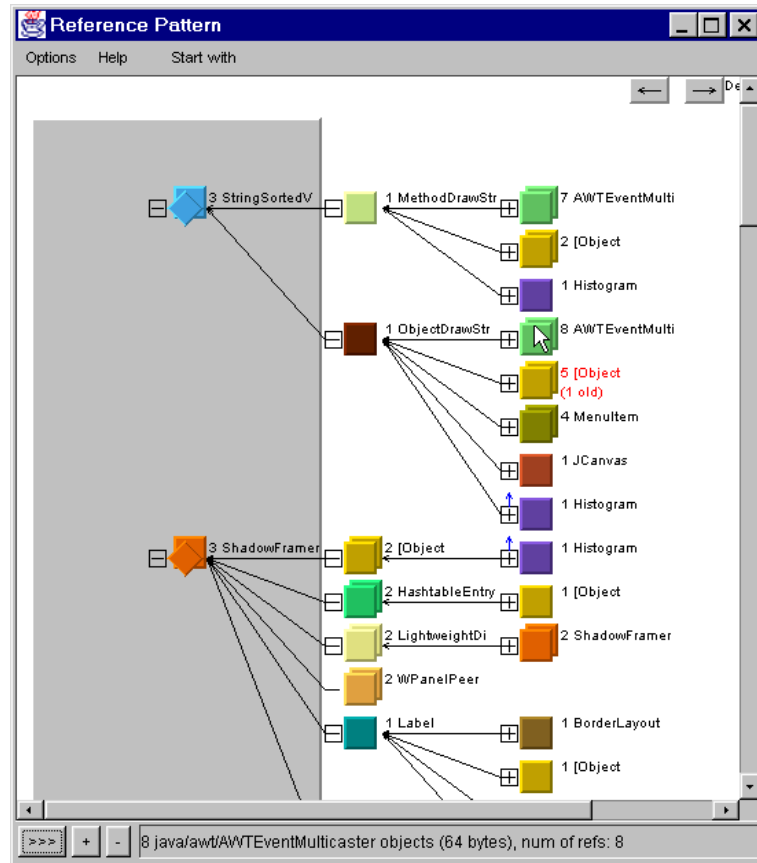


Fig. 6. Visualization of reference patterns

The arrows between the nodes are references. They also indicate the direction of reference between groups of objects, in this example from right to left. The direction can be also reversed to show the pattern of references from (rather than to) the starting set.

By default, each tree is shown expanded to an abbreviated depth, which may be set by the user. Every node which is marked with “+” may be expanded individually by clicking there.

While the aggregation of objects into groups is the key to uncovering patterns, sometimes this same grouping can hide important differences among individual objects. For this reason we allow the user to “ungroup” a node to work with objects individually. The view will then show a separate icon for each of the objects in the group. To the right of each of these icons will be the pattern of references to that object only. Figure 7 shows the result of ungrouping the set of five [Object] arrays (the notation means array of Object) from Figure 6.

References between objects form a directed graph. Presenting a graph in a

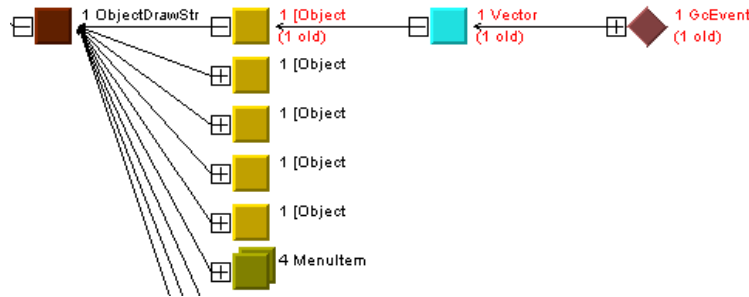


Fig. 7. Result of ungrouping the set of five *[Object]* arrays

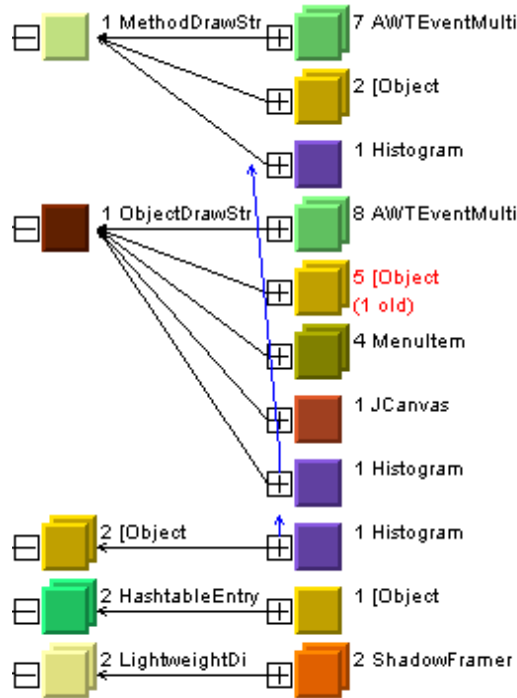
clear and uncluttered form can be a challenge when there are a large number of nodes and interconnections. Therefore, we have chosen to lay out the reference pattern graph as a set of trees. We also believe this format will lend itself naturally to the ways users will explore the data. We expect the user will often begin exploring from a group of objects of interest, and then proceed to progressively expand in a given direction, to see, for example, the objects referring to them. We describe some of these scenarios in more detail in later sections.

Because the view presents the graph as a forest of trees, the same node can appear more than once. An example is shown in Figure 6, where a single *Histogram* object, in purple, appears three times in the view. The way we indicate that an identical node (representing all the same objects) has appeared earlier in the view is with a small blue arrow to the left of the icon, pointing upward. Clicking on the arrow will expand it to point to the first occurrence of this object or set of objects, as shown in Figure 8.

Each node is labeled with the class and the number of objects it represents. The color of the label distinguishes old from new objects. A red label indicates that at least one object in the group is an old object, created before the critical operation began. A node with a black label has only new objects, created during the critical operation. These cues are intended to make it easier to find the offending reference causing a memory leak. In this example, one object labeled in red of type *[Object]* is still pointing to an object labeled in black, preventing that object from being collected. Another aid to finding these references quickly is an option which reduces the view to only those nodes leading to old or class objects within a given depth.

#### 4 Solving memory leaks in practice in the basic memory leak scenario

So far we have described a basic scheme for solving a memory leak by comparing memory before and after a user-identified critical operation, and we have presented new techniques for exploring the results of this step given their complexity. In this section we describe our methodology, including some additional



**Fig. 8.** Clicking on the short blue arrow points to an earlier occurrence of this node

techniques to find and solve memory leaks in practice. We still restrict ourselves in this section to our basic memory leak scenario, where the user is able to identify a critical operation which allocates memory that is unintentionally retained afterward.

The user runs his or her program under Jinsight's instrumented Java virtual machine. Just before the critical operation, the user issues a command to take a snapshot of objects. After the critical operation has completed, the user issues another command to take a snapshot of objects and references. The user then loads the traces into the Jinsight visualizer, and opens the Reference Pattern View, the view we described in the previous section.

The user then selects a starting set of objects from a number of choices. The starting set will typically be all the new objects created during the critical operation that are still in existence afterward. This starting set will then appear in the gray area on the left, showing the objects that are leaking. The user then looks on the right side of the visualization for objects with a red label (these are the old objects) or for diamonds (class objects). These are more permanent objects which are referring, directly or indirectly, to one of the new, temporary objects.

The example in Figure 7 shows an old *[Object* (an array) referring to a new *ObjectDrawStr* object, which is no longer needed and should have been reclaimed. After expanding the references we see that *[Object* is referred to by a *Vector*, which is part of a *GcEvent*. The source code for this example revealed that *GcEvent*'s *Vector* (which has an array inside) is a registry of subscribers to that event type. The *ObjectDrawStr* object had at one time been a subscriber to *GcEvent*, but in this case the programmer forgot to have the *ObjectDrawStr* cancel its subscription to that event.

The user can also look for nodes marked with a small colored circle, as shown in Figure 9. These are objects held in memory by a source other than a Java object, such as a local variable on the stack or a native method.

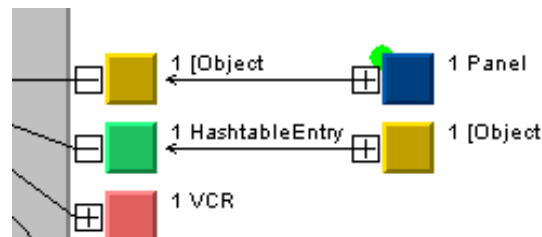


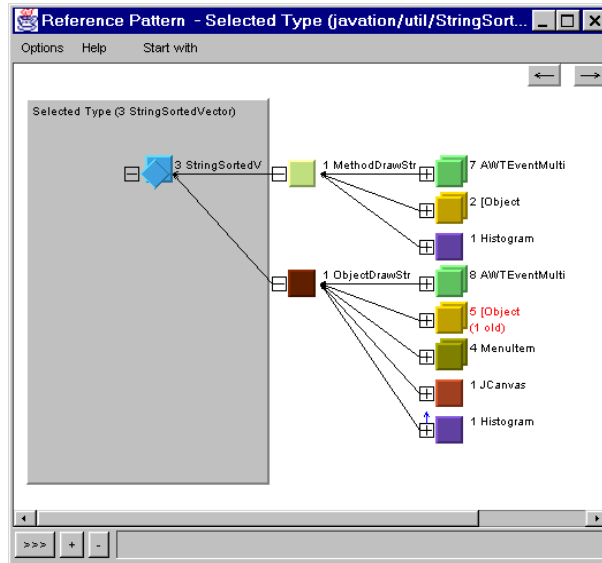
Fig. 9. *Panel* object (with green circle) is pinned by a variable on a Java stack

The tool also allows the user to focus on the new objects of a single class. For example, to find out who is pointing to all the new, leaking *StringSortedV* objects, the user can select just these objects as the starting set, as shown in Figure 10.

Typically there will be a large number of interrelated objects leaking together. In most cases, fixing one spurious reference will have a domino effect; freeing one object will usually allow the garbage collector to free a chain of objects. Therefore we have found it is a good idea to fix one problem at a time, and rerun the program after each fix before performing any further analysis.

With this in mind, it is a good idea to focus first on problems with application-level objects, since these objects will often prevent lower-level library objects from being reclaimed. Jinsight allows the user to start with the set of new objects, where JDK and array objects have been excluded.

So far our approach has been to start with the objects that are leaking and then see who is pointing to them. A complementary approach is to start with the sources that could anchor objects in memory, and see which objects they are holding on to. It may be useful, for example, to look at all objects that are directly or indirectly referred to by static data members. To do this, our tool allows the user to explore reference patterns starting from the class objects. In this case, the user sets the direction of pattern extraction to look for references from, rather than to, the starting set of class objects. Similarly, the user can explore the pattern of references from local variables and from native code, as



**Fig. 10.** All the new, leaking *StringSortedV* objects, and the objects holding on to them

shown in Figure 11.

When a spurious reference has been found, the user may want to fix the code by setting this reference to null. If it is not immediately apparent where the appropriate place in the code is, the user may need to explore further to better understand the context of the object. One way to do this is to continue exploring with the reference pattern visualization, expanding references to or from this object to see what other objects it is connected to.

Another way to understand the wider context is to use some of Jinsight's other capabilities. The Jinsight visualization environment can be used to reveal thread behavior, execution patterns, object population, performance bottlenecks, reference patterns and memory leaks. It has a number of views, each allowing the user to explore the run-time behavior of a program from a different angle. The user can navigate from one view to another and correlate data across views. The instrumented Java virtual machine has a number of user-settable options for tracing a program: it can trace method enters and exits, object creation and collection, or take snapshots of object population and object references.

These different features of Jinsight can be used separately or in combination. For example, to find memory leaks, as we have seen so far, we only need to take two population snapshots - no further tracing is required. However, if we would like to understand the execution sequence which is causing the leak, the user can rerun the program with full tracing turned on during the appropriate operation, in addition to taking the memory snapshots. This extra information will allow the user to find out the execution history for the objects of interest.

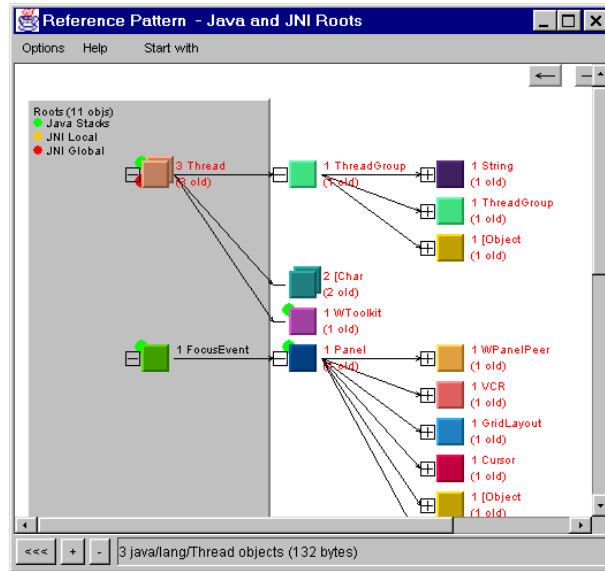


Fig. 11. All objects pinned by Java local variables and JNI roots

## 5 Applications beyond the basic memory leak scenario

### 5.1 Finding and solving other types of memory leaks

We have premised our work so far on the observation that the expected lifetime of objects will often match the duration of an operation which the user can identify. While we found this scenario occurs often in practice, we also know that there are many situations which do not fit this mold, where memory leaks occur. Even so, the techniques presented so far, with some variation, can still be useful to solve these more complex cases.

One common variation is where a critical operation is identified in which only some of the objects it creates are meant to disappear at the end, while other objects are intended to be more permanent. To solve a memory leak in this scenario we can still apply our methodology to reveal the new objects remaining after the critical operation. However, these new objects will now be a mix of correctly allocated objects and objects that should have been released. Our experience with real cases has been that the reference pattern visualization provides an easy way to sift through these results and determine which objects are legitimately persisting and which are memory leaks. Once the leaking objects are identified, the methodology is the same as in the basic scenario for discovering the offending references to these objects.

Another very common scenario is a repetitive operation which may be causing a memory leak. For example, a server is handling requests from clients, and runs out of memory after a while. In this case, we can examine the incremental effect of one or more of these requests. In order to do that, we can take a snapshot

before and after a few cycles of the repetitive operation. We can then see which objects were created during the interval that could not be reclaimed, and who is holding on to them. Since we are dealing with repetitive operations, we expect that fixing the source code for one cycle will apply to all cycles.

Note also that we may have a combination of a repetitive operation and the previous scenario, where some of the objects are meant to persist beyond each cycle. Again, exploring the pattern of references visually helps differentiate the leaking objects from the others.

## 5.2 Exploring data structures

Besides solving memory leaks, there are many other reasons why a programmer may need to understand the data structures of a program: for debugging, performance analysis, or understanding the way an existing program works in order to maintain it. The pattern extraction and visualization techniques we have introduced can be useful in these cases also, enabling the exploration of large, complex data structures in general.

We give an example of how visualization of reference patterns can be used to understand the structure of a hash table. A programmer may want to look at the physical structure of a hash table to determine if a hash function is producing a balanced distribution. Figure 12 shows a *Hashtable* of *JObjects* (values), keyed by *String*. This particular hash table is composed of about 1500 objects.

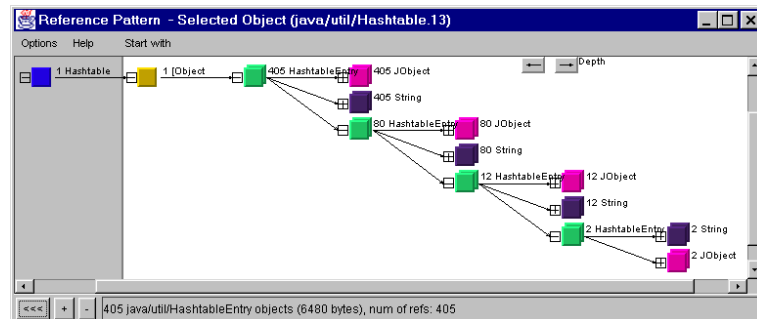


Fig. 12. Reference pattern of a hash table

To begin, the user selects the *Hashtable* object as the starting set. The user then sets the direction of arrows to point from left to right, to see the pattern of references from the *Hashtable*. We see the *Hashtable* object is shown on the left, and it is referring to an array object (*Object*). This array is the main table of linked lists in the hash table. Each linked list represents a hash bucket, and is composed of *HashtableEntry* objects, shown in green. Each *HashtableEntry* has a (key, value) pair, in this case (*String*, *JObject*). From the visualization we can see that 405 *HashtableEntry* elements are at the head of a linked list, 80 elements

are at the second position of their list, 12 are at the third position and 2 are at the fourth position. We can quickly see that the distribution of this hash table is reasonable, since most of the entries are at the head of their list. The user may want to explore some individual objects further by ungrouping a particular node.

## 6 Implementation and experience

The visualization of reference patterns is implemented as part of the Jinsight research tool, available for free from IBM alphaWorks [3]. Jinsight traces the target program using an instrumented version of the Sun JDK Java virtual machine. The user can specify various tracing options to control the type of information recorded. For the analysis of memory problems, we recommend taking snapshots of objects and references, although it is also possible in Jinsight to gather some of this information with a more detailed trace of the execution sequence. The advantage of a snapshot over full tracing is that the amount of trace information collected will usually be significantly smaller. In addition, there will be no perturbation of the running program except while the snapshots are taken, and this usually takes less than a second. This is crucial for analyzing programs that are running for long periods of time.

If needed, the user may turn on full tracing in parts of the program in addition to taking the memory snapshots. This extra information can give the user more insight into the execution history of the objects of interest. The user could find out which objects created these objects, for example, or what sequences of methods were invoked on them.

The Jinsight visualizer is built in 100% pure Java. Figure 13 shows the basic architecture of Jinsight. A transceiver reads events from a trace file or socket and populates Jinsight's dictionaries and models. The Jinsight dictionaries store information about basic object-oriented entities, such as objects, methods, classes, and threads. The models contain more abstract information about the execution of a program, such as the history of each thread's stack over the course of execution, or the references between objects. Modeling the information formally allows us to flexibly combine the information and present it from many angles in the various views. Each view highlights different aspects of the target program behavior, such as the pattern of messages between objects, the program's time and resource bottlenecks, or, as we have described here, the structure of references between objects.

We have used the memory leak tool successfully on many large projects, and it has scaled well to the demands of real-world applications. In an early field test Jinsight was used on a large commercial Java product, where a team of ten developers had been struggling to diagnose memory leaks which would have delayed the delivery of the product. Using Jinsight, the team was able to increase the rate of memory leak discovery and diagnosis by two orders of magnitude, and was able to ship the product on time.

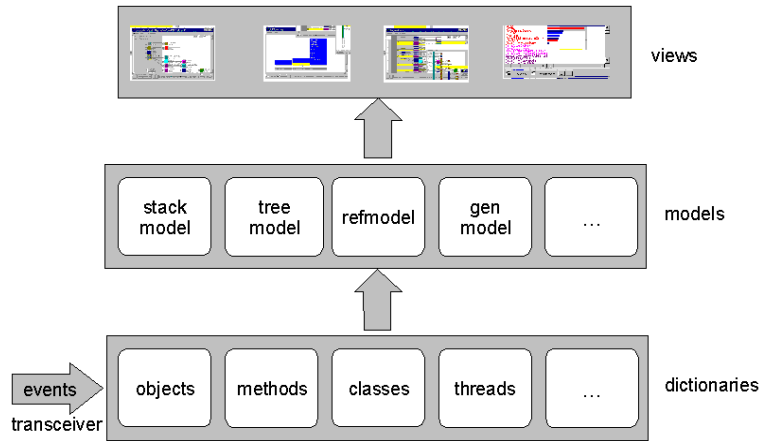


Fig. 13. Architecture of the Jinsight visualizer

Table 1 illustrates some statistics from three projects where Jinsight was used to solve memory leaks. The first one shown was a word processor, and the second an industrial-strength web server. The third was an application of Jinsight to itself to diagnose memory leaks. All of these numbers were obtained running the target program and Jinsight on a 200 MHz Pentium Pro PC with 128 MB of RAM.

Program	Total number of objects	Number of leaking objects	Time to read trace	Time to calculate & show leaks	Memory needed by Jinsight	Trace length
Word processor	25 K	3300	15 s	2 s	17 MB	1.2 MB
Web server	20 K	35	15 s	< 1 s	13 MB	0.9 MB
Jinsight	110 K	800	90 s	40 s	40 MB	3.5 MB

Table 1: Three applications where Jinsight was used to diagnose memory leaks

## 7 Related work

A significant amount of work has been done on tools that perform static checking of source code. Some of these use annotations [5] to provide ways of expressing assumptions about memory allocation, initialization and sharing, or assumptions about interfaces, variables and types. In [5] the tool calculates the constraints to satisfy these assumptions at compile-time and flags the corresponding places as possible bugs. In [6] B. Blanchet describes an Escape Analysis, to determine if the lifetime of data exceeds its static scope. This technique can be used for stack allocation. This work takes into account polymorphism. Static analysis tries to cover all possible executions of a program. In order to achieve this, however,

it must be very conservative. Therefore it may be very difficult in practice to predict the actual behavior of a program. Dynamic analysis examines exactly this behavior. Although dynamic analysis depends on a test set, and it will not catch all possible problems, it is very often a more efficient aid in solving memory problems.

The Heap Analysis Tool [7] from Sun is an example of a dynamic analysis tool. It allows the programmer to create snapshots of the object population and references, and to browse through the results. However, the amount of information can still be overwhelming for anything other than a simple case, since the browsing interface is very limited. The inability to see the high-level structure of objects and references, and to explore the result set interactively is a hindrance in practice when working with memory problems. Vendors KL Group (JProbe [8]) and Intuitive Systems (Optimizeit [9]) also provide memory heap browsers as part of their performance analysis tools. Although these tools have interactive browsing interfaces, neither provides a way to see overall patterns when there is a large numbers of objects, so finding the cause of a memory leak can still be a slow process.

## 8 Conclusion

In this paper we have presented a new methodology for solving memory leak problems in Java. The combination of the following three novel features enables us to tackle memory problems in very complex applications:

- We allow the user to identify a critical operation in which temporary objects are expected to be created and released.
- We exploit the fact that there is usually much repetition in the object structures of a complex system, and we extract patterns from the object reference space. Instead of individual objects, the user can now work with groups of objects with similar reference patterns.
- In order to present the results from this synthesized, more abstract space of reference patterns, we introduce a new visual syntax. This visualization points the user immediately to the causes of a memory leak. In addition, it allows the user to navigate and explore the results, in order to understand the dynamic context in which the leak occurred.

Although our initial and motivating objective was to solve a restricted, though still important, class of memory problems, we showed how these techniques were extended and applied to a wider variety of memory problems, such as solving more complex memory leak scenarios, and examining large data structures.

We have tested and integrated the tool into our visualization system, Jinsight. It has been successfully used by many users to solve memory problems in large, real-world systems.

In the future we would like to expand the range of memory problems which we are able to address. An area we are currently pursuing is to allow the user

to specify complex query criteria in the visualization of reference patterns. One application of this will be to help the user distinguish temporary from permanent objects in much more complex memory leak scenarios. Another use will be to allow the user much greater flexibility in the exploration of complex data structures in general.

Another area of future work is the ability to perform what-if studies on large data structures. By simulating the Java garbage collection algorithm within our visualization system, we will enable the user to experiment by observing the effect that severing a reference would have on the reclamation of objects.

Our techniques could be applied to other object-oriented languages with garbage collection and run-time type information, such as Smalltalk and Eiffel. The basic functionality that we would need is the ability at run time to scan the heap for all live objects and to report all references between these objects.

## Acknowledgements

We would like to thank our colleagues Olivier Gruber, Erik Jensen, Ravi Konuru, John Vlissides and Mark Wegman for their participation in the practical work realizing Jinsight and for many valuable technical discussions. We are also grateful to all the people using Jinsight who have given us feedback and suggestions on our strategy for solving memory leaks.

## References

1. Wilson, P.: Uniprocessor garbage collection techniques. Proceedings of the International Workshop on Memory Management, St. Malo, France, September 1992. Lecture Notes in Computer Science, Vol. 637. Springer-Verlag (1992) 1-42
2. Jones, R., Lins, R.: Garbage Collection. John Wiley and Sons (1996)
3. De Pauw, W., Jensen, E., Konuru, R., Gruber, O., Sevitsky, G. and Vlissides, J.: Jinsight, A Visual Tool for Optimizing and Understanding Java Programs. IBM Corporation, Research Division. Information and tool available at <http://www.alphaWorks.ibm.com/formula/jinsight> (1998)
4. De Pauw, W., Lorenz, D., Vlissides, J., and Wegman, M.: Execution patterns in object-oriented visualization. In Proceedings of the Fourth Conference on Object-oriented Technologies and Systems (COOTS), Santa Fe, New Mexico (1998) 219-234
5. Evans, D.: Static detection of dynamic memory errors. In Programming Language Design and Implementation, May 1996. ACM SIGPLAN Notices, 31(5) (1996) 44-53
6. Blanchet, B.: Escape analysis: correctness, proof, implementation and experimental results. ACM SIGPLAN-SIGACT, Proceedings of the 25th Annual Symposium on Principles of Programming Languages, San Diego, CA, January 1998. (1998) 25-37
7. Foote, W.: Heap Analysis Tool. Sun Microsystems, Inc. Described in <http://developer.java.sun.com/developer/earlyAccess/hat> (1998)
8. JProbe. KL Group, Inc. Described in <http://www.klg.com/jprobe> (1999)
9. Optimizeit. Intuitive Systems, Inc. Described in <http://www.optimizeit.com> (1999)

## Appendix. Algorithm for the extraction of reference patterns from an object space

We build a forest of trees with nodes representing object sets, using the following algorithm:

1. The user first indicates a starting set  $A$  of objects. The choice of this starting set will be motivated by the specific problem at hand. For example, the user might ask to start the exploration of the object reference space from the “new” objects.
2. Partition the starting set  $A$  into subsets,  $A_1, A_2, \dots, A_n$ , of objects grouped by class. Each of these subsets will become the root of a reference pattern tree.
3. To each of these starting subsets  $A_1, A_2, \dots, A_n$ , apply the recursive operation (step 4):
4. For a subset of objects,  $S_i$  (containing objects of the same class), which is a node in a tree:
  - (a) Create a new set,  $R_i$ , of all the objects referring to any object in  $S_i$ .
  - (b) Partition this set  $R_i$  into subsets of objects, grouped by class:  $R_{i1}, R_{i2}, \dots, R_{im}$ .
5. These subsets  $R_{i1}, R_{i2}, \dots, R_{im}$ . will become the child nodes of the node representing  $S_i$ .
6. Apply the recursive operation (step 4) to every child node in the tree, until reaching a minimum tree depth, set by the user, or as needed when a user interactively expands a node of the tree in the visualization.

To reverse the direction of references in this algorithm, replace “referring to” in step 4(a) by “referred to by”.